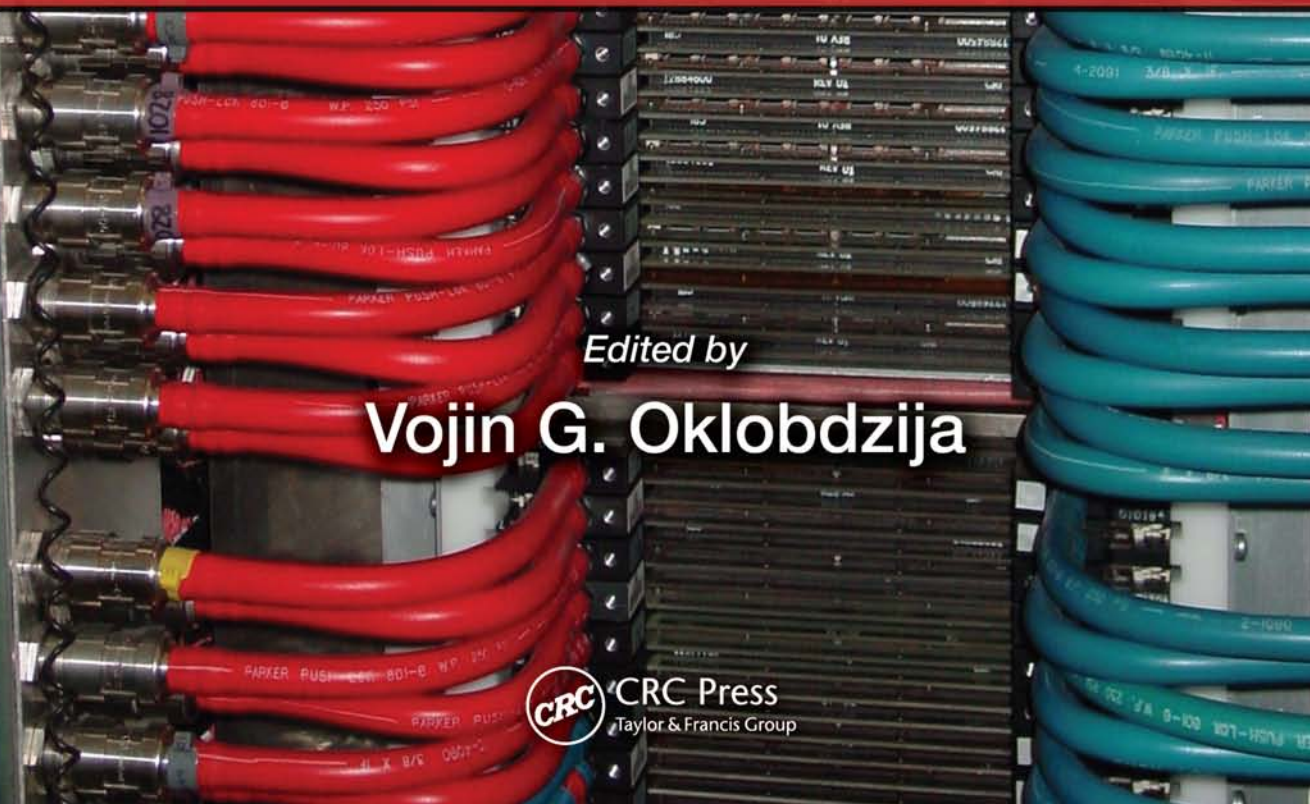
The Computer Engineering Handbook

Second Edition

# Digital Systems *and* Applications



*Edited by*

Vojin G. Oklobdzija

CRC Press
Taylor & Francis Group

**How to go to your page**

In this eBook, each chapter has its own page numbering scheme, consisting of a chapter number and a page number, separated by a hyphen.

For example, to go to page 5 of Chapter 1, type 1-5 in the "page #" box at the top of the screen and click "Go." To go to page 5 of Chapter 2, type 2-5… and so forth.

# The Computer Engineering Handbook

## Second Edition

*Edited by*

Vojin G. Oklobdzija

Digital Design and Fabrication

Digital Systems and Applications

# Computer Engineering Series

Series Editor: Vojin G. Oklobdzija

*Coding and Signal Processing for Magnetic Recording Systems*
**Edited by Bane Vasic and Erozan M. Kurtas**

*The Computer Engineering Handbook Second Edition*
**Edited by Vojin G. Oklobdzija**

*Digital Image Sequence Processing, Compression, and Analysis*
**Edited by Todd R. Reed**

*Low-Power Electronics Design*
**Edited by Christian Piguet**

# DIGITAL SYSTEMS AND APPLICATIONS

*Edited by*

**Vojin G. Oklobdzija**

University of Texas


CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2008 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-0-8493-8619-0 (Hardcover)

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

No part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC) 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

**Trademark Notice:** Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

**Library of Congress Cataloging-in-Publication Data**

Digital systems and applications / editor, Vojin Oklobdzija.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-8493-8619-0 (alk. paper)
1. Computer engineering--Management. 2. Systems engineering--Management. I. Oklobdzija, Vojin G. II. Title.

TK7885.D56 2008
621.39--dc22 2007023257

**Visit the Taylor & Francis Web site at**
**http://www.taylorandfrancis.com**

**and the CRC Press Web site at**
**http://www.crcpress.com**

# Preface

## Purpose and Background

Computer engineering is a vast field spanning many aspects of hardware and software; thus, it is difficult to cover it in a single book. It is also rapidly changing requiring constant updating as some aspects of it may become obsolete. In this book, we attempt to capture the long-lasting fundamentals as well as the new trends, directions, and developments. This book could easily fill thousands of pages. We are aware that in this book, some areas were not given sufficient attention and some others were not covered at all. We plan to cover these missing parts as well as more specialized topics in more detail with new books under the computer engineering series and new editions of the current book. We believe that the areas covered by this new edition are covered very well because they are written by specialists, recognized as leading experts in their fields.

## Organization

This book deals with systems, architecture, and applications and contains seven sections.

Section I is dedicated to computer architecture and computer system organization, a top-level view. Several architectural concepts and organizations of computer systems such as superscalar and vector processor, VLIW architecture, servers, parallel systems, as well as new trends in multithreading and multiprocessing, are described. Implementation and performance-enhancing techniques such as branch prediction, register renaming, virtual memory, and system design issues are also addressed. The section ends with a description of performance evaluation measures and techniques, which are the ultimate measure from the user's point of view.

Section II deals with embedded systems and applications. As the ability to integrate more transistors continues, the chip is turning into a system containing various elements needed to serve a particular application.

Section III describes important digital signal processing applications and low-power implementations.

Section IV deals with communication and networks, followed by Section V, which deals with input and output issues such as circuit implementation aspects, parallel I/O, algorithms, read channel recording, and issues related to read channel and disk drive technology.

Section VI is dedicated to operating systems, which manage the computer system operation and host the application software.

The final section (Section VII) is dedicated to new directions in computing. Given the rapid development of computer systems and their penetration into many new fields and aspects of our everyday life, this section is rich with chapters describing many diverse aspects of computer usage and potentials for use. It describes programmable and reconfigurable computing, media signal processing, processing of

audio signals, Internet, home entertainment, communications, including video-over-mobile network, and data security. This section illustrates deep penetration of computer systems into the consumer's market enabled by advances in signal processing and embedded applications.

## Locating Your Topic

Several avenues are available to access the desired information. A complete table of contents is presented at the front of the book. Each of the sections is preceded with an individual table of contents. Finally, each chapter begins with its own table of contents. Each contributed chapter contains comprehensive references. Some of them contain a "To Probe Further" section, in which a general discussion of various sources such as books, journals, magazines, and periodicals is located. To be in tune with the modern times, some of the authors have also included Web pointers to valuable resources and information. We hope our readers will find this to be appropriate and of much use.

A subject index has been compiled to provide a means of accessing information. It can also be used to locate definitions. The page on which the definition appears for each key defining term is given in the index.

This book is designed to provide answers to most inquiries and to direct inquirers to further sources and references. We trust that it will meet the needs of our readership.

## Acknowledgments

The value of this book is based entirely on the work of people who are regarded as top experts in their respective fields, and their excellent contributions. I am grateful to them. They contributed their valuable time without compensation and with the sole motive to provide learning material and help enhance the profession. I would like to thank Saburo Muroga, who provided editorial advice, reviewed the content of the book, made numerous suggestions, and encouraged me. I am indebted to him as well as to other members of the advisory board. I would like to thank my colleague and friend Richard Dorf for asking me to edit this book and trusting me with this project. Kristen Maus worked tirelessly on the first edition of this book and so did Nora Konopka of CRC Press. I am also grateful to the editorial staff of Taylor & Francis, Theresa Delforn and Allison Shatkin in particular, for all the help and hours spent on improving many aspects of this book. I am particularly indebted to Suryakala Arulprakasam and her staff for a superb job of editing, which has substantially improved this book over the previous one.

**Vojin G. Oklobdzija**
*Berkeley, California*

# Editor

**Vojin G. Oklobdzija** is a fellow of the Institute of Electrical and Electronics Engineers and distinguished lecturer of the IEEE Solid-State Circuits and IEEE Circuits and Systems Societies. He received his PhD and MSc from the University of California, Los Angeles in 1978 and 1982, as well as a Diplom-Ingenieur (MScEE) from the Electrical Engineering Department, University of Belgrade, Yugoslavia in 1971.

From 1982 to 1991, he was at the IBM T.J. Watson Research Center in New York where he made contributions to the development of RISC architecture and processors. In the course of this work he obtained a patent on register-renaming, which enabled an entire new generation of superscalar processors.

From 1988 to 1990, he was a visiting faculty at the University of California, Berkeley, while on leave from IBM. Since 1991, Professor Oklobdzija has held various consulting positions. He was a consultant to Sun Microsystems Laboratories, AT&T Bell Laboratories, Hitachi Research Laboratories, Fujitsu Laboratories, Samsung, Sony, Silicon Systems/Texas Instruments Inc., and Siemens Corp., where he was also the principal architect of the Siemens/Infineon's TriCore processor.

In 1996, he incorporated Integration Corp., which delivered several successful processor and encryption processor designs.

Professor Oklobdzija has held various academic appointments, in addition to the one at the University of California. In 1991, as a Fulbright professor, he helped to develop programs at universities in South America. From 1996 to 1998, he taught courses in Silicon Valley through the University of California, Berkeley Extension, and at Hewlett–Packard. He was visiting professor in Korea, EPFL in Switzerland and Sydney, Australia. Currently he is Emeritus Professor at the University of California and Research Professor at the University of Texas at Dallas.

He holds 14 U.S. and 18 international patents in the area of computer architecture and design.

Professor Oklobdzija is a member of the American Association for the Advancement of Science, and the American Association of University Professors.

He serves as associate editor for the *IEEE Transactions on Circuits and Systems II*; *IEEE Micro*; and *Journal of VLSI Signal Processing*; *International Symposium on Low-Power Electronics*, *ISLPED*; *Computer Arithmetic Symposium*, *ARITH*, and numerous other conference committees. He served as associate editor of the *IEEE Transactions on Computers* (2001–2005), *IEEE Transactions on Very Large Scale of Integration (VLSI) Systems* (1995–2003), the *ISSCC Digital Program Committee* (1996–2003), and the first *Asian Solid-State Circuits Conference, A-SSCC* in 2005. He was a general chair of the 13th Symposium on Computer Arithmetic in 1997.

He has published over 150 papers in the areas of circuits and technology, computer arithmetic, and computer architecture, and has given over 150 invited talks and short courses in the United States, Europe, Latin America, Australia, China, and Japan.

# Editorial Board

**Krste Asanović**
University of California at Berkeley
Berkeley, California

**William Bowhill**
Intel Corporation
Shrewsbury, Massachusetts

**Anantha Chandrakasan**
Massachusetts Institute of Technology
Cambridge, Massachusetts

**Hiroshi Iwai**
Tokyo Institute of Technology
Yokohama, Japan

**Saburo Muroga**
University of Illinois
Urbana, Illinois

**Kevin J. Nowka**
IBM Austin Research Laboratory
Austin, Texas

**Takayasu Sakurai**
Tokyo University
Tokyo, Japan

**Alan Smith**
University of California at Berkeley
Berkeley, California

**Ian Young**
Intel Corporation
Hillsboro, Oregon

# Contributors

**John F. Alexander**
University of North Florida
Jacksonville, Florida

**Krste Asanović**
University of California at Berkeley
Berkeley, California

**Ming Au-Yeung**
San Francisco State University
San Francisco, California

**Pervez M. Aziz**
Agere Systems
Allentown, Pennsylvania

**Raymond Barrett**
University of North Florida
Jacksonville, Florida

**Lejla Batina**
Katholieke Universiteit Leuven
Leuven, Belgium

**Mario Blaum**
IBM Almaden Research Center
San Jose, California

**Pradip Bose**
IBM T.J. Watson Research Center
Yorktown Heights, New York

**Don Bouldin**
University of Tennessee
Knoxville, Tennessee

**E. Bozorgzadeh**
University of California
Los Angeles, California

**Tzi-cker Chiueh**
State University of New York at Stony Brook
Stony Brook, New York

**Adam Dabrowski**
Poznan University of Technology
Poznan, Poland

**Babak Daneshrad**
University of California
Los Angeles, California

**Miroslav Despotović**
University of Novi Sad
Novi Sad, Yugoslavia

**Jozo J. Dujmović**
San Francisco State University
San Francisco, California

**Mohammad Faheemuddin**
King Fahd University of Petroleum & Minerals
Dhahran, Saudi Arabia

**Manoj Franklin**
University of Maryland
College Park, Maryland

**Matthew Franklin**
University of California at Davis
Davis, California

**Borko Furht**
Florida Atlantic University
Boca Raton, Florida

**Jean-Luc Gaudiot**
University of California at Irvine
Irvine, California

**Ricardo E. Gonzalez**
Tensilica, Inc.
Santa Clara, California

**Anna Hać**
University of Hawaii
Honolulu, Hawaii

**Siamack Haghighi**
Intel Corporation
Santa Clara, California

**Yoshiaki Hagiwara**
Sony Corporation
Tokyo, Japan

**Ali Ibrahim**
Advanced Micro Devices
Sunnyvale, California

**Mohammad Ilyas**
Florida Atlantic University
Boca Raton, Florida

**Bruce Jacob**
University of Maryland
College Park, Maryland

**Lizy Kurian John**
University of Texas at Austin
Austin, Texas

**R. Kastner**
University of California
Los Angeles, California

**Ruby Lee**
Princeton University
Princeton, New Jersey

**Worayot Lertniphonphun**
Georgia Institute of Technology
Atlanta, Georgia

**Tomasz Marciniak**
Poznan University of Technology
Poznan, Poland

**Brian Marcus**
IBM Almaden Research Center
San Jose, California

**Daniel Martin**
Infineon
Mountain View, California

**Binu Mathew**
Apple Inc.
Cupertino, California

**James H. McClellan**
Georgia Institute of Technology
Atlanta, Georgia

**S.O. Memik**
University of California
Los Angeles, California

**Milica Mitić**
University of Niš
Niš, Serbia

**John Morris**
Auckland University
Auckland, New Zealand

**Samiha Mourad**
Santa Clara University
Santa Clara, California

**Danny F. Newport**
University of Tennessee
Knoxville, Tennessee

**Garret Okamoto**
Santa Clara University
Santa Clara, California

**Ara Patapoutian**
Maxtor
Shrewsbury, Massachusetts

**Gerald G. Pechanek**
BOPS, Inc.
Chapel Hill, North Carolina

**Donna Quammen**
George Mason University
Fairfax, Virginia

**Todd R. Reed**
University of Hawaii at Manoa
Honolulu, Hawaii

**Peter Reiher**
University of California
Los Angeles, California

**Eric Rotenberg**
North Carolina State University
Raleigh, North Carolina

**Abdul H. Sadka**
University of Surrey
Surrey, England

**Sadiq M. Sait**
King Fahd University of Petroleum & Minerals
Dhahran, Saudi Arabia

**Kazuo Sakiyama**
Katholieke Universiteit Leuven
Leuven, Belgium

**M. Sarrafzadeh**
University of California
Los Angeles, California

**Thomas C. Savell**
Creative Advanced Technology Center
Scotts Valley, California

**Necip Sayiner**
Agere Systems
Allentown, Pennsylvania

**Giovanni Seni**
Motorola Human Interface Labs
Palo Alto, California

**Vojin Šenk**
University of Novi Sad
Novi Sad, Yugoslavia

**Dezsö Sima**
Budapest Polytechnic
Budapest, Hungary

**Kevin Skadron**
University of Virginia
Charlottesville, Virginia

**Mark Smotherman**
Clemson University
Clemson, South Carolina

**Emina Šoljanin**
Lucent Technologies
New Vernon, New Jersey

**Zoran Stamenković**
IHP Gmbh—Innovations for High Performance Microelectronics
Frankfurt (Oder), Germany

**Mile Stojčev**
University of Niš
Niš, Serbia

**Jayashree Subrahmonia**
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

**David Tarjan**
University of Virginia
Charlottesville, Virginia

**Fred J. Taylor**
University of Florida
Gainesville, Florida

**Daniel N. Tomasevich**
San Francisco State University
San Francisco, California

**Jonathan W. Valvano**
University of Texas at Austin
Austin, Texas

**Peter J. Varman**
Rice University
Houston, Texas

**Bane Vasić**
University of Arizona
Tucson, Arizona

**Ingrid Verbauwhede**
Katholieke Universiteit Leuven and UCLA
Leuven, Belgium

**Jeffrey Scott Vitter**
Purdue University
West Lafayette, Indiana

**Albert Wang**
Tensilica, Inc.
Santa Clara, California

**Alice Wang**
Texas Instruments
Dallas, Texas

**Shoichi Washino**
Tottori University
Tottori City, Japan

**Wayne Wolf**
Princeton University
Princeton, New Jersey

**Thucydides Xanthopoulos**
Cavium Networks
Marlboro, Massachusetts

**Larry Yaeger**
Indiana University
Bloomington, Indiana

**Chik-Kong Ken Yang**
University of California
Los Angeles, California

**Habib Youssef**
King Fahd University of Petroleum & Minerals
Dhahran, Saudi Arabia

# Contents

## SECTION I Computer Systems and Architecture

## SECTION II Embedded Applications



## SECTION III Signal Processing



## SECTION IV Communications and Networks



## SECTION V Input/Output

# I

# Computer Systems and Architecture

# 1

# Computer Architecture and Design


Jean-Luc Gaudiot
*University of California at Irvine*

Siamack Haghighi
*Intel Corporation*

Binu Mathew
*Apple Inc.*

Krste Asanović
*University of California at Berkeley*

Manoj Franklin
*University of Maryland*

Donna Quammen
*George Mason University*

Bruce Jacob
*University of Maryland*

Lejla Batina
*Katholieke Universiteit Leuven*

Kazuo Sakiyama
*Katholieke Universiteit Leuven*

Ingrid Verbauwhede
*Katholieke Universiteit Leuven and UCLA*

# Introduction

*Jean-Luc Gaudiot*

It is a truism that computers have become ubiquitous and portable in the modern world: personal digital assistants (PDAs), as well as many various kinds of mobile computing devices are easily available at low cost. This is also true because of the ever-increasing presence of the Wide World Web connectivity. One should not forget, however, that these life changing applications have only been made possible by the phenomenal advances that have been made in device fabrication and more importantly in the architecting of these individual components into powerful systems.

In the 1980s, advances in computer architecture research were most pronounced on two fronts: on the one hand, new architectural techniques such as RISC made their appearance and revolutionized single processor design and allowed high performance for the single chip microprocessors which first came out as system components in the 1970s. At the same time, large-scale parallel processors became mature and could be used by researchers in many high-end, computationally intensive, scientific applications.

In recent times, the appetite of Internet surfers has been fueling the design of architectures for powerful servers: in Section 1.1 Siamack Haghighi emphasizes the unique requirements of server design and identifies the characteristics of their applications.

In Section 1.2, Binu Matthew describes the very long instruction word (VLIW) processor model, compares it to more traditional approaches of Instruction Level Parallelism extraction, and demonstrates the future of VLIW processors, particularly in the context of multimedia applications.

Similarly, multimedia applications have promoted a dual architectural approach. In Section 1.3, Krste Asanovic traces the ancestry of vector processors to the supercomputers of the 1980s (Crays, Fujitsu, etc.) and describes the modern applications of this architecture model.

Architectures cannot be evaluated independently of the underlying technology. Indeed, nowadays, while deep-submicron design rules for VLSI circuit are allowing increasing numbers of devices on the same chip, techniques of multiprocessing are seeing additional applications in different forms which range from networks of workstations. Portability, all the way to multiprocessing on a chip, is the topic of Section 1.4 by Manoj Franklin.

Taking concurrent processing to the next level, Donna Quammen surveys parallel systems in Section 1.5 including large-scale tightly coupled parallel processors.

Finally, in Section 1.6 Bruce Jacob surveys the concepts underlying virtual memory systems and describes the tremendous advances this approach has undergone since first being proposed in the late 1960s.

## 1.1 Server Computer Architecture

*Siamack Haghighi*

### 1.1.1 Introduction

Widespread availability of inexpensive high-performance computers and Internet access have resulted in considerable business productivity improvement and cost savings. Many companies use high-performance computing and networking technologies for highly efficient electronic or e-commerce. As a result, most modern businesses rely on enterprise information technology (IT) computing and communication infrastructure for the backbone of their operation. The cost-savings potential has required many modern companies to fully automate their traditional manual order entry, processing, inventory management, and operations via web-based technologies. Current e-commerce revenue estimates exceed hundred billion dollars in the United States alone.

Availability of low-cost, robust, reliable, and secure IT infrastructure is one of the key drivers of the new Internet-based businesses. Customer usage models and applications affect IT infrastructure

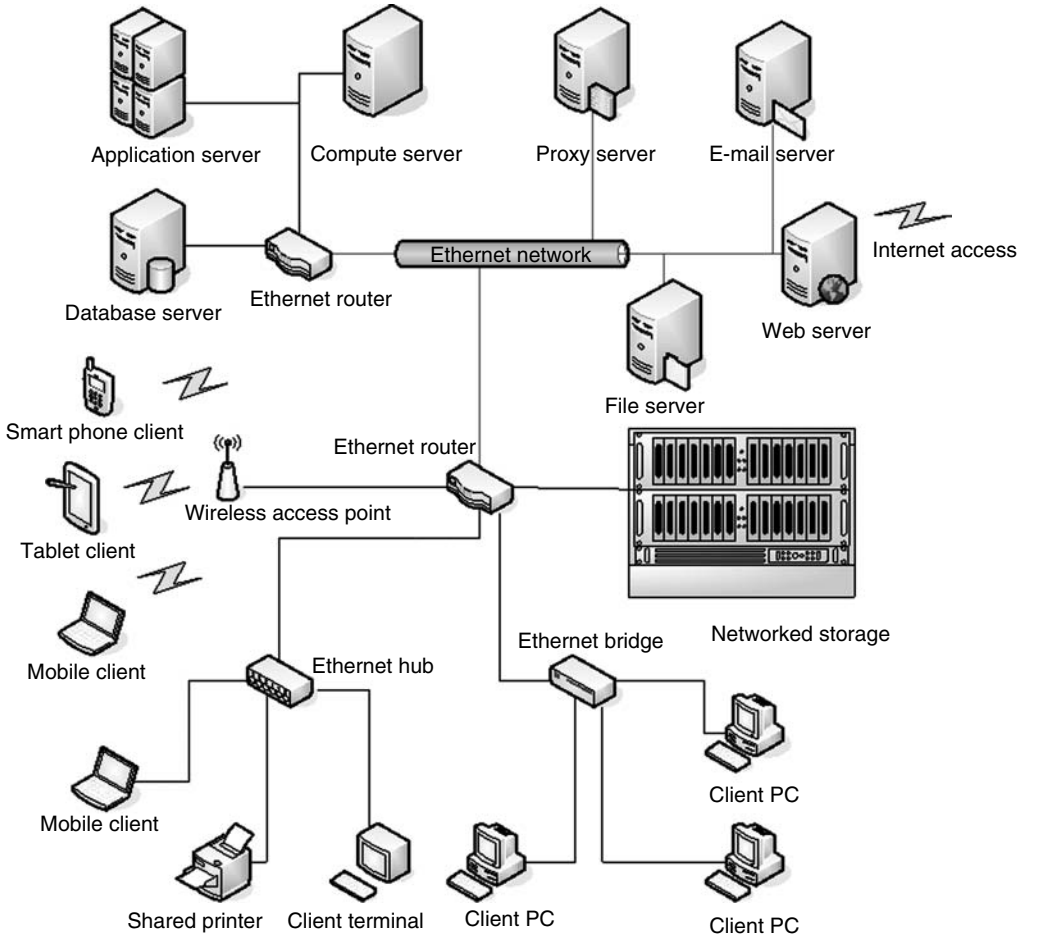



**FIGURE 1.1** Client–server computing infrastructure.

performance, operation, and cost. The requirements of many modern IT deployments can be cost-effectively met with client–server computing technologies. Although not a new idea, availability of inexpensive high-performance commodity microprocessors, scalable computer architecture, storage, and high-speed networks make client–server computing model an ideal fit for enterprise electronic business data processing or e-commerce. Other client–server computing advantages are shared data storage and back up, improved infrastructure reliability, availability, serviceability, manageability, and cost amortization over large number of client devices and users. Figure 1.1 illustrates an example client–server architecture computing deployment.

High-performance servers are built from multiple interconnected processors, high-performance memory systems, scalable networking, local storage subsystems, advanced software, and packaging. This section provides an overview of server architecture design, deployment, and the associated challenges.

### 1.1.2 Client–Server Computing

Client–server computing was developed to address cost-effective computing and communication capability for multiple users. Clients use variety of devices and terminal types to access shared servers.

Hence, users get access to high-performance services economically since infrastructure costs are shared and amortized among many users.

During 1970s, business computing infrastructure consisted of centralized mainframe computers connected to user terminals via networks. Mainframes provided high-performance centralized processing facilities for compute intensive tasks as well as data storage, external network interface, and task management. In 1980s, business computing evolved to distributed model due to advent of low-cost, high-performance personal computers (PCs) fueled by the availability of inexpensive powerful microprocessors. In this architecture, many of the computing tasks previously serviced by the mainframes are performed locally by the PCs. Recently, e-commerce and rapid growth of Internet as the common communication protocol have resulted in another change in business computing infrastructure. World Wide Web (WWW) optimized applications and low-cost computing have facilitated businesses adoption of client–server computing architecture. The following are the modern IT infrastructure elements:

- Simple and robust standardized web-based user interfaces
- Support for variety of access devices such as mobiles, desktop computers, personal digital assistants, and smart phones
- Wired and wireless high-speed data and communication networking
- Shared data storage and peripheral connectivity
- Centralized server array (sometimes referred to as server farm) configured and optimized for enterprise applications

### 1.1.3 Server Types

Modern servers are designed and optimized for low cost, high performance, low maintenance, and, in many cases, specific application usage models. There are a variety of server types, e.g., proxy, application, web cache, compute, communication, security, video, file, and streaming media. A typical server consists of several high-performance CPUs, large centralized or distributed system memory, high-speed local storage subsystem, and network interfaces. Specialization is achieved through selection of elements such as number of CPUs, size, type, and speed of system memory, operating system, number, and speed of network interfaces, local storage subsystem capacity, type, and access speed. As an example, an e-commerce server requires fast network interface, modest system memory, and multiple CPUs for high-throughput transaction processing whereas a file server benefits from large networked storage subsystem and a compute server benefits from many CPUs and large system memory.

Servers also differ based on form factors. Physical size and configuration are important considerations for high-density (high computing capability) server infrastructure deployments because power delivery, thermal cooling, and standardized installation are often the dominant concern factors. While some servers are designed to fit cabinets, others are designed to fit rack mounted enclosures.

In summary, configurability, form factor, scalable hardware, and software are required for optimized high-performance server deployment and operation.

### 1.1.4 Server Deployment Considerations

In addition to form factor, optimal server deployment and operation requires hardware, software features, and flexible configuration. In this section, some of these aspects are detailed.

#### 1.1.4.1 Server Features

Most servers have reliability, availability, serviceability, and manageability (RASM) features.

1. Reliability: Servers are expected to operate reliably with the ability for manual or automatic diagnosis and isolation of errors and failures. For example, banking and investment brokerage computing facilities require rapid diagnosis and isolation of hardware and software failures.

Example reliability features are hardware, software error or failure event detection, response mechanisms such as error correction codes (ECC) and error detection codes, transaction integrity checks, checksum, and multiple redundancies. Example event response mechanisms are event logging, failure source isolation, provisioning, and fail-over switching. Desired reliability features are selected based on cost, complexity, and server application usage model considerations as follows:

- Redundant hardware and software (e.g., independent operating system images on multiple server nodes)
- Server network interface and local storage subsystem integrity check mechanisms
- Fault detection, isolation, and mitigation
- ECC memory scrubbing to detect and correct bit errors that may cause system crash due to charged particle–induced errors
- System management software to collect detected errors and isolate faults
- Networked storage systems that use redundant array of independent disks (RAID) technology for data storage integrity assurance

2. Availability: The rapid rise in business reliance on computing infrastructure has resulted in demand for nonstop computing operations. Servers with such capabilities are referred to as high-availability servers. In banking and investment brokerage businesses, even brief service interruptions are detrimental and cost prohibitive. High-availability servers require specially configured deployments such as multiple backup systems, load balancing, and fail-over switching capabilities. Two key metrics for measuring the value and potential cost of high-availability computing are average downtime per year (in seconds) and potential revenue loss due to service interruption. Other high-availability server features are service provisioning, user task isolation and migration, traffic differentiation, dynamic prioritization, ability to quickly detect, and remedy failures. Scheduled maintenance and upgrade of hardware and software elements may also decrease potential for failures and increase server availability.
3. Serviceability: Continuous trouble-free server operation requires routine maintenance, error and failure monitoring, and the ability to quickly fix or replace defective hardware or software components. The mechanisms that provide such facilities are generally referred to as serviceability options. In many cases, the failure source can be isolated to one unit or subsystem, e.g., one dynamic memory module in system memory. Software mechanisms (e.g., real-time diagnostic tools, alerting, and dynamic server configuration) may be used manually or automatically to isolate, disable a faulty unit, swap backup units and prepare for service or faulty unit replacement before an error or failure becomes catastrophic and propagates to entire server or computing facilities. Features that may assist rapid replacement of faulty components are traffic isolation and hot replacement. Plug and play subsystem capabilities also improve ease of service. Hot replacement allows changing faulty subsystems without the need to power down or reboot the server. Other services such as scheduled downtime to do off-line enhancement may also be necessary.
4. Manageability: Routine and emergency system operation requires management facilities such as
   - Server performance monitoring and key application tuning
   - Capacity planning for existing and future clients, users, and applications
   - Manual or automatic load balancing, distribution, task migration, and scheduling for efficient operation of the enterprise resources and applications
   - Special accommodation of circumstances requiring increased alerting and manageability capabilities (e.g., virus protection, intrusion detection, etc.)
   - Rapid installation and configuration of new applications and systems (e.g., software upgrade and installation)
   - Automatic and preventive operator notification applications and services

- Mechanisms for rapid recovery from service outages
- Remote or local server management despite faulty server components and errors

The following are the other important and desirable deployment features:

5. Scalability: High-performance IT infrastructure can be built in two ways. In one approach, few servers each configured with large number of CPUs and powerful input/output (I/O) capability can be used. Alternatively, large number of servers, each containing a few CPU may be clustered for high-performance computing. A combination of both approaches may also be used.
6. Security: In routine and emergency cases, access to system resources and facilities such as user authentication, intrusion detection, and privileged access may be needed. Cryptographic technologies such as encryption and decryption may also be used to enhance the overall system security. In some cases, cooperation with local and government officials may be required for intrusion detection and prevention.

#### 1.1.4.2 Operation

An important server deployment issue is the form factor and installation requirements. A typical server board contains multiple CPUs, system and peripheral connection bridges, networking, display, and local storage peripherals. In deployments such as data centers, large number of server modules may be housed in racks or cabinets. In dense server deployments, rack or cabinet mounting, operation, maintenance, thermal management, power delivery, and wiring management are major challenges. The proximity of data centers to major customer sites is also important. Other considerations are as follows:

1. Power: A typical server board may consume several hundred watts of power. Providing power to large server racks may be a significant challenge. Power provisioning includes accommodating outages, voltage regulation, power delivery, uninterrupted supply, and, if necessary, battery backup.
2. Thermal: Servers generate large amounts of heat. Large server installations demand planning and accommodation for heat dissipation and cooling. In many cases, thermal dissipation and cooling solution limits server deployment size. Since high-performance server thermal management is a major challenge, many new servers are built from low-power consumption VLSI building blocks. Development of low-power CPUs and chipsets that lower the need for active cooling can effectively address thermal limitations.
3. Total cost of ownership: An important consideration for enterprise servers is the total cost of ownership (TCO). TCO is a metric used to estimate overall IT infrastructure operational costs such as hardware and software purchases, services, required personnel, and downtimes. In each enterprise deployment, one or more TCO factors may be dominant. For example, in an online investment brokerage server installation, the downtime is a major consideration. In many cases, the downtime costs may easily justify additional backup servers.
4. Server clustering: Many business applications such as manufacturing, financial, health care, and telecommunication require mission-critical servers. Telecommunication billing and banking servers are examples of server clustering. Mission-critical servers may be designed by connecting several servers and providing fail-over switching capabilities. If one server crashes, others can continue operation of key applications. Server clustering may be used to mitigate hardware (server components, storage, networking hardware), operating system, and application software failures. Variety of hardware and software automatic fault detection, isolation, and fail-over switching mechanisms are available and used by various mission-critical server manufacturers.

### 1.1.5 Server Architecture

#### 1.1.5.1 Hardware Architecture

Even though servers may be built using custom very large scale integration (VLSI) devices, economic considerations necessitate the use of commodity hardware VLSI whenever possible. Figures 1.2 and 1.3

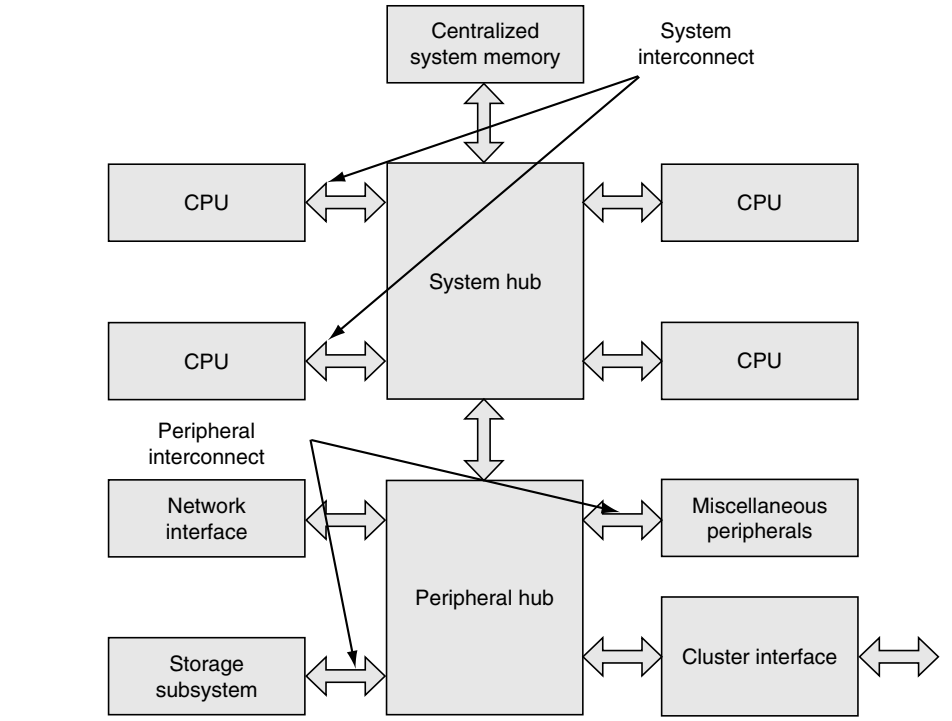



**FIGURE 1.2** Centralized shared memory server architecture.

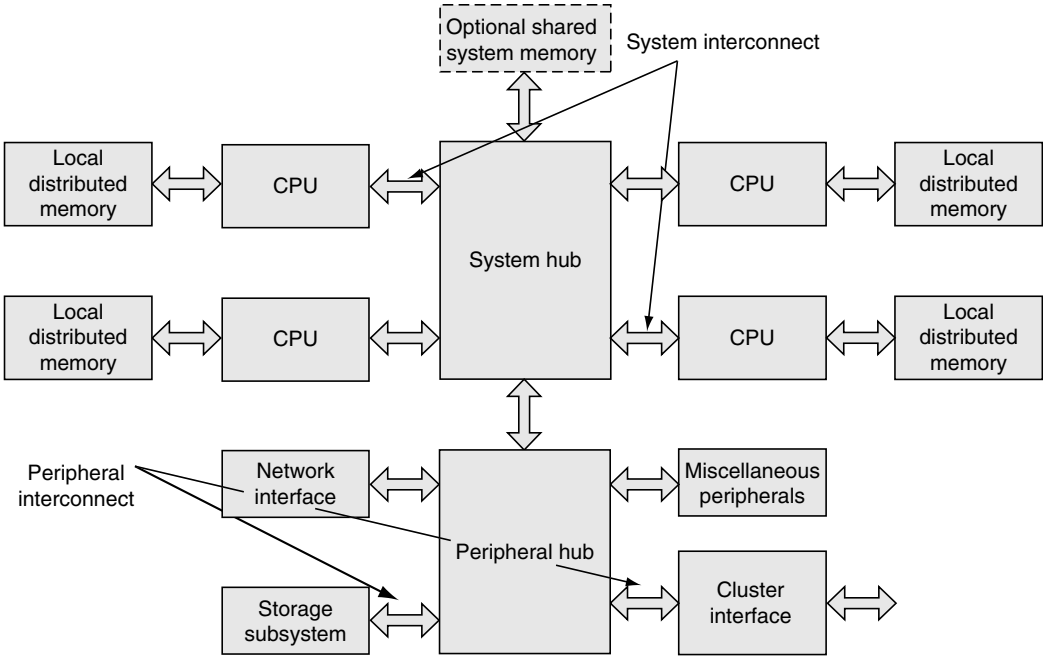



**FIGURE 1.3** Distributed memory server architecture.

illustrate shared centralized and distributed memory multiprocessor server architectures. In both cases, the major building block components are central processing unit (CPU), memory, system, and peripheral hubs, interconnects, and peripherals. Each of these components are now be described in more depth.

#### *1.1.5.1.1 CPU*

Most servers contain multiple CPUs. For economic reasons, servers use special configurations of commodity desktop or mobile PC CPUs. Server CPUs differ from desktop or mobile PC CPUs due to additional features such as larger on-chip caches, hardware multiprocessing, hardware cache coherency support and high-performance system interconnects. Most current servers have symmetric multiprocessing architecture whereby all CPUs are of the same type and configuration. A server may be built from heterogeneous CPUs. In such architecture, the few used CPU types can be optimized, each for specific application classes.

Server CPUs have fast execution capability and multiple levels of hierarchically organized on-chip cache memory. Fast execution capability is required for high-performance application processing. Large, high-performance on-chip cache memory ensures sustained high-performance CPU operation. Modern CPU architectures have multiple execution cores, each operating at several gigahertz speed and include several megabytes of on-chip high-speed cache memory.

Figure 1.4 illustrates a simplified internal organization of a modern server CPU. Most commodity CPUs have 64-bit addressing capability and can easily accommodate processing of large data set applications, support for many clients, applications, and users. Each CPU processing core contains several arithmetic logic units (ALU), multi-ported register files, floating-point multiply, and sophisticated branch prediction and execution units. Most server CPUs execute program instructions out-of-order (OOO) and several operations at a time (super scalar).

High-performance CPU execution rate requires sustained high-bandwidth instruction and data delivery. Caches are useful for high-speed storage and retrieval of frequently used instruction and data. At each level, disjoint or integrated instruction and data caches may be available. Caches are enumerated in increasing order with the lowest level closest to the CPU execution units. In Fig. 1.4, three levels of cache hierarchy are enumerated as L1–L3. As the cache hierarchy level increases, the size is also increased; typically 2–10 times the size of the preceding cache level. Current high-end server microprocessors use 3–4 cache hierarchy levels. Cache organization optimization parameters are capacity, associativity, line size, speed, number of access ports, and line replacement policy. These parameters are determined based on variety of application execution characteristics, performance simulation models and measurements. Current server CPU costs are dominated by the on-chip cache size and optimized for state-of-the-art VLSI processing technology capabilities, circuit design, power consumption, and salient software application characteristics.

The numbers of CPUs used in a server determine the desired server performance, cost, form factor, and thermal and power requirements. Most server designs contain sockets for additional CPUs and scalable computing performance. Architecture and design of high-performance multiprocessor servers are an active area of research.

#### *1.1.5.1.2 Memory*

A critical server building block is the system memory. Server systems use several channels of dynamic random access memory (DRAM) modules. The larger the number of independent memory channels, the larger is the total bandwidth available to devices requesting memory access (such as CPUs and peripheral devices).

A server system memory may be centralized or distributed. Centralized memory organization facilitates simple software architecture. Additional memory modules may be added to the central memory array, benefiting the overall system. The main disadvantages of the centralized system memory architecture are memory access contention and latency. Distributed system memory, as shown in Fig. 1.3, enables lower latency memory access if the CPU to local memory access traffic can be localized.

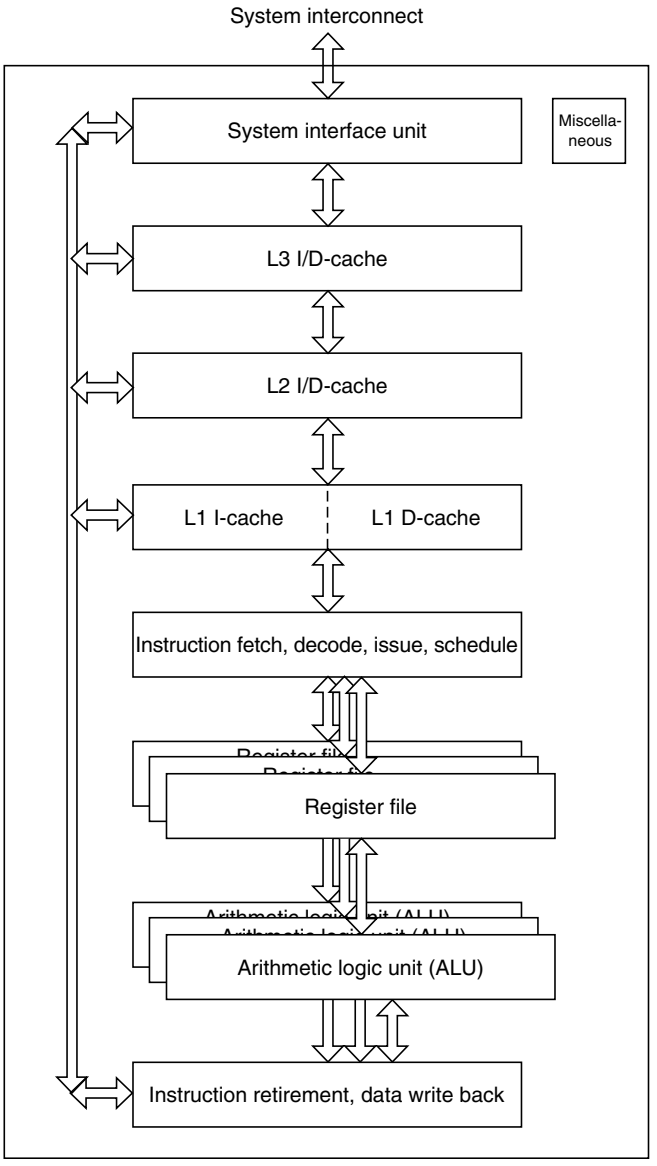



**FIGURE 1.4** Internal server CPU organization.

If a CPU needs to access indirectly attached memory, the request will be routed to the destination CPU via intersystem interconnects and system hubs.

There are several DRAM memory configurations such as single in-line memory module (SIMM) or dual in-line memory module (DIMM). A server system memory may support multiple DRAM types using high-performance open standard or proprietary memory system interface. This type of memory is generally referred to as buffered memory. Most servers have the capacity for several gigabytes of system memory.

For improved reliability and robustness, server system memory includes fault tolerance features such as ECC. Errors can happen for a variety of reasons such as DRAM memory charged particle induced or intra-chip physical layer random errors. One example robustness metric is the number of error bits that

can be detected and corrected. Low-end servers use single error correct, double error detect (SECDED) ECC. Another robustness feature is the chip kill. Chip kill feature allows isolation and correction of multi-bit faults stemming from failure of a single memory chip.

Since servers have large system memory capacity, high-speed auto-initialization of memory to known values, e.g., during boot time may be beneficial. At runtime, accessed memory may be checked for initialization, providing additional robustness. Other robustness features include memory mirroring, redundant memory-bit steering, and soft-error (charge particle–induced error) scrubbing.

#### *1.1.5.1.3 System Interconnects*

For high-performance server operation, the connections between CPUs and the system hub and system to peripheral hubs should be high bandwidth and low latency. Figures 1.2 and 1.3 illustrate example system and peripheral interconnects. Modern servers use point-to-point, pipelined, high-speed interconnects operating at several gigahertz speed. Modern system interconnects are built using state-of-the-art high-speed serial physical layer signaling capable of error detection, correction, and support for cache coherency hardware protocols. System interconnects need to be low cost, low latency, high performance, require inexpensive circuit board design and have low-power consumption. In most cases, some of these requirements are conflicting, hence the need for trade-offs.

#### *1.1.5.1.4 System Hub*

The design of low-cost, high-performance multiprocessor server system hub is an engineering trade-off challenge. On one hand, high-performance system hub needs to deliver peak server performance. On the other hand, system hub needs to be low cost, scalable to multiple CPUs, configurable for various server types and provide variety of connectivity interfaces. In a server, the CPUs compete for low-latency, high-throughput access to shared resources such as system memory. The system hub provides communication mechanisms between the CPUs, system memory, peripheral subsystem, and potentially graphics controllers. The following are some of the considerations for optimum system hub design:

1. Servers with centralized shared memory as in Fig. 1.2, use integrated memory controller system hub. The system hub is capable of supporting multiple memory modules via high-performance links. In distributed shared memory servers as in Fig. 1.3, memory controllers are integrated within each CPU. Closely coupled CPU and memory controller enables low-latency access. Typical system hub features, used in centralized shared memory servers, are multiple, fast, wide, and independent memory channels and memory interleave support, high-speed pipelined memory and CPU interconnects.
2. Shared system resources, such as memory are accessed by various system devices (CPU, networking, storage, etc.). Hence, if one device (e.g., a CPU) is to have highest performance access to system memory, all other devices also competing for system memory access need to be held off. For applications that require protracted high-traffic system memory access, many requesting devices may have to stay idle while the highest priority traffic is serviced, potentially causing severe performance and efficiency loss. Accessing shared system resources is a dynamic application and usage model-dependent event. The main memory scheduling and arbitration access policy optimization are determined through extensive computer simulations that include dynamic models of the application, operating system (OS), and hardware components. Real-time application servers require additional mechanisms such as admission control, quality of service (QoS), traffic differentiation, and bandwidth reservation.
3. Many server system hubs do not provide extensive high-performance graphics capabilities. The reason is the limited need for high-performance graphics in server applications.

#### *1.1.5.1.5 Peripheral Hub*

In addition to traditional user and system connectivity devices (flash, keyboard controller, mouse, graphics, etc.), many server peripheral subsystems have extensive high-performance I/O capabilities.

A server's I/O capability is useful for supporting multiple high-speed network interfaces, storage arrays, and clustering interfaces. Example I/O technologies are peripheral component interface (PCI), PCI-X, and PCI-Express standard interfaces. Many modern peripheral hubs include additional features for server manageability, serviceability, peer-to-peer communication between I/O interfaces, and the ability to isolate, disable, and reroute high-performance I/O traffic.

#### *1.1.5.1.6 Peripherals*

Because of extensive computational capabilities, server peripheral subsystems are more extensive than desktop or mobile PCs. Server peripherals include storage, Ethernet network interface controllers (NIC), clustering interface, and archival storage devices. Some or all traditional peripheral elements such as boot flash, keyboard, mouse, and graphics processing capabilities may also be available. Server peripherals provide desired capabilities by supporting proprietary or open standard intersystem interconnect interfaces such as PCI-Express. Following are the examples of peripherals:

1. Data storage and retrieval: Server storage and archival systems may be centralized or distributed. High-performance, fault tolerant disk-storage access is achieved using RAID technology. Other data-storage and retrieval technologies are network-attached storage (NAS), small computer system interface (SCSI), and fiber channel storage area networking (SAN).
2. Network interface: High-performance servers require high-speed networking interfaces. Most servers include several gigabit or higher speed Ethernet standard interfaces.
3. Clustering interfaces: Variety of proprietary or industry standard interfaces are available to support clustering of multiple servers. Some clustering interfaces are based on switching fabric technologies to enable multiple server node connection. Other proprietary server clustering interfaces are the direct attach interface, optimized for large data transfer capabilities compared to switching fabric interfaces.
4. Miscellaneous peripherals: Servers may include peripherals such as keyboard, mouse, and graphics controller devices. Other special function peripherals such as encryption and decryption accelerators may also be used.

### 1.1.5.2 Software Architecture

The server software architecture is very different from desktop or mobile PCs due to enterprise requirements. In addition to supporting large number of users, server software are required to be robust, secure, scalable, fault tolerant, and optimized for large database or transaction processing tasks. In most cases, software architecture closely matches server types, e.g., web and proxy servers.

#### *1.1.5.2.1 Operating System*

Server operating systems are optimized for supporting large number of users, applications, CPU utilization, large capacity system memory, extensive I/O, networking, and multiprocessor scheduling. Clustered servers use independent and potentially different OS for each server node. Internet working issues in such heterogeneous environment requires additional consideration and optimization. Many OS and software programs use advanced caching techniques for high-performance access of large data sets and databases.

#### *1.1.5.2.2 Applications*

There are a variety of server applications depending on the server type and the enterprise needs, e.g., database management, transaction processing, and decision support. Most server applications support large number of simultaneous users, provide isolation among users accessing and sharing the same database, contain extensive security, error or fault tolerance features. E-commerce servers have optimized features for high-throughput transaction processing.

There are also many specialized server software applications typically referred to as middleware used for application development and delivery process tools for web and application servers.

#### 1.1.5.3 Applications Usage Models

Server applications support variety of usage models and states since each customer may use it differently and the applications have multitude of supported operational modes. For example, a user may be updating the database with recent entries while others are accessing the same database. Application usage models and states have specific system demand characteristics. Robust IT infrastructure deployment and operation ensures that the servers, network, and client architecture are performance tuned and optimized for major usage models of interest. Example server configuration parameters are the number of CPUs, system memory type and size, system and peripheral hubs, peripheral storage size, type, and network interface speed. Software tuning mechanisms include optimizing compilers for parallel processing, multithreading, and customizable libraries.

Server application usage models can be characterized in many ways. Some usages require high-bandwidth communication between CPUs, system memory, and storage devices. Other usages require extensive execution processing rate. Some servers, such as web cache, require high-performance network interfaces.

### 1.1.6 Future Directions

Advanced server design requires enhancements in several areas:

1. Low-cost, high-performance, scalable server system design is an active area of research. Delivering high user-perceived performance while overcoming system deployment limitations such as cost, security, capacity, thermal, and power consumption are example challenges that need to be addressed.
2. New server board material and design technologies that facilitate higher computing density servers at lower cost, lower power consumption, or denser packaging would benefit future high-performance server designs.
3. With advancements in semiconductor processing technology, VLSI feature sizes (line width, transistor size, etc.) are becoming ever smaller. Small geometry devices are more sensitive to charged particle–induced errors and high-speed signaling faults. Hence, for reliable operation, the future server CPUs may need to include internal circuit error detection and correction or other fault tolerance mechanisms. System software and hardware architecture of fault-tolerant CPU servers are active areas of investigation.
4. With advancements in semiconductor processing technology, server CPUs will have ever-increasing processing capabilities. Software and tools that can expose and exploit the increased performance for key enterprise applications are highly desirable. Example requirements are parallelizing compilers, advanced operating systems, development and debug tools, parallel file systems, and high-speed networking.
5. Advancements in new enterprise class usage models are an active research area. Example improvements are enhanced server security, combining real-time, data and transaction processing enterprise applications, and support for more client devices.
6. Scalable and high-performance networking and clustering technologies to interconnect components and servers are also active areas of research.

## 1.2 Very Large Instruction Word Architectures

*Binu Mathew*

### 1.2.1 What Is a VLIW Processor?

Recent high-performance processors have depended on instruction-level parallelism (ILP) to achieve high execution speed. ILP processors achieve their high performance by causing multiple operations to execute in parallel, using a combination of compiler and hardware techniques. Very long instruction

word (VLIW) is one particular style of processor design that tries to achieve high levels of ILP by executing long instruction words composed of multiple operations. The long instruction word called a MultiOp consists of multiple arithmetic, logic, and control operations each of which would probably be an individual operation on a simple RISC processor. The VLIW processor concurrently executes the set of operations within a MultiOp thereby achieving instruction level parallelism. The remainder of this section discusses the technology, history, uses, and the future of such processors.

### 1.2.2 Different Flavors of Parallelism

Improvements in processor performance come from two main sources: faster semiconductor technology and parallel processing. Parallel processing on multiprocessors, multicomputers, and processor clusters has traditionally involved a high degree of programming effort in mapping an algorithm to a form that can better exploit multiple processors and threads of execution. Such reorganization has often been productively applied, especially for scientific programs. The general-purpose microprocessor industry, on the other hand, has pursued methods of automatically speeding up existing programs without major restructuring effort. This leads to the development of ILP processors that try to speedup program execution by overlapping the execution of multiple instructions from an otherwise sequential program.

A simple processor that fetches and executes one instruction at a time is called a simple scalar processor. A processor with multiple function units has the potential to execute several operations in parallel. If the decision about which operations to execute in an overlapped manner is made at run time by the hardware, it is called a super scalar processor. In a simple scalar processor, a binary program represents a plan of execution. The processor acts as an interpreter that executes the instructions in the program one at a time. From the point of view of a modern super scalar processor, an input program is more like a representation of an algorithm for which several different plans of execution are possible. Each plan of execution specifies when and on which function unit each instruction from the instruction stream is to be executed.

Different types of ILP processors vary in the manner in which the plan of execution is derived, but it typically involves both the compiler and the hardware. In the current breed of high-performance processors like the Intel Pentium and the MIPS R18000, the compiler tries to expose parallelism to the processor by means of several optimizations. The net result of these optimizations is to place as many independent operations as possible close to each other in the instruction stream. At run time, the processor examines several instructions at a time, analyses the dependences between instructions, and keeps track of the availability of data and hardware resources for each instruction. It tries to schedule each instruction as soon as the data and function units it needs are available. The processor's decisions are complicated by the fact that memory accesses often have variable latencies that depend on whether a memory access hits in the cache or not. Since such processors decide which function unit should be allocated to which instruction as execution progresses, they are said to be dynamically scheduled. Often, as a further performance improvement, such processors allow later instructions that are independent to execute ahead of an earlier instruction which is waiting for data or resources. In that case, the processor is said to be out-of-order.

Branches are common operations in general-purpose code. On encountering a branch, a processor must decide whether or not to take the branch. If the branch is to be taken, the processor must start fetching instructions from the branch target. To avoid delays due to branches, modern processors try to predict the outcome of branches and execute instructions from beyond the branch. If the processor predicted the branch incorrectly, it may need to undo the effects of any instructions it has already executed beyond the branch. If a super scalar processor uses resources that may otherwise go idle to execute operations the result of which may or may not be used, it is said to be speculative.

Out-of-order speculative execution comes at a significant hardware expense. The complexity and nonscalability of the hardware structures used to implement these features could significantly hinder the

performance of future processors. An alternative solution to this problem is to simplify processor hardware and transfer some of the complexity of extracting ILP to the compiler and run time system—the solution explored by VLIW processors.

Joseph Fisher, who coined the acronym VLIW, characterized such machines as architectures that issue one long instruction per cycle, where each long instruction called a MultiOp consists of many tightly coupled independent operations each of which execute in a small and statically predictable number of cycles. In such a system, the task of grouping independent operations into a MultiOp is done by a compiler or binary translator. The processor freed from the cumbersome task of dependence analysis has to merely execute in parallel the operations contained within a MultiOp. This leads to simpler and faster processor implementations. In later sections, we see how VLIW processors try to deal with the problems of branch and memory latencies and implement their own variant of speculative execution. But, first, we present a brief history of VLIW processors.

### 1.2.3 A Brief History of VLIW Processors

For various reasons, which were appropriate at that time, early computers were designed to have extremely complicated instructions. These instructions made designing the control circuits for such computers difficult. A solution to this problem was microprogramming, a technique proposed by Maurice Wilkes in 1951. In a microprogrammed CPU, each program instruction is considered a macroinstruction to be executed by a simpler processor inside the CPU. Corresponding to each macroinstruction, there will be a sequence of microinstructions stored in a microcode ROM in the CPU.

Horizontal microprogramming is a particular style of microprogramming where bits in a wide microinstruction are directly used as control signals within the processor. In contrast, vertical microprogramming uses a shorter microinstruction or series of microinstructions in combination with some decoding logic to generate control signals. Microprogramming became a popular technique for implementing the control unit of processors after IBM adopted it for the System/360 series of computers.

Even before the days of the first VLIW machines, there were several processors and custom computing devices that used a single wide instruction word to control several function units working in parallel. However, these machines were typically hand-coded and the code for such machines could not be generalized to other architectures. The basic problem was that compilers at that time looked only within basic blocks to extract ILP. Basic blocks are often short and contain many dependences and, therefore, the amount of ILP that can be obtained inside a basic block is quite limited.

Joseph Fisher, a pioneer of VLIW, while working on PUMA, a CDC-6600 emulator, was frustrated by the difficulty of writing and maintaining 64-bit horizontal microcode for that processor. He started investigating a technique for global microcode compaction—a method to generate long horizontal microcode instructions from short sequential ones. Fisher soon realized that the technique he developed in 1979, called trace scheduling, could be used in a compiler to generate code for VLIW-like architectures from a sequential source since the style of parallelism available in VLIW is very similar to that of horizontal microcode. His discovery led to the design of the ELI-512 processor and the Bulldog trace-scheduling compiler.

Two companies were founded in 1984 to build VLIW-based mini supercomputers. One was Multiflow, started by Fisher and colleagues from Yale University. The other was Cydrome founded by VLIW pioneer Bob Rau and colleagues. In 1987, Cydrome delivered its first machine, the 256-bit Cydra 5, which included hardware support for software pipelining. In the same year, Multiflow delivered the Trace/200 machine, which was followed by the Trace/300 in 1988 and Trace/500 in 1990. The 200 and 300 series used a 256-bit instruction for 7 wide issue, 512 bits for 14 wide issue, and 1024 bits for 28 wide issue. The 500 series only supported 14 and 28 wide issues. Unfortunately, the early VLIW machines were commercial failures. Cydrome closed in 1988 and Multiflow in 1990.

Since then, VLIW processors have seen a revival and some degree of commercial success. Some of the notable VLIW processors of recent years are the IA-64 or Itanium from Intel, the Crusoe processor from

Transmeta, the Trimedia media-processor from Philips, and the TMS320C62x series of DSPs from Texas Instruments. Some important research machines designed during this time include the Playdoh from HP labs, Tinker from North Carolina State University, and the Imagine stream and image processor currently developed at Stanford University.

## 1.2.4 Defoe: An Example VLIW Architecture

We now describe Defoe, an example processor used in this section to give the reader a feel for VLIW architecture and programming. Although it does not exist in reality, its features are derived from those of several existing VLIW processors. Later sections that describe the IA-64 and the Crusoe, will contrast those architectures with Defoe.

### 1.2.4.1 Function Units

Defoe is a 64-bit architecture with the following function units:

- Two load/store units.
- Two simple ALUs that perform add, subtract, shift, and logical operations on 64-bit numbers and packed 32-, 16-, and 8-bit numbers. In addition, these units also support multiplication of packed 16- and 8-bit numbers.
- One complex ALU that can perform multiply and divide on 64-bit integers and packed 32-, 16-, and 8-bit integers.
- One branch unit that performs branch, call, and comparison operations.

There is no support for floating point. Figure 1.5 shows a simplified diagram of the Defoe architecture.

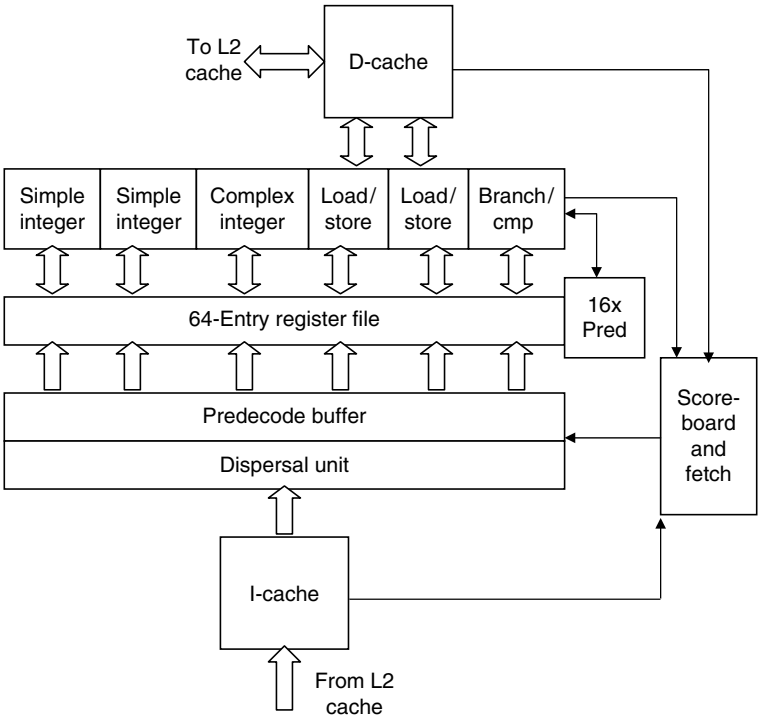



**FIGURE 1.5** Defoe architecture.

#### 1.2.4.2 Registers and Predication

Defoe has a set of 64 programmer visible general-purpose registers that are 64 bits wide. As in the MIPS architecture, register R0 always contains zero. Predicate registers are special 1-bit registers that specify a true or false value. There are 16 programmer visible predicate registers in the Defoe named from PR0 to PR15. All operations in Defoe are predicated, i.e., each instruction contains a predicate register (PR) field that contains a PR number. At run time, if control reaches the instruction, the instruction is always executed. However, at execution time, if the predicate is false, the results are not written to the register file and side effects such as TLB miss exceptions are suppressed. In practice, for reasons of efficiency, this may be implemented by computing a result, but writing back the old value of the target register. Predicate register 0 (PR0) always contains the value 1 and cannot be altered. Specifying PR0 as the predicate performs unconditional operations. Comparison operations use a predicate register as their target register.

#### 1.2.4.3 Instruction Encoding

Defoe is a 64-bit compressed VLIW architecture. In an uncompressed VLIW system, MultiOps have a fixed length. When suitable operations are not available to fill the issue slots within a MultiOp, NOPs are inserted into those slots. A compressed VLIW architecture uses variable length MultiOps to get rid of those NOPs and achieve high code density. In the Defoe, individual operations are encoded as 32-bit words. A special stop bit in the 32-bit word indicates the end of a MultiOp. Common arithmetic operations have an immediate mode, where a sign or zero extended 8-bit constant may be used as an operand. For larger constants of 16, 32, or 64 bits, a special NOP code may be written into opcode field of the next operation and the low-order bits may be used to store the constant. In that case, the predecoder concatenates the bits from two or more different words to assemble a constant (Fig. 1.6 depicts the instruction format).

#### 1.2.4.4 Instruction Dispersal and Issue

A traditional VLIW with fixed width MultiOps has no need to disperse operations. However, when using a compressed format like that of the Defoe, there is a need to expand the operations, and insert NOPs for function units to which no operations are to be issued. To make the dispersal task easy, we make the following assumptions:

- A few bits in the opcode specify the type of function unit (i.e., load/store, simple arithmetic, complex arithmetic, or branch) the operation needs.
- The compiler ensures that the instructions that comprise a MultiOp are sorted in the same order as the function units in the processor. This reduces the circuit complexity of the instruction dispersal stage. For example, if a MultiOp consists of a load, a 32-bit divide, and a branch, then the ordering (load, multiply, branch) is legal, but the ordering (load, branch, multiply) is not legal.
- The compiler ensures that all the operations in the same MultiOp are independent.
- The compiler ensures that the function units are not oversubscribed. For example, it is legal to have two loads in a MultiOp, but it is not legal to have three loads.
- It is illegal to not have a stop bit in a sequence of more than six instructions.
- Basic blocks are aligned at 32-byte boundaries.

Apart from reducing wastage of memory, another reason to prefer a compressed format VLIW over an uncompressed one is that the former provides better I-cache utilization. To improve performance,

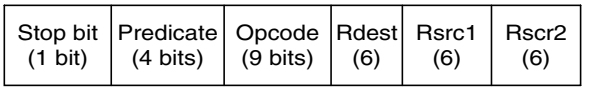



**FIGURE 1.6** Instruction encoding.

we use a predecode buffer that can hold up to eight uncompressed MultiOps. The dispersal network can use a wide interface (say 512 bits) to the I-cache to uncompress up to two MultiOps every cycle and save them in the predecode buffer. Small loops of up to eight MultiOps (maximum 48 operations) will experience repeated hits in the predecode buffer. It may also help to lower the power consumption of a low-power VLIW processor.

Defoe supports in-order issue and out-of-order completion. Further, all the operations in a MultiOp are issued simultaneously. If even one operation cannot be issued, issue of the whole MultiOp stalls.

#### 1.2.4.5 Branch Prediction

Following the VLIW philosophy of enabling the software to communicate its needs to the hardware, branch instructions in Defoe can advise the processor about their expected behavior. A 2-bit hint associated with every branch may be interpreted as follows:

| Opcode Modifier | Meaning |
|---|---|
| Stk | Static prediction. Branch is usually taken. |
| Sntk | Static prediction. Branch is usually not taken. |
| Dtk | Dynamic prediction. Assume branch is taken if no history is available. |
| Dntk | Dynamic prediction. Assume branch is not taken if no history is available. |

Implementations of the Defoe architecture may provide branch prediction hardware, but a branch predictor is not required in a minimal implementation. If branch prediction hardware is provided, static branches need not be entered in the branch history table, thereby freeing up resources for dynamically predicted branches.

#### 1.2.4.6 Scoreboard

To accommodate branch prediction and the variable latency of memory accesses because of cache hits and misses, some amount of scoreboarding is required. Although we will not describe the details of the scoreboard here, it should be emphasized that the scoreboard and control logic for a VLIW processor like the Defoe is much simpler than that of a modern super scalar processor because of the lack of out-of-order execution and speculation.

#### 1.2.4.7 Assembly Language Syntax

The examples that follow use the following syntax for assembly language instructions:

```
(predicate_reg) opcode.modifier Rdest = Rsource1, Rsource2
```

If the predicate register is omitted, PR0 will be assumed. In addition, a semicolon following an instruction indicates that the stop bit is set for that operation, i.e., this operation is the last one in its MultiOp. The prefix "!" for a predicate implies that the opcode actually depends on the logical and not on the value of the predicate register.

#### Example 1

This example demonstrates the execution model of the Defoe by computing the following set of expressions.

$a = x + y - z$
$b = x + y - 2 \times z$
$c = x + y - 3 \times z$

Register assignments: r1 = $x$, r2 = $y$, r3 = $z$, r32 = $a$, r33 = $b$, r34 = $c$.

| *Line #* | *Code* | *Comments* |
|---|---|---|
| 1. | `add r4 = r1, r2` | `// r4 = x + y` |
| 2. | `shl r5 = r3, 1` | `// r5 = z << 1, i.e. z * 2` |
| 3. | `mul r6 = r3, 3;` | `// r6 = z * 3. Stop bit.` |
| 4. | `sub r32 = r4, r3` | `// r5 = a = gets x + y - z` |
| 5. | `sub r33 = r4, r5;` | `// r33 = b = x + y - 2 * z.` `// Stop bit.` |
| 6. | `sub r34 = r4, r6;` | `// r34 = c = x + y - 3 * z.` `// Stop bit.` |

The first three lines are followed by a stop bit to indicate that these three operations constitute a MultiOp and that they should be executed in parallel. Unlike a super scalar processor where independent operations are detected by the processor, the programmer/compiler has indicated to the processor by means of the stop bit that these three operations are independent. The multiply operation will typically have a higher latency than the other instructions. In that case, we have two different ways of scheduling this code. Since Defoe already uses scoreboarding to deal with variable load latencies, it is only natural for the scoreboard to stall the issue till the multiply operation is done. In a traditional VLIW processor, the compiler will insert additional NOPs after the first MultiOp. Lines 4–6 show how structural hazards are handled in a VLIW system. The compiler is aware that Defoe has only two simple integer ALUs. Even instruction 6 is independent of instructions 4 and 5 because of the unavailability of a suitable function unit; instruction 6 is issued as a separate MultiOp, one cycle after its two predecessors. In a super scalar processor, this decision will be handled at run time by the hardware.

### Example 2

This example contrasts the execution of an algorithm on Defoe and a super scalar processor (Intel Pentium). The C language function absdiff computes the sum of absolute difference of two arrays A and B, which contain 256 elements each.

```
int absdiff(int *A, int *B)
{
  int sum, diff, i;
  sum = 0;
  for(i = 0; i<256; i++)
{
    diff = A[i] - B[i];
    if(A[i] > = B[i])
    sum = sum + diff;
    else
    sum = sum - diff;
}
  return sum;
}
```

A hand-assembled version of absdiff in Defoe assembly language is shown below. For clarity, it has been left unoptimized. An optimizing compiler will unroll this loop and software pipeline it.

Register assignment: On entry, r1 = *a*, r2 = *b*. On exit, sum is in r4.

| *Line #* | *Code* | *Comment* |
|---|---|---|
| 1. | add r3 = r1, 2040 | // r3 = End of array A |
| 2. | add r4 = r0, r0; | // sum = r4 = 0 |
| .L1: | | |
| 3. | ld r5 = [ r1] | // load A[ i] |
| 4. | ld r6 = [ r2] | // load B[ i] |
| 5. | add r1 = r1, 8 | // Increment A address |
| 6. | add r2 = r2, 8 | // Increment B address |
| 7. | cmp.neq pr1 = r1, r3; | // pr1 = (i ! = 255) |
| 8. | sub r7, r5, r6 | // diff = A[ i] - B[ i] |
| 9. | cmp.gte pr2 = r5, r6; | // pr2 = (A[ i] >= B[ i] ) |
| 10. | (pr2) add r4 = r4, r7 | // if A[ i] >= B[ i]<br>// sum = sum + diff |
| 11. | (!pr2) sub r4 = r4, r7 | // else sum = sum - diff |
| 12. | (pr1) br.sptk .L1; | |

The corresponding code for an Intel processor is shown below. This is a snippet of actual code generated by the GCC compiler.

Stack assignment: On entry, 12(%ebp) = B, 8(%ebp) = A. On exit, sum is in the eax register.

| *Line #* | *Code* | | *Comment* |
|---|---|---|---|
| 1. | movl | 12(%ebp), %edi | // edi = B |
| 2. | xorl | %esi, %esi | // esi sum = 0 |
| 3. | xorl | %ebx, %ebx | // ebx = 0 |
| .p2align 2 | | | |
| .L6: | | | |
| 4. | movl | 8(%ebp), %eax | // eax = A |
| 5. | movl | (%eax,%ebx,4), %edx | // edx = A[ i] |
| 6. | movl | %edx, %ecx | // ecx = A[ i] |
| 7. | movl | (%edi,%ebx,4), %eax | // eax = B[ i] |
| 8. | subl | %eax, %ecx | // ecx = diff = A[ i] − B[ i] |
| 9. | cmpl | %eax, %edx | // A[ i] < B[ i] |
| 10. | jl | .L7 | // goto .L7 is A[ i] < B[ i] |
| 11. | addl | %ecx, %esi | // sum = sum + diff |
| 12. | jmp | .L5 | |
| .p2align 2 | | | |
| .L7: | | | |
| 13. | subl | %ecx, %esi | // sum = sum − diff |
| .L5: | | | |
| 14. | incl | %ebx | // i++ |
| 15. | cmpl | $255, %ebx | // i <= 255 ? |
| 16. | jle | .L6 | |
| 17. | popl | %ebx | |
| 18. | Movl | %esi, %eax | |

The level of parallelism available in the Defoe listing lines 3–7 (five issue) can be achieved on a super scalar processor only if the processor can successfully isolate the five independent operations fast enough to issue them all during the same cycle. Dependency checking in hardware is extremely complex and adds to the delay of super scalar processors. The ×86 being a register deficient CISC architecture also incurs additional penalties because of register renaming and CISC to internal RISC format translation.

It is worth noticing that the Defoe listing contains only one branch (on line 12) whereas the ×86 listing contains three branches. On a VLIW processor, we can often use predicated instructions to

eliminate branches. In both listings, line 9 corresponds to the comparison of A[$i$] and B[$i$]. The Pentium version does a conditional jump based on the result of the comparison. On the other hand, the VLIW uses the result of the comparison to set a predicate. The predicate is then used to selectively write back the result of either the add or the subtract operation and the result of the other operation is discarded. This technique of converting a control dependence into a data dependence is called "if conversion." The benefits go beyond the single cycle saved by not doing a jump as in the case of the super scalar processor. The jumps on lines 10 and 12 in the second listing depend on the condition code, which in turn depends on the data. Such data-dependent branches are difficult to predict. Assuming that $A[i] < B[i]$ and $A[i] \geq B[i]$ are equally likely, the super scalar processor is likely to experience a branch misprediction and the resulting branch penalty half of the time.

Going by the VLIW philosophy of conveying performance-critical information from the compiler to the hardware, the final branch on line 12 uses the opcode modifier "sptk" to inform the processor that the branch is statically predicted to be taken. For that particular loop, a VLIW processor can therefore predict the loop accurately 255 times out of 256 loop iterations without any hardware branch predictor. Even when a hardware branch predictor is available, the instruction advises the processor not to waste a branch history table entry on that branch because its behavior is already known at compile time.

### 1.2.5 Intel Itanium Processor

The Itanium-1 processor is Intel's first implementation of the IA-64 ISA. IA-64 is an ISA for the explicitly parallel instruction computing (EPIC) style of VLIW developed jointly by Intel and HP. It is a 64-bit, 6 issue VLIW processor with four integer units, four multimedia units, two load/store units, two extended precision floating-point units, and two single-precision floating-point units. This processor running at 800 MHz on a 0.18 $\mu$ process has a 10-stage pipeline.

Unlike the Defoe, the IA-64 architecture uses a fixed-width bundled instruction format. Each MultiOp consists of one or more 128-bit bundles. Each 128-bit bundle consists of three operations and a template. Unlike the Defoe where the opcode in each operation specifies a type field, the template encodes commonly used combinations of operation types. Since the template field is only 5 bits wide, bundles do not support all possible combinations of instruction types. Much like the Defoe's stop bit, in the IA-64, some template codes specify where in the bundle a MultiOp ends. In IA-64 terminology, MultiOps are called instruction groups. Like Defoe, the IA-64 uses a decoupling buffer to improve its issue rate. Although the IA-64 registers are nominally 64 bits wide, there is a hidden 65th bit called NaT (Not a Thing). This is used to support speculation. There are 128 general-purpose registers and another set of 128, 82-bit wide floating-point registers. Like the Defoe, all operations on the IA-64 are predicated. However, the IA-64 has 64 predicate registers.

The IA-64 register mechanism is more complex than the Defoe's because it implements support for software pipelining using a method similar to the overlapped loop execution support pioneered by Bob Rau and implemented in the Cydra 5. On the IA-64, general-purpose registers GPR0 to GPR31 are fixed. Registers 32–127 can be renamed under program control to support a register stack or to do modulo scheduling for loops. When used to support software pipelining, this feature is called register rotation. Predicate registers from 0 to 15 are fixed whereas PRs from 16 to 63 can be made to rotate in unison with the general-purpose registers. The floating-point registers also support rotation.

Modulo scheduling is a software pipelining technique that can support overlapped execution of loop bodies while reducing tail code. In a pipelined function unit, each stage can hold a computation, and successive items of data may be applied to the function unit before previous data is completely processed. To take advantage of pipelined operation, in a modulo scheduled loop, the loop body is unrolled and split into several stages. The compiler can schedule multiple iterations of a loop in a pipelined manner as long as data outputs of one stage flow into the inputs of the next stage in the software pipeline. Traditionally, this required unrolling the loop and renaming the registers used in successive iterations. IA-64 reduces the overhead of such a loop and avoids the need for register

renaming by rotating registers forward, i.e., the rotating register base is incremented in the direction of increasing register index. After rotating by $n$ registers, the value that was in register X + $n$ can be accessed from register X. When used in conjunction with predication, this allows a natural expression of software pipelines similar to their hardware counterparts.

The IA-64 supports software directed control and data speculation. To do control speculation, the compiler moves a load before its controlling branch. The load is then flagged as a speculative load. The processor does not signal exceptions on a speculative load. If the controlling branch is taken, the compiler uses a special opcode named check.s to determine if an exception occurred. If an exception occurred, the check operation transfers control to exception handling code.

To support data speculation, the processor supports a special kind of load called an advance load. If the compiler cannot disambiguate between the addresses of a store and a later load, it can issue an advance load ahead of the store. The processor uses a special hardware structure called the ALAT to keep track of whether a later store wrote to the same location as the advance load. In the original location where the load might naturally have been placed, the compiler inserts a special check operation to see if a store invalidated the result of the advance load. If the advance load was invalidated, the check operation transfers control to recovery code.

As in the case of Defoe, the IA-64 too supports both static and dynamic hints for branches. It also makes use of hardware branch prediction. There are also hints in load and store instructions that inform the processor about the cache behavior of a particular memory operation.

The IA-64 also includes SIMD instructions suitable for media processing. Special multimedia instructions similar to the MMX and SSE extensions for 80 × 86 processors treat the contents of a general-purpose register as two 32-bit, four 16-bit, or eight 8-bit operands and operate on them in parallel.

To improve performance, the IA-64 architecture includes several features that are not found in a traditional VLIW architecture. The Intel Itanium processor is probably the most complex VLIW ever designed. It is a matter of debate whether some of the control complexity of the IA-64 is justifiable in a VLIW architecture and whether the enhancements deliver commensurate performance improvements. Next, we will look at a simpler VLIW processor that has been designed with a totally different goal—that of reducing power consumption.

### 1.2.6 Transmeta Crusoe Processor

The Crusoe processor from Transmeta Corporation represents a very interesting point in the development of VLIW processors. Traditionally, VLIW processors were designed with the goal of maximizing ILP and performance. The designers of the Crusoe, on the other hand, needed to build a processor with moderate performance compared to the CPU of a desktop computer, but with the additional restriction that the Crusoe should consume very little power since it was intended for mobile applications. Another design goal was that it should be able to efficiently emulate the ISA of other processors, particularly the 80 × 86 processors and the Java virtual machine.

The designers left out features like out-of-order issue and dynamic scheduling that require significant power consumption. They set out to replace such complex mechanisms of gaining ILP with simpler and more power-efficient alternatives. The end result was a simple VLIW architecture. Long instructions on the Crusoe are either 64 or 128 bits. A 128-bit instruction word called a molecule in Transmeta parlance encodes four operations called atoms. The molecule format directly determines how operations get routed to function units. The Crusoe has two integer units, a floating-point unit, a load/store unit, and a branch unit. Like the Defoe, the Crusoe has 64 general-purpose registers and supports strictly in order issue. Unlike the Defoe, which uses predication, the Crusoe uses condition flags that are identical to those of the ×86 for ease of emulation.

Binary ×86 programs, firmware and operating systems are emulated with the help of a run time binary translator called code morphing software. This makes the classical VLIW software compatibility problem a nonissue. Only the native code morphing software needs to be changed when the Crusoe

architecture or ISA changes. As a power and performance optimization, the hardware and software together maintain a cache of translated code. The translations are instrumented to collect execution frequencies and branch history and this information is fed back to the code morphing software to guide its optimizations.

To correctly model the precise exception semantics of the ×86 processor, the part of the register file that holds ×86 register state is duplicated. The duplicate is called a shadow copy. Normal operations only affect the original registers. At the end of a translated section of code, a special commit operation is used to copy the working register values to the shadow registers. If an exception happens while executing a translated unit, the run time software uses the shadow copy to recreate the precise exception state. Store operations are implemented in a similar manner using a store buffer. As in the case of IA-64, the Crusoe provides alias detection hardware and data speculation primitives.

## 1.2.7 Scheduling Algorithms for VLIW

The difficulty of programming VLIW processors by hand should be evident even from the simple Defoe programming examples. One reason programming VLIWs is more difficult than writing code for a super scalar processor is that the program for a super scalar processor is inherently sequential and it is left to the hardware to extract parallelism from the sequential program. On the other hand, when generating code for a VLIW processor, the assembly language programmer or the compiler is faced with the task of extracting parallelism from a sequential algorithm and scheduling independent operations concurrently. For this reason, instruction scheduling algorithms are critical to the performance of a VLIW processor. We next describe three important scheduling algorithms starting with the classic trace scheduling algorithm.

### 1.2.7.1 Trace Scheduling

Many compilers for first-generation ILP processors used a three phase method to generate code. The following were the phases:

- Generate a sequential program. Analyze each basic block in the sequential program for independent operations.
- Schedule independent operations within the same block in parallel if sufficient hardware resources are available.
- Move operations between blocks when possible.

This three phase approach fails to exploit much of the ILP available in the program for two reasons.

- Often times, operations in a basic block are dependent on each other. Therefore sufficient ILP may not be available within a basic block.
- Arbitrary choices made while scheduling basic blocks make it difficult to move operations between blocks.

Trace scheduling is a profile-driven method developed by Joseph Fisher to circumvent this problem. In trace scheduling, a set of commonly executed sequence of blocks is gathered together into a trace and the whole trace is scheduled together.

The trace scheduling algorithm works as follows:

1. Generate a possibly unoptimized version of the program, run it on sample input and collect statistics. Estimate the probability of each conditional branch.
2. From the basic block level data precedence graph of the program (also commonly called DAG for directed acylic graph), select a loop-free linear sequence of basic blocks, which have a high probability of execution. Such a sequence is called a trace. The compiler may use other optimizations like loop unrolling or procedure inlining to generate DAGs from which suitable traces can be selected.

3. Consider the trace as if it were a basic block. Build a DAG for it considering branches like all other operations. If an operation controlled by a conditional jump could overwrite a value that is live on the off-trace edge, add an edge that makes the operation dependent on the branch so that the operation cannot be moved ahead of the branch. Also, add edges to preserve the relative order of conditional branches.
4. Schedule the resulting DAG as if it were a basic block doing register allocation and function unit selection as each operation is scheduled.
5. Generate compensation code for mistakes made by considering the trace as a basic block. In particular
   a. If an operation that used to precede a conditional branch in the sequential code is moved after that branch, then add a copy of that operation preceding the off-trace target of the conditional jump.
   b. If an operation that succeeded a point of entry into the trace from outside the trace is moved ahead of that point of entry, place a copy of that operation outside the trace, on the path that leads to that point of entry.
   c. Ensure that the rejoins used to enter the trace enter the new trace only at a point after which no operation is found in the new trace that were not below the rejoin point in the old trace.
6. Link the new trace back into the old DAG.
7. After scheduling the very first trace, new operations would have been added to the original DAG. Pick a different frequent trace and schedule it. Repeat till the DAG has been covered using disjoint traces and no unscheduled operations remain.

#### 1.2.7.2 Trace Scheduling-2

Trace scheduling-2 goes beyond the original trace scheduling in that it allows nonlinear code motion, i.e., it allows operations from both sides of a conditional branch to be moved above the branch. Trace scheduling usually misses code motions that are speculative or moves operations from one trace to another. Trace scheduling-2, on the other hand, uses an expected value function called speculative yield to consider the cost of speculative execution and decide whether or not to move operations from one block to another. Unlike trace scheduling, which operates on a linear sequence of blocks, the newer algorithm works by picking clusters of operations where each cluster is a maximal set of operations that are connected without back edges in the flow graph of the program. The actual details of the algorithm are beyond the scope of this section.

#### 1.2.7.3 Superblock Scheduling

Superblock scheduling is a region scheduling algorithm developed in conjunction with the IMPACT compiler at the University of Illinois. Like trace scheduling, superblock scheduling is based on the premise that extracting ILP from sequential programs requires code motion across multiple basic blocks. Unlike trace scheduling, superblock scheduling is driven by static branch analysis, not profile data. A superblock is a set of basic blocks in which control may enter only at the top, but may exit at more than one point. Superblocks are identified by first identifying the traces and then eliminating side entries into a trace by a process called tail duplication. Tail duplication works by creating a separate off-trace copy of the basic blocks in between a side entrance and the trace exit and redirecting the edge corresponding to the side entry to the copy. Traces are identified using static branch analysis based on loop detection, heuristic hazard avoidance, and heuristics for path selection. Loop detection identifies loops and marks loop back edges as taken and loop exits as not taken. Hazard avoidance uses a set of heuristics to detect situations like ambiguous stores and procedure calls that could cause a compiler to use conservative optimization strategies and then predicts the branches so as to avoid having to optimize hazards. Path selection heuristics use the opcode of a branch, its operands, and the contents of its successor blocks to predict its direction if no other method can be used to predict the outcome of the branch. These are based on common programming patterns like the fact that pointers are unlikely to be NULL,

floating-point comparisons are unlikely to be equal etc. Once branch information is available, traces are grown and superblocks created by tail duplication followed by scheduling of the superblock. Studies have shown that static analysis-based superblock scheduling can achieve results that are comparable to profile-based methods.

#### 1.2.7.4 Future of VLIW Processors

VLIW processors have enjoyed moderate commercial success in recent times as exemplified by the Philips Trimedia, TI TMS320C62x DSPs, Intel Itanium, and, to a lesser extent, the Transmeta Crusoe. However, the role of VLIW processors has changed since the days of Cydrome and Multiflow. Even though early VLIW processors were developed to be scientific supercomputers, newer processors have been used mainly for stream, image and digital signal processing, multimedia codec hardware, low-power mobile computers etc. VLIW compiler technology has made major advances during the last decade. However, most of the compiler techniques developed for VLIW are equally applicable to super scalar processors. Stream and media processing applications are typically very regular with predictable branch behavior and large amounts of ILP. They lend themselves easily to VLIW style execution. The ever-increasing demand for multimedia applications will continue to fuel development of VLIW technology. However, in the short term, super scalar processors will probably dominate in the role of general-purpose processors. Increasing wire delays in deep submicron processes will ultimately force super scalar processors to use simpler and more scalable control structures and seek more help from software. It is reasonable to assume that in the long run, much of the VLIW technology and design philosophy will make its way into main stream processors.

## References

1. J.A. Fisher, *Global code generation for instruction-level parallelism: Trace scheduling-2.* Technical Report HPL-93-43, Hewlett–Packard Laboratories, June 1993.
2. J.A. Fisher, *Very long instruction word architectures and the ELI-512*, Proceedings of the 10th Symposium on Computer Architectures, pp. 140–150, IEEE, June 1983.
3. J.A. Fisher, *Very Long Instruction Word Architectures and the ELI-512. 25 Years ISCA*: Retrospectives and Reprints 1998: 263–273.
4. M. Schlansker, B.R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S.G. Abraham, *Achieving high levels of instruction-level parallelism with reduced hardware complexity.* Technical Report HPL-96-120, Hewlett–Packard Laboratories, February 1997.
5. M. Schlansker and B.R. Rau, *Epic: an architecture for instruction level parallel processors.* Technical Report HPL-1999-111, Hewlett–Packard Laboratories, February 2000.
6. S. Rixner, W.J. Dally, U.J. Kapasi, K. Brucek, A. Lopez-Lagunas, P.R Mattson, and J.D. Owens, *A bandwidth-efficient architecture for media processing.* Proceedings of the 31st Annual International Symposium on Microarchitecture, Dallas, TX, November 1998.
7. Intel Corporation. *Itanium Processor Microarchitecture Reference for Software Optimization.* Intel Corporation, Santa Clara, CA, March 2000.
8. Intel Corporation. *Intel IA-64 Architecture Software Developer's Manula, Volume 3: Instruction Set Reference.* Intel Corporation, Santa Clara, CA, January 2000.
9. Intel Corporation. *IA-64 Application Developer's Architecture Guide.* Intel Corporation, Santa Clara, CA, May 1999.
10. P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg, The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7, 1993.
11. R.E. Hank, S.A. Mahlke, J.C. Gyllenhaal, R. Bringmann, and W.W. Hwu, *Superblock formation using static program analysis*, Proceedings of the 26th Annual International Symposium on Microarchitecture, Austin, TX, pp. 247–255, December 1993.

12. S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann, *Effective compiler support for predicated execution using the hyperblock*, Proceedings of the 25th International Symposium on Microarchitecture, pp. 45–54, December 1992.
13. J.C. Dehnert, P.Y.-T. Hsu, and J.P. Bratt, *Overlapped loop support in the Cydra 5*, Proceedings of ASPLOS 89, pp. 26–38, 1989.
14. A. Klaiber, *The Technology Behind Crusoe Processors*. Transmeta Corporation, Santa Clara, CA, 2000.

## 1.3 Vector Processing

*Krste Asanović*

### 1.3.1 Introduction

For over 30 years, vector processing has been used in the world's fastest supercomputers to accelerate applications in scientific and technical computing. More recently, vector-like extensions have become popular in desktop and embedded microprocessors to accelerate multimedia applications. In both cases, architects are motivated to include data-parallel instructions because they enable large increases in performance at much lower cost than alternative approaches to exploiting application parallelism. This chapter reviews the development of data-parallel instruction sets from the early SIMD (single instruction, multiple data) machines, through the vector supercomputers, to the new multimedia instruction sets.

### 1.3.2 Data Parallelism

An application is said to contain data parallelism when the same operation can be carried out across arrays of operands, for example, when two vectors are added element by element to produce a result vector. Data-parallel operations are usually expressed as loops in sequential programming languages. Data-parallel instructions can be used to execute the code, if each loop iteration is independent of the other. The following vector add code written in C is a simple example of a data-parallel loop:

```
for (i = 0; i<N; i++)
C[i] = A[i] +B[i];
```

Provided that the result array C does not overlap the source arrays A and B, the individual loop iterations can be run in parallel. Many compute-intensive applications are built around such data-parallel loop kernels. One of the most important factors in determining the performance of data-parallel programs is the range of vector lengths observed for typical data sets. Vector lengths vary depending on the application, how the application is coded, and also on the input data for each run. In general, the longer the vectors, the greater the performance achieved by a data-parallel architecture, as any loop start-up overheads will be amortized over a larger number of elements.

The performance of a piece of vector code running on a data-parallel machine can be summarized with a few key parameters. $R_n$ is the rate of execution (for example, in MFLOPS) for a vector of length $n$. $R_\infty$ is the maximum rate of execution achieved assuming infinite length vectors. $N_{\frac{1}{2}}$ is the number of elements at which vector performance reaches one half of $R_\infty$. $N_{\frac{1}{2}}$ indirectly measures start-up overhead, as it gives the vector length at which the time lost to overheads is equal to the time taken to execute the vector operation at peak speed ignoring overheads. The larger the $N_{\frac{1}{2}}$ for a code kernel running on a particular machine, the longer the vectors must be to achieve close to peak performance.

### 1.3.3 History of Data-Parallel Machines

Data-parallel architectures were first developed to provide high throughput for supercomputing applications. There are two main classes of data-parallel architectures: distributed-memory SIMD

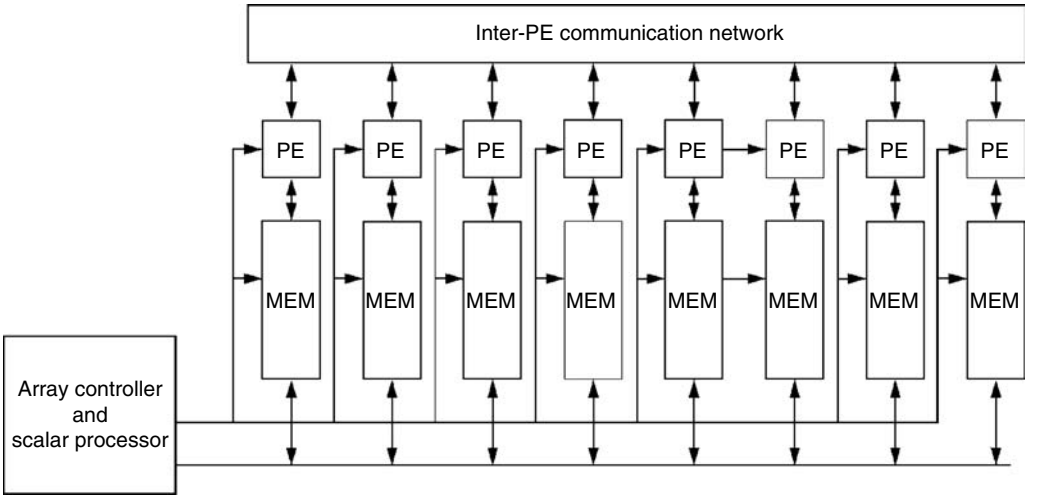



**FIGURE 1.7** Structure of a distributed-memory SIMD (DM-SIMD) processor.

(DM-SIMD) [1] architecture and shared-memory vector architecture. An early example of a DM-SIMD architecture is the Illiac-IV [2]. A typical DM-SIMD architecture has a general-purpose scalar processor acting as the central controller and an array of processing elements (PEs) each with its own private memory, as shown in Fig. 1.7. The central processor executes arbitrary scalar code and also fetches instructions, and broadcasts them across the array of PEs, which execute the operations in parallel and in lockstep. Usually the local memories of the PE array are mapped onto the central processor's address space so that it can read and write any word in the entire machine. PEs can communicate with each other using a separate parallel inter-PE data network. Many DM-SIMD machines, including the ICL DAP [3] and the Goodyear MPP [4], used single-bit processors connected in a 2-D mesh, providing communications well matched to image processing or scientific simulations that could be mapped to a regular grid. The later Connection Machine design [5] added a more flexible router to allow arbitrary communication between single-bit PEs, although at much slower rates than the 2-D mesh network. One advantage of single-bit PEs is that the number of cycles taken to perform a primitive operation, such as an add can scale with the precision of the operands, making them well suited to tasks such as image processing, where low-precision operands are common. An alternative approach was taken in the Illiac-IV where wide 64-bit PEs could be subdivided into multiple 32-bit or 8-bit PEs to give higher performance on reduced precision operands. This approach reduces $N_{\frac{1}{2}}$ for calculations on vectors with wider operands but requires more complex PEs. This same technique of subdividing wide datapaths has been carried over into the new generation of multimedia extensions (MX) for microprocessors. The main attraction of DM-SIMD machines is that the PEs can be much simpler than the central processor because they do not need to fetch and decode instructions. This allows large arrays of simple PEs to be constructed, for example, up to 65,536 single-bit PEs in the original Connection Machine design.

Shared-memory vector architectures (henceforth abbreviated as vector architectures) also belong to the class of SIMD machines, as they apply a single instruction to multiple data items. The primary difference in the programming model of vector machines versus DM-SIMD machines is that vector machines allow any PE to access any word in the system's main memory. Because it is difficult to construct machines that allow a large number of simple processors to share a large central memory, vector machines typically have a smaller number of highly pipelined PEs.

The two earliest commercial vector architectures were the CDC STAR-100 [6] and the TI ASC [7]. Both these machines were vector memory–memory architectures, where the vector operands to a vector instruction were streamed in and out of memory. For example, a vector add instruction would specify the start addresses of both source vectors and the destination vector; during execution, elements were

fetched from memory before being operated on by the arithmetic unit, which produced a set of results that had to be written back to main memory.

The Cray-1 [8] was the first commercially successful vector architecture and introduced the idea of vector registers. A vector register architecture provides vector arithmetic operations that can only take operands from vector registers, with vector load and store instructions that only move data between the vector registers and memory. Vector registers hold short vectors close to the vector functional units, shortening instruction latencies and allowing vector operands to be reused from registers thereby reducing memory bandwidth requirements. These advantages have led to the dominance of vector register architectures and vector memory–memory machines are ignored for the rest of this section.

DM-SIMD machines have two primary disadvantages compared to vector supercomputers when writing applications. The first disadvantage is that the programmer has to be extremely careful in selecting algorithms and mapping data arrays across the machine to ensure that each PE can satisfy almost all of its data accesses from its local memory, while ensuring the local data set still fits into the limited local memory of each PE. In contrast, the PEs in a vector machine have equal access to all main memories, and the programmer only has to ensure that data accesses are spread across all the interleaved memory banks in the memory system.

The second disadvantage is that DM-SIMD machines typically have a large number of simple PEs and so to avoid having many PEs sit idle, applications must have long vectors. For the large-scale DM-SIMD machines, $N_{\frac{1}{2}}$ can be in the range of tens of thousands of elements. In contrast, the vector supercomputers contain a few highly pipelined PEs and have $N_{\frac{1}{2}}$ in the range of tens to hundreds of elements.

To make effective use of a DM-SIMD machine, the programmer has to find a way to restructure code to contain very long vector lengths, while simultaneously mapping data structures to distributed small local memories in each PE. Achieving high performance under these constraints has proven difficult except for a few specialized applications. In contrast, the vector supercomputers do not require data partitioning and provide reasonable performance on much shorter vectors and so require much less effort to port and tune applications. Although DM-SIMD machines can provide much higher peak performances than vector supercomputers, sustained performance was often similar or lower, and programming effort was much higher. As a result, although they achieved some popularity in the 1980s, DM-SIMD machines have disappeared from the general-purpose computing market with no current commercial manufacturers, whereas high-end vector supercomputers are still being produced by Cray and NEC. DM-SIMD architectures remain popular in a few niche special-purpose areas, particularly in image processing and in graphics rendering, where the natural application parallelism maps well onto the DM-SIMD array, providing extremely high throughput at low cost.

Although data-parallel instructions were originally introduced for high-end supercomputers, they can be applied to many applications outside of scientific and technical supercomputing. Beginning with the Intel i860 released in 1989, microprocessor manufacturers have introduced data-parallel instruction set extensions which allow a small number of parallel SIMD operations to be specified in single instruction. These microprocessor SIMD ISA (instruction set architecture) extensions were originally targeted at multimedia applications and supported only limited-precision, fixed-point arithmetic, but now support single- and double-precision floating-point arithmetic and hence a much wider range of applications. In this chapter, SIMD ISA extensions are viewed as a form of short vector instruction to allow a unified discussion of design trade-offs.

### 1.3.4 Basic Vector Register Architecture

Vector processors contain a conventional scalar processor that executes general-purpose code together with a vector processing unit that handles data-parallel code. Figure 1.8 shows the general architecture of a typical vector machine. The vector processing unit includes a set of vector registers and a set of vector functional units that operate on the vector registers. Each vector register contains a set of two or more data elements. A typical vector arithmetic instruction reads source operand vectors from two vector registers, performs an operation pairwise on all elements in each vector register, and writes a result

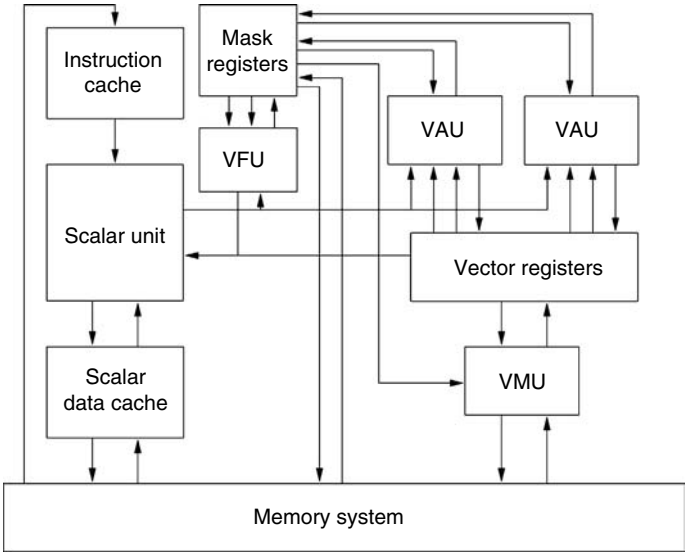



**FIGURE 1.8** Structure of a vector machine. This example has a central vector register file, two vector arithmetic units (VAU), one vector memory load/store unit (VMU), and one vector flag/mask unit (VFU) that operates on the mask registers. (Adapted from Asanovic, K., *Vector Microprocessors*, Ph.D. Thesis, University of California, Berkeley, 1998. With permission.)

vector to a destination vector register, as shown in Fig. 1.9. Often, versions of vector instructions are provided that replace one vector operand with a scalar value; these are termed vector–scalar instructions. The scalar value is used as one of the operand inputs at each element position.

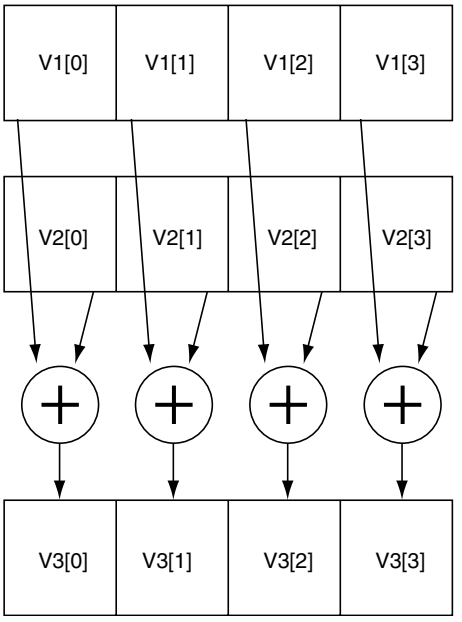



**FIGURE 1.9** Operation of a vector add instruction. Here, the instruction is adding vector registers 1 and 2 to give a result in vector register 3.

The vector ISA usually fixes the maximum number of elements in each vector register, although some machines, such as the IBM vector extension for the 3090 mainframe, support implementations with differing numbers of elements per vector register. If the number of elements required by the application is less than the number of elements in a vector register, a separate vector length register (VLR) is set with the desired number of operations to perform. Subsequent vector instructions only perform this number of operations on the first elements of each vector register. If the application requires vectors longer than will fit into a vector register, a process called *strip mining* is used to construct a vector loop that executes the application code loop in segments that each fit into the machine's vector registers. The MX ISAs have very short vector registers and do not provide any vector length control. Various types of vector load and store instruction can be provided to move vectors between the vector register file and memory. The simplest form of vector load and store transfers a set of elements that are contiguous in memory to successive elements of a vector register. The base address is usually specified by the contents of a register in the scalar processor. This is termed a unit-stride load or store, and is the only type of vector load and store provided in existing MX instruction sets.

Vector supercomputers also include more complex vector load and store instructions. A strided load or store instruction transfers memory elements that are separated by a constant stride, where the stride is specified by the contents of a second scalar register. Upon completion of a strided load, vector elements that were widely scattered in memory are compacted into a dense form in a vector register suitable for subsequent vector arithmetic instructions. After processing, elements can be unpacked from a vector register back to memory using a strided store.

Vector supercomputers also include indexed load and store instructions to allow elements to be collected into a vector register from arbitrary locations in memory. An indexed load or store uses a vector register to supply a set of element indices. For an indexed load or gather, the vector of indices is added to a scalar base register to give a vector of effective addresses from which individual elements are gathered and placed into a densely packed vector register. An indexed store, or scatter, inverts the process and scatters elements from a densely packed vector register into memory locations specified by the vector of effective addresses.

Many applications contain conditionally executed code; for example, the following loop clears values of A[i] smaller than some threshold value:

```
for (i=0; i<N; i++)
    if (A[i] <threshold)
        A[i] =0;
```

Data-parallel instruction sets usually provide some form of conditionally executed instruction to support parallelization of such loops. In vector machines, one approach is to provide a mask register that has a single-bit per element position. Vector comparison operations test a predicate at each element and set bits in the mask register at element positions where the condition is true. A subsequent vector instruction takes the mask register as an argument, and at element positions where the mask bit is set, the destination register is updated with the result of the vector operation, otherwise the destination element is left unchanged. The vector loop body for the previous vector loop is shown below (with all strip mining loop code removed).

```
lv va, (ra) # Load slice of vector A from memory
cmp.lt.vs va, rt # Set mask where A[i] <threshold
move.vs.m va, r0 # Clear elements of A[i] under mask
sv va, (ra) # Store updated slice of A to memory
```

### 1.3.5 Vector Instruction Set Advantages

Vector instruction set extensions provide a number of advantages over alternative mechanisms for encoding parallel operations. Vector instructions are compact, encoding many parallel operations in a

single, short instruction, compared with superscalar or very long instruction word (VLIW) instruction sets, which encode each individual operation using a separate collection of instruction bits.

They are also expressive, relaying much useful information from software to hardware. When a compiler or programmer specifies a vector instruction, they indicate that all of the elemental operations within the instruction are independent, allowing hardware to execute the operations using pipelined execution units, or parallel execution units, or any combination of pipelined and parallel execution units, without requiring dependency checking or operand bypassing for elements within the same vector instruction. Vector ISAs also reduce the dependency checking required between two different vector instructions. Hardware only has to check dependencies once per vector register, not once per elemental operation. This dramatically reduces the complexity of building high-throughput execution engines compared with reduced instruction set computer (RISC) or VLIW scalar cores, which have to perform dependency and interlock checking for every elemental result. Vector memory instructions can also relay much useful information to the memory subsystem by passing a whole stream of memory requests together with the stride between the elements in the stream.

Another considerable advantage of a vector ISA is that it simplifies scaling of implementation parallelism. As described in the next section, the degree of parallelism in the vector unit can be increased while maintaining object–code compatibility.

### 1.3.6 Lanes: Parallel Execution Units

Figure 1.10a shows the execution of a vector add instruction on a single-pipelined adder. Results are computed at the rate of one element per cycle. Figure 1.10b shows the execution of a vector add

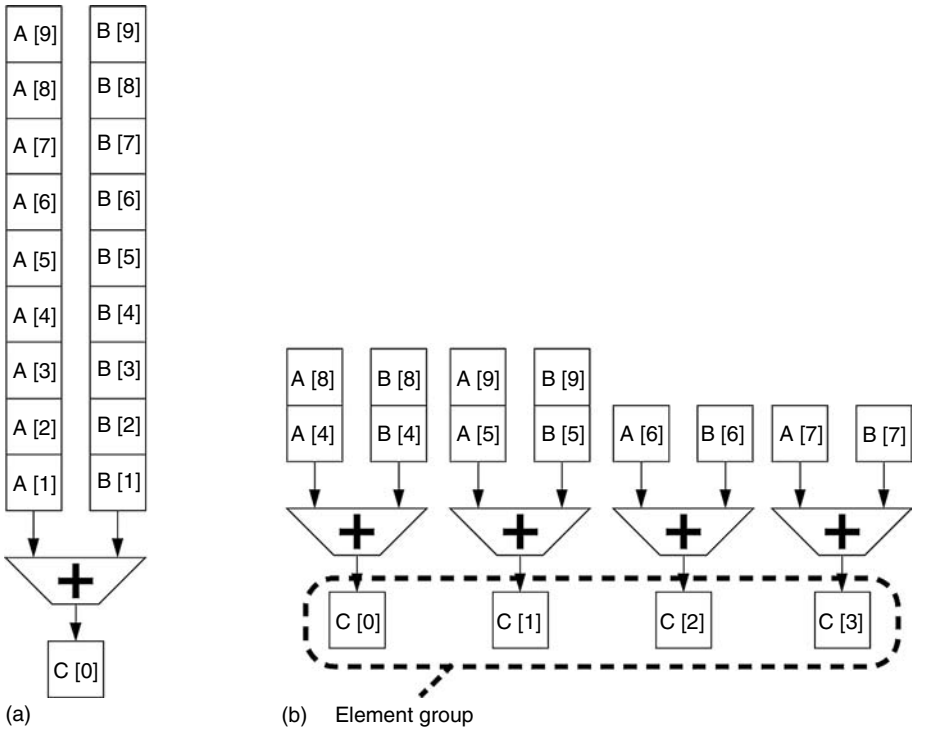



**FIGURE 1.10** Execution of vector add instruction using different numbers of execution units. The machine in (a) has a single adder and completes one result per cycle, while the machine in (b) has four adders and completes four results every cycle. An element group is the set of elements that proceed down the parallel pipelines together. (From Asanovic, K., *Vector Microprocessors*, Ph.D. Thesis, University of California, Berkeley, 1998. With permission.)

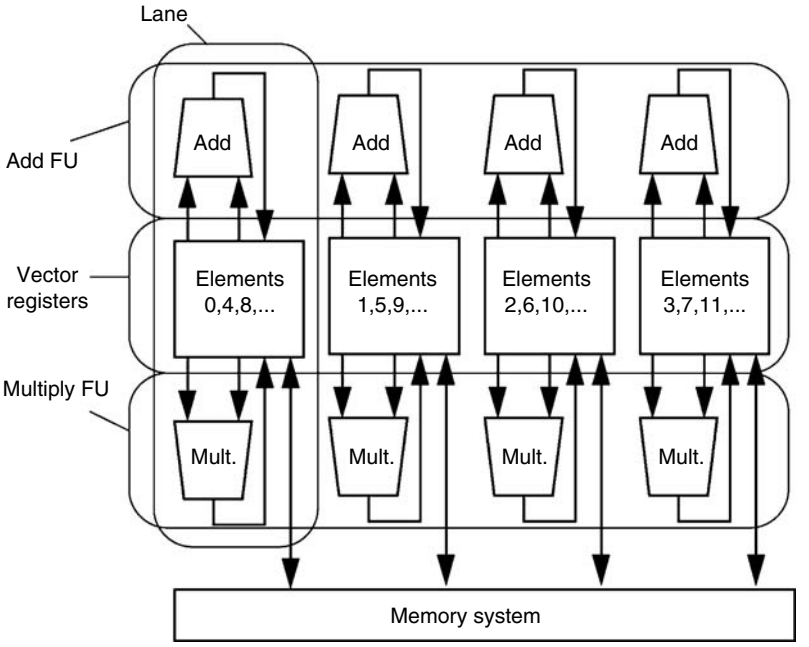



**FIGURE 1.11** A vector unit constructed from replicated lanes. Each lane holds one adder and one multiplier as well as one portion of the vector register file and a connection to the memory system. The adder functional unit (adder FU) executes add instructions using all four adders in all four lanes.

instruction using four parallel pipelined adders. Elements are interleaved across the parallel pipelines allowing up to four element results to be computed per cycle. This increase in parallelism is invisible to software except for the increased performance.

Figure 1.11 shows how a typical vector unit can be constructed as a set of replicated lanes, where each lane is a cluster containing a portion of the vector register file and one pipeline from each vector functional unit. Because of the way the vector ISA is designed, there is no need for communication between the lanes except via the memory system. The vector registers are striped over the lanes, with lane 0 holding all elements 0, $N$, $2N$, etc., lane 1 holding elements 1, $N+1$, $2N+1$, etc. In this way, each elemental vector arithmetic operation will find its source and destination operands located within the same lane, which dramatically reduces interconnect costs.

Vector supercomputers can have as many as 16 parallel 64-bit lanes in each CPU. For example, the NEC SX-5 can complete 16 loads, sixteen 64-bit floating-point multiplies, and sixteen 64-bit floating-point adds each clock cycle.

Many data-parallel systems, ranging from vector supercomputers, such as the early CDC STAR-100, to the MX ISAs, such as AltiVec, provide variable precision lanes, where a wide 64-bit lane can be subdivided into a larger number of lower precision lanes to give greater performance on reduced precision operands. This technique can be traced back to the MIT Lincoln Labs TX-2 computer from 1957 (famous as the machine used to run Ivan Sutherland's seminal Sketchpad program), which had a 36-bit datapath that could be divided into two 18-bit lanes, or four 9-bit lanes [9].

### 1.3.7 Vector Register File Organization

Vector machines differ widely in the organization of the vector register file. The important software-visible parameters for a vector register file are the number of vector registers, the number of elements in each vector register, and the width of each element. The Cray-1 had eight vector registers each holding sixty four 64-bit elements (a total of 4096 bits). The AltiVec MX for the PowerPC has 32 vector

registers each holding 128-bits that can be divided into four 32-bit elements, eight 16-bit elements, or sixteen 8-bit elements. Some vector supercomputers have extremely large vector register files organized in a vector register hierarchy, e.g., the NEC SX-5 has 72 vector registers (8 foreground plus 64 background) that can each hold five hundred and twelve 64-bit elements.

For a fixed vector register storage capacity (measured in elements), an architecture has to choose between few longer vector registers and more shorter vector registers. The primary advantage of lengthening a vector register is that it reduces the instruction bandwidth required to attain a given level of performance because a single instruction can specify a greater number of parallel operations. Increases in vector register length give rapidly diminishing returns, as amortized start-up overheads become small and as fewer applications can take advantage of the increased vector register length.

The primary advantage of providing more vector registers is that it allows more temporary values to be held in registers, reducing data memory bandwidth requirements. For machines with only eight vector registers, vector register spills have been shown to consume up to 70% of all vector memory traffic, while increasing the number of vector registers to 32 removes most register spill traffic [10,11]. Adding more vector registers also gives compilers more flexibility in scheduling vector instructions to boost vector instruction-level parallelism. In 2002, Cray announced a completely redesigned vector instruction set for their new Cray X1 vector computer [12]. The X1 instruction set increases the number of software-visible vector registers from eight, used on all earlier Cray machines since the Cray-1, to a more modern 32.

Some vector machines provide a configurable vector register file to allow software to dynamically choose the optimal configuration. For example, the Fujitsu VPP 5000 allows software to select vector register configurations ranging from 256 vector registers each holding 128 elements to eight vector registers holding 4096 elements each. For loops where few temporary values exist, longer vector registers can be used to reduce instruction bandwidth and strip mining overhead, whereas for loops where many temporary values exist, the number of shorter vector registers can be increased to reduce the number of vector register spills and, hence, the data memory bandwidth required. The main disadvantage of a configurable vector register file is the increase in control logic complexity and the increase in machine state to hold the configuration information.

### 1.3.8 Traditional Vector Computers versus Microprocessor Multimedia Extensions

Traditional vector supercomputers were developed to provide high performance on data-parallel code developed in a compiled high-level language (almost always a dialect of FORTRAN) while requiring only simple control units. Vector registers were designed with a large number of elements (64 for the Cray-1). This allowed a single-vector instruction to occupy each deeply pipelined functional unit for many cycles. Although only a single instruction could be issued per cycle, by starting separate vector instructions on different vector functional units, multiple vector instructions could overlap in execution at one time. In addition, adding more lanes allows each vector instruction to complete more elements per cycle.

MX ISAs for microprocessors evolved at a time when the base microprocessors were already issuing multiple scalar instructions per cycle. Another distinction is that the MX ISAs were not originally developed as compiler targets, but were intended for use in a few key library routines. This helps in explaining why MX ISAs, although sharing many attributes with earlier vector instructions, have evolved differently. The very short vectors in MX ISAs allow each instruction to specify only one or two cycle's worth of work for the functional units. To keep multiple functional units busy, the superscalar dispatch capability of the base scalar processor is used. To hide functional unit latencies, the multimedia code must be loop unrolled and software pipelined. In effect, the multimedia engine is being programmed in a microcoded style with the base scalar processor providing the microcode sequencer and each MX instruction representing one microcode primitive for the vector engine.

This approach of providing only primitive microcode level operations in the MX also explains the lack of other facilities standard in a vector ISA. One example is vector length control. Rather than using long

vectors and a VLR, the MX ISAs provide short vector instructions that are placed in unrolled loops to operate on longer vectors. These unrolled loops can only be used with long vectors that are a multiple of the intrinsic vector length multiplied by the unrolling factor. Extra code is required to check for shorter vectors and to jump to separate code segments to handle short vectors and the remnants of any longer vector that were not handled by the unrolled loop. This overhead is greater than for the strip mining code in traditional vector ISAs, which simply set the VLR appropriately in the last iteration of the strip-mined loop.

Vector loads and stores are another place where functionality has been moved into software for the MX ISAs. Most MX ISAs only provide unit-stride loads and stores that have to be aligned on boundaries corresponding to the vector length, not just aligned at element boundaries as in regular scalar code. For example, a unit-stride load of four 16-bit quantities has to be aligned at 64-bit boundaries in most MX instruction sets, although in some cases hardware will handle misaligned loads and stores at a slower rate. To help handle misaligned application vectors, various shift and align instructions have been added to MX ISAs to allow misalignment to be handled as part of the software microcoded loop. This approach simplifies the hardware design, but unfortunately these misaligned vectors are common in application code, and significant slowdown occurs when performing alignment in software. This encourages the use of loops optimized for certain operand alignments, which leads to an increase in code size and also in loop start-up time to select the appropriate routine. In certain cases, the application can constrain the layout of the data elements to ensure alignment at the necessary boundaries, but typically this is only possible when the entire application has been optimized for these MX instructions, for example, in a dedicated media player. Strided and indexed operations are also usually coded as scalar loads and stores with a corresponding slowdown over full vector mode.

### 1.3.9 Memory System Design

Perhaps the biggest difference between microprocessors and vector supercomputers is in the capabilities of the vector memory system. Vector supercomputers usually forgo data caches and rely on many banks of interleaved main memory to provide high memory bandwidth, whereas microprocessors rely on multilevel cache hierarchies to isolate the CPU from memory latencies and limited main memory bandwidth. A modern high-end vector supercomputer provides over 50 GB/s of main memory bandwidth per CPU, whereas high-end microprocessor systems provide only around 5 GB/s per CPU. For applications that require nonunit stride accesses to large data sets, the bandwidth discrepancy is even larger, because microprocessors access memory using long cache lines that waste bandwidth when there is little spatial locality. A modern vector CPU might sustain 16 or 32 nonunit stride memory operations every cycle pipelined out to main memory, with hundreds or thousands of outstanding memory accesses, whereas a microprocessor usually can only have a total of four–eight cache line misses outstanding at any time. This large difference in nonunit stride memory bandwidth is the main reason that vector supercomputers remain popular for certain applications, including car crash simulation and weather forecasting.

Traditional vector ISAs use long vector registers to help hide memory latency. MX ISAs have only very short vector registers and so require a different mechanism to hide memory latency and make better use of available main memory bandwidth. Various forms of hardware and software prefetching schemes have become popular with microprocessor designers to hide memory latency. Hardware prefetching schemes dynamically inspect memory references and attempt to predict the data that will be needed next, fetching these into the cache before requested by the application. This approach has the advantage of not requiring changes to software, but can be inaccurate and can consume excessive memory bandwidth for incorrectly speculated prefetches.

Software prefetching schemes can be very accurate as the compiler knows the reference patterns of each piece of code, but the software prefetch instructions have to be carefully scheduled so that data are not brought in too early, perhaps evicting useful data, or too late, which will leave some memory latency exposed. The optimal schedule depends on the CPU and memory system implementation,

which implies that prefetch code optimized for one generation of CPU or one particular memory system will not be optimal for another.

For either hardware or software prefetching schemes, it is essential that the memory controller can support many outstanding requests; otherwise, high memory bandwidths cannot be sustained from a typical high-latency memory system.

### 1.3.10 Future Directions

Microprocessor architects are continually searching for techniques that can take advantage of ever-increasing transistor counts to improve application performance. Data-parallel ISA extensions have proven effective on a wide range of applications, and hardware designs scale well to more parallel lanes. Existing supercomputers have sixteen 64-bit lanes while microprocessor MX implementations have expanded to two 64-bit lanes. It is likely that there will be further expansion of MX units to four or more 64-bit lanes. At higher lane counts, efficiencies drop, partly because of limited application vector lengths and partly because additional lanes do not help nondata parallel portions of each application.

An alternative approach to attaining high throughput on data-parallel applications is to build a multicore chip, i.e., multiple independent CPUs each with vector units, and to parallelize loops at the thread level. This technique also allows independent CPUs to run different tasks to improve system throughput. The main disadvantages of this multiprocessor approach compared to simply increasing the number of lanes are the hardware costs of additional scalar processor logic and the additional inter-CPU synchronization overhead. The relative cost of adding more CPUs is reduced as lane counts grow, particularly, when the cost of providing sufficient main memory bandwidth is considered. The inter-CPU synchronization cost is a more serious issue as it adds to vector start-up latencies and can increase $N_{\frac{1}{2}}$ dramatically, reducing the effectiveness of multiprocessors on shorter vectors. For this reason, multiprocessor vector supercomputers have added dedicated hardware for fast inter-CPU synchronization. For example, the Cray X1 gangs together four 2-lane processors in software to appear as a single 8-lane processor to the user [12]. It should be expected that some form of fast inter-CPU synchronization primitive will be added to general-purpose ISAs as chip-scale multiprocessors become common, as these primitives can also be applied to many types of thread-level parallel code.

Increased CPU clock frequencies and increased lane counts combine to dramatically increase the memory bandwidth required by a vector CPU. The cost of a traditional vector style memory system has become prohibitive even for high-end vector supercomputers. Even if the cost could be justified, the high memory latency of a flat memory system will hamper performance for applications that have lower degrees of parallelism and that can fit in caches, and a continued move toward cached memory hierarchies for vector machines is to be expected, leading to a merging of vector supercomputer and microprocessor design points. The Cray X1 relies on a cached memory hierarchy, for example, and cannot sustain full vector load and store bandwidth from main memory.

MX for microprocessors have undergone considerable changes since their introduction. The current designs provide low-level arithmetic and memory system primitives that are intended to be used in hand-microcoded loops. These result in high start-up overheads and large code size relative to traditional vector extensions as discussed above. A possible future direction that could merge the benefit of vector ISAs and out-of-order superscalar microprocessors would be to add vector-style ISA extensions, but have these interpreted by microcode sequencers that would produce internal elemental microoperations that would be passed through the regular register renaming and out-of-order dispatch stages of a modern superscalar execution engine. This would be similar to the way that legacy complex instruction set computers (CISC) string operations are handled by modern implementations.

### 1.3.11 Conclusions

Data-parallel instructions have appeared in many forms in high-performance computer architectures over the last 30 years. They remain popular because many applications are amenable to data-parallel

execution, and because data-parallel hardware is the simplest and cheapest way to exploit this type of application parallelism. As MX evolve, they are likely to adopt more of the characteristics of traditional shared-memory vector ISAs to reduce loop start-up overhead and decrease code size. However, these new multimedia vector ISAs will be shaped by the need to coexist with the speculative out-of-order execution engines used by the superscalar processors.

### References

1. Flynn, M.J., Very high-speed computing systems, *Proceedings of the IEEE*, 54, 1901, 1966.
2. Barnes, G.H. et al., The Illiac IV computer, *IEEE Transactions on Computers*, C-17, 46, 1968.
3. Reddaway, S.F., DAP–A distributed array processor, in *Proceedings of the 1st Annual Symposium on Computer Architectures*, Gainesville, FL, 61, 1973.
4. Batcher, K.E., Architecture of a massively parallel processor, in *Proceedings of the 7th Annual Symposium on Computer Architecture*, ACM Press, La Baule, NY, 168–173, 1980.
5. Hillis, W.D., *The Connection Machine*, MIT Press, Cambridge, MA, 1985.
6. Hintz, R.G. and Tate, D.P., Control data STAR-100 processor design, in *Proceedings of the 6th Annual IEEE Computer Society*, International Conference (COMPCON), CA, 1972, 1.
7. Cragon, H.G. and Watson, W.J., A retrospective analysis: The TI advanced scientific computer, *IEEE Computer*, 22, 55, 1989.
8. Russel, R.M., The CRAY-1 computer system, *Communications of the ACM*, 21, 63, 1978.
9. Frankovich, J.M. and Peterson, H.P., A functional description of the Lincoln TX-2 computer, *Western Joint Computer Conference*, Los Angeles, CA, 1957.
10. Espasa, R., *Advanced Vector Architectures*, Ph.D. Thesis, Universitat Politecnica de Catalunya, Barcelona, 1997.
11. Asanovic, K., *Vector Microprocessors*, Ph.D. Thesis, University of California, Berkeley, CA, 1998.
12. Cray, Inc., Cray X1 System Overview, Technical manual S-2346–22, 2002.

## 1.4 Multithreading, Multiprocessing

*Manoj Franklin*

### 1.4.1 Introduction

A defining challenge for research in computer science and engineering has been the ongoing quest for faster execution of programs. There is broad consensus that barring the use of novel technologies such as quantum computing and biological computing, the key to further progress in this quest is to do parallel processing of some kind.

The commodity microprocessor industry has been traditionally looking to fine-grained or instruction level parallelism (ILP) for improving performance, with sophisticated microarchitectural techniques (such as pipelining, branch prediction, out-of-order execution, and superscalar execution) and sophisticated compiler optimizations. Such hardware-centered techniques appear to have scalability problems in the sub-micron technology era and are already appearing to run out of steam. According to a recent position paper by Dally and Lacy [4], "Over the past 20 years, the increased density of VLSI chips was applied to close the gap between microprocessors and high-end CPUs. Today this gap is fully closed and adding devices to uniprocessors is well beyond the point of diminishing returns." We view ILP as the main success story form of parallelism thus far, as it was adopted in a big way in the commercial world for reducing the completion time of general purpose applications. The future promises to expand the "parallelism bridgehead" established by ILP with the "ground forces" of thread-level parallelism (TLP), by using multiple processing elements to exploit both fine-grained and coarse-grained parallelism in a natural way.

Current hardware trends are playing a driving role in the development of multiprocessing techniques. Two important hardware trends in this regard are single chip transistor count and clock speed, both of which have been steadily increasing due to advances in sub-micron technology. The Semiconductor Industry Association (SIA) has predicted that by 2012, industry will be manufacturing processors containing 1.4 billion transistors and running at 10 GHz [39]. DRAMs will grow to 4 Gbits in 2003. This increasing transistor budget has opened up new opportunities and challenges for the development of on-chip multiprocessing.

One of the challenges introduced by sub-micron technology is that wire delays become more important than gate delays [39]. This effect is predominant in global wires because their length depends on the die size, which is steadily increasing. An important implication of the physical limits of wire scaling is that, the area that is reachable in a single clock cycle of future processors will be confined to a small portion of the die [39].

A natural way to make use of the additional transistor budget and to deal with the wire delay problem is to use the concept of *multithreading* or *multiprocessing** in the processor microarchitecture. That is, build the processor as a collection of independent *processing elements* (*PEs*), each of which executes a separate *thread* or flow of control. By designing the processor as a collection of PEs, (a) the number of global wires reduces, and (b) very little communication occurs through global wires. Thus, much of the communication occurring in the multi-PE processor is *local* in nature and occurs through short wires.

In the recent past, several multithreading proposals have appeared in the literature. A few commercial processors have already started implementing some of these multithreading concepts in a single chip [24,34]. Although the underlying theme behind the different proposals is quite similar, the exact manner in which they perform multithreading is quite different. Each of the methodologies has different hardware and software requirements and trade-offs. The objective of this chapter is to present a common framework for studying different multiprocessing and multithreading techniques, and to discuss existing multithreaded processors and futuristic proposals in the light of this framework. The following are some of the questions that are specifically addressed in the common framework:

- Parallel programming model
- Nature of threads
- PE Interconnects
- Role of the compiler

Section 1.4.1 has highlighted the importance of multithreading and multiprocessing. The rest of this chapter is organized as follows. Section 1.4.2 presents a common framework for studying different multithreading and multiprocessing approaches, and highlights software issues that are important to consider while examining them. Section 1.4.3 presents a common framework for studying parallel processor hardware configurations. Section 1.4.4 provides a survey of existing multithreaded processors and proposals. In particular, it describes how multithreading is employed in the multiscalar processor, the superthreaded processor, the trace processor, the M-machine, and some of the other multithreaded microarchitectures. Finally, it presents a qualitative comparison and discusses future trends.

## 1.4.2 Parallel Processing Software Framework

In this section we discuss our framework for studying multithreading and multiprocessing. We also identify three key issues related to multithreading: thread granularity, parallel programming model, and program partitioning into threads. We shall discuss each of these issues in detail. Not all of these issues are entirely orthogonal to each other, and it is our objective to highlight how each issue bears on other related issues.

*In this section, we use the terms *multithreading, multiprocessing*, and *parallel processing* interchangeably. Similarly, we use the generic term *threads* whenever the context is applicable to processes, light-weight processes, and light-weight threads.

We define a *thread* as a flow of control through a program and that flow's current state (represented by a current program counter, a call/return stack and, occasionally, some thread-private data). The central idea behind multithreading and multiprocessing is to have multiple flows of control within a process, allowing parts of the process to be executed in parallel. A process can have one or more threads doing its work. Threads that execute in parallel are invariably control-independent, in which case the decision to execute a thread does not depend on the other active threads. Thus, instructions that are control-dependent on a conditional branch invariably belong to the thread to which that branch belongs.

#### 1.4.2.1 Parallel Programming Model

An important attribute of any multiprocessing/multithreading system is its parallel programming model, embodied in a parallel language or programming environment. This model specifies the names (such as registers and memory addresses) the thread can access, the operations it can perform on the named data, and the ordering semantics among these operations, particularly those done by distinct threads. (In the simplest case, the model assumes multiprogramming, which has no inter-thread communication and synchronization.) First, we will discuss thread sequencing model, which specifies ordering constraints (if any) on multiple threads. Then, we discuss inter-thread communication, which deals with passing data values among two or more threads. Finally, we discuss synchronization aspects of the programming model, which cause running threads to wait for one another, and waiting threads to resume execution at the proper time. Orchestrating the inter-thread ordering often requires explicit synchronization operations when the ordering implicit in the basic operations is not sufficient.

##### *1.4.2.1.1 Thread Granularity and Management*

Thread-level parallelism (TLP) is more coarse-grained than ILP, and has wide variance in granularity. We categorize the TLP granularities into three levels as described below. Depending on the granularity, thread management (including run-time thread scheduling) is done by the operating system or the runtime hardware.

- *Processes*: In this case, a thread is a process itself. Parallel processing then involves executing multiple processes in parallel, which is traditionally known as *multitasking* or *multiprogramming*. This is perhaps the most common form of parallel processing, as even most of the uniprocessor operating systems implement this (by time sharing). Multiple processes can be created using the fork system call. Processes can be thought of as heavy-weight threads, as their creation entails duplicating the memory address space, and can take hundreds of thousands of CPU clock cycles. Management and scheduling of processes is done by the operating system. In a multiprogramming environment, parallelly executing processes either do not communicate, or communicate through operating system features such as pipes.
- *Light-weight processes or threads*: A light-weight process (also called *thread*) has a granularity somewhat finer than a process. The concept of light-weighted processes has been implemented in a number of operating systems (SUN Solaris, IBM AIX, and Microsoft Windows NT), thread libraries, and parallel programming languages. Such threads are used in today's symmetric multiprocessor workstations and servers. An important characteristic is that these threads share a common memory address space, and are nonspeculative from the control point of view.
- *Fine-grain threads*: These threads are much smaller (of the order a few hundred instructions, at most) and are not generally known to the operating system. Thread management and scheduling are typically done by the run-time hardware. In many cases, such threads share a common register space, besides sharing a common memory address space. Furthermore, the threads are often speculative from the control point of view.

For a particular TLP granularity, the system performance will depend to a large extent on the nature of the application and the level of the memory hierarchy at which the PEs are interconnected.

#### *1.4.2.1.2 Thread Sequencing Model*

The commonly used model for control flow among threads is the *parallel threads* model (also called the *control operators based parallel control flow* model). In this model, a *fork* instruction or a variant specifies the creation of new threads and their starting addresses. The parent thread as well as the forked threads are allowed to execute in parallel until they reach a *join* instruction, after which only one of them can continue. Thus, the join operation serves as a synchronizing point. Apart from the join, other explicit synchronization operations can be introduced using *locks* and *barriers*. Computation inside each thread is based on sequential control flow. This thread sequencing model is illustrated in Fig. 1.12.

Compilers and programmers have made significant progress in parallelizing regular numeric applications for the parallel threads model; however, they have had little or no success in doing the same for highly irregular numeric or especially non-numeric applications [18]. In such applications memory addresses are difficult (if not impossible) to statically predict—in part because they often depend on run-time inputs and behavior—that makes it extremely difficult for the compiler to statically prove whether or not potential threads are independent. Given the size and complexity of real non-numeric programs, parallelization appears to be an unrealistic goal if we stick to the parallel threads model. For such applications, we can use a different thread control flow model called *sequential threads* model. This model is closer to sequential control flow, and envisions a strict sequential ordering among the threads. That is, threads are extracted from sequential code and run in parallel, without violating the sequential program semantics. The control flow of the sequential code imposes an order on the threads and, therefore, we can use the terms *predecessor* and *successor* to qualify the relation between any given pair of threads. This means that inter-thread communication between any two threads (if any) is strictly in one direction, as dictated by the sequential thread ordering. Thus, no explicit synchronization operations are necessary, as the sequential semantics of the threads guarantee proper synchronization. This relaxation allows us to "parallelize" non-numeric applications into threads without explicit synchronization, even if there is a potential inter-thread data dependence. Program correctness will not be violated if at run time there is a true data dependence between two threads. The purpose of identifying threads in such a model is to indicate that those threads are good candidates for parallel execution.

Examples for multithreading proposals using sequential threads are the multiscalar model [8,9,30], the superthreading model [35], the trace processing model [28,36], and the dynamic multithreading model [1]. When using the sequential threads model, we can have threads that are *nonspeculative* from the control point of view, as well as threads that are *speculative* from the control point of view. The latter model is often called *speculative multithreading* (*SpMT*). This model is particularly important to deal with the complex control flow present in typical non-numeric programs. The multiscalar architecture [8,9,30] provided a complete design and evaluation of an SpMT architecture. Since then, many other proposals have extended the basic idea of SpMT [5,19,22,28,31,35,36]. One such extension is threaded

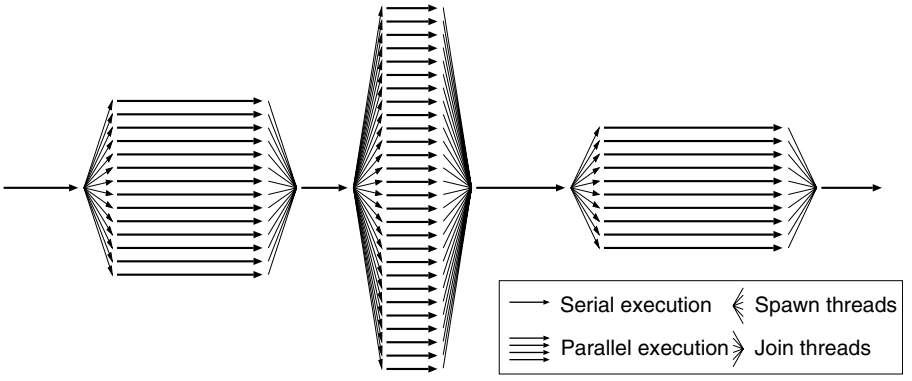



**FIGURE 1.12** Parallelism profile for a parallel threads model.

multipath execution (TME) [38], in which the speculative threads are the alternate paths of hard-to-predict branches. A simple form of the SpMT model uses loop-based threads only [15,22].

#### *1.4.2.1.3 Inter-Thread Communication*

Inter-thread communication refers to passing data values between two or more threads. One of the key issues in a parallel programming model is the name levels at which sharing takes place between threads. Communication can take place at the level of register space, memory address space, and I/O space, with the registers being the level closest to the processor. If sharing can happen at a particular level, it can also happen at a more distant level. Parallel programming models can be classified into three categories, based on the sharing level that is closest to the processor:

- Shared register model
- Shared memory model
- Message passing model

In the *shared register model*, multiple threads share the same register space (or a portion of it). Interthread communication happens implicitly due to reads and writes to the shared registers (and to shared memory locations). This model typically uses fine-grain threads, because it is difficult to have long threads that communicate at the low level of registers, granularity is small. This class of parallel processors is fairly new and has evolved as an extension of single-threaded ILP processors. Examples are the multiscalar execution model [8,9,30], the trace execution model [28,36], and the dynamic multithreading model (DMT) [1].

In the *shared memory model*, multiple threads share a common memory address space (or a portion of it). Inter-thread communication occurs implicitly as a result of conventional memory access instructions to shared memory locations. That is, writes to a logically shared address by one thread are visible to reads of the other threads, provided there are no other prior writes to that address as per the memory consistency/synchronization model.

In the *message passing model*, inter-thread communication occurs only through explicit I/O operations called messages. That is, the inter-thread communication is integrated at the I/O level rather than at the memory level. The messages are of two kinds—*send* and *receive*—and their variants. The combination of a send and a matching receive accomplishes a pairwise synchronization event. Several variants of the above synchronization event are possible. Message passing has long been used as a means of communication and synchronization among cooperating processes. Operating system functions such as *sockets* serve precisely this function.

#### *1.4.2.1.4 Inter-Thread Synchronization*

Synchronization involves coordinating the results of a set of parallel threads into some merged result. An example is waiting for one thread to finish filling a buffer before another begins using the data. Synchronization is achieved in different ways:

- *Control Synchronization*: Control synchronization depends only on the threads' control state and is not affected by the threads' data state. This synchronization method requires a thread to wait until other thread(s) reach a particular control point. Examples for control synchronization operations are *barriers* and *critical sections*. With barrier synchronization, all parallel threads have a common barrier point. Each thread is allowed to proceed after the barrier only after all of the spawned threads have reached the barrier point. This type of synchronization is typically used when the results generated by the spawned threads need to be merged. With critical section type synchronization, only one thread is allowed to enter into the critical section code at a time. Thus, when a thread reaches a critical section, it will wait if another thread is currently executing the same critical section code.
- *Data Synchronization*: Data synchronization depends on the threads' data values. This synchronization method requires a thread to wait at a point until a shared name is updated with a particular value (by another thread). For instance, a thread executing a wait ($x = 0$) statement

will be delayed until x becomes zero. Data synchronization operations are typically used to implement *locks, monitors*, and *events*, which, in turn, can be used to implement *atomic operations* and *critical sections*. When a thread executes a sequence of operations as an atomic operation, other threads cannot access any of the (shared) names updated during the atomic operation until the atomic operation has been completed.

#### 1.4.2.2 Coherence and Consistency

The last aspect that we will consider about the parallel programming model is coherence and consistency when threads share a name space. Coherence specifies that the value obtained by a read to a shared location should be the latest value written to that location. Notice that when a read and a write are present in two parallel threads, coherence does not specify any ordering between them. It merely states that if one thread sees an updated value at a particular time, all other threads must also see the updated value from that time onward (until another update happens to the same location).

The consistency model determines the time at which a written value will be made visible to other threads. It specifies constraints on the order in which operations to the shared space must appear to be performed (i.e., become visible to other threads) with respect to one another. This includes operations to the same locations or to different locations, and by the same thread or different threads. Thus, every transaction (or parallel transactions) transfers a collection of threads from one consistent state to another. Exactly what is consistent depends on the consistency model. Several consistency models have been proposed:

- *Sequential Consistency*: This is the most intuitive consistency model. As per sequential consistency, the reads and writes to a shared address space from all threads must appear to execute serially in such a manner as to conform to the program orders in individual threads. This implies that the overall order of memory accesses must preserve the order in each thread, regardless of how instructions from different threads are interleaved. A multiprocessor system is *sequentially consistent* if it always produces results that are same as what could be obtained when the operations of all threads are executed in some sequential order [20]. Sequential consistency is very restrictive and prevents the multiprocessor hardware from performing many optimizations to improve performance.
- *Weak Consistency*: This consistency model [6] relaxes the constraints imposed by sequential consistency by relating memory access order to synchronization points in the program. That is, sequential consistency is maintained among the synchronization accesses. In addition, a synchronization access serves as a barrier by enforcing that all previous memory accesses must be completed before performing a synchronization access, and no subsequent memory accesses can be performed before completing a synchronization access.

In addition to weak consistency, several other relaxed consistency models have been proposed—*release consistency* [12], *processor consistency* [13], etc.

#### 1.4.2.3 Partitioning a Program into Threads

Thread selection involves partitioning a control flow graph (CFG) into threads. Given a particular parallel programming model (inter-thread communication model as well as thread sequencing model), how should the parallelizer go about deciding where the thread boundaries should be? Perhaps the most important issue in multiprocessing/multithreading is the basis used for partitioning a program into threads. The criterion used for partitioning is very important, because an improper partitioning could in fact result in high inter-thread communication and synchronization, thereby degrading performance! True multithreading should not only aim to distribute instructions evenly among the threads, but also aim to minimize inter-thread communication by localizing a major share of the inter-instruction communication occurring in the processor to within each PE. In order to achieve this, mutually data dependent instructions are most likely allocated to the same thread. This is somewhat hard,

because programs are currently written in control-driven form, which often causes individual strands of data-dependent instructions to be spread over a large segment of code. Thus, the partitioning software has to first construct the data flow graph (DFG), and then do the program partitioning. Notice that if programs were specified in data-driven form as in the dataflow computation model [17], taking data dependences into account would have been simpler.

Thread selection is a difficult problem, because we need to consider many issues such as PE utilization, load balancing, control independence of threads (thread prediction accuracy for SpMT models), and inter-thread data dependences. Often, trying to make optimizations for one area will have a negative effect on another.

#### *1.4.2.3.1 Who Does the Program Partitioning?*

Program partitioning can be done by the programmer, compiler, or run-time hardware. Depending on who does the partitioning, the type of analysis that can be done will be different.

- *Programmer*: In this approach, the programmer explicitly represents the threads in the high-level language program. In order to do this, three types of extensions are provided at the high-level language level: (i) multithreading library, (ii) language extensions, and (iii) compiler directives. Examples for this approach are EARTH [21] and XMT [37]. All of these use the parallel threads model. Notice that the compiler has to be modified to handle these extensions. The compiler does not, however, make decisions on where to do the partitioning. It is interesting to note that although conventional multiprocessors have been commercially available for quite some time, only a small fraction of the software has been written so far to exploit parallelism.
- *Compiler*: In this case, the compiler takes a sequential program and partitions it into threads. The main advantage of deferring program partitioning to the compiler is that it frees the programmer from reasoning about parallel threads. Its main advantages with respect to hardware-based partitioning are that it does not add to the complexity of the processor, and that it has the ability to perform complex pre-partitioning and post-partitioning optimizations that are difficult to perform at run-time. Compiler-directed partitioning algorithms are generally insensitive to the number of PEs in the processor, however, its partitioning decisions need to be conveyed to the multithreaded hardware, possibly by making it part of the ISA (at the expense of incompatibility for existing binaries). Parallelizing compilers have been successful in parallelizing many numeric applications for the parallel threads model. As pointed out earlier, their success has not been spectacular when it comes to non-numeric applications and the parallel threads model. Several researchers are currently working on parallelizing compilers that parallelize such applications for the sequential threads model.
- *Hardware*: It is also possible to let the run-time hardware do the program partitioning. If partitioning decisions are taken by the hardware, the multithreaded processor provides object code compatibility to existing sequential code. Furthermore, it has the ability to adapt to run-time behavior. Hardware-based partitioning is typically done only if the thread granularity is small, and if sequential control flow is used. The main limitation is the significant impact it may have on clock cycle time. In order to simplify the dynamic partitioning hardware and to reduce the impact on clock cycle time, the partitioning job is often split into two parts—a static part (which is done by pre-processing hardware) and a dynamic part. The static part collects information that is static in nature (such as register dependences in a straightline piece of code) and stores it in a special i-cache structure, often after performing some additional processing. The dynamic part uses this information while deciding the final partitioning at run-time. Examples of multithreaded processors that use hardware-based partitioning are trace processor [28,36], speculative multithreading processor [22], and dynamic multithreading processor [1].

#### *1.4.2.3.2 Compiling for Multithreading*

Most of the multithreading approaches perform partitioning at compile time, possibly with some help from the programmer; it is somewhat unrealistic at this time to expect programmers to write only

parallel programs. The hardware is also limited in its program partitioning capability. Therefore, the compiler has the potential to play a significant role in multithreading. Besides program partitioning, it can schedule threads as well as the instructions within threads.

The task of the compiler is to identify sufficient parallelism to keep the processors busy, while minimizing the effects of synchronization and communication latencies on the execution time of the program. To accomplish this objective, a parallelizing compiler typically performs the following functions:

1. Identify the parallelism inherent in the program. This phase has received the most attention in parallel compiler research to date [25,26]. Many varied program transformations have been developed to unearth parallelism buried in the semantics of sequential programs.
2. Partition the program into multiple threads for parallel execution. This is perhaps the most crucial phase. Many factors must be considered, such as inter-thread dependences, intra-thread locality, thread size, critical path, and deadlock avoidance.
3. Schedule the concurrent execution of threads; the final scheduling is often determined by the run-time environment. The compiler must assign threads to processors in a way that maximizes processor utilization without severely restricting the amount of parallelism to be exploited.
4. After program partitioning, the compiler can schedule the instructions in a thread so as to reduce inter-thread wait times. For instance, if a shared value is produced very late in one thread, but is needed very early in another thread, very little parallelism will be exploited by the hardware. This problem is likely to surface frequently, if the compiler assumed a single-threaded processor in the code generation phase. In such situations, post-partitioning scheduling can help minimize the waiting time of instructions by ensuring that shared values required in other threads are produced as early as possible. Post-partitioning scheduling is especially beneficial if PEs execute their instructions in strict serial order.

#### *1.4.2.3.3 Object Code Compatibility*

Another important issue, especially from the commercial point of view, is the level of compatibility that the multithreaded processor provides. We can think of three levels of compatibility in the context of multithreaded processors: full compatibility, family-wide compatibility, and no compatibility.

- *Full Compatibility*: In some multithreaded processors, the multithreading aspect is strictly a microarchitectural phenemenon and is invisible at the ISA level. Such processors provide full compatibility, i.e., both backward compatibility and forward compatibility. Existing executable binaries can be run on them, and their executable binaries can be run on existing processors. Furthermore, these processors also provide compatibility across all multithreading models (of the same ISA) that provide full compatibility. In these processors, the thread partitioning is done by offline hardware or run-time hardware. Fully compatible multithreaded processors have a higher chance for commercial success.
- *Family-Wide Compatibility*: Although full compatibility is desirable, some multithreaded processors opt for ISA-level changes so as to benefit from compiler techniques to extract additional performance. Processors in the family-wide compatibility category provide compatibility within its multithreading family. Thus, they do not require recompilation to be performed when the number of PEs is changed. Generally, these processors also provide limited backward compatibility (albeit at reduced performance). For example, if an existing binary executable is given to the multiscalar processor, it can execute the entire program as a single task. This will not give good performance, but it can run the old binaries.
- *No Compatibility*: In spite of the benefits of object code compatibility, some multithreaded processors, such as the M-machine, go in for significant changes at the ISA-level, which preclude any possibility of backward compatibility or family-wide compatibility. The motivation is to tap into sophisticated compiler techniques to extract performance.

Several techniques for binary translation have been proposed recently to address the object code compatibility problem. These include static approach as in the FX!32 [16], dynamic approach as in the DAISY [7] and hardware-based schemes such as DIF [23]. Object code compatibility may become a less important issue in the future when these techniques become more mature and efficient. This will also open up more opportunities to tap architecture specific optimizations for multithreading in the future.

## 1.4.3 Parallel Processing Hardware Framework

The previous section discussed a common framework for parallel programming and compilation. This section discusses a common framework for parallel processing hardware. In our hardware framework, regardless of the specific implementation, a multithreaded processor consists of multiple PEs, possibly along with a few centralized resources such as the thread allocation mechanism and parts of the memory subsystem, as shown in Fig. 1.13. The PEs work independently of each other (subject only to inter-PE synchronization) and usually contain multiple execution units (EUs). The PEs are interconnected by some network or through centralized resources such as register file and memory, for inter-PE communication.

Our definition of a PE is somewhat loose. On one extreme, the PEs in some multithreaded processors are separate processor-memory systems with their own instruction cache, decode unit, register file, and execution units; on the other extreme, the PEs in some multithreaded processors even share the execution units, as in the dynamic multithreading processor [1]. Such a loose definition allows us to discuss a wide spectrum of multithreaded processors under a common framework.

### 1.4.3.1 Number of PEs and PE Organization

The number of PEs in a multiprocessor is an important hardware parameter. This number is strongly tied to the perceived parallelism in the targeted application domain, and also the nature of the threads. On one extreme, we have single-PE multithreaded processors that perform time sharing. On the other extreme, we have massively parallel processors (MPPs) consisting of thousands of PEs, which are the most powerful machines available today for many time-critical applications [4]. Because of the sharp increase in the number of transistors integrated in a single chip, there is significant interest in integrating multiple PEs in the same chip. This has been the motivation behind many of the SpMT processing models.

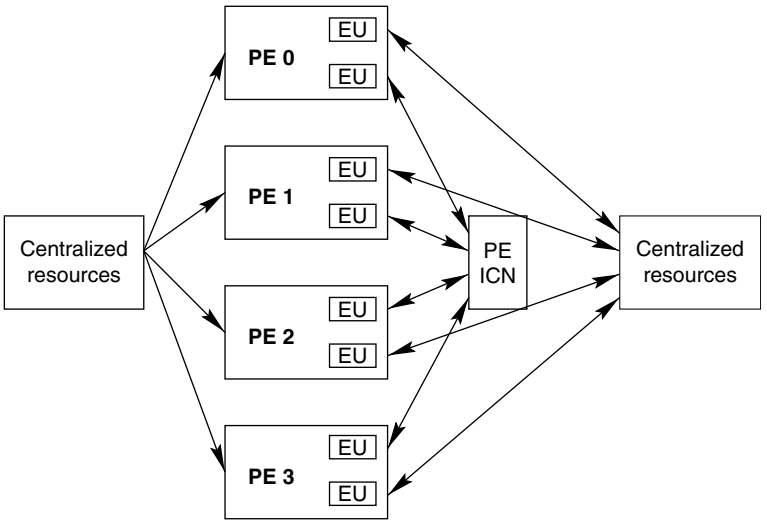



**FIGURE 1.13** A generic 4-PE multiprocessor.

#### *1.4.3.1.1 Processor Context Interleaving*

When the number of parallel threads exceeds the number of PEs, it is possible to time-share a single PE among multiple threads in a way that minimizes the time required to switch threads. This is accomplished by sharing as much as possible of the program execution environment between the different threads so that very little state needs to be saved and restored when changing threads. This type of low-overhead interleaving is given the name *multithreading* in many circles [2,3,17]. Interleaving-based multithreading differs from conventional multitasking (or multiprogramming) in that the concurrent threads share more of their environment with each other than do concurrent tasks under multitasking. Threads may be distinguished only by the value of their program counters and stack pointers while sharing a single address space and set of global variables. As a result, there is very little protection of one thread from another, in contrast to multitasking. Interleaving-based multithreading can thus be used for very fine-grain multitasking, at the level of a few instructions, and so can hide latency by keeping the processor busy after one thread issues a long-latency instruction on which subsequent instructions in that thread depend.

- *Cycle-level interleaving*: In this scheme, a PE switches to a different thread after each instruction fetch; i.e., an instruction of another thread is fetched and fed into the execution pipeline in the next clock cycle. Cycle-level interleaving is typically used for coarse-grain threads—processes or light-weight processes. The motivation for this is that it eliminates control and data dependences between the instructions that are simultaneously active in the pipeline. Thus, there is no need to build complex forwarding paths, permitting a simple and potentially fast pipeline. Furthermore, the context switch latency is zero cycles. Memory latency is tolerated by not scheduling a thread until the memory access has been completed. For this interleaving to work well, there must be as many threads as the worst-case latencies experienced by the instructions. Interleaving the instructions from many threads limits the processing speed of a single thread, thereby degrading single-thread performance. The most well-known examples of cycle-level interleaving processors are HEP [29], Horizon [33], and Tera MTA [2].
- *Block interleaving*: In this scheme, the instructions of a thread are executed successively until a long-latency event occurs, which causes a context switch. A typical long-latency operation is a remote memory access. Compared to the cycle-level interleaving technique, a smaller number of threads is sufficient, and a single thread can execute at full speed until the next context switch. The events that cause a context switch can be determined statically or dynamically.

When hardware technology allows more PEs to be integrated in a processor, PE interleaving becomes less attractive, because computational throughput will clearly improve when multiple threads execute in parallel on multiple PEs instead of time-sharing a single PE. As we look into the future, and the prospect of a billion transistors on a single chip, it seems inevitable that microprocessors will have multiple PEs.

#### *1.4.3.1.2 PE Organization*

The next issue of importance in a multithreaded processor is the organization of the PEs. This issue is strongly tied to the PE interconnect used. Most of the sequential threads model based processors organize the PEs as a circular queue, as shown in Fig. 1.14. The circular queue imposes a sequential order among the PEs, with the head pointer indicating the oldest active PE. When the tail PE is idle, a thread allocation unit (TAU) invokes the next thread (as per the sequential thread ordering) on the tail PE and advances the tail pointer. Completed threads are retired from the head of the PE queue, enforcing the required sequential ordering. Although this PE organization is tailored for sequential threads (from a sequential program), this multithreaded hardware can also execute multiple threads from different processes, if required.

An important issue that needs to be considered when organizing the PEs as a circular queue is *load balancing*. If some PEs have long threads assigned to them, and the rest have short ones, only modest performance will be obtained. If threads are not close to the same size, a short thread may complete soon and perform no useful computation while it waits for longer predecessor threads to retire. To get good

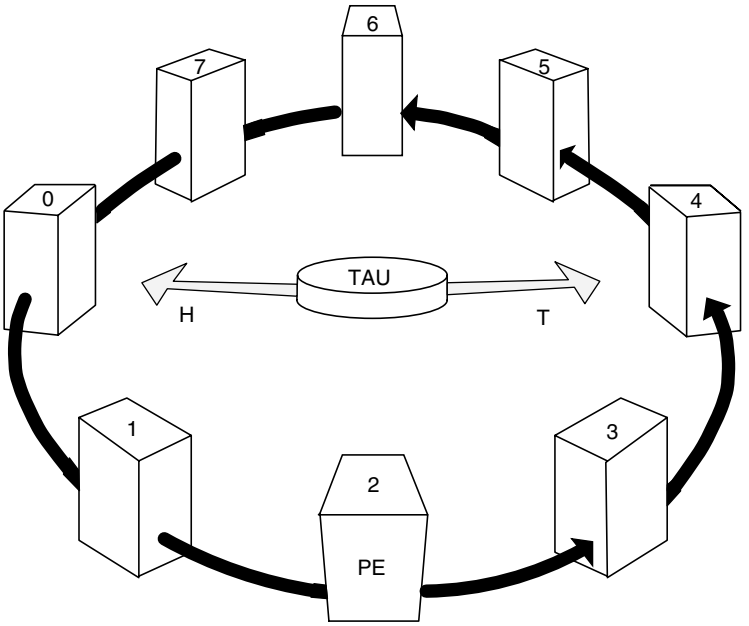



**FIGURE 1.14** Organizing the PEs of a multithreaded processor as a circular queue.

performance, threads should be of uniform length.* One option to deal with load balancing, albeit with additional hardware complexity, is to let each physical PE have multiple virtual PEs and assign a thread to each of the virtual PEs.

### 1.4.3.2 Inter-PE Register Communication and Synchronization

As discussed earlier, a few multithreading approaches have a shared register space for all threads, and the rest do not. When threads share a common register space, the thread sequencing model has always been the sequential threads model. Because the semantics of this model are in line with sequential control flow, synchronization happens automatically, once inter-PE register communication is handled properly.

#### *1.4.3.2.1 Register File Implementation*

When threads do not share a common register space, it is straightforward to implement the register file (RF)—each PE can have its own register file, thereby providing fast register access. When threads share a common register space, it is important that we still provide a separate register file in each PE to support fast register access, as it is difficult for a centralized register file to provide a 1-cycle multi-port access time with today's high clock rates. This decentralization can be achieved in two ways, both of which provide faster register access times due to physical proximity and fewer access ports per physical register file.

- *RF Partitioning*: In this approach, each physical register file implements (or maps) an independent set of ISA-visible registers. Notice that a PE may occasionally need a register value stored in a nonlocal register file, in which case the value is fetched through an interconnection network that interconnects the PEs.
- *RF Replication*: With the replication scheme, a physical copy of the register file is kept in each PE, so that each PE has a local copy of the shared set register space. These register file replica maintain

*The actual, more stringent, requirement is that the thread execution times should be matched across all PEs. This is a more difficult problem, because it depends on intra- and inter-PE data dependences as well.

different *versions* of the register space, i.e., the multiple copies of the register file store register values that correspond to the processor state at different points in a sequential execution of the program. In general, replication avoids unnecessary communication; however, if not done carefully, it might increase communication by replicating data that is not used in the future. A multithreaded processor that uses the replication scheme is the multiscalar processor [9].

#### *1.4.3.2.2 PE Interconnect for Register Values*

When threads share a common register space, and a distributed RF structure is used, an important hardware attribute is the type of interconnect used to send register values from one PE to another. The interconnects that have been proposed in the context of multithreaded processors are bus, ring (unidirectional and bi-directional), crossbar, mesh, and hypercube; of course, it is possible to use other types of interconnects as well.

Bus: The bus is a simple, fully connected network. However, it permits only one data transmission at any time, providing a bandwidth of only O(1). In fact, the bandwidth scaling is worse than O(1) because of reduction in bus operating speed with the number of ports, due to increase in capacitance. Therefore, it may be a poor choice as an interconnect for inter-PE register communication, which may be nontrivial, especially when using a large number of PEs.

Crossbar: A crossbar interconnect also provides full connectivity from every PE to every other PE. It provides O($N$) bandwidth, but the cost of the interconnect is proportional to the number of cross-points, or O($N^2$). When using a crossbar, all PEs are of same proximity to each other; hence the thread allocation algorithm becomes straightforward; however, a crossbar may not scale as easily as a ring or mesh. It is important to note that fast crossbars can be built on a single chip. With a crossbar-type interconnect, there is no notion of neighboring PEs, so all PEs become equally far away. Therefore, the cross-chip wire delays begin to dominate the inter-PE communication latency.

Ring: With a ring-type interconnect, the PEs are connected as a circular loop, and there is a notion of neighboring PEs and distant PEs. Routing in a ring is trivial because there is exactly one route between any pair of nodes (two routes if it is a bi-directional ring). The ring can be easily laid out with O($N$) space using short wires (as depicted in Fig. 1.14), which can be easily widened. A ring is ideal if most of the inter-PE register communication can be localized to neighboring PEs (which is typically the case in a sequential threads processor that uses the circular queue PE organization [36]), but is a poor choice if a lot of communication happens across distant PEs. An advantage of the ring is that it easily supports the scaling up of the number of PEs, as allowed by technological advances.

Mesh: Rings generalize naturally to higher dimensions, including 2D grids and 3D cubes (with end-around connections). The main advantages of mesh are its regular structure and its ability to provide full connectivity between four neighboring PEs (as opposed to two PEs with the ring). Similar to a ring, a mesh can easily support the scaling up of the number of PEs. The mesh suffers from the same disadvantages of a ring in communicating with distant PEs. Moreover, thread allocation for a mesh topology is more complex than that for ring and crossbar.

### 1.4.3.3 Inter-PE Memory Communication and Synchronization

When threads do not share a common memory address space (as in the message passing model), it is straightforward to provide a memory system for each PE, as we do not need to worry about inter-thread memory communication and synchronization.

#### *1.4.3.3.1 Memory System Implementation*

When threads do share a common memory address space, the multithreaded processor needs to provide appropriate mechanisms for inter-thread memory communication as well as synchronization. One option is to provide a central memory system, in which all memory accesses roughly take the same amount of time. Such a system is called *uniform memory access* (*UMA*) system. An important class of UMA systems is the *symmetric multiprocessor* (*SMP*).

A UMA system may provide uniformly slow access time for every memory access. Instead of slowing down every access, we can provide fast access time for most of the accesses by distributing the memory system (or at least the top portions of the memory hierarchy system). Shared memory multiprocessors that use partitioning are called *distributed shared memory* (*DSM*) systems. As with the register file structure, we can use two techniques—partitioning and replication—to distribute the memory.

- *Memory Partitioning*: Partitioning is useful if it is possible to confine most of the memory accesses made in one PE to its partition. Partitioning the top portion of the memory hierarchy may not be attractive, at least for irregular, non-numeric applications, because it may be difficult to do this confinement due to not knowing the addresses of most of the loads and stores at compile time. Partitioning of the lower portion of the memory hierarchy is often done, however, as this portion needs to handle only those accesses that missed in the PEs' local caches.
- *Memory Replication*: It is impractical to replicate the entire memory system. Therefore, only the top part of the memory hierarchy is replicated. The basic motivation behind replicating the top portion of the memory hierarchy among local caches is to satisfy most of the memory accesses made in a PE with its local cache. Notice that a replicated cache structure must maintain proper coherency among all the duplicate copies of data.

DSMs often use a combination of partitioning and replication, i.e., portions of the memory hierarchy are replicated and the rest are partitioned. One type uses replicated cache memories and partitioned main memories. One interesting variation is the *cache only memory architecture* (*COMA*) system. A COMA multiprocessor partitions the entire memory system across the PEs; however, there is no fixed partition assigned for a particular memory location. Rather, the partition associated with a memory location is dynamically changed based on the PEs that access that location. Several other shared memory organizations are also possible [3,17].

#### *1.4.3.3.2 Inter-PE Data Dependence Speculation*

In the parallel threads model, synchronization of threads is carried out with the use of special mechanisms such as locks and barriers. In the sequential threads model, ensuring sequential semantics ensures proper memory synchronization. However, this means that when a load instruction is encountered in a PE, it has to ensure that its producer store has been already executed. This is difficult to determine if the producer store belongs to another thread, as memory addresses are calculated at run-time, and it is possible that the producer store instruction may not have even been fetched. In order to overcome this problem, sequential threads based processors incorporate some form of *thread-level data speculation* [11]. The idea is to speculate if a memory operation has to wait for inter-thread synchronization. This speculation can be as simple as predicting that the producer store has been already executed, or it can be more complex, based on past behavior of the load instruction. Below we discuss some of the hardware schemes proposed for carrying out thread-level data speculation.

- *Address Resolution Buffer* (*ARB*): The ARB [11] is a hardware buffer for storing different versions of several memory locations as well as information regarding the loads and stores executed from the currently active threads. Each entry in the ARB buffers all versions of the same memory location. When a load request is issued for a particular memory address, the corresponding ARB entry is checked to see if a prior store has been done to the same address; if so, the value written by the latest store is returned by the ARB; if not, the request is sent to the next lower level of the memory hierarchy. In either case, the state information for that location is updated to reflect the fact that a load has been made by the current thread. When a store operation is performed, the ARB checks if any sequentially successor loads have been prematurely performed. If so, that is an incorrect data dependence speculation, and the ARB hardware initiates a recovery action such as partially re-executing the thread containing the incorrect load (and subsequent threads). A centralized hardware approach such as the ARB has the danger of increasing the load latency due to long latency incurred because of long wires.

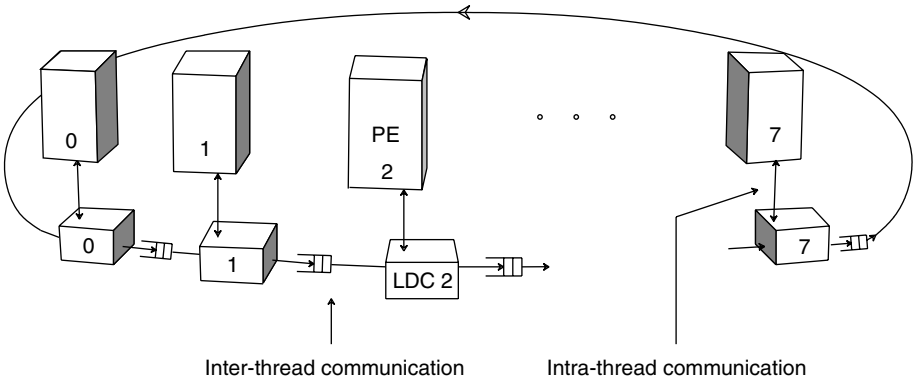



**FIGURE 1.15** Block diagram of a multi-version cache in a sequential thread based multithreaded processor.

- *Multi-Version Cache* (*MVC*): The MVC uses a decentralized approach by using a local data cache (LDC) for each PE [10]. Each LDC thus stores a different version for each mapped memory location. The local data caches are interconnected by a unidirectional ring, as shown in Fig. 1.15. The loads and stores generated in a PE are serviced directly from its local data cache. When a load request is issued to a local data cache, it provides a value if it has a copy; otherwise, the request is sent to the next lower level of the memory hierarchy. In either case, the state information for that location in the data cache is updated to reflect the fact that a load has been made by the current thread. When a store operation is performed, the value is written in its local data cache. The last updates to each memory location (in a thread) are forwarded to the subsequent LDCs through the ring-type interconnect. When a forwarded value reaches an LDC, it checks for incorrect speculations and takes appropriate recovery actions.
- *Speculative Versioning Cache* (*SVC*): The speculative versioning cache is similar to the multi-version cache in many respects [14]. It also keeps a separate private cache for each PE. The differences are mainly in the way the caches are connected and in the methodology by which the caches are kept coherent. SVC uses a bus interconnect for the caches a snooping bus based cache coherence protocol.

### 1.4.4 Concluding Remarks

Multithreaded processors are the future of computer design. The ease of hardware replication has proven to be an ever-increasing impetus toward parallel processor implementations. The goal is to maintain high levels of parallelism (without increasing hardware complexity and the clock rate) by distributing the dynamic instruction stream among several processing elements. The combined issue rates of several processing elements allow large amounts of parallelism to be exploited.

Multithreading and multiprocessing, as with other complex engineering problems, undergo an ongoing process of reinventing, borrowing, and adapting.

Looking forward to the future of multithreaded processors, the pace of change makes for rich opportunities and also for great challenges. Although it is difficult to precisely predict where this field will go, this final section seeks to outline some of the key areas of development in multithreaded processors. Whatever technological breakthroughs occur and whatever directions the market takes, the fundamental issues addressed in Section 1.4.2 will still apply. The realization of multithreaded processors will still rest upon good techniques to perform thread selection, inter-PE communication, and synchronization. The core techniques for addressing these issues will remain valid; however, the way that they are employed will surely change as the critical parameters of clock speeds and wire delays continue to change.

It is difficult to obtain good performance without having complexity somewhere in the hardware-software multithreaded system! In a high-performance multithreaded processor, the complexity could be

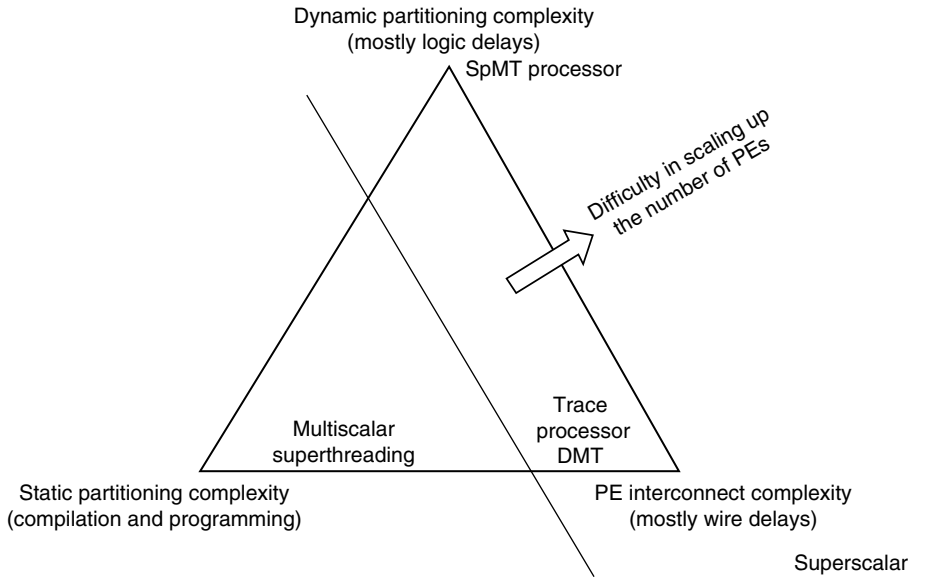



**FIGURE 1.16** Complexity in multithreading/multiprocessing.

at the static partitioning side (programming or compiler), at the dynamic partitioning hardware side, or at the PE interconnect side. Figure 1.16 illustrates this concept. To support hardware scalability, complexity at the dynamic partitioning hardware and the PE interconnect act as hurdles. In the long run, as more transistors are integrated into a processor chip, it can be expected that the number of PEs would be scaled up. However, the trend towards higher clock rates will make it more difficult to support complexity in the dynamic partitioning hardware and in the PE interconnect.* Thus, the end result of the trends in high transistor count and high clock rates (which encourage multithreading/multiprocessing) is a shift towards doing more and more things statically, as opposed to dynamically. This means that program partitioning will eventually be done only at compilation time, and perhaps more and more at programming time.

## To Probe Further

Multiprocessing has been around for a long time, and so naturally the computer literature has an overabundance of articles and textbooks on this subject. The multiprocessing community consists of different camps, which often use different terminology for the same concepts. This lack of consensus makes it somewhat difficult to merge the ideas presented in different papers or books. Nevertheless, we list a few helpful references to which interested readers can refer. Two recent good textbooks on this subject are *Parallel Computer Architecture: A Hardware/Software Approach* [3] and *Scalable Parallel Computing* [17]. Two important journals dealing with parallel processing are *Journal of Parallel and Distributed Computing* and *IEEE Transactions on Parallel and Distributed Systems*. In addition, readers can keep abreast of the latest research developments by reading the yearly proceedings of *International Conference on Parallel Processing, International Conference on Supercomputing*, and *Supercomputing*.

## Acknowledgments

The author thanks U.S. National Science Foundation (NSF grants MIP 9702569, CCR 9711566, and CCR 0073582) for supporting this work.

*Although it is possible to pipeline a crossbar interconnect so that it can accept new requests every cycle, the long inter-PE latency that it causes would increase the number of clock cycles required to execute a program, compared with what is obtained with scalable interconnects [27].

## References


1. H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor," *Proceedings of 31st International Symposium on Microarchitecture*, 1998.
2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and J.B. Smith, "The Tera Computer System," *Proceedings of International Conference on Supercomputing*, pp. 1–6, 1990.
3. D.E. Culler and J.P. Singh, Parallel Computer Architecture A Hardware/Software Approach. Morgan Kaufmann, 1999.
4. W.J. Dally and S. Lacy, "VLSI Architecture: Past, Present, and Future," *Proceedings of Advanced Research in VLSI Conference*, 1999.
5. P. Dubey, K. O'Brien, K.M. O'Brien, and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading," *Proceedings of International Conference on Parallel Architecture and Compilation Techniques* (*PACT'95*), 1995.
6. M. Dubois, C. Scheurich, and F.A. Briggs, "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 434–442, 1986.
7. K. Ebcioğlu and E.R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 26–37, 1997.
8. M. Franklin and G.S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism," *Proceedings of 19th International Symposium on Computer Architecture*, pp. 58–67, 1992.
9. M. Franklin, "The Multiscalar Architecture," Ph.D. Thesis, Technical Report TR 1196, Computer Sciences Department, University of Wisconsin, Madison, 1993.
10. M. Franklin, "Multi-Version Caches for Multiscalar Processors," *Proceedings of International Conference on High Performance Computing*, 1995.
11. M. Franklin and G.S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.
12. K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 15–25, 1990.
13. J.R. Goodman, "Cache Consistency and Sequential Consistency," Technical Report 61, IEEE SCI Committee, 1990.
14. S. Gopal, T.N. Vijaykumar, J.E. Smith, and G.S. Sohi, "Speculative Versioning Cache," *Proceedings of 4th International Symposium on High Performance Computer Architecture* (*HPCA-4*), 1998.
15. L. Hammond, B.A. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *IEEE Computer*, September 1997.
16. R. Hookway, "Running 32-bit ×86 Applications on Alpha NT," *Proceedings of IEEE COMPCON 97*, pp. 37–42, 1997.
17. K. Hwang and Z. Xu, *Scalable Parallel Computing*, WCB McGraw-Hill, New York, 1998.
18. R. Joy and K. Kennedy, *President's Information Technology Advisory Committee* (*PITAC*)—*Interim Report to the President*. National Coordination Office for Computing, Information and Communication, 4201 Wilson Blvd, Suite 690, Arlington, VA 22230, August 10, 1998.
19. V. Krishnan and J. Torellas, "A Chip Multiprocessor Architecture with Speculative Multithreading," *IEEE Transactions on Computers*, September 1999.
20. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, pp. 690–691, September 1979.
21. O.C. Maquelin, H.H.J. Hum, and G.R. Gao. "Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors," *Proceedings of 1st International EURO-PAR Conference*, 1995.
22. P. Marcuello, A. Gonzalez, and J. Tubella, "Speculative Multithreaded Processors," *Proceedings of International Conference on Supercomputing*, 1998.
23. R. Nair and M.E. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 13–25, 1997.

24. N. Nishi et al., "A 1-GIPS 1-W Single-Chip Tightly Coupled Four-Way Multiprocessor with Architecture Support for Multiple Control-Flow Execution," *Proceedings of the 47th International Solid-States Circuits Conference*, pp. 418–475, 2000.
25. D. Padua, "Polaris: An Optimizing Compiler for Parallel Workstations and Scalable Multiprocessors," Technical Report 1475, University of Illinois at Urbana-Champaign, Center for Supercomputing Research & Development, January 1996.
26. C. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung, and D.A. Schouten, "The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran," *Languages and Compilers for Parallel Computing*, MIT Press, Cambridge, MA, 1990.
27. N. Ranganathan and M. Franklin, "An Empirical Study of Decentralized ILP Execution Models," *Proceedings of 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-VIII*), pp. 272–281, 1998.
28. E. Rotenberg, Q. Jacobson, Y. Sazeides, and J.E. Smith, "Trace Processors," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 138–148, 1997.
29. B.J. Smith, "The Architecture of HEP," *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pp. 41–55, MIT Press, Cambridge, MA.
30. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, 1995.
31. J.G. Steffan and T.C. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," *Proceedings of 4th International Symposium on High Performance Computer Architecture*, 1998.
32. K.K. Sundararaman and M. Franklin, "Multiscalar Execution along a Single Flow of Control," *Proceedings of International Conference on Parallel Processing* (*ICPP*), pp. 106–113, 1997.
33. M. Thistle and B.J. Smith, "A Processor Architecture for Horizon," *Proceedings of Supercomputing '88*, pp. 35–41, 1988.
34. M. Tremblay et al, "The MAJC Architecture: A Synthesis of Parallelism and Scalability," *IEEE MICRO*, pp. 12–25, November/December 2000.
35. J.-Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques* (*PACT '96*), pp. 35–46, 1996.
36. S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 1–12, 1997.
37. U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman, "Explicit Multi-threaded (XMT) Bridging Models for Instruction Parallelism," *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures* (*SPAA*), pp. 140–151, 1998.
38. S. Wallace, B. Calder, and D. Tullsen, "Threaded Multiple Path Execution," *Proceedings of the 25th Annual International Symposium on Computer Architecture,* pp. 238–249, 1998.
39. "The National Technology Roadmap for Semiconductors," Semiconductor Industry Association, 1997.

## 1.5 Survey of Parallel Systems

*Donna Quammen*

### 1.5.1 Introduction

Computers have long been considered "a solution looking for a problem," but because of limits found by *complexity theory* and limits on computing power some problems that were presented could not be solved. Multimedia problems, image processing and recognition, AI application, and weather prediction may not

be accomplished unless processing power is increased. There are many varieties of parallel machines, each has the same goal, to complete a task quickly and inexpensively. Modern physics has continually increased the speed and capacity of the media on which modern computer chips are housed, usually VLSI, and at the same time decreased the price. The challenge of the computer engineers is to use the media effectively. Different components may be addressed to accomplish this, such as, but not limited to:

- Functionality of the processors—floating point, integer, or high level function, etc.
- Topology of the network which interconnects the processors
- Instruction scheduling
- Position and capability of any master control units that direct the processors
- Memory address space
- Input/output features
- Compilers and operating systems support to make the parallel system accessible
- Application's suitability to a particular parallel system
- Algorithms to implement the applications

As can be imagined there is an assortment of chooses for each of these components. This provides for the possibility of a large variety of parallel systems. Plus more chooses and variations are continually being developed to utilize the increased capacity of the underlining media.

Mike Flynn, in 1972 [Flynn72], developed a classification for various parallel systems, which has remained authoritative. It is based on the number of instruction streams and the number of data streams active in one cycle. A sequential machine is considered to have *single instruction stream executing on a single data stream;* this is called *SISD.* An *SIMD* machine has a *single instruction stream executing on multiple data streams* in the same cycle. *MIMD* has *multiple instruction streams executing on multiple data streams* simultaneously. All are shown in Fig. 1.17. An *MISD* is not shown but is considered to be a *systolic array.*

Four categories of *MIMD* systems, *dataflow, multithreaded, out of order execution, and very long instruction words* (*VLIW*), are of particular interest, and seem to be the tendency for the future. These categories can be applied to a single CPU, providing parallelism by having *multiple functional units.* All four attempt to use fine-grain parallelism to maximize the number of instructions that may be executing in the same cycle. They also use fine-grain parallelism to assist in utilizing cycles, which possibly could be lost due to large *latency* in the execution of an instruction. Latency increases when the execution of one instruction is temporarily staled while waiting for some resource currently not available, such as the results of a cache miss, or even a cache fetch, the results of a floating point instruction (which takes longer than a simpler instruction), or the availability of a needed functional unit. This could cause delays in the execution of other instructions. If there is very fine grain parallelism, other instructions can use available resources while the staled instruction is waiting. This is one area where much computing power has been reclaimed.

Two other compelling issues exist in parallel systems. *Portability,* once a program has been developed it should not need to be recoded to run efficiently on a parallel system, and *scalability,* the performance of a system should increase proportional to the size of the system. This is problematic since unexpected bottlenecks occur when more processors are added to many parallel systems.

### 1.5.2 Single Instruction Multiple Processors (SIMD)

Perhaps the simplest parallel system to describe is an SIMD machine. As the name implies all processors execute the same instruction at the same time. There is one master control unit, which issues instructions, and typically each processors also has its own local memory unit. SIMD processors are fun to envision as a school of fish that travel closely together, always in the same direction. Once one turns, they all turn. In most systems, processors communicate only with a set of nearest neighbors; *grids, hypercubes,* or *torus* are popular. In the most generic system, shown in Fig. 1.17b, no set communication pattern is

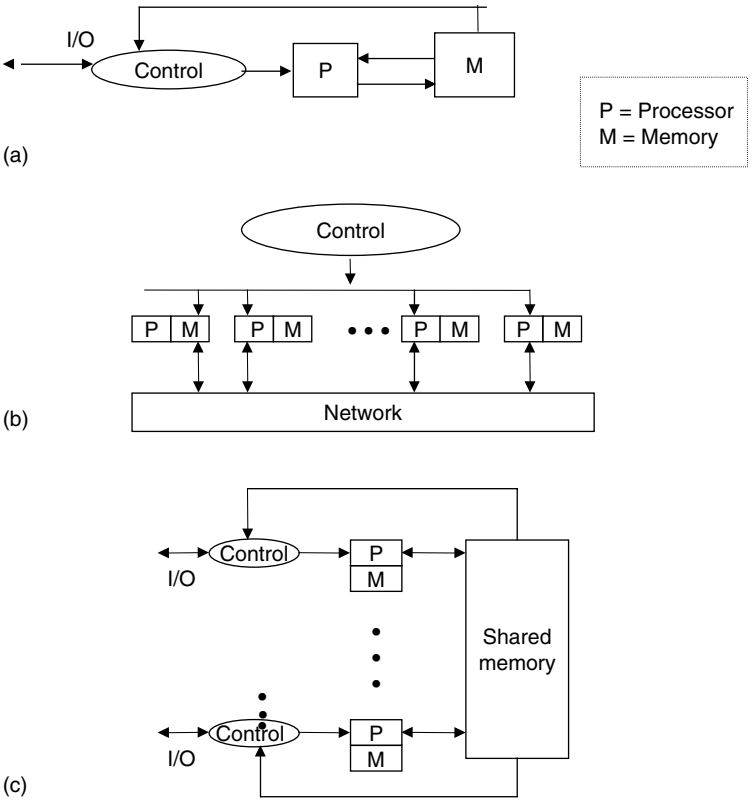



**FIGURE 1.17** (a) SISD uniprocessor architecture. (b) General SIMD with distributed memory. (c) Shared memory MIMD.

dictated. Because different algorithms do better on different physical topologies (algorithms for sorting do well on tree structures, but array arithmetic does well on grids), reconfigureable networks are ideal but hard to actually implement. A variety of structures can be built on top of grids if the programmer is resourceful, but some processing power would be lost. It is difficult for a compiler to decide which substructure would be optimal to use. A mixture of a close connections supplemented with longer connections seems to be most advantageous. *MasPar* [MasPar91] has SIMD capability and uses a grid structure supplemented with longer connections. The *Thinking Machines CM-5* has SIMD capability using a *fat-tree* network, also a mix of closer and longer connections. The PEC network also has this quality [Kirkman91, Quammen96].

Of course, SIMD machines have limits. Logical arrays may not meet the physical topology of the processors but require folding or skewing. *If* statements may cause different paths to be followed on different processors, and since it is necessary to always have synchronization, some processing power will be lost. *Masks* are used to inhibit issued instructions on processors on which they should not be executed. A single control unit becomes a bottleneck as an SIMD system expands. If an SIMD system were very large, it would be desirable to be able to use it as a multiprogrammed machine where different programs would be allocated different "farms" of processors for their use as a dedicated array. A large SIMD system should be sub-dividable for disjoint multiuser applications. The operating system would have to handle this allocation.

On an SIMD computer a loop such as the one below can be translated to one SIMD instruction as is shown. The form $A(1{:}N)$ means array $A$ indexes 1 to $N$:

for $I = 1$ to $N$ do
$A(I) = \mathrm{B}(I) + C(I)$; $A(1{:}N) = B(I{:}N) + C(1{:}N)$;
endfor;

The code below would take four steps:

$F(0) = 0$;
for $I = 1$ to $N$ do $F(0) = 0$;
$A(I) = \mathrm{B}(I)/C(I)$; $A(1{:}N) = B(I{:}N)/C(1{:}N)$;
$D(I) = A(I) * E(I)$; $D(1{:}N) = A(1{:}N) * E(1{:}N)$;
$F(I) = D(I) + F(I-1)$; $F(1{:}N) = D(1{:}N) + F(0{:}N-1)$;
endfor;

Compilers can identify loops such as the ones presented above [Wolfe91]. However, many loops are not capable of executing in an SIMD fashion because of reverse dependencies. Languages have been developed for SIMD programming, which allows the programmer to specify data locations and communication patterns.

### 1.5.3 Multiple Instruction Multiple Data

Perhaps the easiest MIMD processor to envision is a shared memory multiprocessor such as shown in Fig. 1.17c. With this machine, all processors access the same memory bank with the addition of local caches. This allows the processors to communicate by placing data in the shared memory. However, sharing data causes problems. *Data* and *cache coherence* are of major concern. If one processor is altering data that another processor wishes to use, and the first processor is also holding the current updated value for this data in its cache, there is a need to guard access to the stale value which is being held in the shared memory. This creates a need for *locks* and *protocols* to protect communal data. Inefficient algorithms to handle cache coherence can cause delays, or invalidate results. In addition, if more than one processor wishes to access the same locked memory location, a fairness issue occurs as to which processor should be allowed first access after the location becomes unlocked [Hwang93]. Further delays in accessing the shared memory occur due to the use of a single bus. This arrangement is described as a uniform memory access (UMA) time approach and avoids worst-case communication scenarios possible in other memory arrangements.

To reduce contention on the bus as a MIMD memory system scales, a distributed memory organization can be used. Here, clusters of processors share common memories, and the clusters are connected to allow communications between clusters. This is called an NUMA (nonuniform memory access) organization [Gupta91]. If a MIMD machine is to be *scalable,* this approach must be used. Machines within the same cluster will be able to share data with less latency than machines housed on different memory banks. It will be possible to access all data. This creates questions as to which sets of data should be placed on which processor cluster. Compilers can help with this by locating a code that uses common data. If data is poorly placed, the worst-case execution time could be devastating.

Message passing systems, such as the *Transputer* [May88], have no shared memory but handle communications using message passing. This can cause high latency while waiting for requested data; however, each processor can hold multiple threads, and may be able to occupy itself while waiting for remote data. Deadlocks are a problem.

Another variation of memory management is cache only memory access (COMA). Memory is distributed, but only held in cache [Saulsbury95]. The Kendall Square machine [KSR91] has this organization. On the KSM distributed memory is held in the cache of each processor, which is connected by a ring. The caches of remote processors are accessed using this ring.

### 1.5.4 Vector Machines

A *vector machine* creates a series of functional units and pumps a stream of data through the series. Each stage of the pipe will store its resulting data in a *vector register,* which will be read by the next stage.

In this way the parallelism is equal to the number of stages in the pipeline. This is very efficient if the same functions are to be preformed on a long stream of data. The Cray series computer [Cray92] is famous for this technique. It is becoming popular to make an individual processor of a MIMD system a vector processor.

### 1.5.5 Dataflow Machine

The von Neumann approach to computing has one control state in existence at one time. A program counter is used to point to the single next instruction. This approach is used in traditional machines, and is also used in most of the single processors of the multiple processor systems described earlier. A completely different approach was developed at the Massachusetts Institute of Technology [Dennis91, Arvind90, Polychronopoulos89]. They realized that the maximum amount of parallelism could be realized if at any one point *all instructions that are ready to execute* were executed. An instruction is *ready* to execute if the data that is required for its complete execution is available. Therefore, execution of an instruction is not governed by the sequential order, but by its readiness to execute, that is, when both operands are available. A table is kept of the instructions that are *about ready* to execute, that is, one of the two operands needed for the *assembly language level instruction* is available. When the second operand is found, this instruction is executed. The result of the execution is passed to a control unit, which will select a set of new instructions to be *about ready* to execute, or mark an instruction as *ready* (because the second operand needed has arrived).

This approach yields the maximum amount of parallelism. However, it runs into problems with "run away execution." Too many instructions may be *about ready*, and clog the system. It is a fascinating approach, and machines have been developed. It has the advantage that no, or very little, changes need to be made to *old dusty decks* to extract parallelism. Steps can be made to avoid "run away execution."

### 1.5.6 Out of Order Execution Concept

An approach similar to the *dataflow* concept is called *out of order execution* [Cintra00]. Here again, program elements that are ready to execute may be executed. It has a big advantage when multiple functional units are available on the same CPU, but the functional units have different latency values. The technique is not completely new but similar to issuing a *load instruction*, which has high latency, well before the result of the *load* is required. By the time the *load* is completed the code has reached the location where it is used. Also a *floating point instruction,* again a class of instructions with high latency, is frequently started before *integer instructions* coded to execute first are executed. By the time the *floating point* is complete, its results are ready to be used. The compiler can make this decision statically. In *out of order execution* the hardware has more of a role in the decision of what to execute. This may include both the *then* and the *else* parts of an *if* statement. They can both be executed, but not be committed to until the correct path is determined. This technique is also called *speculative execution*. Any changes that have been made by a wrong path must be capable of being rolled *back*. Although this may seem to be extra computation, it will decrease execution time if done well. Other areas of the program may also be executed, if it is determined that their execution will not affect the final result or *can be rolled back*. The temporary results may be kept in registers. The Alpha Computer [Kessler99] as well as the Intel Pentium Pro [Intel97] use this technique. This method is becoming popular to fully utilize increasingly powerful computers.

Compiler techniques can be used to try to determine which streams should be chosen for advance execution. If the wrong choice is made, there is a risk of extremely poor performance due to continual rollbacks. Branch prediction, either statically by the compiler, or dynamically by means of architecture prediction flags, is a useful technique for increasing the number of instructions, which may be beneficial to execute prematurely.

Since assembler instructions contain precise register addresses, set by the compiler, and it is unknown which assembler instructions will be caught in partial execution at the same time, a method

called *register renaming* is used. A *logical* register address is mapped to a *physical* register chosen from a *free list*. The mapping is then used throughout the execution of the instruction, and released again to the *free list*.

### 1.5.7 Multithreading

*Multithreading* [Tullsen95] is another method to hide the latency endured by various instructions. More than one chain, or thread, of execution is active at any one point. The states of different chains are saved simultaneously [Lo97, Miller90] in their own *state space*. Modern programs are being written as a collection of modules, either *threads* or *objects*. Because one of the main advantages of this form of programming is data modulization, many of these modules could be ready to execute concurrently. While the processor is waiting for one thread's data (for example, a cache miss or even a cache access), other threads, which have a full state in their dedicated space, can be executed. The compiler cannot determine which modules will be active at the same time, that will have to be done dynamically. The method is somewhat similar to the *multiprogramming* technique of changing context while waiting for I/O; however, it is at a *finer grain*. Multiple access lines to memory are beneficial since many threads may be waiting for I/O. The *Tera machine* [Smith90] is the prime example of this technique. This approach should help lead to Teraflops performance.

### 1.5.8 Very Long Instruction Word (VLIW)

A VLIW will issue an instruction to multiple functional units in parallel. Therefore, if the compiler can find one operation for each of the functional unit internal to a processor (these instructions are usually RISC-like), which will be able to execute at the same time (that is, the data for their execution are statically determined to be available in registers), and none of the instructions depend on an instruction being issued in the same cycle, then you can execute them in parallel. All the sub-instructions will be issued by one long instruction. The name VLSI comes from the need that the instruction be long enough to hold multiple operation codes, one for each functional unit, which will be used, and the register identifiers, which they need. Unlike the three methods described previously, the compiler is responsible for finding instructions that do not interfere with each other and assigning the registers for these instructions [Rau93, Park97]. The compiler packs these instructions statically into the VLIW. The Intel iWarp can control a *floating point multiplier, floating point adder, integer adder, memory loader, increment unit,* and *condition tester* on each cycle [Cohn89]. The instruction is 96 bits long and can, with compiler support, execute nine instructions at once. The code for the following loop can be turned into a loop of just one VLSI instruction as opposed to a loop of at least nine RISC size instructions.

for $I := 0$ to $N-1$ do
$A(2 * I) := C + B(I) * D$;

It is difficult for a compiler to find instructions that can fill all the fields statically, so frequently some of the functional units go unoccupied. There are many techniques to find qualified instructions, and frequently the long instruction can be filled. One technique is to mine separate threads [Lam88, Bakewell91], another successful technique tries together several basic Oblocks into a *hyperblock*. The hyperblock has one entrance but may have several exits. This creates a long stream of code, which would normally be executed sequentially, and allows the compiler to choose instructions over this larger range. Roll-back code must be added to the hyperblock's exit to undo the affects of superfluous code that was executed, but would not execute sequentially. Loops are frequently *unrolled,* several iterations considered as straight code, to form hyperblocks. *Branch prediction* can also help create beneficial hyperblocks.

A newer approach is to dynamically pack the VLIW, using a preprocessor that accesses multiple queues, one for each functional unit. Realize that using queues is similar to *out of order execution*.

### 1.5.9 Interconnection Network

An interconnection network is a necessary component of all parallel processing systems. Several features govern the choice of a network. A scalable interconnection network for parallel processor would be ideal if it meets the following requirements for a large range of system sizes. For instance, it may be scalable by reasonably small increments from $2^4$ to perhaps $2^{20}$ processors.

- Have a low average and maximum diameter (the distance between the furthest two nodes) to avoid communication latency.
- Minimize routing constraints (have many routes from *A* to *B*).
- Have a constant number of I/O port (channels) per node to allow for expansion without retrofit.
- Have a simple wire layout to allow for expansion and to avoid wasting VLSI space.
- Be inherently fault tolerant.
- Be sub-dividable for disjoint multiuser applications.
- Be able to handle a large range of algorithms without undo overhead.

The most popular parallel networks—hypercube, quad tree, fat-tree, binary tree, mesh, and torus—fail in one or more of these items.

Meshes have a major disadvantage: they lack support for long distance connections. Hypercubes have excellent connectivity by guaranteeing a maximum distance between any two nodes of log *N* where *N* is the number of nodes. Also, many paths exist between any two nodes making it fault tolerant and amenable to low contention. However, the number of I/O ports per node is log *N*. As a system scales, each node would need to be retrofit to add additional ports. In addition, the wire layout is complex, making this network expensive in space. Tree structures are popular and have a maximum long distance connection of O(2 log *N*). Communications on a tree, however, can be complicated by the fact that although many neighbors are close, many can only communicate through the root. This causes contention at the root. Fattrees reduce this contention by increasing the bandwidth, as the network approaches the root [Leiserson92].

The extreme ideal network would allow all nodes to connect to all others. This is not practical for large system. However, one class of networks, the *multistage network,* uses an internal switching system and allows constant access time between any two nodes. Several arrangements are available for multistage networks. All are similar. One such network, the *baseline network,* is shown in Fig. 1.18. It can be proved that a *N*-by-*N* network can be totally connected with log *N* switches steps [Seigel89]. Each step is through a row of switches, with $N/2$ switches in each row. This requires quite a bit of hardware and does allow for a connection in log *N* steps. This is not very scalable.

*Optical* technology [Yuan97] has shown to be promising for the implementation of all networks. Instead of a wire, an optical "beam" is used to make the connection. This is fast and has several

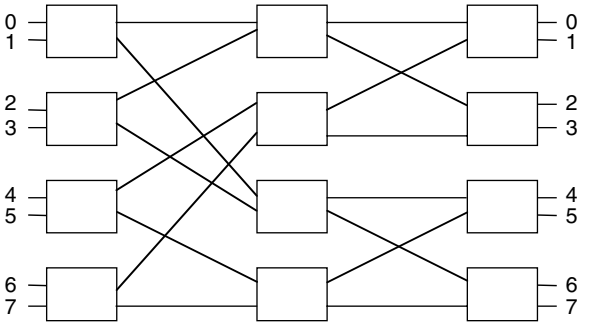



**FIGURE 1.18** An 8 × 8 baseline network.

advantages. One, a broadcast can be made from one node to many nodes at once (although, one node cannot receive many inputs at once), and two, the transmission can be sent through clear space.

### 1.5.10 Conclusion

Parallel systems are going to become almost universal in computer systems. Desktop computers are now frequently being delivered with more than one CPU and definitely with more than one functional unit. The new models of SUN, MIPS, Intel, and Macintosh desktop computers currently are providing parallel computing capabilities. This feature is no longer limited to just super computers. In order to use these systems efficiently, the consumer should be aware of how their programs are going to utilize the systems. Compilers, operating systems, and program design are all things that should be examined.

Most parallel processing is being aimed at hiding latency; however, embedded systems and super computing implementations also need the ability to execute separate lines of control running independently, which must communicate with each other. Embedded systems need this to assure the strict adherence to real-time deadlines. Super computer applications need the additional processing power.

This section is only briefly discussed this field. A full addressing of the field is covered in large volumes of books and journals. Topics, such as compiler techniques, dedicated languages, communication techniques, multi-user facilities, algorithms, memory hierarchies, I/O facilities, programming tools, fault tolerant, power consumption, and debugging technique, are only an abbreviated list of topics which need to be examined. Plus, each of these subtopics has many aspects. All computer users should certainly be aware of this field.

## References

[Arvind90] Ardvin and Nikhil, "Executing a Program on the MIT Tagged Dataflow Architecture," *IEEE Trans. Computer,* 1990.

[Bakewell91] H. Bakewell, D. Quammen, and P. Wang, "Mapping Concurrent Programs to VLIW Processors" *Proc. Principles and Practices of Parallel Programming,* Williamsburg, Virginia, pp. 21–27, April 1991.

[Cintra00] M. Cintra, J. Martinez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelism in Shared-Memory Multiprocessors," *Int. Symp. Computer Architecture,* 2000.

[Cohn89] R. Cohn, T. Gross, M. Lam, and P. Tseng, "Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor," *Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* 1989.

[Cray92] Cray, *Cray/MPP Announcement,* Cray Research, Inc. Eagan, MN, 1991.

[Dennis91] J. Dennis, "The Evolution of Static Dataflow Architecture," in *Gaudiot and Bic, Advanced Topics in Dataflow Computing,* Prentice-Hill, Englewood Cliffs, NJ, 1991.

[Flynn72] M.J. Flynn, "Some Computer Organization and their Effectiveness," *IEEE Trans. Computer,* 21 (9): 948–960, 1972.

[Gupta91] A. Gupta, J. Hennessy, et al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. Int. Symp. Computer Architecture,* May 1991.

[Hwang93] Kai Hwang, *Advanced Computer Architecture,* McGraw-Hill, Inc, New York, 1993.

[Intel97] Intel Corporation, *Pentium(R) Pro Processor Developmer's Manual,* McGraw-Hill, New York, June 1997.

[Kessler99] R.E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro,* 1999.

[Kirkman91] W. Kirkman and D. Quammen, "Packed Exponential Connections—A Hierarchy of 2D Meshes," *Proc. 5th Int. Parallel Proceedings Symposium,* Anaheim, pp. 464–470, California, April 1991.

[KSR91] KSR-1 *Overview,* Internal Report, Kendall Square Research Co.170 Tracer Lane, Waltham, MA 02154, 1991.

[Lam88] M. Lam Software Pipelining, "An Effective Scheduling Technique for VLIW Machines," *ACM Sigplan 1088 Conf. or Programming Language Design and Implementation,* 1988.

[Leiserson92] C.E. Leiserson, "The Network Architecture of the Connection Machine, CM-5," *Proc. ACM Symp. Computer Architecture,* 1992.

[Lo97] J. Lo, S. Eggers, et al., "Tuning Compiler Optimization for Simultaneous Multithreading," *Int. Symp. on Microarchitecture,* 1997.

[MasPar91] MasPar, "The MasPar Family of Data Parallel Computer," Technical summary, MasPar Computer Corporation, Sunnyvale, CA, 1991.

[May88] D. Pountain and D. May, *Occam Programming,* INMOS, Oxford England, 1988.

[Miller90] D.R. Miller and D.J. Quammen. "Exploiting Large Register Sets," *Int. J. Microprocessors and Microsystems,* pp. 333–340, August 1990.

[Park97] S. Park, S. Shim, and S. Moon, Evaluation of Scheduling Techniques on a SPARC-Based VLIW Testbed, Micro-30, 1997.

[Polychronopulos89] C.D. Polychronopulos, et al., "Paradrase 2: An Environment for Parallel Zing, Partitioning, Synchronizing Programs on Multiprocessors," *Proc. Int. Conf. on Parallel Processing,* 1989.

[Quammen96] D. Quammen, J. Stanley, and P. Wang, "The Packed Exponential Connection Network," *Pre. Int. Symp. Parallel Arch. Algorithms and Networks,* 1996.

[Rau93] B. Rau and J. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *J. Supercomputing: Special Issue on Instruction-Level Parallelism,* 1993.

[Saulsbury95] A. Saulsbury, T. Wilkinson, et al., "An Argument for a Simple COMA," *Symp. Int. on High Perf. Computer Architecture,* 1995.

[Siegel89] H.J. Siegel, "A model of SIMD Machines and a Comparison of Various Interconection Networks," *IEEE Trans. Computer,* 28(12) 1989.

[Smith90] J.E. Smith, "Future General-Purposes Supercomputer Architecture," *Proc. ACM Supercomputing Conf.,* 1990.

[Tullsen95] D. Tullsen, et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 23rd Int. Symp. on Computer Architecture*, 1995.

[Wolfe91] M.E. Wolfe and M. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel Distri Systems,* 3(10), 1991.

[Yuan97] X. Yuan, R. Melhem, and R. Gupta, "Distributed Path Reservation Algorithms for Multiplexing All-Optical Interconnection Networks," *Proc. Symp. High-Performance Comp. Architecture,* 1997.

## 1.6 Virtual Memory Systems and TLB Structures

*Bruce Jacob*

### 1.6.1 Virtual Memory, a Third of a Century Later

Virtual memory was designed in the late 1960s to provide automated storage allocation. It is a technique for managing the resource of physical memory that provides to the application an illusion of a very large amount of memory—typically, much larger than is actually available. In a virtual memory system, only the most often used portions of a process's address space actually occupy physical memory; the rest of the address space is stored on disk until needed. When the mechanism was invented, computer memories were physically large (one kilobyte of memory occupied a space the size of a refrigerator), they had access times comparable to the processor's speed (both were extremely slow), and they came with astronomical price tags. Due to space and monetary constraints, installed computer systems typically had very little memory—usually less than the size of today's on-chip caches, and far less than the users of the systems would have liked. The virtual memory mechanism was designed to solve this problem, by using a system's disk space as if it were memory and placing into main memory only the data used most often.

Since then, we have seen constant evolution (and revolution) in the computer industry. Typical microprocessors today have more on-chip cache than the core memory found in multimillion-dollar

systems of yesterday and cost orders of magnitude less. Today, memory takes up very little space: you can easily hold a gigabyte of DRAM in your hand. In recent decades, processor designers have focused on improving speed while memory-chip designers have focused on improving storage size, and, as a result, memory is now extremely slow compared to processor speeds. Due to rapidly decreasing memory prices, it is usually possible to have enough memory in one's machine to avoid using the disk as a backup memory space. Many of today's machines generate 64-bit addresses, some even larger; most modern machines therefore reference 16 exabytes (16 giga-gigabytes) or more of data in their address space directly. The list goes on. In fact, one of the few things that has not changed since the development of virtual memory is the basic design of the virtual memory mechanism itself, and the one problem it was invented to solve—too little memory—is no longer a factor in most systems. However, the virtual memory mechanism has proven itself valuable in other areas besides extending the memory space. Today it is used in nearly every modern operating system because of the convenience offered by its features: It simplifies memory allocation and memory protection, and it provides an intuitive programming interface to the application—the "virtual machine" interface—that simplifies program design and provides a natural path to multitasking.

### 1.6.2 Caching the Process Address Space

A process operates in its own little world; this is the *virtual machine* paradigm, illustrated in Fig. 1.19. Each running process generates addresses for loads and stores as if it has the entire machine to itself—as if the computer offers an extremely large amount of memory and no other processes are executing or consuming resources. This makes the job of the programmer and compiler much easier, because no details of the hardware or memory organization are necessary to build a program.

The operating system divides the process address space into equal-sized portions for ease of management; these divisions are called *virtual pages*. A page is usually a multiple of the unit of transfer that hard disks use, and in most operating systems ranges from several kilobytes to several dozen kilobytes. A page is never fragmented; if any data in a virtual page are in physical memory then all the data in that page are, and if any of the data in a virtual page are nonexistent or being held on disk then all the data are. When the word *page* is used in a verb form, it means to allow a section of memory to be virtual—to

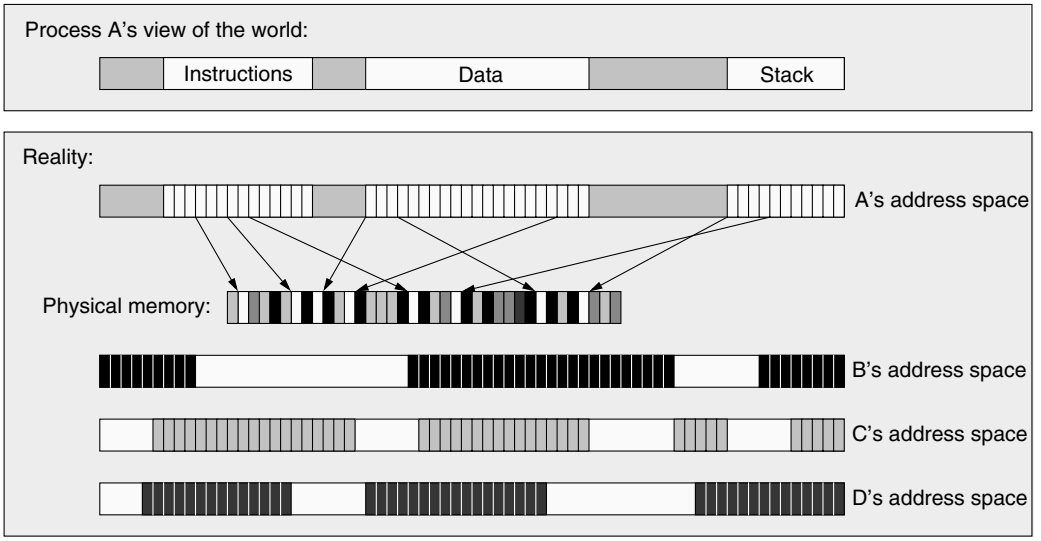



**FIGURE 1.19** The virtual machine paradigm. A process operates in its own virtual environment, unaware that other processes are executing and contending for the same limited resources. The operating system views each process address space as a collection of pages that can be cached in physical memory, or left in backing store.

allow it to move freely between physical memory and disk. This allows the physical memory to be used more efficiently: When a region of memory has not been used recently, the space can be freed up for more active pages, and pages that have been migrated to disk are brought back in as soon as they are needed again.

How is this done? The ultimate home for the process's address space is *backing store*, usually a disk drive; this is where the process's instructions and data come from and where all of its permanent changes go to. Every hardware memory structure between the CPU and the backing store is a cache for the instructions and data in the process's address space. This includes main memory—main memory is really nothing more than a cache for a process's virtual address space. A cache operates on the principle that a small, fast storage device can hold the most important data found on a larger, slower storage device, effectively making the slower device look fast. The large storage area in this case is the process address space, which can be many gigabytes in size. Everything in the address space initially comes from the program file stored on disk or is created on demand and defined to be zero. Figure 1.20 illustrates:

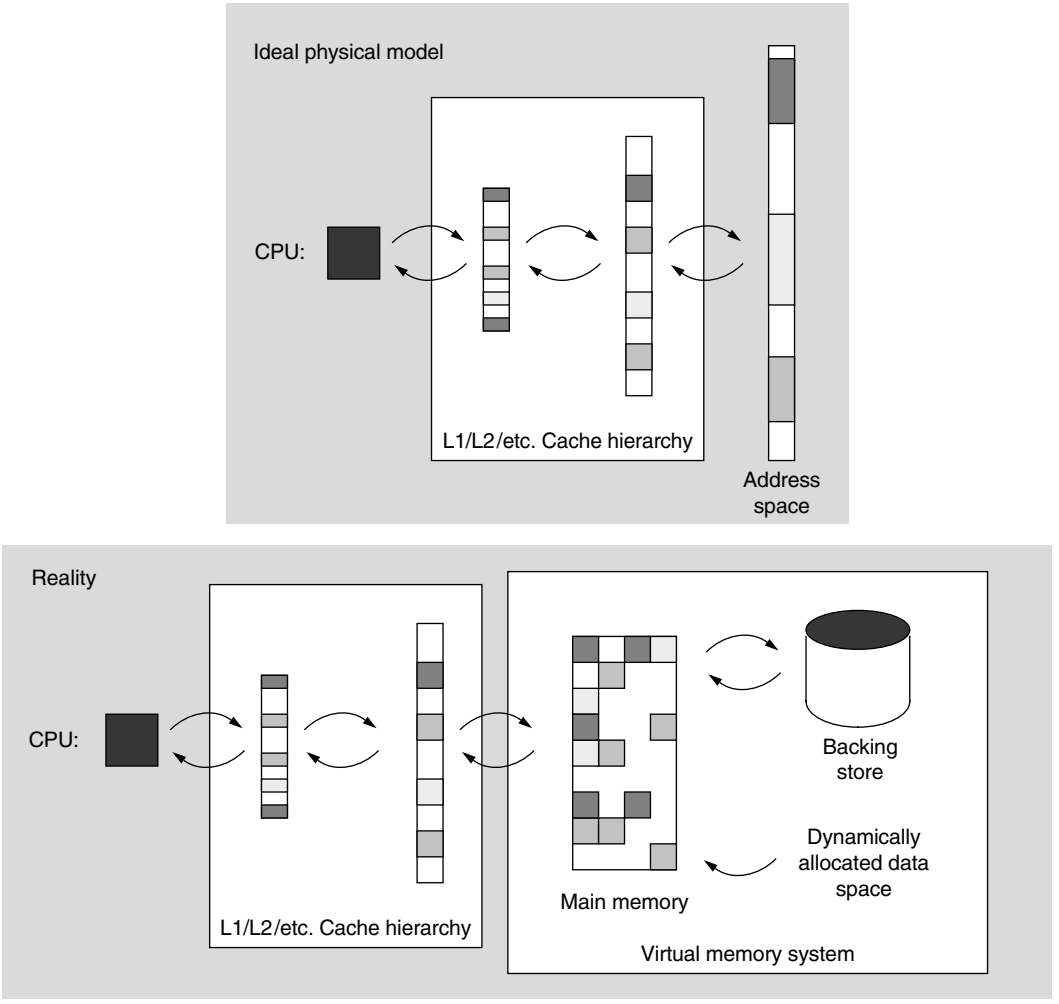



**FIGURE 1.20** Caching the process address space. In the first view, a process is shown referencing locations in its address space. Note that all loads, stores, and fetches use virtual names for objects, and many of the requests can be satisfied by a cache hierarchy. The second view shows that the address space is not a linear object stored on some device, but is instead scattered across hard drives and dynamically allocated when necessary.

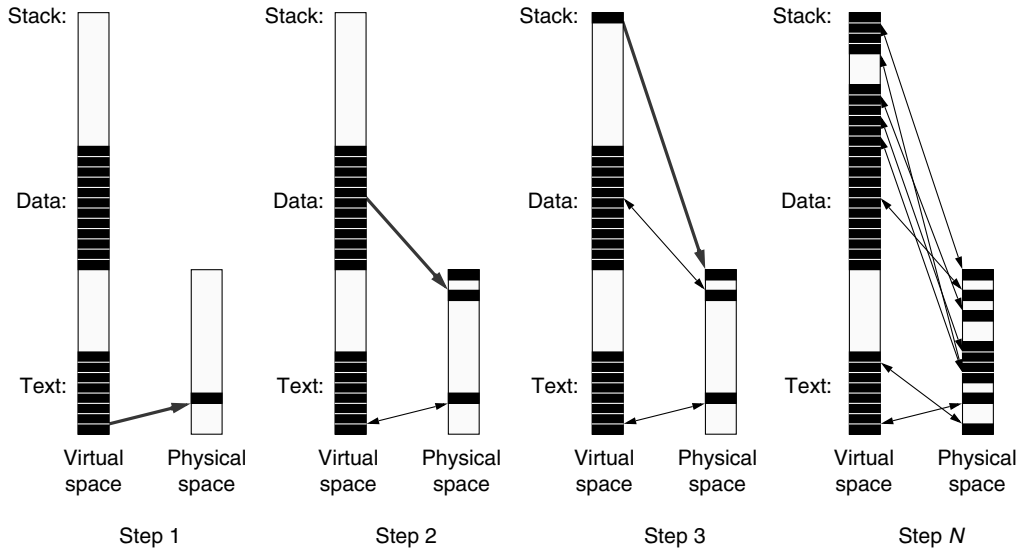



**FIGURE 1.21** Demand paging at process start-up. In step 1, the operating system loads the first page of the process's instructions into physical memory, and sets the program counter to the first instruction in the program. This first instruction references a location in the process's data area, so in step 2 the operating system brings the corresponding data page into physical memory. The next instruction references a location on the process's stack, so in step 3 the operating system has allocated a stack page for the process and placed it into the process address space and main memory. Succeeding instructions reference more locations in the stack area, jump to instructions that lie outside of the initial page of instructions, and allocate extra data storage area on the heap. In step *N* (many steps later), these pages have been brought into main memory.

There really is no linear array of data that houses the process address space. Its illusion is actually manufactured by the operating system through the virtual memory mechanism.

When a program first begins executing, the operating system copies a small portion of the process address space from the program file stored on disk into main memory. This typically includes the first page of instructions in the program and possibly a small amount of data that the program needs at startup. Then, as more instructions or data are needed, the operating system brings in pages from the process's address on demand. This process, called *demand paging*, is depicted in Fig. 1.21.

In step 1 of the figure, the operating system initializes a process address space and loads the first page of instructions into physical memory. The operating system then sets the hardware program counter to the first instruction in the program, which sets the process running. Assuming that one of the first few instructions references the initialized data area, the uninitialized data area, or the (so far nonexistent) stack, the operating system will have to bring in a page of data from the program file or create an uninitialized-data page or stack page and link it into the process address space. This is shown in steps 2 and 3 of the figure. When a process references an item in its address space that is not currently in physical memory, the reference causes a *page fault*, and the operating system loads the necessary pages from backing store into main memory. Clearly, the term *demand paging* refers to the fact that pages are allocated or brought into physical memory on demand. Step *N* of the figure shows a process that has been executing for some time, as it has several pages of data in its stack area and several pages in its data area that were not there when the process began executing. All of these pages were dynamically allocated by the operating system as the process needed or asked for them.

As has been pointed out before, the process is unaware of the operating system activity that moves pages in and out of main memory on its behalf. It typically does not know whether or not any given page is memory-resident or where it is located if it is memory-resident. Figure 1.19 at the beginning of the section illustrates this by showing a process address space from two points of view. The first point of

view is from the process itself; in most operating systems a process sees its address space as a contiguous span of memory locations from minimum to maximum. Somewhere in the address space is the program's instructions, or *text*; somewhere else is the program's data. Most operating systems also create a stack area, a heap area, and possibly one or more dynamically loaded libraries containing system-supplied utilities such as input/output routines or networking functions. The advantage of the virtual machine paradigm is that these can be arranged in physical memory, which is most convenient, rather than having to fit things together like the pieces of a puzzle, as would be the case without address translation.

The second point of view in the figure is from the operating system. In reality, the process address space is not a large contiguous segment in physical memory but is partially cached by physical memory. Portions of the process address space are scattered about physical memory and are likely to be not contiguous at all. The process is unaware of where in the system any particular portion of its address space is being held; some portions can be on disk (for example, the portions of the program that have not been used yet), some can be in main memory, and some can be in hardware caches. The operating system maintains a map for each address space so that, for every virtual page in the address space, it can tell where in memory or on disk the page can be found. As the figure suggests, the virtual machine paradigm allows each process to behave as if it owns the entire machine; each process is protected from all others and does not even know that other processes exist—for example, a process cannot spoof the identity of another process, and the resource-management mechanisms implemented by the operating system to support the illusion that each process own all physical resources means that no process may dominate system resources. One of the many benefits of this organization is that it makes facilities such as multitasking very easy to implement, because process protection, resource sharing, and a clean division of process identity are provided as side effects of the virtual machine paradigm by definition.

The mapping information that tells the location of pages in memory or on disk is organized into *page tables*, which are collections of *page table entries* (*PTEs*). Virtual addresses (shown in Fig. 1.22) are mapped at the granularity of *pages*; at its simplest, virtual memory is then a mapping of *virtual page numbers* (*VPNs*) to *page frame numbers* (*PFNs*), shown in Fig. 1.23. "Frame" in this context means "slot"—physical memory is divided into frames that hold pages. The page table holds one PTE for every mapped virtual page; an individual PTE indicates whether its virtual page is in memory, on disk, or not allocated yet. The logical PTE therefore contains the VPN and either the page's location in memory (a PFN), or its location on disk (a disk block number). Depending on the organization, some of this information is redundant; actual implementations do not necessarily require both the VPN and the PFN. Later developments in virtual memory added such things as page-level protections; a modern PTE usually contains protection information as well, such as whether the page contains executable code, whether it can be modified, and if so by whom.

The mapping is a function; any virtual page can have only one location. However, the inverse map is not necessarily a function; it is possible and sometimes advantageous to have several virtual pages mapped to the same page frame (to share memory between processes or threads, or to allow different views of data with different protections, for example). Shared memory is one of the more commonly used features of page tables. It is a mechanism whereby two address spaces that are protected from each other are allowed to intersect at points, still retaining protection over the nonintersecting regions. Several processes sharing portions of their address spaces are pictured in Fig. 1.24. The shared memory mechanism only opens up a pre-defined portion of a process's address space; the rest

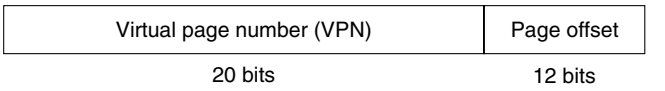



**FIGURE 1.22** Virtual addresses. A virtual address is divided into two components: the virtual page number and the page offset. The virtual page number identifies the page's location within the address space. The page offset identifies a byte's location within the page. Bit widths are shown for a 32-bit address and a 4 kbyte page size.

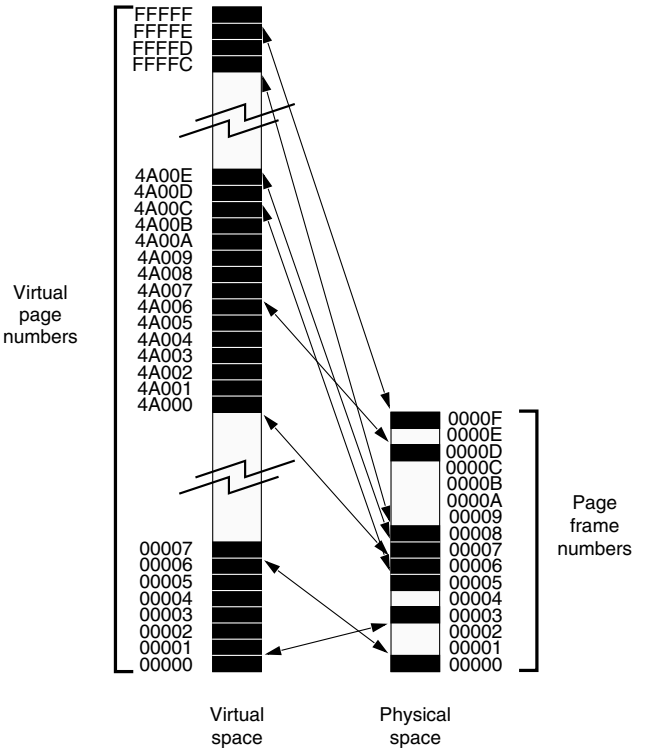



**FIGURE 1.23** Page numbers (for 32-bit virtual addresses). Every page in an address space is given a virtual page number (VPN). Every page in physical memory is given a physical page number, called a page frame number (PFN).

of the address space is still protected, and even the shared portion is only unprotected for those processes sharing the memory. For instance, in the figure, the region of A's address space that is shared with process B is unprotected from whatever actions B might want to take, but it is safe from the actions of any other processes. Shared memory is therefore useful as a simple, secure means for

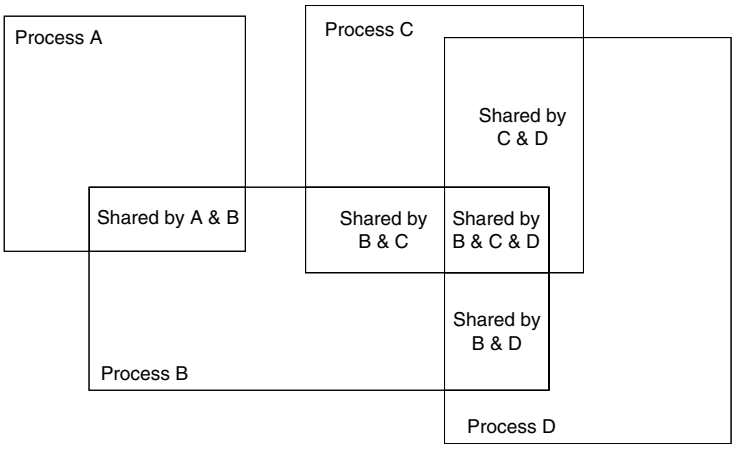



**FIGURE 1.24** Shared memory. Shared memory allows processes to overlap portions of their address space while retaining protection for the nonintersecting regions; this is a simple and effective method for inter-process communication. Pictured are four process address spaces that have overlapped. The darker regions are shared by more than one process, while the lightest regions are still protected from other processes.

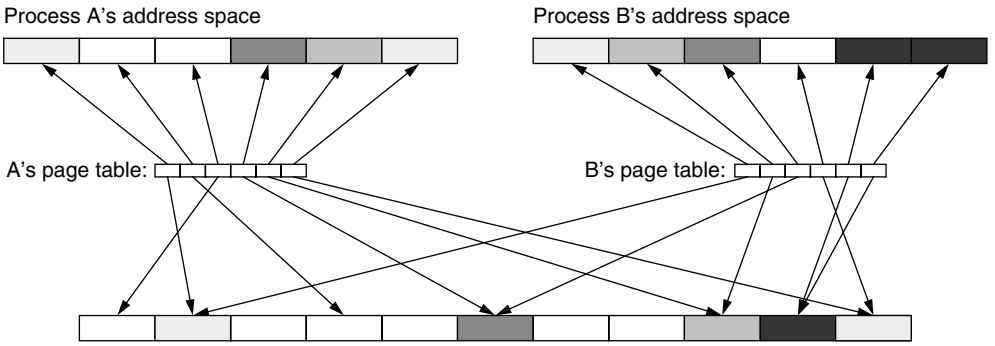



**FIGURE 1.25** How page tables support shared memory. Two process address spaces are shown sharing several pages. Their page tables maintain information on where virtual pages are located in physical memory. The darkened pages are mapped to several locations; note that the darkest page is mapped at two locations in the same address space.

inter-process communication. Shared memory also reduces requirements for physical memory; for example, in most operating systems, the text regions of processes are shared whenever multiple instances of a single program are run, or when multiple instances of a common library are used in different programs.

The mechanism works by ensuring that shared pages map to the same physical page; this is done by simply placing the same page frame number in the page tables of two processes sharing a page. A simple example is shown in Fig. 1.25. Here, two very small address spaces are shown overlapping at several places, and one address space overlaps with itself; two of its virtual pages map to the same physical page. This is not just a contrived example; many operating systems allow this, and it is useful, for example, in the implementation of user-level threads.

### 1.6.3 An Example Page Table Organization

So now the question is: How do page tables work? If we think of main memory as the data array of a cache, then the page table is the cache's corresponding *tags array*—it is a lookup table that tells one what is currently stored in the data array. The traditional design of virtual memory uses a fully associative organization for main memory: Any virtual object can be placed at (more or less) any location in main memory, which reduces contention for main memory and increases performance. An idealized fully associative cache is pictured in Fig. 1.26. A data tag is fed into the cache; the first stage compares the input tag to the tag of every piece of data in the cache. The matching tag points to the data's location in the cache. The goal of the page table organization is to support this lookup function as efficiently as possible.

To access a page in physical memory, it is necessary to look up the appropriate PTE to find where the page resides. This lookup can be simplified if PTEs are organized contiguously so that a page number can be used as an offset to find the appropriate PTE. This leads to two primary types of page table organization: the *forward-mapped* or *hierarchical page table*, indexed by the virtual page number, and the *inverse-mapped* or *inverted page table*, indexed by the physical page number (page frame number). Each design has its strengths and weaknesses. The hierarchical table supports a simple lookup algorithm and simple sharing mechanisms and can require a significant fraction of physical memory. The inverted table supports efficient hardware table-walking mechanisms and requires less physical memory than a hierarchical table but inhibits sharing by not allowing the mappings for multiple virtual pages to exist in the table simultaneously, if those pages map to the same page frame. Detailed descriptions of these can be found elsewhere (Jacob and Mudge 1998a).

Instead of describing all possible page table organizations, we will look in some detail at a concrete example: the virtual memory implementation of one of the oldest and simplest virtual memory systems,

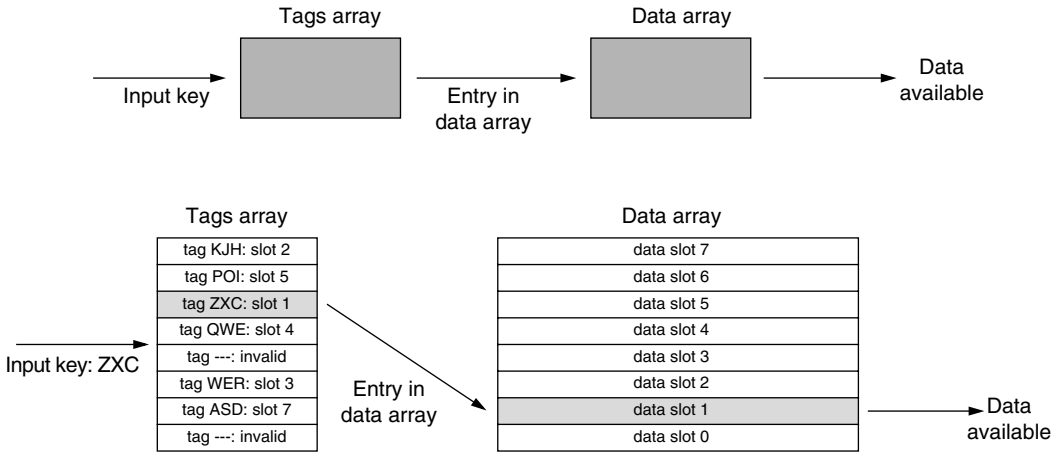



**FIGURE 1.26** An idealized fully associative cache lookup. A cache is comprised of two parts: the tags array and the data array. The tags act as a database; they accept as input a key (a virtual address) and output either the location of the item in the data array, or an indication that the item is not in the data array. A fully associative cache allows an item to be located at any slot in the data array, thus the input key is compared against every key in the tags array.

4.3BSD Unix (Leffler et al. 1989). The intent is to show how mapping information is used by the operating system and how the physical memory layout is organized. Version 4.3 of Berkeley Unix provides support for shared text regions, address space protection, and page-level protection. There is a separate page table for every process, and the page tables cannot be paged to disk. As we will see, address spaces are organized to minimize memory requirements.

BSD defines segments to be contiguous regions of virtual space. A process address space is composed of five primary segments: the *text* segment, holding the executable code; the *initialized data* segment, containing those data that are initialized to specific nonzero values at process start-up; the *bss* segment, containing data initialized as zero at process start-up; the *heap* segment, containing uninitialized data and the process's heap; and the *stack*. Beyond the stack is a region holding the kernel's stack (used when executing system calls on behalf of this process, for example) and the *user struct*, a kernel data structure holding a large quantity of process-specific information. Figure 1.27 illustrates the layout of these segments in a process's address space: The initialized data segment begins immediately after the text segment, the bss segment begins immediately after the initialized data segment, and the heap segment begins immediately after the bss segment. This is possible because the text, initialized data, and bss regions by definition cannot change size during the execution of a process. The heap segment can grow larger, as can the stack. Therefore, these two begin at opposite ends of the address space and grow towards each other. Beyond the 2 GB point, the address space belongs to the kernel; a user reference causes an exception.

This design makes sense for a number of reasons. When the operating system was designed, memory was at a premium. The choice was made to wire down the page tables. Given this, it makes most sense to restrict an address space to be composed of a minimal number of contiguous regions; this would ensure a compact page table (contiguous pages imply densely packed PTEs). The process model includes a

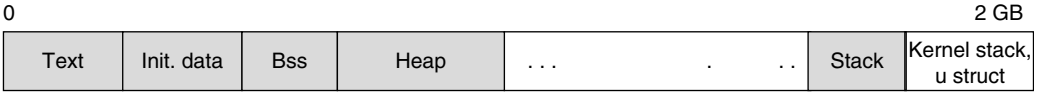



**FIGURE 1.27** The 4.3BSD per-process virtual address space.

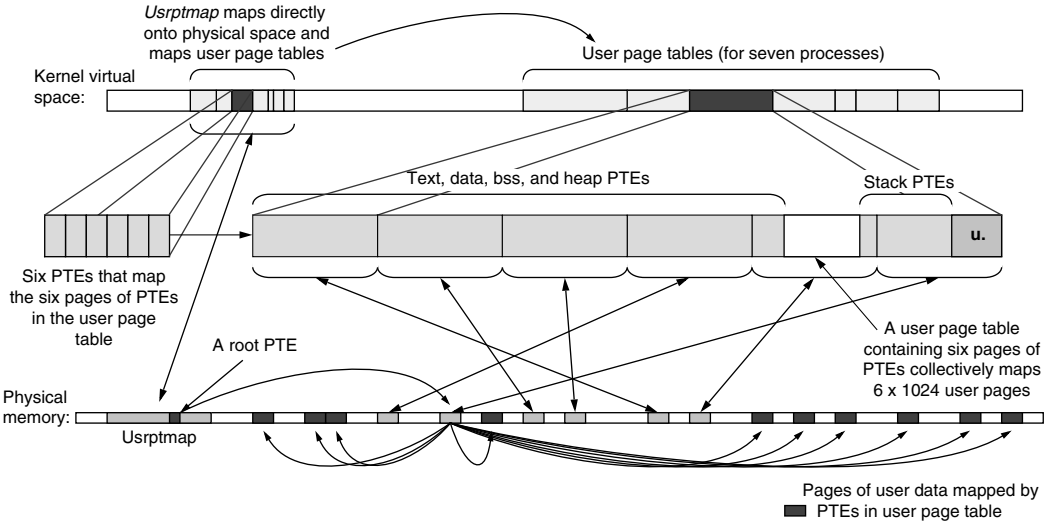



**FIGURE 1.28** User-process page tables in 4.3BSD Unix.

single thread of execution per address space; 4.3BSD did not have multiple threads within an address space, nor did it use dynamically loaded libraries. Therefore, there was no need to support sparsely populated address spaces.

Figure 1.28 depicts the layout of process address spaces and the associated process page tables. The page tables are kept in the kernel's virtual address space and are relocatable even if wired down. As shown in the figure, each user-process page table mirrors the process's address space; the PTEs that map the text, data, bss, and heap segments are at the bottom end of a contiguous range of PTEs (which are held in the kernel's virtual pages), and the PTEs that map the user's stack are near the top of the range of PTEs. A user page table is therefore as compact as it can be, with no more than a page of wasted space; the empty space between the ranges of PTEs allows for expansion of the heap and stack segments.

When a process needs to expand its address space beyond the confines of its user page table, the operating system adds an additional page to the page table and shifts all following process page tables up by one virtual page. This is the advantage of placing the user page tables in virtual space; the displaced data need not be recopied. The disadvantage is that there needs to be another level of mapping to determine where in the physical memory the pages that comprise a process's user page table are located. The *Usrptmap* is a structure that mirrors the entire set of user page tables, and for every page in a process's user page table, there is one PTE in the *Usrptmap*.

When a user reference requires a lookup in the page table, the operating system first determines which process caused the fault; this identifies the appropriate page table within the region of user page tables. The operating system then determines whether the access was to the user's stack or one of the text, bss, or data segments. If the access is to the user's stack, the operating system indexes backward from the top of the appropriate user page table to find the PTE; if the access is to the text, data, bss, or heap segment, the operating system indexes forward from the bottom of the user page table.

The *usrptmap* begins at a known location in physical memory; therefore, any process address space can be mapped. The appropriate root PTE within the *usrptmap* can always be found, given a process ID, and each root PTE points to a page of PTEs in physical memory, each of which then points to a page in the user address space.

### 1.6.4 Translation Lookaside Buffers: Caching the Page Table

There is an obvious question of performance to consider: If every memory access by a user program requires a lookup to the page table, how does anything ever get done? The answer is a familiar one: we

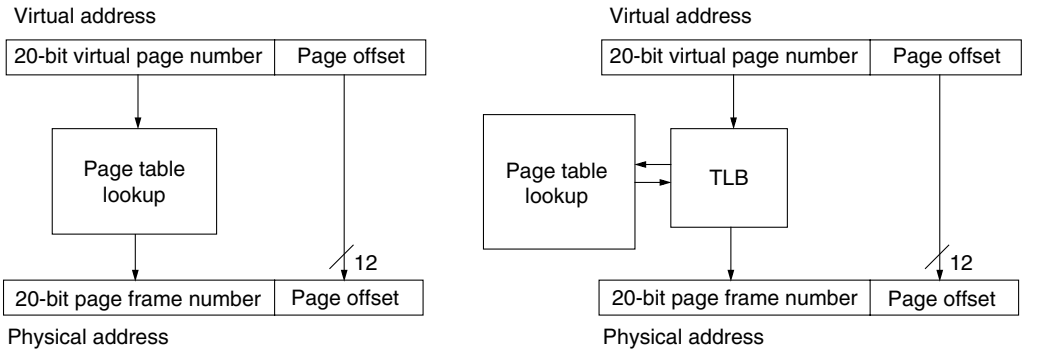



**FIGURE 1.29** Address translation with and without a TLB. Address translation without a TLB is shown on the left; translation with a TLB is shown on the right. The only difference is that the TLB caches the most recently used entries in the page table, and the page table is only referenced when a lookup misses the TLB.

cache things. Rather than perform a page-table lookup on every memory reference (which returns a PTE that gives us mapping information), we cache the most frequently used PTEs in hardware. The hardware structure is called a *translation lookaside buffer* (*TLB*), and because it holds mapping information, the hardware can perform the address translations of those PTEs that are currently cached in the TLB without having to access the page table (see Fig. 1.29). If the appropriate PTEs are stored in hardware, a memory reference completes at the speed of hardware, rather than being limited by the speed of looking up PTEs in the page table.

Most architectures provide a TLB to support memory management; the TLB is a special-purpose cache that holds only virtual-physical mappings. When a process attempts to load from or store to a virtual address, the hardware searches the TLB for the virtual address's mapping. If the mapping exists in the TLB, the hardware can translate the reference to a physical address without the aid of the page table. If the mapping does not exist in the TLB (an event called a *TLB miss*), the process cannot continue until the correct mapping information is loaded into the TLB.

Translation lookaside buffers are fairly large; they usually have in the order of 100 entries, making them several times larger than a register file. They are typically fully associative, and they are often accessed every clock cycle. In that clock cycle they must translate both the I-stream and the D-stream. Thus, they are often split into two halves, each devoted to translating either instruction or data references. They can constrain the chip's clock cycle as they tend to be fairly slow, and they are also power hungry (both are a function of the TLB's high degree of associativity).

In general, if the necessary translation information is on-chip in the TLB, the system can translate a virtual address to a physical address without requiring an access to the page table. In the event that the translation information is not found in the TLB, one must search the page table for the translation and insert it into the TLB before processing can continue. This activity can be performed by the operating system or by the hardware directly; a system is said to have a *software-managed TLB* if the OS is responsible, or a *hardware-managed TLB* if the hardware is responsible. The classic hardware-managed design, as seen in the DEC VAX, GE 645, PowerPC, and Intel ×86 architectures (Clark and Emer 1985, Organick 1972, IBM and Motorola 1993, Intel 1993), provides a hardware state machine to perform this activity; in the event of a TLB miss, the state machine would walk the page table, locate the translation information, insert it into the TLB, and restart the computation. Software-managed designs are seen in the Compaq Alpha, the SGI MIPS processors, and the Sun SPARC architecture (Digital 1994, Kane and Heinrich 1992, Weaver and Germand 1994).

The performance difference between the two is due to the page table lookup and the method of operation. In a hardware-managed TLB a hardware state machine walks the page table; there is no interaction with the instruction cache. By contrast, the software-managed design uses the general

interrupt mechanism to invoke a software TLB miss-handler—a primitive in the operating system usually 10–100 instructions long. If this miss-handler is not in the instruction cache at the time of the TLB miss-exception, the time to handle the miss can be much longer than in the hardware-walked scheme. In addition, the use of the general-purpose interrupt mechanism adds a number of cycles to the cost by draining the pipeline and flushing a possibly large number of instructions from the reorder buffer; this can add up to something on the order of 100 cycles. This is an overhead that the hardware-managed TLB does not incur; when hardware walks the page table, the pipeline is not flushed, and in some designs (notably the Pentium Pro (Upton 1997)), the pipeline keeps processing, in parallel with the TLB miss-handler, those instructions that are not dependent on the one that caused the TLB miss. The benefit of the software-managed TLB design is that it allows the operating system to choose any organization for the page table, while the hardware-managed scheme defines an organization for the operating system. If TLB misses are infrequent, the flexibility afforded by the software-managed scheme can outweigh the potentially higher per-miss cost of the design. For the interested reader, a survey of hardware mechanisms is provided in (Jacob and Mudge 1998b), and a performance comparison of different hardware/operating system combinations is provided in (Jacob and Mudge 1998c).

Lastly, to put modern implementations in perspective, note that TLBs are not a necessary component for virtual memory, though they are used in every contemporary general-purpose processor. Virtually addressed caches would suffice because they are indexed by the virtual address directly, requiring address translation only on the (hopefully) infrequent cache miss. Such a scheme is detailed and evaluated in (Jacob and Mudge 2001).

## References

D.W. Clark and J.S. Emer. "Performance of the VAX-11/780 translation buffer: Simulation and measurement." *ACM Transactions on Computer Systems*, 3(1), 1985.

Digital. *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual*, Digital Equipment Corporation, Maynard, MA, 1994.

IBM and Motorola. *PowerPC 601 RISC Microprocessor User's Manual*. IBM Microelectronics and Motorola, 1993.

Intel. *Pentium Processor User's Manual*. Intel Corporation, Mt. Prospect, IL, 1993.

Bruce Jacob and Trevor Mudge. "Virtual memory: Issues of implementation." *IEEE Computer*, 31(6), pp. 33–43, June 1998a. <http://www.ece.umd.edu/~blj/papers/computer31-6.pdf>.

Bruce Jacob and Trevor Mudge. "Virtual memory in contemporary microprocessors." *IEEE Micro*, 18(4), pp. 60–75, July/August 1998b. <http://www.ece.umd.edu/~blj/papers/micro18-4.pdf>.

Bruce Jacob and Trevor Mudge. "A look at several memory management units, TLB-refill mechanisms, and page table organizations." In *Proc. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS'98*), pp. 295–306. San Jose, CA, October 1998c.

Bruce Jacob and Trevor Mudge. "Uniprocessor virtual memory without TLBs." *IEEE Transactions on Computers*, 50(5), May 2001. <http://www.ece.umd.edu/~blj/papers/ieeetc50-5.pdf>.

G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Reading, MA, 1989.

E.I. Organick. *The Multics System: An Examination of its Structure*. The MIT Press, Cambridge, MA, 1972.

M. Upton. *Personal communication*. 1997.

D.L. Weaver and T. Germand, editors. *The SPARC Architecture Manual version 9*. PTR Prentice-Hall, Englewood Cliffs, NJ, 1994.

# 1.7 Architectures for Public-Key Cryptography

*Lejla Batina, Kazuo Sakiyama, and Ingrid Verbauwhede*

## 1.7.1 Introduction

The importance of security keeps growing because our ever-increasing dependence on information. Numerous examples of security applications are present in everyday life such as purchasing goods over Internet, secure e-mail exchange, online banking, mobile phone communication, medical applications, etc. More recent applications envision security protocols running even on RFID tags and sensor nodes. For all these applications, there exists a range of algorithms that can provide basic cryptographic services: confidentiality, data integrity, authentication, and nonrepudiation [MOV97]. Cryptographic algorithms include stream and block ciphers, hash functions, digital signatures, public-key algorithms, etc. These algorithms are usually divided into secret-key and public-key algorithms. Stream and block ciphers are examples of the former and they allow for a fast encryption of a large amount of data. However, effective information protection against eavesdropping and modifications in open systems as well as advanced cryptographic service, e.g., digital signatures and key exchange, can only be achieved using public-key algorithms.

The foundations of cryptography originate from Claude Shannon [Sha48] and the basic model is as follows. Alice and Bob (or any two parties) want to exchange messages over an insecure channel in such a way that an adversary Eve is not able to learn the contents of their communication (Fig. 1.30). For that purpose they use a secret key that was a priori exchanged. In modern cryptography, Kerckhoffs's principle is assumed, which states that only the secret key $k$ is not known to an adversary. This rule was established already in the nineteenth century by A. Kerckhoffs.

In this system, a user Alice wants to send a message $m$ to Bob, which is called the *plaintext*. It can be any element of a finite set of messages and here we assume that it was converted to a bit string. Alice is using the secret key $k$ to encrypt the message by an injective mapping $E_k$ to a string $c$, which is called the *ciphertext c*. Therefore, one can write $E_k(m) = c$ and this mapping $E_k$ is called the *encryption* operation. Since $E_k$ is injection, the inverse of it exists, i.e., the mapping $D_k$ which is called the *decryption* operation. The same key $k$ will be used by Bob for decryption of $c$, i.e., one can write $D_k(c) = D_k(E_k(m)) = m$.

This system is the model for symmetric-key (or secret-key) cryptography. This scheme for two parties, who want to communicate securely, is based on a shared secret key. Although symmetric cryptosystems allow for large amounts of data to be transferred efficiently, key management and key distribution problems do not scale well in the case of a large number of users.

### 1.7.1.1 History of Public-Key Cryptography

Diffie and Hellman introduced the idea of public-key cryptography [DH76] in the mid 1970s. They showed that one can eliminate the need for prior agreement of a key, that is, an evident limiting factor in

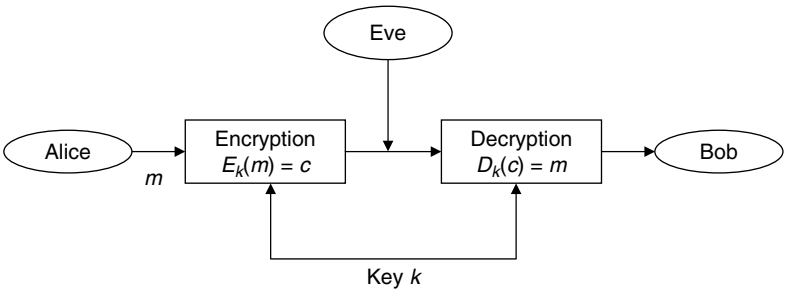



**FIGURE 1.30** Basic model for a cryptosystem.

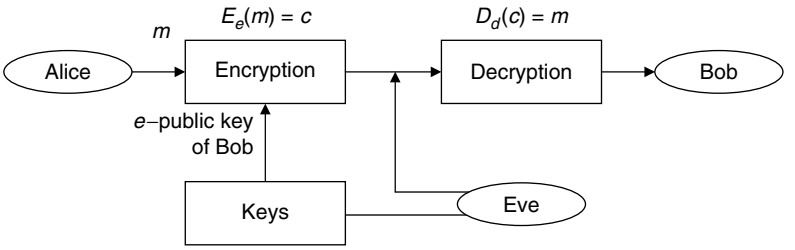



**FIGURE 1.31** A public-key encryption algorithm.

the setting of private-key cryptography. The system is shown in Fig. 1.31. There exists a pair of keys ($E_e$, $D_d$) for each user instead of the unique key that all of them should own.

Here $E$ and $D$ are, respectively, the encryption and decryption mappings and $e$ and $d$ are called the public and the secret key, respectively. The pair ($E_e$, $D_d$) should be easy to generate. In order to achieve secret communication, the condition, $D_d\,(c) = D_d\,(E_e(m)) = m$, is required and it is hard to derive $d$ from $e$. The setting of a public-key cryptosystem also allows for the digital signatures; they were introduced by Diffie and Hellman to uniquely bind a message to the sender. Until today numerous public-key cryptosystems have been proposed. Most of the schemes used today base their security on a small number of mathematical problems. The best-known and most commonly used public-key cryptosystems are based on factoring (RSA) and on the discrete logarithm problem in a large prime field (Diffie-Hellman, ElGamal, Schnorr, DSA) [MOV97]. The RSA public-key cryptosystem is named after its inventors Rivest, Shamir, and Adelman [RSA78]. Elliptic curve cryptography (ECC), which was proposed in the mid 1980s by Miller [Mil85] and Koblitz [Kob87], is based on a different algebraic structure, i.e., on an abelian group of points on an elliptic curve.

#### 1.7.1.2 Applications from High-End to Extremely Constrained Devices (RFIDs and Sensor Nodes)

Public-key cryptosystems (PKC) are present today in almost all spheres of digital communication, e.g., for financial, governmental, and medical applications; they form an essential building block for network security protocols (e.g., SSL/TLS, IPsec, SSH). These are often implemented on general-purpose computers or high-end custom chips where the throughput is the function to optimize. However, more recent embedded security applications such as mobile phones, PDAs, consumer electronics, automotive, and wireless applications imply much more challenging design tasks for PKC implementations. In all these cases, there are firm constraints on area, power, energy, and so on. Therefore, pervasive security is posing difficult demands on cryptography engineering. Especially extreme constraints as imposed by RFID technology and sensor networks present open problems for cryptographic protocols as well as implementations.

#### 1.7.1.3 Various Architectural Options for PKC

Algorithms for RSA and ECC are based on arithmetic in finite fields. RSA algorithm requires only arithmetic in an integer ring where all operations are modulo $N$, where $N$ is an integer (and a product of two large prime numbers) at least 1024 bits long for security applications nowadays. On the other hand, ECC exists over a prime and a binary field where elements are at least 160 bits long.

More precisely, recommended key-lengths for RSA are at least 1024 bits currently. This estimate depends on the difficulty to factor integers and the current progress of factorization efforts, e.g., NFSNET (see www.nfsnet.org). The security of 1024-bit RSA is usually compared to 160-bit ECC and with 80 bit of a symmetric key algorithm such as AES. However, Lenstra and Verheul estimated that with respect to computationally equivalent security, 1024- and 1375-bit RSA are comparable to 139- and

**TABLE 1.1** Comparison of the Key-Lengths for RSA and ECC

| Security Level $n$ Symm. Key Algorithms, e.g., AES | RSA | ECC |
|---|---|---|
| 80 | 1248 | 160 |
| 112 | 2432 | 224 |
| 128 | 3248 | 256 |

160-bit ECC, respectively [LV00]. On the other hand, for cost equivalent security they suggested slightly different corresponding bit-lengths. Within the ECRYPT project—European Network of Excellence for Cryptology [ECRYPT-AZT] the numbers as in Table 1.1 were suggested. Here, the security level $n$ means that $\Theta(2^n)$ operations are needed by the best-known algorithms to break the system.

As both RSA and ECC rely on integer arithmetic modulo, a large number, the crucial operation for implementations is modular multiplications. In the case of RSA and ECC over a prime field, the algorithm of Montgomery [Mon85] appears to be the best solution.

An architecture based on Montgomery's algorithm is probably the best-studied architecture in hardware. Differences appeared because of various approaches for avoiding long carry chains.

Most common ways to do so are systolic array and redundant representation, e.g., residue number systems [PP98]. We discuss the former in more detail in the remainder of this chapter.

## 1.7.2 RSA Algorithm

The private key of a user consists of two large primes $p$ and $q$ and an exponent $d$. The public key consists of a pair $(N, \mathrm{e})$, where $N = p \cdot q$ is the modulus (at least 1024 bits) and an exponent e is such that $\mathrm{e} = d^{-1} \bmod \lambda(N)$. Here, we denote $\lambda(N) = \mathrm{lcm}\,(p-1, q-1)$, where lcm $(a, b)$ is the least common multiple of $a$ and $b$. The corresponding $p$, $q$, and $d$ are kept secret. To encrypt a message $M$, the user Alice computes

$$C = M^e \bmod N$$

and decryption is described by

$$M = C^d \bmod N \equiv M^{(1+k\varphi(N))} \equiv M \bmod N$$

The previous equality follows by Fermat's theorem [Kob94] and the fact that $\lambda$ is a divisor of $\varphi(N) = (p-1)(q-1)$. The RSA function is the modular exponentiation with the public exponent $e$ and the private exponent $d$ is referred to as the trapdoor to invert the function.

Hence, modular exponentiation and also modular multiplication are the most important operations, which have to be considered in detail.

### 1.7.2.1 The RSA Problem and Integer Factoring Problem

*The RSA problem:* Consider a positive integer $N$ (that is a product of two distinct primes $p$ and $q$), a positive integer $e$ such that $\gcd(e, \lambda(N)) = 1$, and an integer $C$; find $M$ such that $M^e \equiv C \bmod N$. So, the RSA problem is the problem of finding $e$th roots modulo a composite number $N$. It is related to the integer factoring problem, in this case, the problem of factoring a composite number $N$, which is the product of two large primes, $p$ and $q$. It can be shown that if the factors of $N$ are known, the RSA problem can be easily solved [MOV97]. The security of the RSA cryptosystem is based on the difficulty of the RSA problem. It is still the most popular cryptosystem, especially for high-end devices that are typically used in e-commerce and virtual private network (VPN) servers.

### 1.7.2.2 Chinese Remainder Theorem (CRT)

By means of the Chinese remainder theorem (CRT), the speed for the RSA decryption scheme can be increased up to four times (Koblitz [Kob94]). This possibility is very attractive in practical applications

especially for hardware implementations. Use of CRT for RSA was proposed in 1982 by Quisquater and Couvreur [QC82].

If the factors of $N$, i.e., $p$ and $q$ are known to Bob, he can compute modular operations with moduli $p$ and $q$ instead of $N$. He computes $M_p \equiv C_1^{\,d} \bmod p$ and $M_q \equiv C_2^{\,d} \bmod q$, (where $C_1 \equiv C \bmod p$ and $C_2 \equiv C \bmod q$. All these calculations are performed modulo integers $p$ and $q$ that are typically half the length of $N$. The original message $M$ is recovered as the linear combination of $M_p$ and $M_q$. The methods to reconstruct the message $M$ are known in the literature as the algorithms of Gauss and Garner [MOV97]. These computations can be performed in $\Theta([\lg n]^2)$ bit operations. (Here, lg denotes the base 2 logarithm.) Altogether, this way of decryption can reduce the workload by a factor of four if cubic complexity of exponentiation is assumed. For hardware implementations that results in a substantial speed-up in performance in the case where two multiplication units are available. More precisely, increase of the area with a factor of two can result in the speed-up in performance of a factor 4.

### 1.7.2.3 RSA Operations

In Fig. 1.32, the structure of operations required for any RSA protocol is depicted. The basic building block consists of modular exponentiation that is based on a number of modular multiplications and squarings. On the bottom level are modular addition, subtraction, and inversion. We remind the reader that all calculations are performed either modulo the composite RSA modulus $N$, or modulo some prime ($p$ or $q$, in the case of CRT). We explain the operations and their realizations in hardware in more detail.

#### *1.7.2.3.1 Modular Exponentiation*

The dominant cost operation in the RSA cryptosystem is modular exponentiation, namely computing $M^e \bmod N$. The basic technique for exponentiation is based on repeated squaring and multiplications (see Knuth [Knu98], p. 461). In [MOV97], this method is called left-to-right binary exponentiation (Algorithm 1). An exponent $e$ is given here in the MSB form and by the radix 2 representation. A similar algorithm is also used for point/divisor multiplication in ECC. In this case the analogous scheme is called double-and-add or the binary method (Algorithm 2) [BSS99].

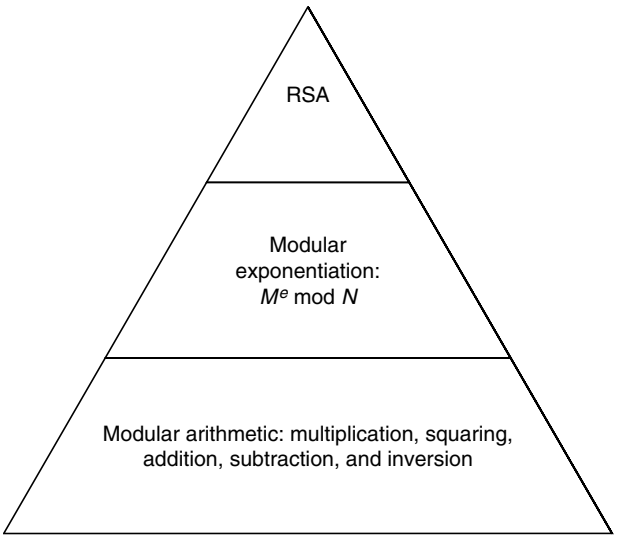



**FIGURE 1.32** Hierarchy of RSA operations.

**Algorithm 1: Modular exponentiation**
Input: $0 \leq M < N$ and $0 < e < N$, $e = (e_{t-1}, \ldots, e_1, e_0)$, $e_i \in \{0,1\}$, $e_{t-1} = 1$, and $N$
Output: $R = M^e \bmod N$

1. $R \leftarrow M$
2. for $i$ from $t-2$ down to 0 do
3. $R \leftarrow R \cdot R \bmod N$
4. if $e_i = 1$ then $R \leftarrow R \cdot M \bmod N$
5. end for
6. return $R$

**Algorithm 2: Point multiplication (binary method)**
Input: A point $P$, a $t$-bit integer $k = (k_{t-1}, \ldots, k_1, k_0)$, $k_i \in \{0,1\}$
Output: $Q = kP$

1. $Q \leftarrow \infty$
2. for $i$ from $t-1$ down to 0 do
3. $Q \leftarrow 2Q$
4. if $k_i = 1$ then $Q \leftarrow Q + P$
5. end for
6. return $Q$

Numerous methods for speeding-up exponentiation and scalar multiplication have been proposed in the literature; for a survey, see Gordon [Gor98]. Recently, side-channel security is also considered to be an important factor for the choice of a suitable exponentiation algorithm. As this became an important research area in the last decade that is closely related to implementations, we explain about side-channel attacks in more detail.

In general, attacks on cryptography can be divided into two groups: mathematical attacks (more traditional type of attacks that are usually purely theoretical) and implementation attacks (more practical type that pose a growing threat today). Implementation attacks exploit weaknesses in specific implementations of a cryptographic algorithm. Sensitive information, such as secret keys or a plaintext can be obtained by observing the time consumed, the power consumption, the electromagnetic radiation, etc. This class of attacks is called side-channel attacks. In 1996, Paul Kocher introduced the concept of timing attacks by showing that secret information can be extracted through measurements of the execution time of cryptographic algorithms [Koc96]. Timing attacks are applicable to all implementations that have a nonconstant execution time, which depends on the bits of the secret key. Two years later, Kocher et al. performed successful attacks by measuring the power consumption while the cryptographic circuit is executing the implemented algorithm [KJJ99]. For example, conditional operations that are key-dependent (such as step 4 in Algorithms 1 and 2) can leak bits of the secret key by merely observing power consumption graphs of algorithms being performed. It is evident that constant time-implementations should remove all vulnerabilities of cryptographic applications with respect to timing attacks, and algorithms should always perform the same sequence of operations to counteract simple side-channel attacks.

#### *1.7.2.3.2 Montgomery's Arithmetic*

Modular multiplication forms the basis of modular exponentiation, which is the core operation of the RSA cryptosystem. It is also present in many other cryptographic algorithms, including those based

on ECC. The most popular algorithm for modular multiplication is Montgomery's method [Mon85]. The approach of Montgomery avoids the time-consuming trial division, the common bottleneck of other algorithms. Montgomery's algorithm is especially suitable for hardware implementations because the division with a large number (the modulus or some prime) is replaced by reduction with a power of 2.

We give here all details for Montgomery's arithmetic as commonly used for RSA implementations. Let $N$ be a modulus. For a word base $b = 2^r$, the Montgomery radix (or parameter) $R$ is typically chosen such that $R = (2^r)^n > N$. Let $x$ be an odd integer represented by its radix $b$ representation $x = \sum_{i=0}^{n-1} x_i b^i$. There is a one-to-one correspondence between each $x$ and its representation $X = xR \bmod N$. This representation is usually referred to as the Montgomery representation. Addition and subtraction of two elements in Montgomery representation is again an element in Montgomery representation. For efficient implementation of modular multiplication, the crucial operation is modular reduction, which is replaced by reduction by a number that is a power of 2, as previously mentioned.

In the original algorithm of Montgomery, the requirements are given on the parameters $R$ and $N'$ such that $R > N$ and $R^{-1}$ and $N'$ are satisfying $0 < R^{-1} < N$, $0 < N' < R$ and $RR^{-1} - NN' = 1$. For the computation of the Montgomery product $T = XYR^{-1} \bmod N$, Algorithm 3 was proposed by Montgomery [MOV97].

**Algorithm 3: Montgomery's modular multiplication**
Input: $N$, $N' = -N^{-1} \bmod 2^r$, $X = (x_{n-1} \ldots x_1 x_0)_{2^r}$, $Y = (y_{n-1} \ldots y_1 y_0)_{2^r}$ with $0 < X$, $Y < N$, $R = 2^{rn}$ $\gcd(N, 2) = 1$
Output: $T = XYR^{-1} \bmod N$

1. $T = 0$
2. for $i = 0$ up to $n - 1$ do
3. $m_i = (t_0 + x_i y_0)\ N' \bmod 2^r$
4. $T = (T + x_i Y + m_i N)/2^r$
5. end for
6. if $(T > N)$ then $T = T - N$
7. return $T$

In the original algorithm of Montgomery, a modular reduction is needed in step 6. The reason is in inputs being bounded by $N$, e.g., $X$, $Y < N$ and the output $T$ was bounded by $2N$, so $T < 2\ N$. Hence, if $T > N$, $N$ must be subtracted so that the output can be used as input to the next multiplication. This extra reduction slows down modular exponentiation and it also introduces a vulnerability to side-channel attacks. To avoid this subtraction, a bound for $R$ is given by Walter [Wal02] such that for inputs $X$, $Y < 2\ N$ also the output is bounded: $T < 2\ N$.

One possible way to calculate the Montgomery's modular multiplication (MMM) is to use a digit-serial multiplier. The corresponding idea is given by Algorithm 4 (bit-serial version). It computes bit-serial MMM with only additions and right-shift operations without the final subtraction. As shown in Fig. 1.33, during multiplication of $X$ and $Y$, modulus $N$ is added to the intermediate product of $XY$ so that the LSB becomes 0, which allows for division with 2 (i.e., the right-shift operation).

**Algorithm 4: Bit-serial Montgomery's modular multiplication**
Input: A $k$-bit integer $N$, $X = (x_k \ldots x_1 x_0)_2$, $Y = (y_k \ldots y_1 y_0)_2$ with $0 < X$, $Y < 2N - 1$, $R = 2^{k+2}$, $\gcd(N,2) = 1$
Output: $T = XYR^{-\mathbf{1}} \bmod N$

1. $T = 0$
2. for $i = 0$ up to $k + 2$ do

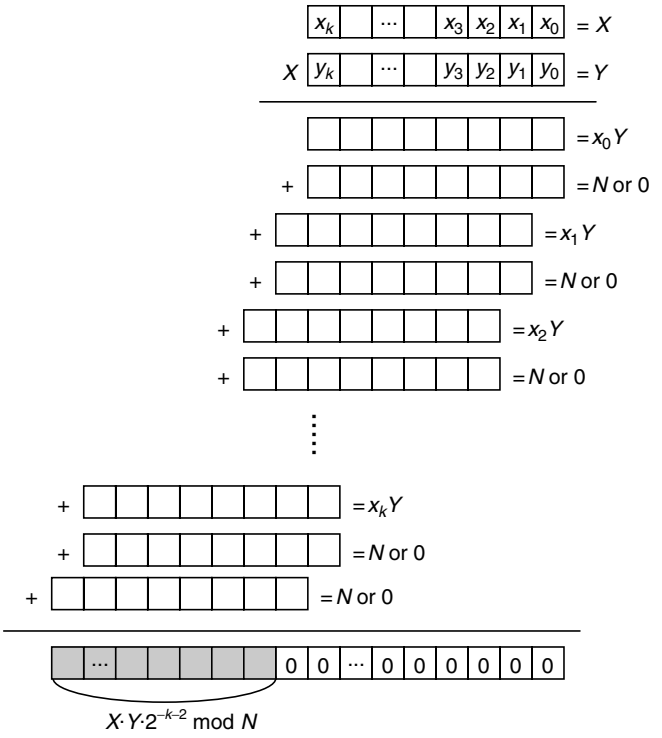



**FIGURE 1.33** Computation flow of the bit-serial Montgomery's modular multiplication.

3. $m_i = t_0 + x_i y_0$
4. $T = (T + x_i y + m_i N)/2$
5. end for
6. return $T$

Step 4 is the most critical computation in Algorithm 4, and it can be implemented with adders and 1-bit right-shift logic (Fig. 1.34). The adder can be implemented with a carry-save adder (CSA) avoid long carry propagation (Fig. 1.35). We explain more about ways to implement modular addition below.

For further speedup, one can use a higher radix instead of the radix 2. In that case an $r \times r$-bit multiplier is used, where $r$ is an arbitrary power of 2.

#### *1.7.2.3.3 Modular Addition and Subtraction*

Modular addition and subtraction are usually performed as in Algorithm 5 and Algorithm 6, respectively [Koc95].

**Algorithm 5: Modular addition**

Input: Integers $A$ and $B$ and modulus $N$
Output: $C = A + B \bmod N$

1. $S' = A + B$
2. $S'' = S' - N$
3. if $S'' < 0$, then $C = S'$
4. else $C = S''$
5. return $C$

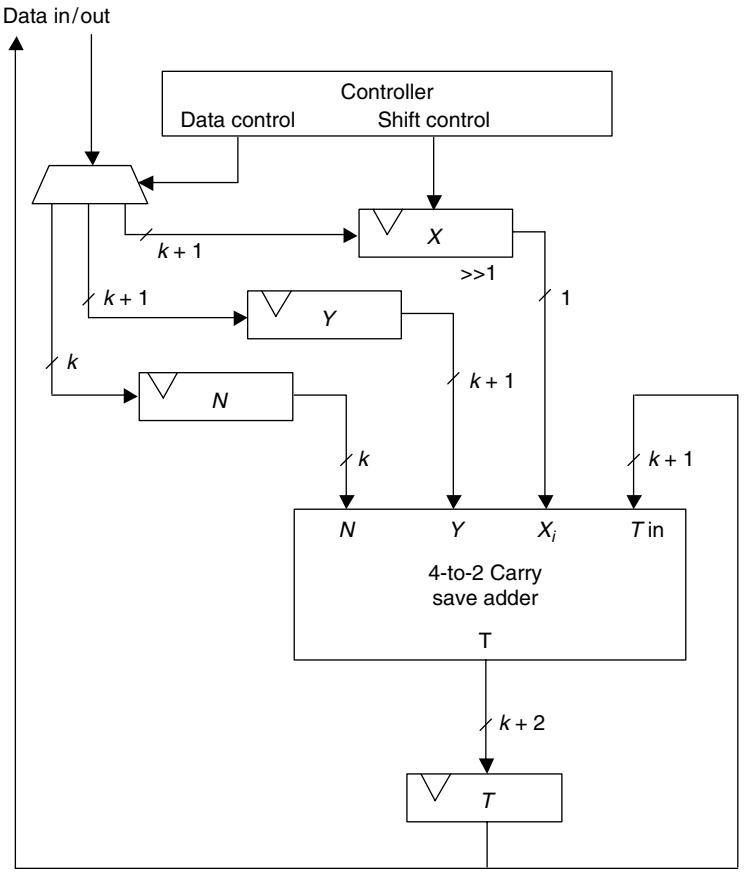



**FIGURE 1.34** Schematic representation of the hardware block that performs MMM.

**Algorithm 6: Modular subtraction**
Input: Integers $A$ and $B$ and modulus $N$
Output: $C = A - B \bmod N$

1. $S' = A - B$
2. $S'' = S' + N$

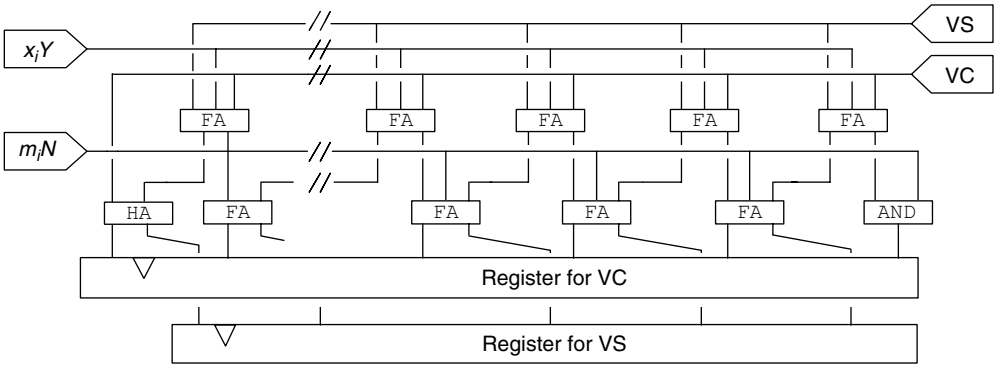



**FIGURE 1.35** Four-to-two CSA-based MMM corresponding to Algorithm 4. The intermediate result, $T$ is represented in carry-save form with VS and VC ($T$ = VS + VC).

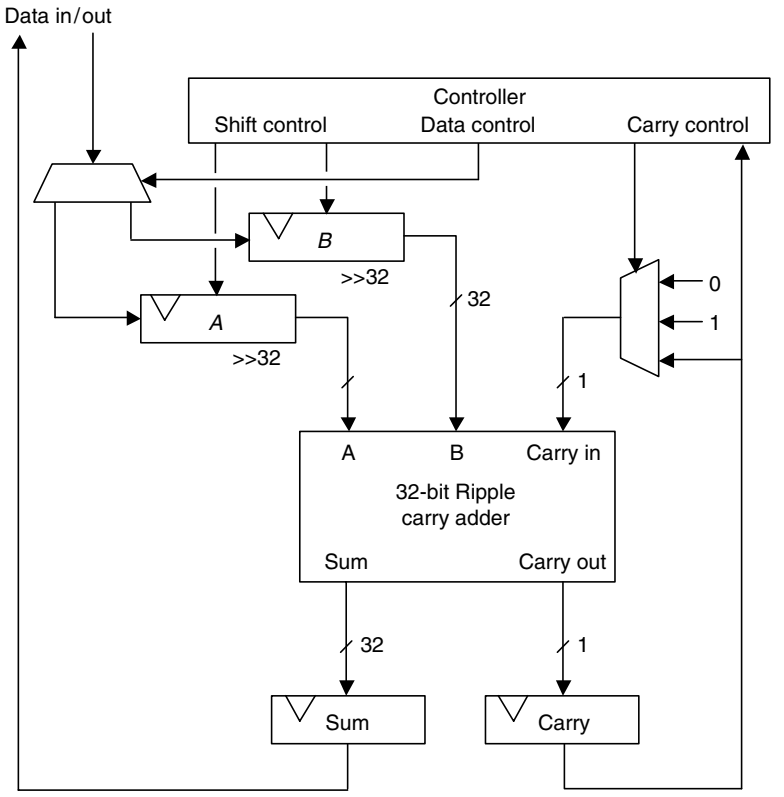



**FIGURE 1.36** Schematic representation of the hardware block that performs modular addition and subtraction.

3. if $S' < 0$, then $C = S''$
4. else $C = S'$
5. return $C$

The numbers are represented in two's complement representation and in this way both the addition and subtraction operation can be combined into one circuit as explained by Mano and Kime [MK01]. One can use a serial adder/subtractor that consists of one full adder, two shift-registers (one for $A$ and $C$ and the other for $B$), one flip-flop (for a carry bit), a counter, and a controller (Fig. 1.36). In this figure, a digit-serial addition is shown that calculates a 32-bit addition by means of ripple carry adder (RCA) [MK01].

### 1.7.2.4 Hardware Architectures for RSA

Soon after its invention, the first proposals for RSA hardware implementations appeared and different architectures were proposed in the past two decades. The systolic array architecture still appears to be the best solution for modular multiplication with very long integers. This architecture has been studied intensively, both from a theoretical and a practical viewpoint.

### 1.7.2.5 Systolic Array Architectures

A systolic array is typically defined as a grid-like structure of special processing elements (PEs) that processes data like an $n$-dimensional pipeline (see Johnson et al. [JHS93]). Each line indicates a communication path and each intersection represents a cell or a systolic element.

The main advantage of this architecture is that it can easily be scaled. Scalability is one of the most important requirements of cryptographic applications nowadays. This results in increased flexibility especially when implemented on FPGA platforms. According to Tenca and Koç [TK99], an arithmetic unit is called scalable if the unit can be reused or replicated in order to generate long-precision results independently of the data path precision for which the unit was originally designed. More precisely, the longest path should be "short" and independent of operands' length and designed such that it fits even in restricted hardware regions [GTK02]. This means that the arithmetic unit can handle arbitrary bit-lengths with the exception of memory limitations. The number of clock cycles per operation depends only on the actual size of the operands. A typical scalable architecture based on a systolic array implementing Montgomery multiplication is shown in Fig. 1.37 [BM02].

The design shows a large number arithmetic unit (LNAU), which is designed as a systolic array. If two such units are available CRT computation can be performed fully in parallel. This array is one dimensional and consists of a fixed number of PEs. A FIFO memory is added to the design to achieve scalability. A PE contains some adders and multipliers that can process $\alpha$ bits of $X$, and $\beta$ bits of $Y$ in one clock cycle. So, in one clock cycle a number of additions and multiplications can be performed, e.g., to execute step 4 in Algorithm 4. More precisely, in each PE, within this architecture, one loop of the Montgomery's algorithm can be performed in one cycle. The architecture shown in Fig. 1.37 is also scalable according to the previous definition. If the operands are too large to fit in the available number of PEs the intermediate result of the last PE is fed into the first PE. These intermediate results are temporarily stored in a FIFO memory, if necessary. The operands of the multiplication are divided into words. The words of $X$ are divided over the PEs. When there are not enough PEs more rounds are needed. In one round, each word of $Y$ has to pass all PEs. Each PE calculation takes one clock cycle within which is computed $\frac{T_j + x_i y_j + m_i N_j}{2^\alpha}$. So, in $P$ clock cycles a word of $Y$ passes all PEs in the array. Here $P$ denotes the number of PEs. When the number of words of $Y$ is larger than the number of PEs the FIFO memory is used to store the intermediate results of the last PE.

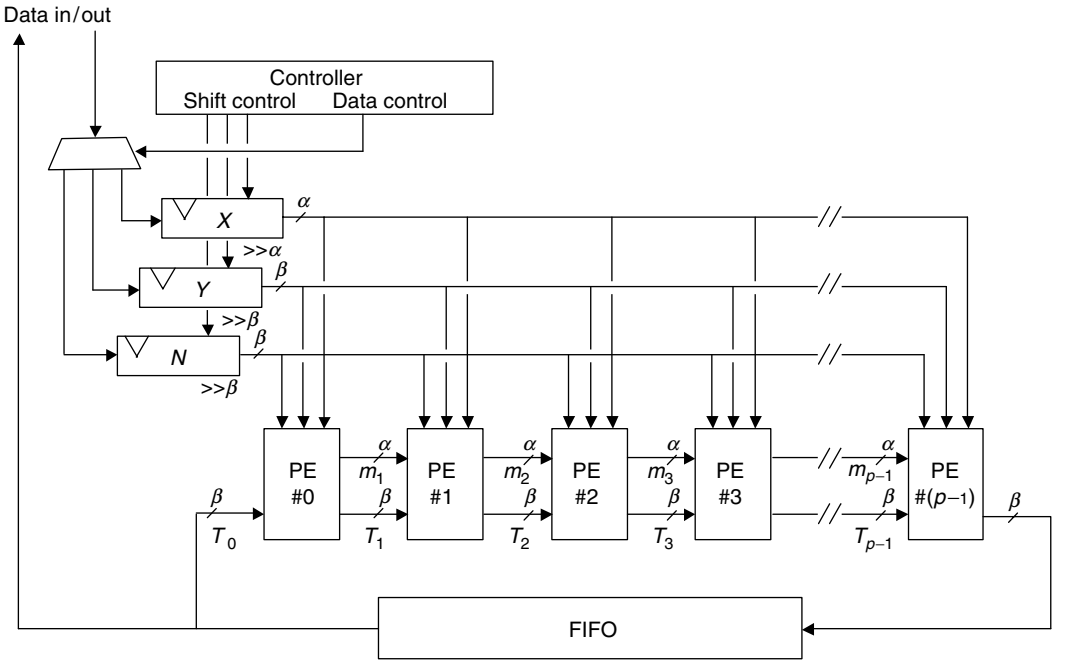



**FIGURE 1.37** Example of a systolic array.

Other related work on systolic array architectures for modular multiplication includes the work of Iwamura et al. [IMI94] and of Eldridge and Walter [EW93].

## 1.7.3 Elliptic Curve Cryptography

Another important public-key cryptosystem is the one based on elliptic curves. It is important to point out that ECC offers equivalent security as RSA for much smaller key sizes as mentioned previously. Other benefits include higher speed, lower power consumption, and smaller certificates, which is especially useful in constrained environments.

### 1.7.3.1 Mathematical Background

Most of the public-key algorithms require the structure of an algebraic group. In the case of ECC, the group of points on an elliptic curve is used. Public and private keys are defined as points on a curve and the one-way function is the multiplication of a point with a scalar. Therefore, Alice having a private key as some integer $e_A$ can send Bob a multiple of a point $P$, so $e_AP$, which is also a point on the same curve as $P$ lies on, due to the properties of a group. The problem of finding the logarithm in this group, e.g., finding $e$ for given $P$ and $eP$ is called the elliptic curve discrete logarithm problem (ECDLP). So, nobody can recover the key of Alice $e_A$ in the example above, by knowing $e_AP$ and $P$ due to the difficulty of the ECDLP.

### 1.7.3.2 Elliptic Curves over Finite Fields

For cryptography, we need a finite cyclic group in which the group operation is efficiently computable, but the discrete logarithm problem is very difficult to solve. Elliptic curve groups appear to meet these criteria when the underlying field is finite. Elliptic curves that are used in most applications are defined over $F_q$ with $q = p^m$ where $p$ is a prime number. In standards such as IEEE [IEE99] and ANSI [ANS], fields for $q = p$ and $q = 2^n$ where $p \approx 2^n$ and $n \geq 160$ are recommended. Also, it appears that elliptic curve systems over both prime ($F_p$) and binary ($F_2{}^n$) fields provide the same level of security but fields $F_2{}^n$ have some implementation advantages. Namely, arithmetic in $F_2{}^n$ can be implemented more efficiently than arithmetic in $F_p$, at least on platforms without specialized arithmetic coprocessors. With respect to theoretical security, it is typically recommended to use fields $F_2{}^p$ where $p$ is a prime. The reason is that it was shown that if $p$ is not a prime, ECDLP is sometimes easier than in a general case.

A set of points on an elliptic curve together with the point at infinity, denoted by $\infty$, and with point addition as binary operation has the structure of an Abelian group. The following equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

is called the Weierstrass equation for an elliptic curve. An elliptic curve $E$ is the set of solutions to the Weierstrass equation, together with the extra point at $\infty$.

First we consider finite fields of characteristic two.

A nonsupersingular elliptic curve $E$ over $F_2{}^n$ is defined as the set of solutions $(x,y) \in F_2{}^n \times F_2{}^n$ to the following equation:

$$y^2 + xy = x^3 + ax^2 + b$$

where $a, b \in F_2{}^n$, $b \neq 0$, together with $\infty$.

In the case of a field $F_p$, we get the following equation:

$$y^2 = x^3 + ax^2 + b$$

where $a, b \in F_p$.

### 1.7.3.3 Algorithms for ECC

The main operation in any curve-based primitive is the scalar multiplication. The hierarchical structure for operations required for implementations of curve-based cryptography is given in Fig. 1.38. Point multiplication is at the top level. It can be implemented using Algorithm 1 which is usually called point double-and-add. At the next (lower) level are the point group operations. The lowest level consists of finite field operations such as addition, subtraction, multiplication, and inversion required to perform the group operations.

#### *1.7.3.3.1 ECC Point Operations in* $\mathbf{F_p}$

When $E$ is a curve defined with the Weierstrass equation, inverse of the point $P=(x_1, y_1)$ is $-P=(x_1, -y_1)$. The sum $P+Q$ of points $P=(x_1, y_1)$ and $Q=(x_2, y_2)$ (assume that $P, Q \neq \infty$) is point $R=(x_3, y_3)$ where

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$
$$\lambda = \begin{Bmatrix} \frac{y_2 - y_1}{x_2 - x_1} & \text{if} & P \neq Q \\ \frac{3x_1^2 + a}{2y_1} & \text{if} & P = Q \end{Bmatrix}$$

The point at $\infty$ plays a role analogous to that of the number 0 in ordinary addition. Thus, $P+\infty = P$ and $P+(-P)=\infty$ for all points $P$. As mentioned above, this operation is an abelian group and Fig. 1.39 shows the group law for the case of an elliptic curve over the set of real numbers. Here $P$ and $Q$ are arbitrary two points on an elliptic curve. Let $p$ be the line through $P$ and $Q$ and $-R$ is the third point on $p$. The sum of $P$ and $Q$ is defined as the point $R$ that is the mirror of $-R$ with respect to $x$-axis. For a point double operation of a given point $P$, one has to draw the tangent line in $P$.

There are many types of coordinates in which an elliptic curve may be represented. In the equations above, affine coordinates are used but the so-called projective coordinates have some implementation advantages. The main advantage is that point addition can be done in projective coordinates using only field multiplications, with no inversions required. Thus, inversions become almost irrelevant as only one inversion needs to be performed at the end of a point multiplication operation. A projective point $(X, Y, Z)$ on the curve satisfies the homogeneous Weierstrass equation:

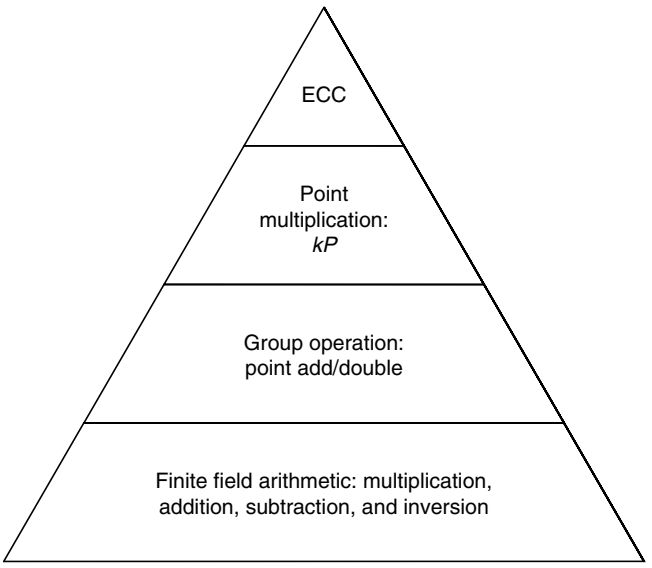



**FIGURE 1.38** Hierarchical structure of ECC.

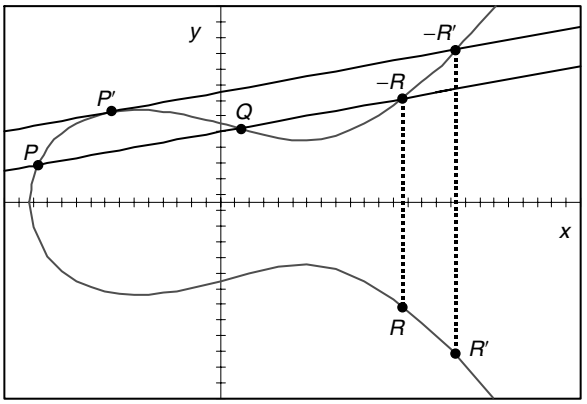



**FIGURE 1.39** Group law for an elliptic curve over the set of real numbers *R*.

$$Y^2Z = X^3 + aX^2 + bZ^3$$

and, when $Z \neq 0$, it corresponds to the affine point $(X/Z, Y/Z)$. It was shown that other projective representations result in more efficient implementations of the group operation [CMO98]. In particular, a weighted projective representation (also referred to as Jacobian representation) is preferred in the sense of faster arithmetic on elliptic curves [BSS99, IEEE]. In this representation, a triplet $(X, Y, Z)$ corresponds to the affine coordinates $(x,y) = (X/Z^2, Y/Z^3)$ for $Z \neq 0$. In this case, we have a weighted projective curve equation of the form:

$$Y^2 = X^3 + aXZ^4 + bZ^6$$

Weighted projective coordinates provide faster arithmetic than the "normal" projective coordinates. Conversion from projective to affine coordinates costs one inversion $(I)$ and four multiplications $(M)$, while vice versa is trivial. If one implements addition and doubling in a way specified in the IEEE standard [IEEE], the total costs for general addition is $I + 3M$ in affine coordinates and $16M$ in projective coordinates. Here, $I$ and $M$ are denoting the modular inversion and multiplication operations, respectively. In the case of doubling (with $a = p - 3$), this relation is $I + 4M$ in affine coordinates against $8M$ in projective coordinates. Thus, the choice of coordinates is determined by the ratio *I*:*M*. Therefore, multiplication in finite field is the most important operation to focus on when working with projective coordinates. On the other hand, the extra inverter is required for the affine coordinates' representation because one inversion has to be performed for every point operation.

#### *1.7.3.3.2 ECC Point Operations in* $\mathbf{F}_{2^n}$

Here we consider a finite field of characteristic 2, i.e., $F_{2^n}$. For this case, we are only interested in curves that are nonsupersingular. A nonsupersingular elliptic curve $E$ over $F_{2^n}$ is defined as the set of solutions $(x,y) \in F_{2^n} \times F_{2^n}$ of the equation $y^2 + xy = x^3 + ax^2 + b$, where $a, b \in F_{2^n}$, b $\neq 0$, together with $\infty$.

The point addition in affine coordinates is performed according to the following formulae. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points on an elliptic curve *E*. Assume $P_1, P_2 \neq \infty$ and $P_1 \neq -P_2$.

The sum $P_3 = (x_3, y_3) = P_1 + P_2$ is computed as follows [BSS99], p. 57:

If $P_1 \neq P_2$

$$\lambda = \frac{y_2 + y_1}{x_2 + x_1}$$
$$x_3 = \lambda^2 + \lambda + x_1 + x_2$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

If $P_1 = P_2$

$$\lambda = \frac{y_1}{x_1} + x_1$$
$$x_3 = \lambda^2 + \lambda + a$$
$$y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

Projective coordinates can be used also in this case to avoid the inversion in a binary field.

### 1.7.3.4 Finite Field Arithmetic for ECC

#### *1.7.3.4.1 Modular Multiplication and Addition*

Simply reducing the operation size of modular arithmetic, ECC over $F_p$ can be implemented with the same hardware algorithm as discussed for RSA implementations. Therefore, this section focuses on hardware architecture over ${F_2}^n$. The hardware complexity is simpler than modular operations in $F_p$ because binary field arithmetic is carry free.

There are many types of basis in which elements of ${F_2}^n$ can be represented. A usual choice is the polynomial basis. In this basis, the basis elements have the form $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$ where $\alpha$ is a root in ${F_2}^n$ of an irreducible polynomial $f$ of degree $n$ over $F_2$. In this basis, the elements of ${F_2}^n$ are polynomials of degree at most $n-1$ over ${F_2}^n$, and arithmetic is carried out modulo an irreducible polynomial $f$ of degree $n$ over $F_2$. According to this representation, an element of ${F_2}^n$ is a polynomial of length $n$ and can be written as

$$A(x) = \sum_{j=0}^{n-1} a_j x^j, \quad \text{where } a_i \in F_2$$

The standard way to compute the product of two elements in this field is the one that is using convolution (Algorithm 7) [BG89]. On the other hand, addition can be implemented by means of XOR gate.

**Algorithm 7: Bit-serial MSB-first polynomial-basis modular multiplication**

Input: Irreducible polynomial $P(x) = x^n + \sum_{j=0}^{n-1} p_j x^j$, $A(x) = \sum_{j=0}^{n-1} a_j x^j$, $B(x) = \sum_{j=0}^{n-1} b_j x^j$, with $A(x)$, $B(x) \in {F_2}^n$

Output: $T(x) = A(x)B(x) \bmod P(x)$

1. $T(x) = 0$
2. for $i = n - 1$ down to 0 do
3. $T(x) = (T(x) \oplus a_i B(x) \oplus t_n P(x))x$
4. end for
5. return $T(x)$

Algorithm 7 computes bit-serial MSB-first polyomial-basis modular multiplication. Different from the MMM, the irreducible polynomial $P(x)$ is XORed to the intermediate result, so that the MSB (the coefficient of $x^n$ in the polynomial $T(x)$) becomes 0 (Fig. 1.40).

#### *1.7.3.4.2 Modular Inversion*

As observed above for implementations of ECC one has to implement the inversion operation. For curves over prime fields the easiest solution is to use Fermat's theorem [Kob94]. In that case inversion is performed by means of repeated multiplications and squarings. More precisely, if $p$ is a prime, it holds $a^p \equiv a \pmod p$. Furthermore, if $p$ is not a divisor of $a$ we have $a^{p-1} \equiv 1 \pmod p$. This fact is usually

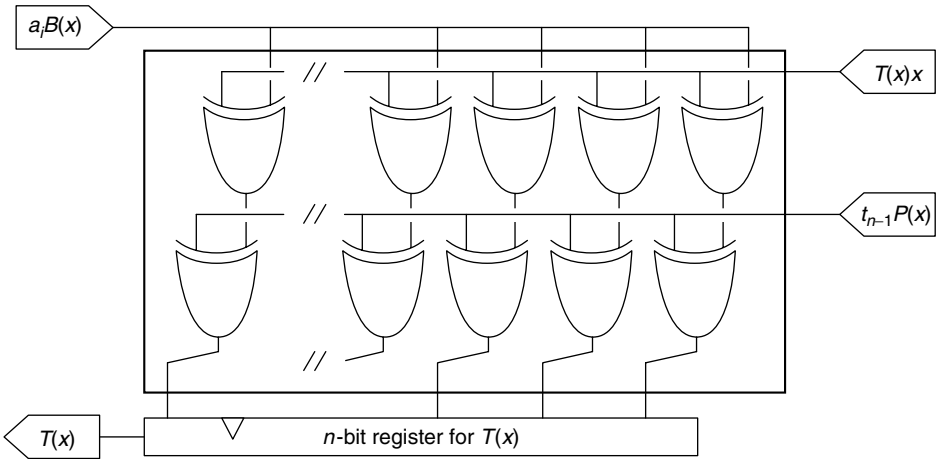



**FIGURE 1.40** Bit-serial MSB-first polynomial-basis modular multiplier.

known as Fermat's little theorem. Then, it follows: $a^{p-2} \equiv a^{-1} \pmod{p}$, which means that one can compute an inverse via a number of modular exponentiations.

In the case of affine coordinates, where many inversions have to be computed (i.e., one for each point operation) a dedicated inverter is often necessary to improve the performance. Some prominent examples from literature include [GTK02], where authors also used the idea of Montgomery. On the other hand, if projective coordinates are deployed, only one inversion is required for the whole point multiplication. In this case, using exponentiation is the most common choice.

For ECC over binary fields, the most efficient algorithm for inversion is extended Euclidean algorithm [MOV97]. The drawback is that it is very difficult to implement it in hardware. Hence, for hardware implementations one also uses Fermat's little theorem, i.e., the following equation $a^{-1} = a^{2n-2}$, for all $a \in F_2{}^n$. The technique to compute this in an optimal way is based on the idea of Itoh and Tsujii [IT88].

### 1.7.4 Architectures Supporting Both RSA and ECC

It can be concluded that elliptic curve cryptosystems also rely on efficient finite field arithmetic, especially on field multiplication. As already mentioned, typical fields are not only prime field as in the case of RSA but also binary fields. The latter is often recommended as the binary fields arithmetic is easier to implement and area and power consumption are smaller than in the case of prime fields. This is believed to be true, but only for platforms where specialized arithmetic coprocessors for finite field arithmetic are not available. On the other hand, an advantage of prime fields is in their suitability for both RSA and ECC with sharing of hardware resources. This trend can be observed mainly for some recent works as ECC has only recently proved its potential and started replacing RSA in some applications. The work of Crowe et al. [CDM05] also proposed a single architecture for RSA and ECC. A hardware optimized version of Montgomery multiplication method is used for modular multiplication. The so-called dual processor could operate in parallel for ECC or in a pipelined series for RSA.

The contribution presented in [BBÖ04] deals with an FPGA implementation of RSA and ECC cryptosystems over a field of prime characteristic. The authors used a systolic array to achieve arbitrary precision in bits; hence easily bridging the gap between the bit-lengths for ECC from 160-bit to 2048-(or higher) bit long moduli for RSA. There exists also some related work on so-called dual field ECC, which deals with processors that can support both type of fields for ECC. Wolkerstorfer [Wol02] proposed the unique arithmetic unit that supports addition and multiplication for prime and binary fields. A scalable dual-field ECC processor is described in the work by Satoh and Takano [ST03]. They proposed a high-speed version of the processor and another, compact one.

### 1.7.5 Concluding Remarks

Owing to not only all the previously mentioned threats but also as a result of various constraints that are imposed by security applications, special care is required when implementing a cryptographic algorithm. Especially, implementations of public-key cryptography present a challenge for most application platforms varying from software to hardware. The reason is that one has to deal with very long numbers (up to 2048 bits) in conditions that can be quite severe in costs, area, and power. Emerging examples are RFID tags and sensor networks. For implementations of cryptographic protocols to achieve various security applications, it is not enough to come up with an efficient implementation but it also has to be secure against side-channel attacks. With respect to this, it is well known that although software platforms offer a cost-effective and flexible solution, only hardware implementations provide a suitable level of security related to side-channel attacks.

## Acknowledgments

Kazuo Sakiyama and Lejla Batina are funded by a research grant from the Katholieke Universiteit (KU) Leuven and Fund for Scientific Research-Flanders (FWO) projects G.0450.04 and G.0475.05. This work was supported in part by the Interuniversity Attraction Pole (IAP) program P6/26 Belgian Fundamental Research on Cryptology and Information Security (BCRYPT) of the Belgian State (Belgian Science Policy), by the European Union Information Society Technologies (EU IST) FP6 projects security for embedded systems on chip (SESOC) and European Network of Excellence for Cryptology (ECRYPT), by KU Leuven, and by the Interdisciplinary Institute for Broadband Technology Quality of Experience (IBBT-QoE) project of the IBBT.

## References

[ANS] ANSI. ANSI X9.62 the elliptic curve digital signature algorithm (ECDSA). http://www.ansi.org.

[BBÖ04] L. Batina, G. Bruin-Muurling, and S.B. Örs. Flexible hardware design for RSA and elliptic curve cryptosystems. In T. Okamoto (Ed.), *Topics in Cryptology-CT-RSA—The Cryptographers' Track at the RSA Conference*, No. 2964 in LNCS, pp. 250–263, Springer-Verlag, Berlin, Heidelberg, 2004.

[BDK98] J.C. Bajard, L.S. Didier, and P. Kornerup. An RNS Montgomery's modular multiplication. *IEEE Transactions on Compute*rs, 19(2): 167–178, 1998.

[BG89] T. Beth and D. Gollmann. Algorithm engineering for public key algorithm, *IEEE Journal on Selected Areas in Communications*, 7(4): 458–465, 1989.

[BM02] L. Batina and G. Muurling. Montgomery in practice: How to do it more efficiently in hardware. In B. Preneel (Ed.), *Topics in Cryptology-CT-RSA—The Cryptographers' Track at the RSA Conference*, No. 2271 in LNCS, pp. 40–52, Springer-Verlag, Berlin, Heidelberg, 2002.

[BSS99] I. Blake, G. Seroussi, and N.P. Smart. *Elliptic Curves in Cryptography*. London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, 1999.

[CDM05] F. Crowe, A. Daly, and W. Marnane. A Scalable Dual Mode Arithmetic Unit for Public Key Cryptosystems, *Proceedings of IEEE International Conference on Information Technology—ITCC'05*, pp. 568–573, 2005.

[CMO98] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei (Eds.), *Proceedings of ASIACRYPT 1998*, No. 1514 in LNCS, pp. 51–65, Springer-Verlag, Berlin, Heidelberg, 1998.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22: 644–654, 1976.

[ECRYPT-AZT] ECRYPT Yearly Report on Algorithms and Keysizes (2004), Document D.SPA.10, available at www.ecrypt.eu.org.

[EW93] S.E. Eldridge and C.D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42: 693–699, 1993.

[Gor98] D.M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27: 129–146, 1998.

[GTK02] A. Gutub, A.F. Tenca, and Ç.K. Koç. Scalable VLSI architecture for GF($p$) Montgomery modular inverse computation, *Proceedings of IEEE Computer Society Annual symposium on VLSI*, pp. 53–58, 2002.

[IEE99] IEEE P1363. Standard specifications for public-key cryptography, 1999.

[IT88] T. Itoh and S. Tsujii. Effective recursive algorithm for computing multiplicative inverses in GF ($2^m$). *Electronics Letters*, 24(6): 334–335, 1988.

[IMI94] K. Iwamura, T. Matsumoto, and H. Imai. Montgomery modular multiplication method and systolic arrays suitable for modular exponentiation. *Electronics and Communications in Japan*, 77(3): 40–50, 1994.

[JHS93] K.T. Johnson, A.R. Hurson, and B. Shirazi. General-purpose systolic arrays. *IEEE Computer*, 26(11): 20–31, 1993.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener (Ed.), *Advances in Cryptology*: *Proceedings of CRYPTO'99*, No. 1666 in LNCS, pp. 388–397, Springer-Verlag, Berlin, Heidelberg, 1999.

[Knu98] D.E. Knuth. *The Art of Computer Programming-Vol. 2—Seminumerical Algorithms*. Addison-Wesley, 3rd ed., 1998.

[Kob87] N. Koblitz. Elliptic curve cryptosystem. *Mathematics of Computation*, 48: 203–209, 1987.

[Koc95] Ç.K. Koç. RSA Hardware implementation, Technical Report, RSA Laboratories, 1995.

[Kob94] N. Koblitz. *A Course in Number Theory and Cryptography*, *Graduate Text in Mathematics*, Vol. 114, 2nd ed., Springer-Verlag, Berlin, Heidelberg, New York, 1994.

[Koc96] P. Kocher. Timing attack on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz (Ed.), *Advances in Cryptology: Proceedings of CRYPTO'96*, No. 1109 in LNCS, pp. 104–113, Springer-Verlag, Berlin, Heidelberg, 1996.

[LV00] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. In H. Imai and Y. Zheng (Eds.), *Proceedings of Third International Workshop on Practice and Theory in Public Key Cryptography* (*PKC 2000*), No. 1751 in LNCS, pp. 446–465, Springer-Verlag, Berlin, Heidelberg, 2000.

[MK01] M.M. Mano and C.R. Kime. *Logic and Computer Design Fundamentals*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 2001.

[Mil85] V. Miller. Uses of elliptic curves in cryptography, In H.C. Williams (Ed.), *Advances in Cryptology: Proceedings of CRYPTO'85*, No. 218 in LNCS, pp. 417–426, Springer-Verlag, Berlin, Heidelberg, 1985.

[Mon85] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170): 519–521, 1985.

[MOV97] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997.

[PP98] K.C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5): 449–454, 1998.

[QC82] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18: 905–907, 1982.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2): 120–126, 1978.

[Sha48] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27: 379–423, 1948. Reprinted with corrections.

[ST03] A. Satoh and K. Takano. A scalable dual-field elliptic curve cryptographic processor. In C. Paar and C.K. Koc (Eds.), *IEEE Transactions on Computers*, 52: 449–460, 2003, Special Issue on Cryptographic Hardware and Embedded Systems.

[TK99] A.F. Tenca and Ç.K. Koç. A scalable architecture for Montgomery multiplication. In C.K. Koc and C. Paar (Eds.), *Proceedings of 1st International Workshop on Cryptographic Hardware and*

*Embedded Systems (CHES)*, No. 1717 in LNCS, pp. 94–108, Springer-Verlag, Berlin, Heidelberg, 1999.

[Wal02] C.D. Walter. Precise bounds for Montgomery modular multiplication and some potentially insecure RSA moduli. In B. Preneel (Ed.), *In Topics in Cryptology-CT-RSA—The Cryptographers' Track at the RSA Conference*, No. 2271 in LNCS, pp. 30–39, Springer-Verlag, Berlin, Heidelberg, 2002.

[Wol02] J. Wolkerstorfer. Dual-field arithmetic unit for GF ($p$) and GF($2^m$). In B.S. Kaliski Jr., C. Koc, and C. Paar (Eds.), *Proceedings of 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, No. 2523 in LNCS, Springer-Verlag, Berlin, Heidelberg, 2002.

# 2

# System Design




Mark Smotherman
*Clemson University*

Dezsö Sima
*Budapest Polytechnic*

Kevin Skadron
David Tarjan
*University of Virginia*

Tzi-cker Chiueh
*State University of New York at Stony Brook*

Binu Mathew
*Apple Inc.*

Ali Ibrahim
*Advanced Micro Devices*


## 2.1 Superscalar Processors

*Mark Smotherman*

### 2.1.1 Introduction

A superscalar processor is designed to fetch, decode, execute, and complete multiple instructions each clock cycle from a single instruction stream. The term superscalar originated in the 1980s to distinguish it as a form of parallelism set apart from vector and array processors as well as an advance beyond pipelined scalar processors. The rationale for such a design can be illustrated by considering the basic computer performance equation, IC × CPI × CCT, that is, execution time is a function of the instruction count (IC) multiplied by the cycles per instruction (CPI) multiplied by the clock cycle time (CCT). The goal of a pipelined scalar processor is to strive toward a minimum CPI of 1.0, while simultaneously

reducing, or at least limiting any expansion of, the instruction count and the clock cycle time. The result is reduced overall execution time. The goal of a superscalar design is to attain a fractional CPI, or, stated as the reciprocal, the goal is to attain a throughput measured in instructions per cycle (IPC) greater than 1.0. With similar reductions, or at least limits on the expansion, of the instruction count and the clock cycle time, the result is an even larger reduction in the execution time than for scalar pipelining alone.

A VLIW (very long instruction word) processor is also designed for fetching, decoding, executing, and completing multiple operations each clock cycle. The difference between a superscalar processor and a VLIW processor is one of implementation and architecture. Superscalar design can be applied as an implementation technique to an existing sequential instruction set, while VLIW design requires that the instruction set architecture, the implementation, and the compilers be specifically designed to support the packaging of multiple independent operations into long instruction words.

Proponents of the VLIW approach rightly contend that VLIW design reduces the control complexity within a processor; however, the corresponding drawbacks are a loss of wide-scale program portability at the binary level and a lack of flexibility. With regard to portability, the control logic added by a superscalar processor is used to dynamically determine opportunities for parallel execution within a conventional instruction stream. Thus, superscalar processors dynamically schedule parallel execution of the instructions of existing executable program files, whereas recompilation into a static representation of parallel execution is a requirement for programs to run on VLIW processors. With regard to flexibility, superscalar processors can easily respond to dynamic events, such as cache misses. Dynamic events present a difficulty for VLIW designs. For example, early VLIW designs avoided data caches so that memory access time would be a known quantity for use in compiler scheduling.

A hybrid approach, called EPIC (explicitly parallel instruction computing), is exemplified in the Itanium Processor Family (IPF, also known as the IA-64) and is an attempt to gain the best of both approaches. Explicit dependence information is incorporated into the instruction formats to reduce the control logic complexity, and some scheduling of dynamic behavior is incorporated to provide flexibility.

## 2.1.2 Instruction-Level Parallelism

Superscalar processors attempt to identify and exploit parallelism in the instruction stream. That is, instructions that are independent should be executed in parallel. We briefly review the concept of dependencies. More details can be found in the Shen and Lipasti text on superscalar processor design [1].

### 2.1.2.1 Dependencies

Dependencies limit the parallelism between instructions because they must be enforced so that the results of program execution will be correct. Indeed, much of the control logic in a superscalar processor is devoted to identifying dependencies, so that execution will produce the same results as if the instruction stream was being executed on a purely sequential computer. Dependencies can be categorized in three ways.

### 2.1.2.2 Data Dependencies

Data dependencies exist between two instructions when the order between the two instructions must be maintained for execution to be correct. The most obvious data dependency is the true data dependency (or RAW: read-after-write dependency) in which the result of one instruction is used as an input operand for the second instruction. To preserve correctness, the first instruction must be executed before the second. The storage location in which the operand is passed can be either a memory location or a CPU register. A forwarding path can also be used to provide the operand to the second instruction without having to wait on the storage update.

Two other cases arise when the second instruction writes to the common storage location. An output dependency (or WAW: write-after-write dependency) occurs when both instructions write to the same storage. To preserve correctness, the result of the second instruction must be the final value of the storage. An anti-dependency (or WAR: write-after-read dependency) occurs when the first instruction reads an input operand from the storage location that will be written with the result of the second

```
True dependency:        add  r1, r2, r3     ; r3 ← r1 + r2
                        sub  r3, r4, r5     ; r5 ← r3 – r4

Anti-dependency:        add  r1, r2, r3     ; r3 ← r1 + r2
                        sub  r4, r5, r1     ; r1 ← r4 – r5

Output dependency:      add  r1, r2, r3     ; r3 ← r1 + r2
                        sub  r4, r5, r3     ; r3 ← r4 – r5
```

**FIGURE 2.1** Example data dependencies.

instruction. To preserve correctness, the first instruction must obtain its input operand before that value is overwritten by a new value from the second instruction. Both these cases are called false data dependencies because they arise from the reuse of storage locations.

See Fig. 2.1 for examples of data dependencies based on register storage. Similar examples using load and store instructions could be given based on memory storage.

#### 2.1.2.3 Control Dependencies

A control dependency occurs when an instruction depends on a conditional branch instruction. It is not known whether the instruction is to be executed or not until the branch is resolved. Thus, the branch must be executed (or predicted) before the instruction execution.

#### 2.1.2.4 Structural Dependencies

A structural dependency occurs when two instructions need the same resource. If the resource is not duplicated, the instructions must execute sequentially, one after the other, rather than in parallel. The resource for which the instructions contend might be an adder, a bus, a register file port, or some other component.

#### 2.1.2.5 Studies of Instruction-Level Parallelism

In the early 1970s, two studies on decoding and executing multiple instructions per cycle were published, one by Tjaden and Flynn on a design of a multiple-issue IBM 7094 [2] and the other by Riseman and Foster on the effect of branches in CDC 3600 programs [3]. The conclusion in both papers was that only a small amount of instruction-level parallelism existed in sequential programs—1.86 and 1.72 instructions per cycle on average, as determined by the respective studies. Thus, these studies clearly demonstrated the limiting effect of data and control dependencies on instruction-level parallelism, and the result was to encourage researchers to look for parallelism in other arenas, such as vector processors and multiprocessors. However, the study by Riseman and Foster did examine the effect of relaxing the control dependencies and found increasing levels of parallelism, up to 51 instructions per cycle, as the number of branches were eliminated (albeit in an impractical way). Later studies, in which false data dependencies as well as control dependencies were eliminated, found much more available parallelism, with the highest published estimate being 90 instructions per cycle by Nicolau and Fisher as part of their VLIW research [4].

#### 2.1.2.6 Techniques to Increase Instruction-Level Parallelism

Just as the limit studies indicated, performance can be increased if dependencies can be eliminated or reduced. Let us consider the dependencies in the reverse order from their enumeration above. First, many structural dependencies can be avoided by providing duplicate copies of necessary resources. Even scalar pipelines provide two paths for memory access (separate instruction and data caches) and multiple adders (branch target adder and main ALU). Superscalar processors have even more resource requirements, and it is not unusual to find duplicated function units and even multiple ports to the data cache (e.g., true multiporting, multiple banks, or accessing a single-ported cache multiple times per cycle).

Control dependencies are eliminated by compiler techniques of unrolling loops and performing optimizations such as "if conversion" (using conditional or predicated execution of instructions so that a control-dependent instruction is transformed into a data-dependent instruction). However, the main approach to reducing the impact of control dependencies is the use of sophisticated branch prediction. For example, the Pentium 4 keeps the history of over 4000 branches [5]. Branch prediction techniques allow instructions from the predicted path to begin before the branch is resolved and execute in a speculative manner. Of course, if a prediction is incorrect, there must be a way to recover and restart execution along the correct path.

False data dependencies can be eliminated or reduced by better compiler techniques (e.g., register and memory allocation algorithms that avoid reuse) or by the use of register and memory renaming hardware on the processor. Register renaming can be accomplished in the hardware by incorporating a larger set of physical registers than are available in the instruction set architecture. Thus, as each instruction is decoded, that instruction's architectural destination register is mapped to a new physical register, and future uses of that architectural register will be mapped to the assigned physical register. Hardware renaming is especially important for older instruction sets that have few architectural registers and for legacy and shrink-wrapped programs that for one reason or another will not be recompiled.

True data dependencies have been viewed as the fundamental limit for program execution; however, value prediction has been proposed in the past few years as somewhat of an analog of branch prediction, in which paths within the instruction stream that depend on easily predicted source values or addresses can be started earlier. As with branch prediction, there must be a way to recover from mispredictions. Another method to reduce the impact of true data dependencies is the use of some form of multi-threading in which instructions from multiple threads are interleaved on a single processor; of course, instructions from different threads are independent by definition.

## 2.1.3 Superscalar Terminology

### 2.1.3.1 Program Order

Figure 2.2 illustrates various terms used in describing superscalar processors. The adjectives *in-order* and *out-of-order* refer to the ordering of instructions as compared to the program. Figure 2.2a shows a simple scalar processor with four stages, and the actions of moving an instruction from one stage to the next are named fetch, issue, and complete, respectively. Instructions are decoded and issued in program order. A simple extension to the scalar pipeline is the use of multiple pipelines, as shown in Fig. 2.2b. This can allow a scalar processor to specialize its execution pipelines (or function units; e.g., integer vs. floating point), or, as of interest here, it can produce an in-order superscalar, in which multiple instructions are fetched, decoded, and issued in program order. The stage-to-stage terminology is the same as for the simple in-order scalar processor.

### 2.1.3.2 Instruction Completion and Precise Exceptions

All processors that attempt to execute instructions in parallel must deal with variations in instruction execution times. That is, some instructions, such as those involving simple integer arithmetic or logic operations, will need only one cycle for execution, while others, such as floating-point instructions, will need multiple cycles for execution. If these different instructions are started at the same time, as in a superscalar processor, or even in adjacent cycles, as in a scalar pipelined processor with separate function units or execution pipelines, a simple instruction can complete earlier than a longer running instruction that appears earlier in the instruction stream. This is called out-of-order completion. We can, of course, prevent out-of-order completion by techniques such as adding delay stages so that all execution paths have the same number of stages.

If we choose to allow a subsequent, simple instruction to write its result to storage before a longer running instruction completes, we may violate a data dependency. Dependency checking hardware can eliminate this problem while still allowing some out-of-order completions; however, dependency checking will not solve the problem of an inconsistent state of storage (registers or memory) if the longer

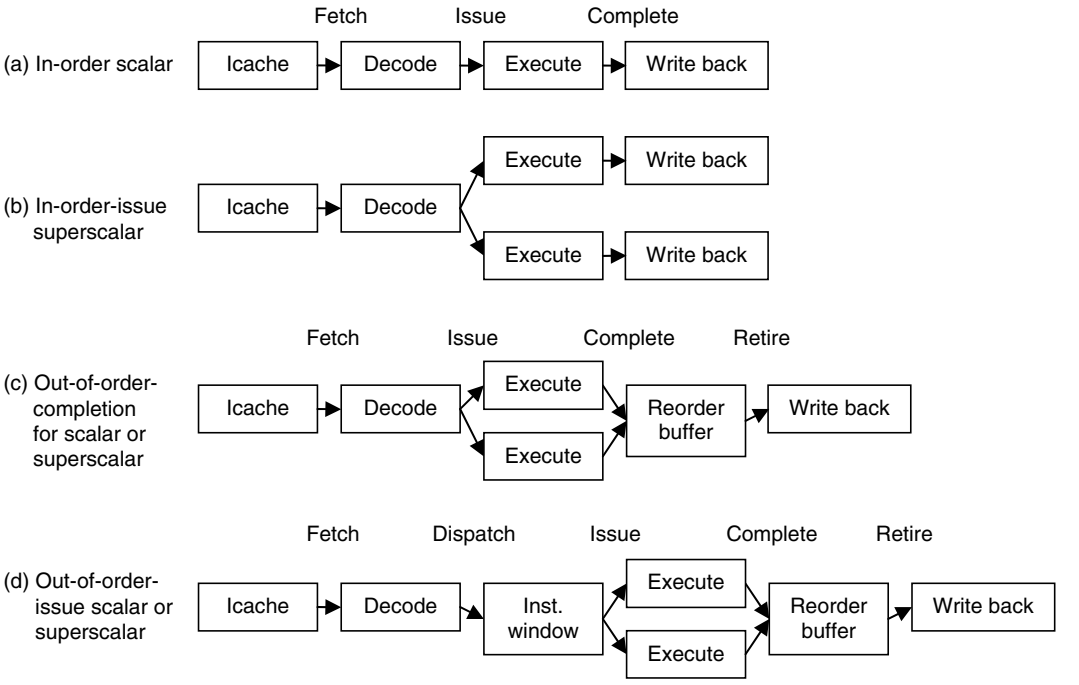



**FIGURE 2.2** Superscalar terminology.

running instruction causes an exception. To handle this exception and to be able to resume the program, we must know the precise state of the storage, that is, we must know which instructions, before the one causing the exception, have not been completed and which instructions, after the one causing the exception, have been completed. To resume at a given point in program order, the processor must restore a consistent state with all previous instructions completed and no subsequent instructions completed. A standard technique to handle this is to provide a form of buffering for the results of instructions, usually called a reorder buffer. This method is depicted in Fig. 2.2c. Instructions completing out-of-order can place their results in preassigned entries in this buffer (assignments are made when instructions are in the decode stage); and, when available, the results are *retired* out of this buffer in program order. (This action is alternatively called commit, completion, or graduation in some processors.) If an exception occurs, or for that matter, a branch or value misprediction occurs, instructions before the one causing the exception or misprediction are allowed to retire and then the contents of the reorder buffer beyond that instruction are flushed. Execution can then be resumed with a consistent state of storage.

Other techniques for dealing with exceptions include the use of a run slow mode bit to switch between in-order and out-of-order instruction completion (IBM RS/6000), the use of a history buffer (Motorola 88110), the use of exception barrier instructions (DEC Alpha), delaying instruction issue until exceptions from previous instructions are guaranteed not to occur (called *safe instruction recognition* in the Intel Pentium), and the use of a future file (UltraSPARC-III).

### 2.1.3.3 Instruction Issue

Up to this point, instructions have been described to issue, that is, start execution, in program order. Consider the case of a long-running dependent instruction pair followed by independent instructions. It would be advantageous if the compiler would statically schedule independent instructions between the two instructions of the dependent pair; however, not all programs will be so scheduled. An alternative is to provide *dynamic instruction scheduling* in the hardware, also known as *out-of-order execution*.

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| load 0(r1), f0 | F | D | E | E | W | | | | | | | | | | | | | | | |
| addf f0, f2, f4 | F | D | | | E | E | E | E | W | | | | | | | | | | | |
| store f4, 0(r1) | | F | | | D | | | | E | E | W | | | | | | | | | |
| add r1, #4, r1 | | F | | | D | | | | | | E | W | | | | | | | | |
| bne r1, r2, loop | | | | | F | | | | | | D | E | W | | | | | | | |
| load 0(r1), f0 | | | | | | | | | | | F | D | E | E | W | | | | | |
| addf f0, f2, f4 | | | | | | | | | | | F | D | | | E | E | E | E | W | |
| store f4, 0(r1) | | | | | | | | | | | | F | | | D | | | | E | E |
| add r1, #4, r1 | | | | | | | | | | | | F | | | D | | | | | |
| bne r1, r2, loop | | | | | | | | | | | | | | | F | | | | | |

**FIGURE 2.3** Superscalar execution with in-order execution.

This requires providing some form of buffering for instructions subsequent to decode, as depicted in Fig. 2.2d, and the necessary control logic to identify and issue instructions that are ready to execute.

The execution of ready instructions in an out-of-order processor is a form of data flow execution, and the dynamic scheduling of the portion of the instruction stream held in the instruction buffer has been called *restricted data flow*. The buffer for the instructions can take the form of a centralized *instruction window* or a decentralized set of *reservation stations*, in which a subset of the instruction buffers are located at each function unit. In Fig. 2.2d, the action of placing a decoded instruction into this buffer is called *dispatch* and the term for the start of execution (i.e., choosing and routing instructions from the instruction buffer to execution units) remains issue. Unfortunately, some authors and processor manuals make the issue and dispatch terms synonymous and some even reverse the above meanings, so the reader is advised to always read the context carefully when encountering these two terms.

Figures 2.3 and 2.5 illustrate superscalar processing with in-order execution and out-of-order execution, respectively, for two iterations of a short loop that adds a value to each element in an array of floating-point values. Throughput rates of two instructions per cycle are assumed for each stage-to-stage action, and branch prediction and operand forwarding are assumed. Loads and stores have two-cycle execution, floating-point add has four-cycle execution, and integer add and branch have single-cycle execution each. The stages for the in-order processor in Fig. 2.3 are FDEW: fetch, decode, execute, and write back. The stages for the out-of-order processor in Fig. 2.5 are FDIECR: fetch, decode, issue (or inst. window), execute, complete (or reorder buffer), and retire (or write back). Renaming is assumed for the out-of-order processor.

Without the ability to dispatch dependent instructions into an instruction window or to reservation stations, the decoder in the in-order processor in Fig. 2.3 stalls from cycle 3 to cycle 5, at which point the floating-point add can be issued. Similar decoder stalls can be observed at cycles 6, 13, and 16. Because of the data dependencies within the loop body, the throughput is less than one instruction per cycle. The overall effect is that the two iterations cannot be finished within 20 cycles. To take better advantage of the in-order processor, a compiler or assembly language programmer would need to unroll the loop. For example, unrolling by a factor of two would lead to the execution diagram in Fig. 2.4. In this case,

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| load 0(r1), f0 | F | D | E | E | W | | | | | | | | | | | | | | | |
| load 4(r1), f2 | F | D | E | E | W | | | | | | | | | | | | | | | |
| addf f0, f4, f6 | | F | D | | E | E | E | E | W | | | | | | | | | | | |
| addf f2, f4, f8 | | F | D | | E | E | E | E | W | | | | | | | | | | | |
| store f6, 0(r1) | | | F | | D | | | | E | E | W | | | | | | | | | |
| store f8, 4(r1) | | | F | | D | | | | E | E | W | | | | | | | | | |
| add r1, #8, r1 | | | | | F | | | | D | E | | W | | | | | | | | |
| bne r1, r2, loop | | | | | F | | | | D | | E | W | | | | | | | | |

**FIGURE 2.4** Superscalar execution with in-order execution and compiler loop unrolling.

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| load 0(r1), f0 | F | D | I | E | E | C | R | | | | | | | | | | | | | |
| addf f0, f2, f4 | F | D | | | | I | E | E | E | E | C | R | | | | | | | | |
| store f4, 0(r1) | | F | D | | | | | | | | I | E | E | C | R | | | | | |
| add r1, #4, r1 | | F | D | I | E | C | | | | | | | | | R | | | | | |
| bne r1, r2, loop | | | F | D | | I | E | C | | | | | | | | R | | | | |
| load 0(r1), f0 | | | | F | D | | I | E | E | C | | | | | | R | | | | |
| addf f0, f2, f4 | | | | F | D | | | | | I | E | E | E | E | C | | R | | | |
| store f4, 0(r1) | | | | | F | D | | | | | | | | | I | E | E | C | R | |
| add r1, #4, r1 | | | | | F | D | I | E | C | | | | | | | | | | R | |
| bne r1, r2, loop | | | | | | F | D | | I | E | C | | | | | | | | | R |

**FIGURE 2.5** Superscalar execution with out-of-order execution and register renaming.

one iteration of the unrolled loop, with two element updates, finishes in cycle 12. There is still some minor stalling at the decoder, as seen in cycles 4 and 6, but the overall performance has improved greatly.

For the out-of-order superscalar processor illustrated in Fig. 2.5, each instruction has to traverse more stages, but dependent instructions can be buffered so that the decoder and execution units can bypass these instructions and uncover independent instructions that will be ready to execute. This can be observed in cycle 4, in which the integer add is issued before any of the three previous instructions complete. (The WAR dependency between the add and store instructions is handled by register renaming.) The integer add completes in cycle 6 but the result has to wait in the reorder buffer until it can retire in-order in cycle 15. Even without compiler unrolling, the two iterations finish by cycle 20. In fact, the use of renaming and dynamic scheduling allows the processor to do *hardware unrolling* of the loop, as seen in cycle 7 in which the load instruction of the second iteration is issued before the first iteration is finished. (The data dependency arising from the write to register f0 in this second load is handled by register renaming.) If the loop has enough iterations, throughput for the out-of-order superscalar will continue to increase as more hardware overlap of iterations occurs and will become more competitive with the unrolled, in-order superscalar throughput. The out-of-order superscalar will also be more tolerant of cache misses than the in-order superscalar.

A scalar processor typically has decode and issue rates of one instruction per cycle. As compared to a scalar design, a superscalar design must provide a greater than one throughput for each action—fetch, dispatch, issue, complete, and retire. Note that while the throughputs rates do not have to be equal, the overall throughput rate for a processor is limited by the smallest throughput rate across the individual stage-to-stage actions. In fact, a processor in the style of Fig. 2.2d that fetched, decoded, and executed multiple instructions per cycle but that could only retire one instruction per cycle would be in effect a scalar processor.

## 2.1.4 Example Machines

### 2.1.4.1 Historical Designs

Dynamic instruction scheduling has been implemented in scalar processors as well as in superscalar processors. For example, the Zuse Z4 (developed in the 1940s) had a limited ability to switch the order of execution for two instructions. The CDC 6600 (announced in 1963) and the IBM S/360 Model 91 (announced in 1964) are the two most well-known scalar processors that use dynamic instruction scheduling. The 6600 had 10 functional units that could execute simultaneously. One instruction was decoded at a time and dispatched (issued in CDC terminology) to the scoreboard, but up to three instructions could start execution, out-of-order, at a time. (The issue rate limit of three instructions per cycle was due to a structural limitation of three sets of read port pairs, called data trunks, provided for the register file in the 6600.) The Model 91 also decoded one instruction at a time but used out-of-order execution in its floating-point unit. The Model 91 technique was called the Tomasulo algorithm, named

after inventor Robert Tomasulo, and provided register renaming by tagging. Up to two instructions, one add and one multiply, could start execution at the same time within the Model 91 FPU. The Model 91 was also infamous among programmers for not providing precise exceptions.

The idea of a superscalar computer originated with John Cocke at IBM in the 1960s. Cocke has said that Gene Amdahl, architect of the IBM 704 and one of the architects of the IBM S/360, postulated a bound on computer performance that included an assumption of a maximum decoding rate of one instruction per cycle on a single processor. Cocke felt that this was not an inherent limit for a single processor. His ideas about multiple instruction decoding became an important part of the IBM ACS-1 supercomputer design, which was started in 1965 but ultimately cancelled in 1969. In this design, up to 16 instructions would be decoded and checked for dependencies of each cycle and up to seven instructions would be issued, out-of-order, to function units [6].

After the ACS cancellation and the publication of the early ILP studies [2,3], the idea of superscalar processing lay dormant until the early 1980s when further research at IBM revived the notion of multiple decoding and multiple issue. John Cocke teamed with Tilak Agerwala at IBM and worked on a series of designs that finally led to the IBM POWER (performance optimized with enhanced RISC) instruction set architecture and the IBM RS/6000 workstation, which was announced in late 1989 and delivered in 1990. Agerwala is credited with coining the term superscalar during a series of talks in 1983–1984. These talks and the related IBM technical report [7] were influential in kindling interest in the approach. In 1989, Intel introduced the first single-chip superscalar microprocessor, the i960CA. Also around this time, LIW efforts, such as H.T. Kung's Warp project, the Intel i860, the Apollo DN10000, and the National Semiconductor Swordfish, and VLIW efforts, such as the Multiflow TRACE computers and the Cydrome Cydra-5, were under way.

### 2.1.4.2 Modern Designs

Most high-performance processors now incorporate some form of superscalar processing. Even many simple processors can decode and execute one integer instruction along with one floating-point instruction per cycle. We briefly survey three representative processors in the following subsections. Other notable superscalar designs include the Compaq Alpha 21 × 64 series, HP 8 × 00 series, and MIPS R1 × 000 series. It should be noted that designers of IBM mainframes developed a superscalar implementation, the IBM ES/9000 Model 520, in 1992, but more recent implementations have reverted to scalar pipelines.

#### *2.1.4.2.1 UltraSPARC-I, 1995*

The UltraSPARC-I [8] is an example of an in-order superscalar processor Fig. 2.2b. It provides four-way instruction issue to nine functional units. The design team extensively simulated many alternatives and concluded that an out-of-order approach would have required a 20% penalty in clock cycle time and increased the time to market by up to half a year. The final design involves a nine-stage pipeline. This includes a decoupled front-end pipeline (fetch and decode stages) that performs branch prediction and places decoded instructions in a 12-entry buffer. A grouping stage then selects up to four instructions in-order from the buffer to be issued in the next cycle. Precise exceptions are supported by padding out most function unit pipelines to four stages each (the required length for the floating-point pipelines) so that most four-instruction groups complete in-order. The final two stages resolve any exceptions in the groups and write back the results.

#### *2.1.4.2.2 PowerPC 750, 1997*

The PowerPC 750 [9] is an example of an out-of-order processor with distributed reservation stations and a reorder buffer (called the completion buffer in the 750). The 750 has six function units, including two integer units. Each unit has one reservation station, except the load/store unit, which has two. Instructions can be issued, when ready, from these reservation stations. (This limited form of out-of-order execution is sometimes called *function unit slip.*) The 750 also includes six rename registers for renaming the 32 integer registers and six rename registers for renaming the 32 floating-point registers.

The overall pipeline works as follows. A decoding stage is not needed since instructions are pre-decoded into a wider representation as they are filled into the instruction cache. Up to four instructions are fetched per cycle into a six-entry instruction buffer. Logic associated with the instruction buffer removes any nops or predict-untaken branches and overwrites predict-taken branches with target-path instructions so that no instruction buffer entries are required for nops or branches. (However, predicted branches are kept in the branch unit until resolution to provide for misprediction recovery.) Up to two instructions can be dispatched per cycle to the reservation stations and are allocated entries in the six-entry completion buffer. The integer units require a single cycle for execution, while the load/store unit and the floating-point unit require two and three cycles, respectively. After execution, results are placed into the assigned entries in the completion buffer. Up to two entries per cycle can be written back from the completion buffer to the register files.

#### *2.1.4.2.3 Pentium 4, 2000*

The Pentium 4 [5] is an example of an out-of-order processor with a centralized instruction window and a reorder buffer. The original Pentium combined two integer pipelines, each similar in design to the pipeline of the 486, and could decode and execute up to two instructions in-order per cycle (similar to Fig. 2.2b). Intel then developed the P6 core microarchitecture, which serves as the basis for the Pentium Pro, Pentium II, Pentium III, and Pentium-M. (A detailed case study of the P6 core appears in Shen and Lipasti [1].) After branch prediction and instruction fetch, the P6 core decodes up to three variable-length Intel IA-32 instructions each cycle and translates them into up to six fixed-length μops (microoperations). Up to three μops are processed by register renaming logic each cycle, and these are placed into the 20-entry centralized instruction window along with being allocated entries in the 40-entry reorder buffer (similar to Fig. 2.2d). The window is scanned each cycle in a pseudo-FIFO manner in an attempt to issue up to four μops. Preference is given to back-to-back μops to reduce the amount of operand forwarding among the execution units. The actual scanning and issue requires two cycles, while most instructions require single-cycle execution. At maximum, the reorder buffer can receive up to three results per cycle and can start retirement of up to three μops per cycle. Retirement requires three cycles. Thus, the overall pipeline has some 14 stages; but, because some of these stages can overlap, the effect is a minimum latency of 12 cycles per instruction.

The Pentium 4 is a redesign of the P6 core microarchitecture. The translation of IA-32 instructions into μops is retained, but instead of repeatedly fetching, decoding, and translating recurring IA-32 instruction sequences, the μops are stored in a separate *trace cache* for repeated access. The trace cache can hold up to 12 K μops and stores frequently traversed sequences (i.e., traces) of μops with any predict-taken branches followed by instructions from the predicted path. The trace cache can provide up to three μops per cycle, which are then routed through reorder-buffer allocation logic, register-renaming logic, and then into μop queues for scheduling. Up to six μops can be issued per cycle, and up to three μops can be retired per cycle. The reorder buffer size is increased from 40 entries in the P6 core to 126 entries for the Pentium 4. The clock rate was also increased on the Pentium 4, with approximately twice the number of pipeline stages as compared to the P6 core. Also by cascading ALUs, two dependent addition or subtraction operations can be performed in each cycle in the Pentium 4.

#### *2.1.4.2.4 POWER4, 2001*

The POWER4 [10] is one of the most aggressive superscalar designs to date with an issue rate of up to eight instructions per cycle per core and two cores on a single chip. Up to 200 instructions can be in flight at any one time on each core. The two cores share a second-level cache, a third-level cache directory and controller, and a communication (fabric) controller.

To limit complexity within a core, instruction groups of up to five instructions each (with a limit of at most one branch per group) are formed after four stages of decoding and are then used to track dispatch, completion, and exceptions through a global completion table. Thus, each core is limited to a throughput of one group (up to five instructions) per cycle. The eight function units per core are two integer units, two floating-point units, two load-store units, a branch unit, and a condition register unit.

These units are fed by a total of 11 issue queues, and instruction-level parallelism is supported by a total of 244 rename resources.

Because of the grouping structure, individual instructions are said to *finish* (or write back), and groups are said to *complete* (i.e., retire). Complex instructions are cracked into multiple internal operations (called IOPS), each of which requires one instruction slot in a group. If an instruction in a multi-instruction group causes an exception, the instructions are reformed into multiple single-instruction groups and redispatched. Integer instructions traverse 15 pipeline stages, and floating-point instructions traverse 20 pipeline stages. The PowerPC 970 (G5) uses an almost identical core pipeline structure, and the recent POWER5 (available in 2005) uses the same core pipeline structure of the POWER4.

## References


1. Shen, J. and Lipasti, M., *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill, New York, 2005.
2. Tjaden, G. and Flynn, M., Detection of parallel execution of independent instructions, *IEEE Transactions on Computers,* C-19, 889, 1970.
3. Riseman, E. and Foster, C., The inhibition of potential parallelism by conditional jumps, *IEEE Transactions on Computers,* C-21, 1405, 1972.
4. Nicolau, A. and Fisher, J., Measuring the parallelism available for very long instruction word architectures, *IEEE Transactions on Computers,* C-33, 968, 1984.
5. Hinton, G. et al., The microarchitecture of the Pentium 4 processor, *Intel Technology Journal,* available on-line, 2001.
6. Schorr, H., Design principles for a high-performance system, in *Proceedings of the Symposium on Computers and Automata*, New York, 1971, 165.
7. Agerwala, T. and Cocke, J., High performance reduced instruction set processors, Technical Report RC12434, IBM Thomas Watson Research Center, 1987.
8. Tremblay, M., Greenly, D., and Normoyle, K., The design of the microarchitecture of the Ultra-SPARC-I, *Proceedings of IEEE*, 83, 1653, 1995.
9. Kennedy, A. et al., A G3 PowerPC superscalar low-power microprocessor, in *Proceedings COMP-CON*, San Francisco, 1997, 315.
10. Tendler, J. et al., POWER4 system microarchitecture, *IBM Journal of Research and Development*, 46, 5, 2002.


# 2.2 Register Renaming Techniques*

*Dezsö Sima*

## 2.2.1 Introduction

Register renaming (or renaming for short) is a widely used technique in advanced instruction level processors (ILP-processors) to remove false data dependencies between register operands of subsequent instructions in a straight line code sequence [1–3]. False data dependencies are write-after-read (WAR) or write-after-write (WAW) dependencies (see Appendix A). After removing false data dependencies by register renaming, on average more instructions are available for parallel execution per cycle, this increases processor performance.

*Portions of this chapter reprinted with permission from Sima, D., The design space of register renaming techniques, *IEEE Micro*, 20, Sept./Oct., 70, 2000, [35] © IEEE.

The principle of register renaming is straightforward. The processor removes false data dependencies by writing the results of the instructions first into dynamically allocated buffers, called rename buffers, rather than into the specified destination registers and forwards these result into the originally specified architectural registers in a later stage of instruction execution. For instance, in the case of the following WAW dependency:

$i_1$: add r1, r2, r3; [r1 $\leftarrow$ (r2) + (r3)]
$i_2$: mul r1, r4, r5; [r1 $\leftarrow$ (r4) $\times$ (r5)]

the processor renames the destination register of $i_2$ that is r1, say to r33. Then after renaming register r1, the instruction $i_2$ becomes

$i'_2$: mul r33, r4, r5; [r33 $\leftarrow$ (r4) $\times$ (r5)]

and the processor writes the result of $i'_2$ into r33 instead of r1. This resolves the previous WAW dependency between $i_1$ and $i_2$. In subsequent instructions, however, references to the source register r2 must be redirected to the rename buffer r33 as long as the renaming remains valid. In the next section we give a detailed description of the whole rename process.

A precursor to register renaming was introduced in 1967 by Tomasulo in the IBM 360/91 [4], a scalar supercomputer of that time, which pioneered both pipelining and shelving. The 360/91 renamed floating-point registers in order to preserve the logical consistency of the program execution, rather than to increase processor performance by removing false data dependencies.

Tjaden and Flynn [5] were the first to suggest the use of register renaming for removing false data dependencies in 1970. They proposed to rename load type instructions, without using the term "register renaming." This specific term was introduced a few years later, in 1975, by Keller [6] who extended renaming to cover all instructions including a destination register. He also described a possible hardware implementation of this technique. Because of the complexity of its implementation, however, about two decades passed until register renaming came into widespread use in superscalars in the middle of the 1990s.

Early superscalar models of significant processor lines, such as the PA 7100, SuperSPARC, Alpha 21064, R8000, and the Pentium, typically did not yet use renaming as indicated in Fig. 2.6.

Renaming appeared gradually, first in a restricted form, called partial renaming, in the beginning of the 1990s, in the IBM RS/6000 (POWER1), POWER2, PowerPC 601, and in NextGen's Nx586 processors, as depicted in Fig. 2.6. Partial renaming restricts renaming to one or to a few data types, such as floating-point loads or floating-point instructions, as detailed in Section 2.2.3.2. Full renaming emerged later, beginning in 1992, first in the high-end models of the IBM mainframe family ES/9000, then in the PowerPC 603. Subsequently, renaming spread into virtually all superscalar processors with the notable exception of Sun's UltraSPARC line. At present, register renaming is considered to be a standard feature of superscalar processors.

## 2.2.2 Overview of the Rename Process

The rename process itself is considerably complex. It consists of a number of rename specific tasks—renaming the destination and the source registers, fetching renamed source operands, updating the rename buffers, releasing allocated rename buffers, recovery of the rename process from faultily executed speculative execution, etc. In addition, each of the rename-specific tasks may be implemented in a number of different ways. Furthermore, also specific features of the underlying microarchitecture affect the rename process. Therefore, each concrete description of the rename process is related to a particular renaming technique employed and the underlying microarchitecture. Thus, before describing the rename process we need to be specific about both possible renaming techniques and types of microarchitectures considered.

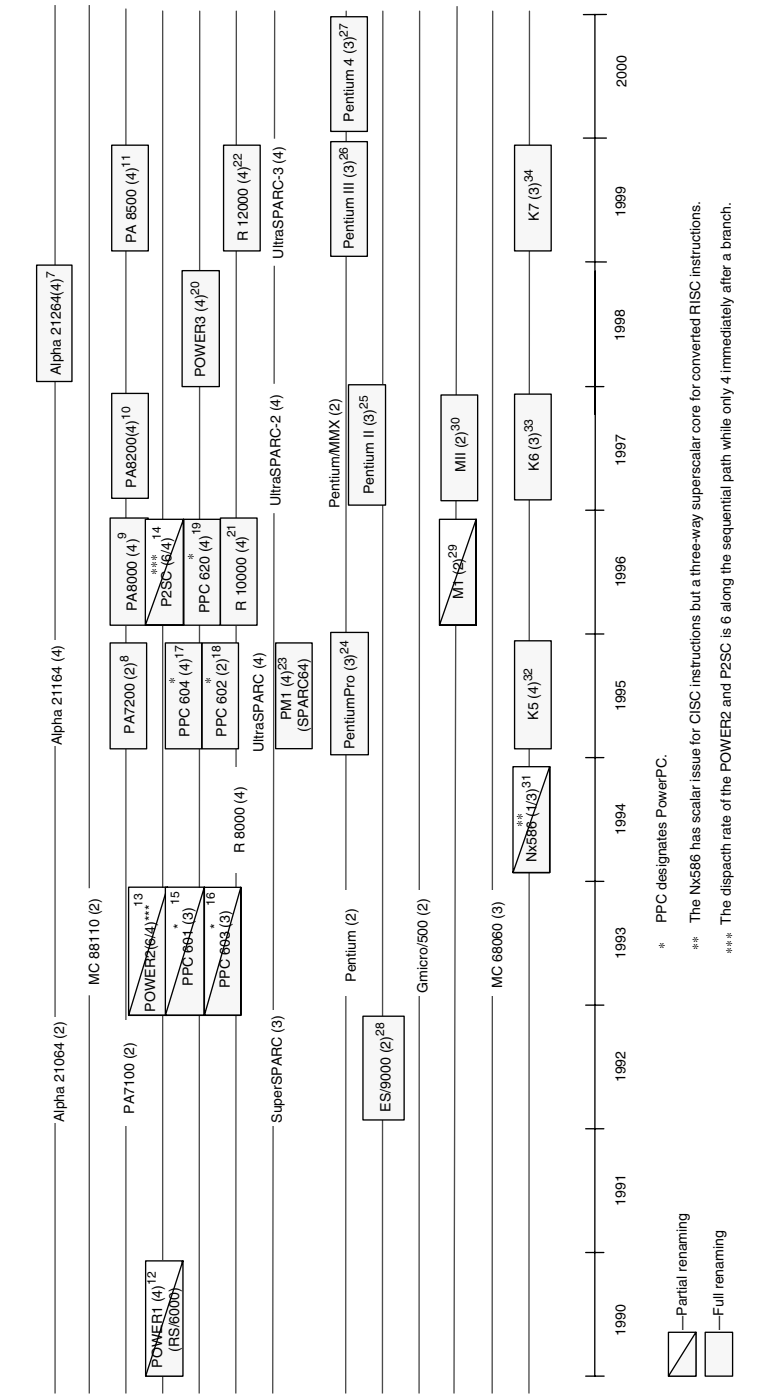



**FIGURE 2.6** Chronology of the introduction of renaming in commercial superscalar processors. As date of introduction we indicate the first year of volume production. Following the model designation we also show the dispatch rate of the processors (in brackets). Concerning the dispatch rate of CISC processors we note that one x86 instruction can be considered to be equivalent of 1.3–1.9 RISC instructions. In this figure we give references to the processors which make use of renaming. (From Gwennap, L., *Microprocessor Report*, 9, 14, 1, 1995.)

Concerning renaming techniques, in a subsequent section, we show that there are nine basic alternatives available. In our description of the rename process, we need to presume one of them. Our choice is the one where (1) renaming is implemented by using rename register files (RRF) and (2) architectural registers are mapped to rename registers by means of mapping tables. Although both terms are explained later in the subsequent section, beforehand we note that RRFs, split to separate fixed-point (FX) and floating-point (FP) RRFs, store the instruction results produced by the execution units temporarily, while the FX- and FP-mapping tables hold the actual mappings of the FX- and FP-architectural registers to the associated rename registers, as indicated in the section on Layout of Register Mapping.

As far as the underlying microarchitecture is concerned, there are two design aspects that affect the implementation of the rename process: (1) whether or not the processor uses shelving (indirect issue, dynamic instruction scheduling, queued issue; see related box) and (2) assuming the use of shelving, what kind of operand fetch policy is employed (see related box). As recent superscalars predominantly make use of shelving, we take this design option for granted throughout this section. Regarding the operand fetch policy, which is one design aspect of shelving, we take into account both alternatives, since superscalar processors make use of both policies. Thus, while we describe the rename process in the subsequent two sections, we do it in two scenarios, first assuming the dispatch-bound fetch policy and then the issue-bound fetch policy. In both the scenarios mentioned, we describe the rename process by focusing only on a small part of the microarchitecture, which is just enough to highlight the implementation of specific tasks of the rename process.

#### 2.2.2.1 Process of Renaming, Assuming Dispatch-Bound Operand Fetching

The considered part of the microarchitecture executes FX-instructions and consists of an architectural register file (ARF) and an execution unit (EU), as shown in Fig. 2.7.

Our subsequent description of the rename process is embedded into the general framework of instruction processing. Here, we distinguish the following four processing phases: (1) decoded instructions are dispatched into the RSs, (2) executable instructions are issued from the RSs to the EUs, (3) the EUs perform the prescribed operations and generate the result of the instructions. At this time the instructions are said to be finished, and finally, (4) the processor completes (commits, retires) instructions in an in-order fashion, irreversibly updating the program state with the results of the instructions.

Assuming the processor core as shown in Fig. 2.7 and dispatch-bound operand fetching, the rename process is carried out as follows:

I. During instruction dispatch, three rename-related tasks must be performed:

a. Destination registers of dispatched instructions (Rd) need to be renamed.
b. Source registers (Rs1 and Rs2) should be renamed in order to redirect the source references to the associated rename registers.
c. Required source operands need to be fetched.

1. *Renaming the destination registers of dispatched instructions*: To rename the destination register of a dispatched instruction, first a free rename register needs to be allocated to the dispatched instruction. This task is accomplished by means of the mapping table. The mapping table keeps track of the actual mappings of the architectural registers to the rename registers. Renaming of the destination register results in writing the identifier of the allocated rename register (Rd′) into the corresponding mapping table entry, and forwarding this identifier also into the corresponding field of the RS. Typically, the processor uses the index of the allocated rename register as Rd′.
2. *Renaming the source registers*: Source registers, for which a valid renaming exists, also need to be renamed. This is carried out by accessing the mapping table with the source register identifiers (Rs1, Rs2) as indices, and fetching the identifiers of the allocated rename registers (designated as Rs′1, Rs′2). If, for a particular source identifier there is no valid renaming, the required source operand will be accessed from the ARF by using the original source register identifier (Rs1 or Rs2).
3. *Fetching the source operands*: Finally, the referenced source operands need to be fetched. However, with renaming, requested source operands may be in one of two possible locations. If there is a

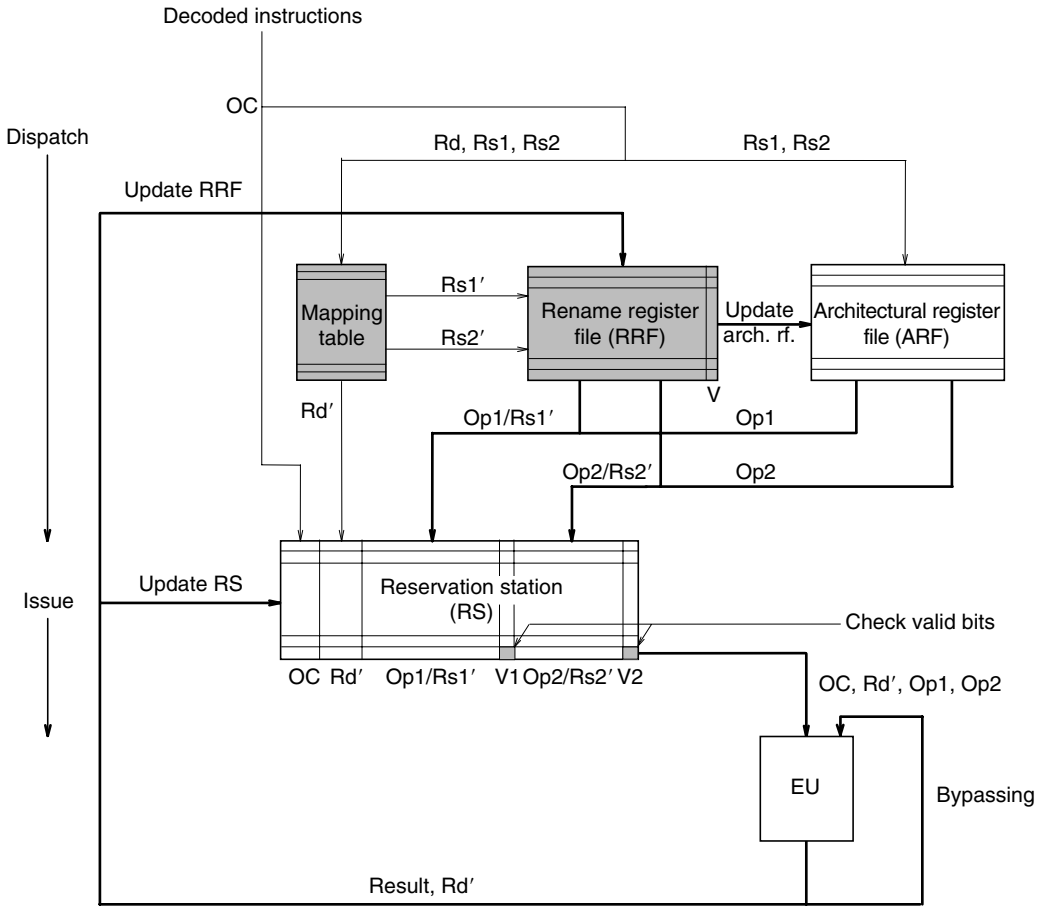



**FIGURE 2.7** Processor core providing shelving with dispatch-bound operand fetching and renaming.

valid renaming, the requested operand needs to be fetched from the RRF, else from the ARF. To fetch a requested operand, usually the processor accesses both the RRF and the ARF simultaneously to shorten the access time. If only the ARF hits, the referenced source register is actually not renamed and the accessed value is the required one. If, however, for a particular source register a valid renaming exists, both register files hit and the processor will give preference to the operand fetched from the RRF. In this case, the RRF may deliver either a valid operand value (Op1/Op2), if it has already been produced by a preceding instruction, or the index of that rename register, which will hold the requested value after its generation ($Rs'1/Rs'2$), if the required result has not yet been calculated. Thus, for each referenced source register either the requested operand value (Op1/Op2) or the appropriate rename register identifier ($Rs'1/Rs'2$) will be written into the RS. The valid bits associated with the source operand fields (V1/V2) indicate whether the related operand field holds a valid source operand value (Op1/Op2) or a rename register identifier ($Rs'1/Rs'2$).

II. Issuing is not at all rename specific. Assuming in-order issuing, the processor inspects the valid bits of the source operands (V1 and V2) of the oldest instruction kept in the RS. If both valid bits of this instruction are set and the EU is also free, the instruction is forwarded to the EU for execution.

III. After the EU has finished the execution of an instruction, both the RS and the RRF need to be updated with the generated result. To update the RS, the generated results and their identifiers (Rd′) are

broadcasted to all the source register entries held in the RS. Through an associative search, all source register identifiers ($Rs'1$, $Rs'2$), which are waiting for the new result, are located. The processor substitutes matching identifiers with the result value and sets the associated valid bits (V1 or V2) to indicate availability. We note that this task is performed basically in the same way with and without renaming. There is, however, a slight difference with renaming, as in this case the search key is the renamed destination register identifier ($Rd'$) rather than the original destination register identifier (Rd) that is used without renaming. The second task is to update the rename register file. This is done simply by writing the new result into the RRF using the identifier accompanying the result produced ($Rd'$) and setting the associated valid bit (V) to signal availability.

IV. When an instruction completes, the processor permanently updates the ARF, and thus the program state, with the content of the associated rename register. This is done by writing the result of the completed instruction from the associated rename register to the addressed destination register. At this stage of the instruction execution, resources bound to the established renaming becomes free. Therefore, the related entry in the mapping table needs to be deleted and the rename register involved can be reclaimed for further use. This is so since (i) after completion, the result of the instruction, that is, the content of the rename register, has already been written into the addressed destination register, and (ii) after finishing the instruction, the generated result has already been transferred to all instructions waiting for this operand in the RS.

During renaming, rename registers take on a sequence of states, as indicated in Fig. 2.8.

During initialization, the processor sets all rename registers into the "available" state. When the processor allocates a rename register to a dispatched instruction, the state of the allocated register will be changed to allocated, not valid and its valid bit will be reset. When this instruction becomes finished, the newly produced result is written into the associated rename register, and its state is set to allocated, valid. Finally, while the instruction completes, the result held temporarily in the rename register is written into the specified architectural register. Thus, the allocated rename register can be reclaimed. Its state is then changed to available. Nevertheless, it can happen that an exception or faulty speculative execution gives rise to flush not yet completed instructions. In this case, a recovery procedure is needed, and the state of the concerned rename registers will be changed from the allocated, not valid or allocated, valid state to the available state and the corresponding mappings between architectural and rename registers will be deleted.

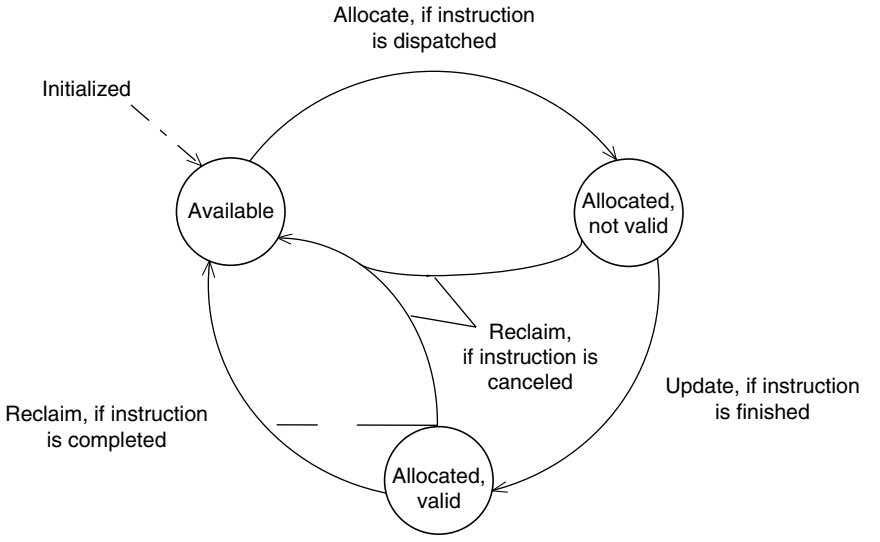



**FIGURE 2.8** State transition diagram of the rename registers, assuming the use of a rename register file (RRF).

#### 2.2.2.2 Process of Renaming, Assuming Issue-Bound Operand Fetching

Assuming basically the same processor core as before, but using the issue-bound operand fetching, the rename process is carried out as follows (see Fig. 2.9):

1. During instruction dispatch, both the destination register (Rd) and the source registers (Rs1 and Rs2) are renamed in the same way as described for dispatch-bound operand fetching. But now, beyond the operation code (OC) and the renamed destination register identifier (Rd′), the renamed source register identifiers (Rs′1 and Rs′2) are written into the RS rather than the operand values (Op1, Op2, if available) as with dispatch-bound operand fetching.
2. During issuing, two tasks need to be performed: (a) the instruction held in the last entry of the RS needs to be checked to see whether it is executable. If so and if the EU is also free, this instruction needs to be forwarded for execution to the EU. (b) During forwarding of the instruction, its operands need to be fetched either from the RRF or from the ARF in the same way as described in connection with the dispatch-bound operation.
3. When the EU finishes its operation, the generated result is used to update the RRF. Updating is performed by writing the result into the allocated rename register using the supplemented register identifier (Rd′) as an index into the RRF and setting the associated valid bit (V-bit).

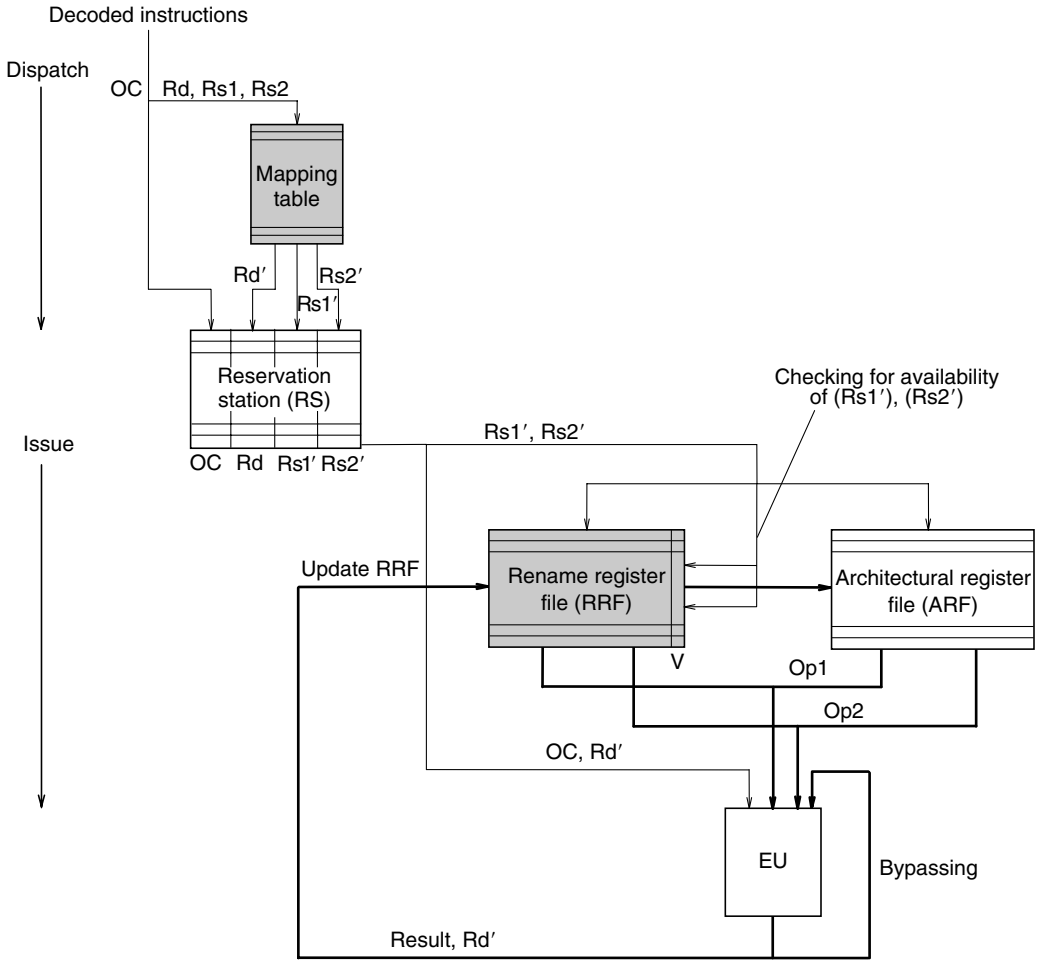



**FIGURE 2.9** Processor core providing shelving with issue-bound operand fetching and renaming.

4. Finally, while the processor completes an instruction, the temporary result held in the associated rename register is written into the architectural register, which is specified in the destination field of the instruction. The only tasks remaining are to delete the corresponding entry in the mapping table and to reclaim the rename register associated with the completed instruction. Reclaiming of the rename register is, however, a far more complex task now than with dispatch-bound operand fetching. Notice that if operands are fetched dispatch bound (a) dispatched instructions immediately access their operands and (b) missing operands are, after their generation, immediately forwarded from the EU to the instructions waiting for these operands in the RS. In this case, after completing an instruction, the allocated rename register can immediately be reclaimed. However, if operands are fetched during issuing, the RS is not automatically updated with the produced results. As a consequence, after an instruction completes, the RS may still contain instructions, which will require the contents of the rename register, that are allocated to the just-completed instruction. Thus, while instructions complete, their allocated rename registers cannot be reclaimed immediately as in the case of the dispatch-bound operand fetching. To resolve this problem, one possible solution is to maintain a counter for each rename register, which keeps track of the number of references made to this register. The counter will be incremented each time if one of the source operands of a dispatched instruction addresses this particular rename register, and will be decremented during issuing of the instructions each time when a source operand is fetched from this register. After all outstanding fetch requests for a particular rename register are satisfied, as indicated by the counter score of zero, and the associated instruction has been completed, the related register becomes eligible for reclaiming. At the first sight, it may seem that this intricate reclaim process can be avoided if during completion the RS would have been searched for all renamed source operand identifiers ($Rs'1$, $Rs'2$), which refer to the rename buffer, allocated to the completing instruction ($Rd'$), and matching renamed source register identifiers would have been remapped to the associated architectural register (Rd). Unfortunately, this idea is not applicable since there is no guarantee that the addressed architectural register would not be rewritten until instructions needing its content are issued.

During the rename process, rename registers will take the same states and the same state transitions will also occur as described earlier in connection with Fig. 2.8. The only difference is that now rename registers are reclaimed according to modified conditions, as discussed previously.

We emphasize that other basic alternatives of register renaming differ mainly in two aspects: (1) the processor can hold renamed values in other structures than rename register files and (2) the processor can use a different scheme for mapping the architectural registers to rename registers as assumed above. In addition, the processor should be able to rename not just one instruction per cycle but all dispatched instructions. Nevertheless, despite these differences, the previous descriptions in the two characteristic scenarios give a good background about how the rename process is carried out in any of the possible implementation schemes.

## 2.2.3 Design Space of Register Renaming Techniques

### 2.2.3.1 Overview

The design space of register renaming has four main dimensions: the scope of register renaming, the layout of the rename registers, the implementation technique of register mapping, and the rename rate, as indicated in Fig. 2.10. These aspects are discussed in the subsequent sections. For the presentation of the design space we make use of DS trees [3,36].

### 2.2.3.2 Scope of Register Renaming

The scope of register renaming indicates how extensively the processor makes use of renaming. In this respect, we distinguish between partial and full renaming. Partial renaming is restricted to one or to only a few instruction types, for instance only to FP-instructions. This incomplete form of implementation was typical for the introductory phase of renaming, at the beginning of the 1990s (see Fig. 2.6).

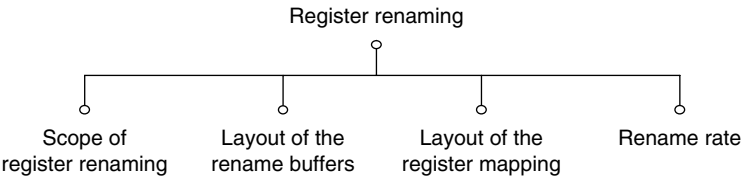
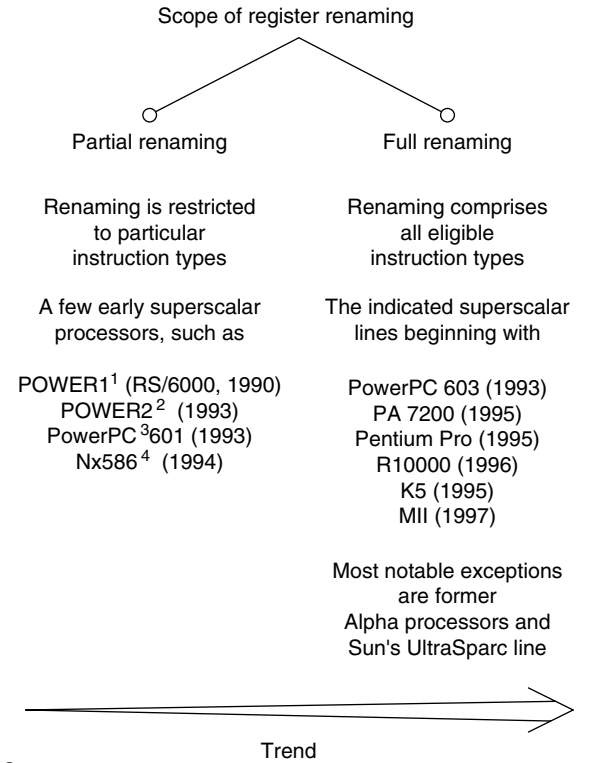



**FIGURE 2.10** Design space of register renaming.

Examples of processors using partial renaming are the POWER1 (RS/6000), POWER2, PowerPC 601, and the Nx586, as shown in Fig. 2.11. Of these, the POWER1 (RS/6000) renames only FP-loads. As the POWER1 has only a single FP-unit, it executes FP-instructions in sequence, so there is no need to rename floating-point register instructions. POWER2 introduces multiple FP-units, consequently it extends renaming to all FP-instructions, whereas the PowerPC 601 renames only the Link and count register. In the Nx586, which includes an integer core, renaming is restricted obviously only to FX-instructions.

Full renaming covers all instructions including a destination register. As Fig. 2.11 demonstrates, virtually all recent superscalar processors employ full renaming. Noteworthy exceptions are Sun's UltraSPARC line and Alpha processors preceding the Alpha 21264.




Comments:

[1] The POWER1 renames only FP-loads.

[2] The POWER2 extends renaming to all FP-instructions.

[3] The PowerPC 601 renames only the Link and count register.

[4] Since the Nx586 has an integer core, it renames only FX-instructions.

**FIGURE 2.11** Scope of register renaming.

## 2.2.4 Layout of the Rename Buffers

### 2.2.4.1 Overview

Rename buffers establish the actual framework for renaming. From their layout we point out three essential design aspects—the type and the number of the rename buffers provided as well as the number of the read and write ports, as shown in Fig. 2.12.

### 2.2.4.2 Types of Rename Buffers

The choice of the type of rename buffers to use in a processor has far-reaching impact on the implementation of the rename process. Given its importance, we will outline the various design options. To simplify our presentation, we initially assume a common architectural register file for all data types processed. We later extend our discussion to the split register scenario that is commonly employed.

As Fig. 2.13 illustrates, there are four fundamentally different ways to implement rename buffers. The range of choices include (1) using a merged architectural and rename register file, (2) employing a stand-alone rename register file, (3) keeping renamed values in the reorder buffer (ROB), or (4) keeping the renamed values in the shelving buffers.

I. In the first approach, rename buffers are implemented along with the architectural registers in the same physical register file, called the merged architectural and rename register file or the merged register file for short. Here, both architectural and rename registers are dynamically allocated to particular registers of the same physical file.

Each physical register of the merged architectural and rename register file is at any time in one of four possible states [28]. These states reflect the actual use of a physical register as follows:

a. Not committed (available state)
b. Used as an architectural register (architectural register state)
c. Used as a rename buffer, but this register does not yet contain the result of the associated instruction (rename buffer, not valid state)
d. Used as a rename buffer, and this register already contains the result of the associated instruction (rename buffer, valid state)

As part of the instruction processing, the states of the physical registers are changed as described below and indicated in the state transition diagram in Fig. 2.14.

During initialization, all available physical registers ($n$) are set to the "available" state. When a dispatched instruction needs a new rename buffer, one physical register is selected from the pool of the available registers and is allocated to the destination register concerned. Accordingly, its state is set to the "rename buffer, not valid" state and its valid bit is reset. After the associated instruction finishes its execution, the produced result is written into the allocated rename buffer. Its valid bit is then set and its state is changed to "rename buffer, valid" state. Later, when the associated instruction completes, the rename buffer, which is allocated to it will be declared to be the architectural register, which implements the destination register specified in the just-completed instruction. Its state then changes to the "architectural register" state to reflect this. Finally, when an old architectural register is reclaimed, its state becomes again "available." Possible schemes for reclaiming old architectural registers are described for both dispatch-bound and issue-bound operand fetching in Section 2.2.2. It can also happen that not yet

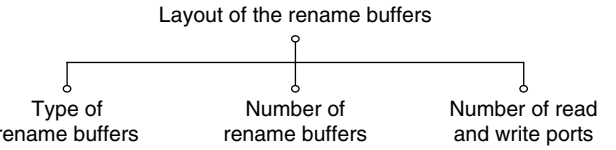



**FIGURE 2.12** Layout of the rename buffers.

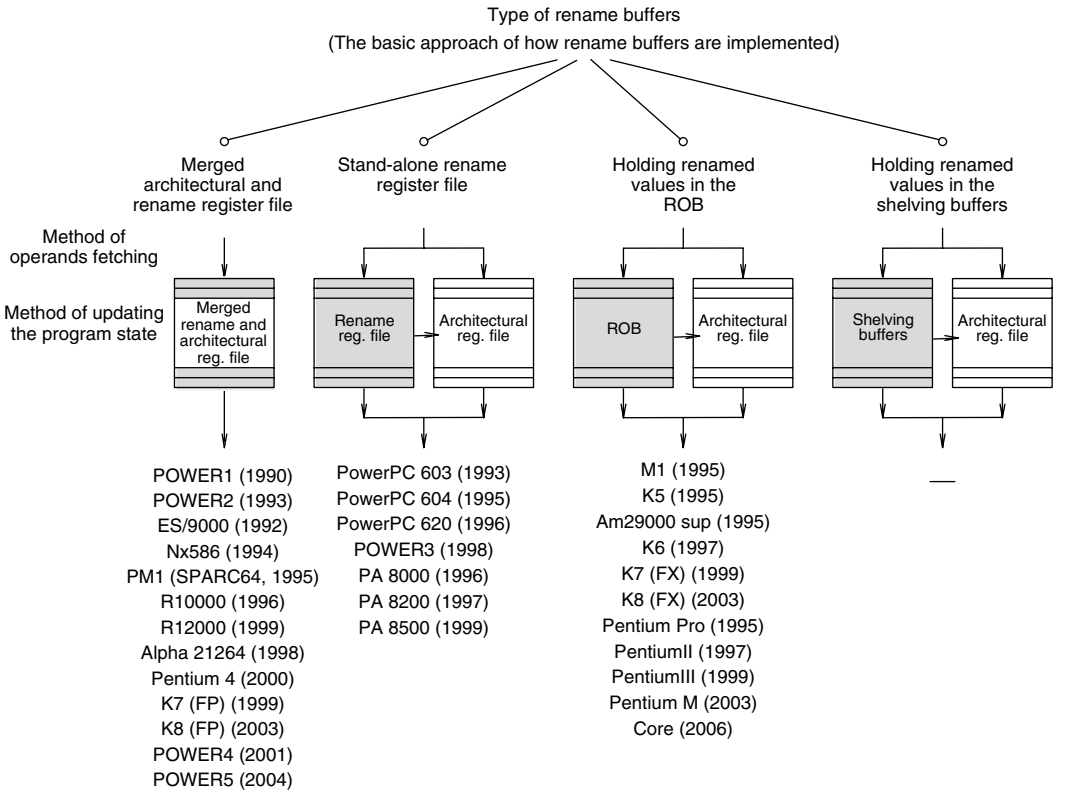



**FIGURE 2.13** Generic types of rename buffers (rename buffers are indicated by shaded boxes).

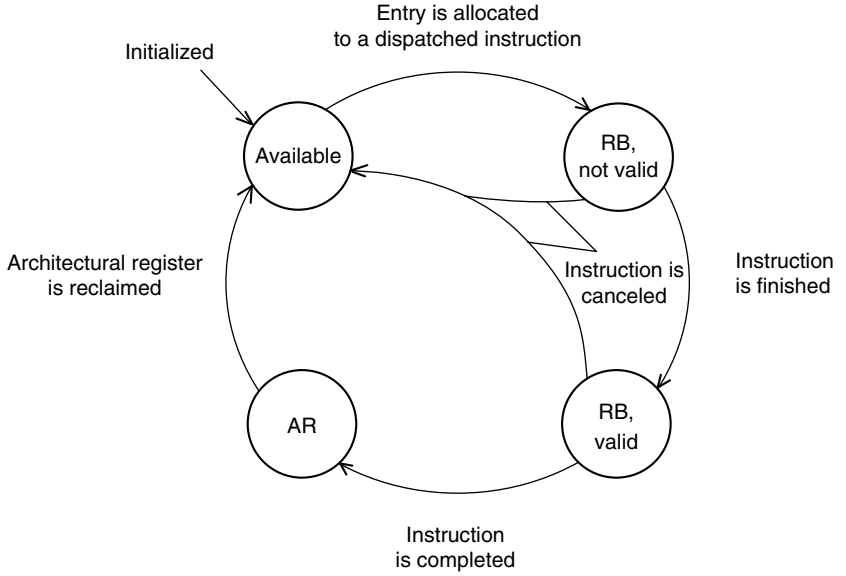



**FIGURE 2.14** State transition diagram of a particular register of the merged architectural and rename register file (AR: architectural register, RB: rename buffer). (From Liptay, J.S., *IBM Journal of Research and Development*, 36, 4, 713, 1992.)

completed instructions should be canceled because of exceptions or faultily executed speculative instructions. In this case, allocated rename buffers in the states "rename buffer, not valid" and "rename buffer, valid," are deallocated and their states are changed to "available." In addition, the corresponding mappings, kept either in the mapping table or in the rename buffer (as discussed later), need to be canceled.

Note that merged architectural and rename register files do not require a physical data transfer to update architectural registers. All that is needed for updating is to change the status of the related registers. By contrast, separate rename register implementations need, for updating the architectural registers, a physical data transfer from the rename buffer file to the architectural register file. This requires additional read and write ports on the rename register file and on the architectural register file, respectively, as well as a dedicated data path. For this reason, recent processors make increasing use of merged architectural and rename register files, e.g., the Alpha 21264, Pentium 4, or the K7, K8 (for renaming floating-point instructions).

Merged architectural and rename register files are employed furthermore, in the high end models (520-based models) of the IBM ES/9000 mainframe family, the POWER and R1x000 lines of processors.

All other alternatives separate rename buffers form architectural registers. Their respective state transition diagram is depicted in Fig. 2.8 and has already been discussed in connection with the overview of the rename process.

II. In the first separated variant, a stand-alone rename register file (or rename register file for short) is used exclusively to implement rename buffers. The PowerPC 603–PowerPC 620 and the PA8x00 line of processors are examples for using rename register files.

III. Alternatively, renaming can also be based on the reorder buffer (ROB); see related box. The ROB has recently been widely used to preserve the sequential consistency of the instruction execution. When using a ROB, an entry is assigned to each dispatched instruction for the duration of its execution. Therefore, it is quite natural to use this entry for renaming as well, basically by extending it with a new field, which will hold the result of that instruction. Examples of processors, which use the ROB for renaming, are the Am 29000 superscalar, the K5, K6, the Pentium Pro, Pentium II, Pentium III, Pentium M, and the Core line.

The ROB can even be extended further to serve as a central shelving buffer. In this case, the ROB is also occasionally designated as the DRIS (deferred scheduling register renaming instruction shelve). The lightning processor proposal [37] and the K6 made use of this solution. As the lightning proposal, which dates back to the early 1990s, was too ambitious in the light of the technology available at that time, it could not be economically implemented and never reached the market.

IV. The last conceivable implementation alternative of rename buffers is to use the shelving buffers for renaming. In this case, each shelving buffer needs to be extended functionally to perform the task of a rename buffer as well. But this alternative has a significant drawback resulting from the different deallocation mechanisms of the shelving and rename buffers. While shelving buffers can be reclaimed as soon as the instruction has been issued, rename buffers can be deallocated only at a later time, not earlier than the instruction has been completed. To date, no processor has chosen this option, so, subsequently we will neglect this alternative.

For the sake of simplicity, we have so far assumed that all data types are stored in a common architectural register file. But usually, processors provide distinct architectural register files for FX- and FP-data; consequently, they typically employ distinct rename register files, as shown in Fig. 2.15.

As depicted in Fig. 2.15, when the processor employs the split register principle, distinct FX- and FP-register files are needed for both the merged files and the stand-alone rename register files. In these cases, separate data paths are also needed to access the FX- and the FP-registers. Recent processors typically incorporate split rename registers with one exception. When renaming takes place within the ROB, usually a single mechanism is maintained for the preservation of the sequential consistency of instruction execution. Then all renamed instructions are kept in the same ROB queue, despite using split architectural register files for FX- and FP-data. In this case, clearly, each ROB entry is expected to be long enough to hold either FX- or FP-data.

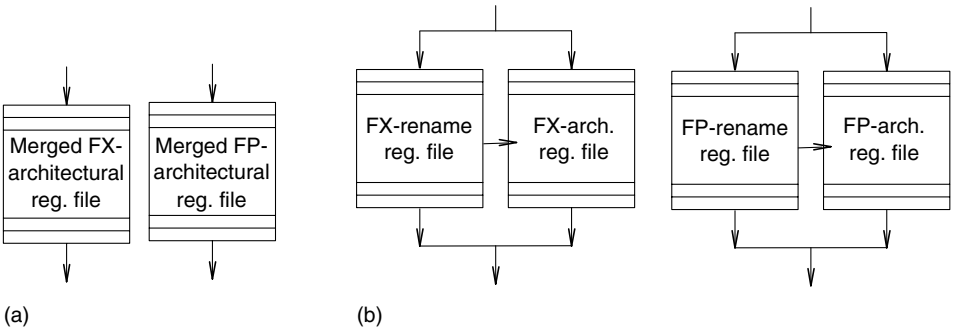



**FIGURE 2.15** Using split registers in the case of (a) merged register files, and (b) stand-alone rename register file.

### 2.2.4.3 Number of Rename Buffers

Rename buffers keep register results temporarily until instructions complete. Assuming not more than a single data result per instruction, a processor needs up to as many rename buffers as the maximum number of instructions in-flight (instructions that have been dispatched but not yet completed). Dispatched but not yet completed instructions are (1) held in shelving buffers waiting for issuing (if shelving is employed) or (2) just in processing in any of the execution units or (3) in the load queue waiting for cash access (if there is a load queue) or finally (4) in the store queue waiting for completion and later for forwarding them into the cache to execute the required store operation (if there is a store queue). Thus, the maximal number of instructions that may have been dispatched but have not yet been completed in the processor ($n_{\text{pmax}}$) is given by

$$n_{\text{pmax}} = w_{\text{dw}} + n_{\text{EU}} + n_{\text{Lq}} + n_{\text{Sq}} \tag{2.1}$$

where

$w_{\text{dw}}$ is the width of the issue window (total number of shelving buffers)
$n_{\text{EU}}$ is the number of the execution units, which may operate in parallel
$n_{\text{Lq}}$ is the number of the entries in the load queue
$n_{\text{Sq}}$ is the number of the entries in the store queue

Assuming a worst case design approach, from Eq. 2.1, we determine the total number of rename buffers required ($n_{\text{rmax}}$) as the instructions held in the store queue that do not require rename buffers.

$$n_{\text{rmax}} = w_{\text{dw}} + n_{\text{EU}} + n_{\text{Lq}} \tag{2.2}$$

Furthermore, if the processor includes a ROB, the ROB needs to maintain an entry for every dispatched but not yet completed instruction. So, on the basis of Eq. 2.1, the total number of ROB entries required ($n_{\text{ROBmax}}$) is

$$n_{\text{ROBmax}} = n_{\text{pmax}} \tag{2.3}$$

Nevertheless, if the processor has fewer rename buffers or fewer ROB entries than expected to have according to the worst case approach (as given by Eqs. 2.2 and 2.3, respectively), missing free rename buffers or ROB entries can cause dispatch blockages. With a decreasing number of entries provided, we expect a smooth and slight performance degradation. Hence, a stochastic design approach is also feasible, where the required number of entries is derived from the tolerated level of performance degradation.

On the basis of Eqs. 2.1 through 2.3, the following relations are typically valid concerning the width of the processor's issue window ($w_{dw}$), the total number of the rename buffers ($n_r$), and the reorder width ($n_{ROB}$), which equals the total number of ROB entries available:

$$w_{dw} < n_r \leq n_{ROB} \tag{2.4}$$

In Table 2.1, we summarize the type and the number of rename buffers provided in recent RISC and x86 superscalars. In addition, we give four key parameters of the enlisted processors: (1) the dispatch rate, (2) the width of the issue window ($w_{dw}$), (3) the total number of rename buffers provided ($n_r$), and (4) the reorder width ($n_{ROB}$).

As the data in Table 2.1 shows, the interrelations (Eq. 2.4) have been taken into account in the design of most processors; however, two obvious exceptions arise. First, the PowerPC 604 provides 20 rename buffers, four more buffers than the reorder width of the processor, which is 16. In the subsequent PowerPC 620, Intel corrected this by decreasing the number of rename buffers to 16. Second, the R10000

**TABLE 2.1** Type and Available Number of Rename Buffers in Recent Superscalars

| Processor Type/Year of Volume | Type of Rename Buffer | Number of Rename Buffers | | Dispatch Rate | Width of the Issue Window | Total Number of Rename Buffers | Reorder Width |
|---|---|---|---|---|---|---|---|
| Shipment | | FX | FP | | ($w_{dw}$) | ($n_r$) | ($n_{ROB}$) |
| *RISC processors* | | | | | | | |
| PowerPC 603 (1993) | ren. reg. file | n.a. | 4 | 3 | 3 | n.a. | 5 |
| PowerPC 604 (1995) | ren. reg. file | 12 | 8 | 4 | 12 | 20 | 16 |
| PowerPC 620 (1996) | ren. reg. file | 8 | 8 | 4 | 15 | 16 | 16 |
| POWER3 (1998) | ren. reg. file | 16 | 24 | 4 | 23 | 40 | 32 |
| POWER4 (2001) | Merged | 80 | 72 | 5 | 78 | 152 | 20*5 |
| POWER5 (2004) | Merged | 120 | 120 | 5 | 82 | 240 | 20*5 |
| R10000 (1996) | Merged | 32 | 32 | 4 | 48 | 64 | 32 |
| R12000 (1998) | Merged | 32 | 32 | 4 | 48 | 64 | 48 |
| Alpha 21264 (1998) | Merged | 48 | 41 | 4 | 35 | 89 | 80 |
| PA 8000 (1986) | ren. reg. file | 56 | 56 | 4 | 56 | 112 | 56 |
| PA 8200 (1987) | ren. reg. file | 56 | 56 | 4 | 56 | 112 | 56 |
| PA 8500 (1989) | ren. reg. file | 56 | 56 | 4 | 56 | 112 | 56 |
| PM1 (1996) | Merged | 38 | 24 | 4 | 36 | 62 | 62 |
| *CISC (x86) processors* | | | | | | | |
| Pentium Pro (1995) | In the ROB | 40 | | 32 | 201 | 40 | 401 |
| Pentium II (1997) | In the ROB | 40 | | 32 | 201 | 40 | 401 |
| Pentium III (1999) | In the ROB | 40 | | 32 | 201 | 40 | 401 |
| Pentium 4 (2000) (Willamette) | Merged | 128 | | 32 | n.a. | 128 | 1261 |
| Core (2006) | In the ROB | 96 | | 4 | 32 | 96 | 96 |
| Pentium 4 (2002) Northwood | Merged | 128 | | 3 | n.a. | 256? | 2*126? |
| Pentium 4 (2004) Prescott | Merged | 256 | | 3 | n.a. | 512? | 4*128? |
| Pentium M (2003) | In the ROB | 40 | | 3 | 24 | 40 | 40 |
| K5 (1995) | In the ROB | 16 | | 42 | 111(?) | 16 | 161 |
| K6 (1996) | In the ROB | 24 | | 32 | 241 | 24 | 241 |
| K7 (1999) | In the ROB/merged3 | 72 n.a. | | 32 | 54 | 88 | 24*3 |
| K8 (2003) | In the ROB/merged | 72 | 120 | 32 | 60 | 192 | 24*3 |

*Note*: In this table we also indicate four related parameters of the enlisted processors.

1 RISC operations.

2 x86 instructions (On average x86 instructions produce 1.3–1.9 RISC operations).38

3 The K7 renames FX operands in the ROB but FP operands in a merged architectural and rename register file, respectively.

? Designates questionable data.

provides only 32 ROB entries. This number is far too low compared to the issue width (48) and to the number of available rename buffers (64). MIPS addressed this disproportion in its following model, the R12000, by increasing the reorder width of the processor to 48.

A further remark relates to multithreading. When the processor makes use of multithreading, each thread represents a different instruction flow and, thus, each thread needs separate rename resources, such as rename registers. As the Northwood and the Prescott cores of the Pentium 4 line are designed to support SMT (symmetrical multithreading, called hyperthreading in Intel's terminology), these cores are assumed to provide multiple sets of rename resources as well [57]. According to the cited publication, the Northwood core provides two sets and the Preston core even four sets of rename registers, presumably to support twofold and fourfold multithreading, respectively. Nevertheless, Intel did not reveal the exact number of rename registers implemented in these cores.

#### 2.2.4.4 Number of Read and Write Ports

By taking into account prevailing practice, in our subsequent discussion we assume split register files.

First, let us focus on the required number of read ports (output ports). Clearly, rename buffers need as many read ports as there are data items to be supplied by the rename buffers in any one cycle. In this respect, we should take into account that rename buffers supply the required operands for the instructions to be executed and also forward the results of the completed instructions to the addressed architectural registers.

The number of operands, which need to be delivered in the same cycle, depends first of all on whether the processor fetches operands during instruction dispatch or during instruction issue.

If operands are fetched dispatch bound, the rename buffers need to supply the operands for all instructions, which are dispatched in the same cycle into the shelving buffers. If there are no dispatch restrictions, both the FX- and the FP-rename buffers are expected to deliver in each cycle all required operands for up to as many instructions as the dispatch rate. For instance, this means that in recent four-way superscalar RISCs, the FX- and the FP-rename buffers typically need to supply 8 and 12 operands, respectively, assuming up to two FX- and three FP-operands for each FX- and FP-instruction, respectively. If, however, there are some dispatch restrictions, obviously less read ports are required.

By contrast, if the processor employs the issue-bound fetch policy, the rename buffers should provide the operands for all instructions, which are forwarded from the issue window (instruction window) for execution in the same cycle. In this case, the FX-rename buffers need to supply the required FX-operands for the integer units and for the load/store units (including register operands for the specified address calculations and FX-data for the FX-store instructions). As far as the FP-rename buffers are concerned, they need to deliver operands for the FP- and MM-units (FP-register data) and also for the load/store units (FP-operands for the FP-store instructions). In the POWER3, for instance, this implies the following read port requirements. The FX-rename buffers need to have 12 read ports (up to $3 \times 2$ operands for the three integer units as well as $2 \times 2$ address operands and $2 \times 1$ data operands for the two load/store units). On the other hand, the FP-rename registers need to have eight read ports (up to $2 \times 3$ operands for the two FP-units and $2 \times 1$ operands for the two load/store units).

In addition, if rename buffers are implemented separately from the architectural registers, the rename buffers need to be able to forward in each cycle as many result values to the architectural registers as the completion rate (retire rate) of the processor. As recent RISCs usually complete up to four instructions per cycle, this task increases the required number of read ports in the rename buffers in these processors by four.

We note here that too many read ports in a register file may unduly increase the physical size of the data path and thus the cycle time as well. To avoid this problem, a few high-performance processors (such as the POWER2, POWER3, and the Alpha 21264) implement two copies of particular register files. The POWER2 duplicates the FX-architectural register file, the POWER3 doubles both the FX-rename and the FX-architectural files, and the Alpha 21264 has two copies of the FX-merged architectural and rename register file. As a result, fewer read ports are needed in each of the copies. For example, with two copies of the FX-merged register file, the POWER3 needs only 10 read ports in each file, instead of

16 read ports in one FX-register file. A drawback of this approach is, however, that a scheme is also required to keep both copies coherent.

Now let us turn to the required number of write ports (input ports). Since rename buffers need to accept in each cycle all results produced by the execution units, they need to provide as many write ports as many results the execution units may produce per cycle. The FX-rename buffers receive results from the available integer-execution units and from the load/store units (fetched FX-data), whereas the FP-rename buffers hold the results of the FP-execution units and of the load/store units (fetched FP-data). Most results are single data items requiring one write port. However, there are a few exceptions. When execution units generate two data items they require two write ports as well; like the load/store units of PowerPC processors. For instance, after execution of the LOAD-WITH-UPDATE instruction, these units return both the fetched data value and the updated address value.

## 2.2.5 Layout of the Register Mapping

### 2.2.5.1 Overview

Register mapping includes three main tasks, as depicted in Fig. 2.16:

1. The processor needs to allocate rename buffers to the destination registers of the dispatched instructions.
2. It also must keep track of the valid mappings for two reasons:
   a. To forward generated results to the right rename buffers.
   b. To deliver the correct operand values when they are needed in the course of instruction processing.
3. It needs to deallocate no longer needed rename buffers.

### 2.2.5.2 Allocation Scheme of Rename Buffers

Processors usually allocate rename buffers to every dispatched instruction rather than only to those including a destination register in order to simplify logic. Although rename buffers are not needed until the instruction results are generated in the last execution cycle, rename buffers are typically allocated to the instructions as early as during instruction dispatch. This kind of register allocation leads to wasted rename register space. Delaying the allocation of rename buffers to the instructions until instructions finish [39] saves rename register space. Various schemes have been proposed for this, such as virtual renaming [39–42] and others [43]. In fact, a virtual allocation scheme has already been introduced in the POWER3 [39].

### 2.2.5.3 Method of Keeping Track of Actual Register Mapping

There are three possibilities for keeping track of the actual mapping of the architectural registers to the allocated rename buffers: (1) The processor can use a mapping table for this, (2) it can simplify the tracking task by means of a future file, or (3) it can track register renames within the rename buffers themselves. In the following section we outline these methods, which are illustrated in Fig. 2.17.

A mapping table has as many entries as there are architectural registers in the instruction set architecture (ISA), usually 32 for RISCs. Each entry holds a status bit (called the entry valid bit in the figure), which indicates whether the associated architectural register is renamed. Operands from

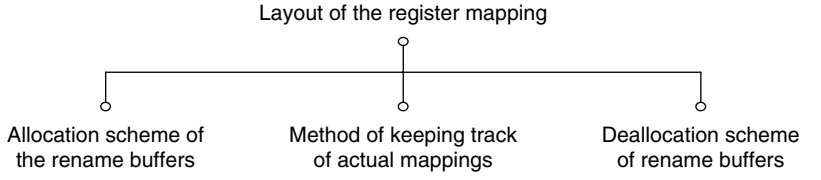



**FIGURE 2.16** Layout of the register mapping.

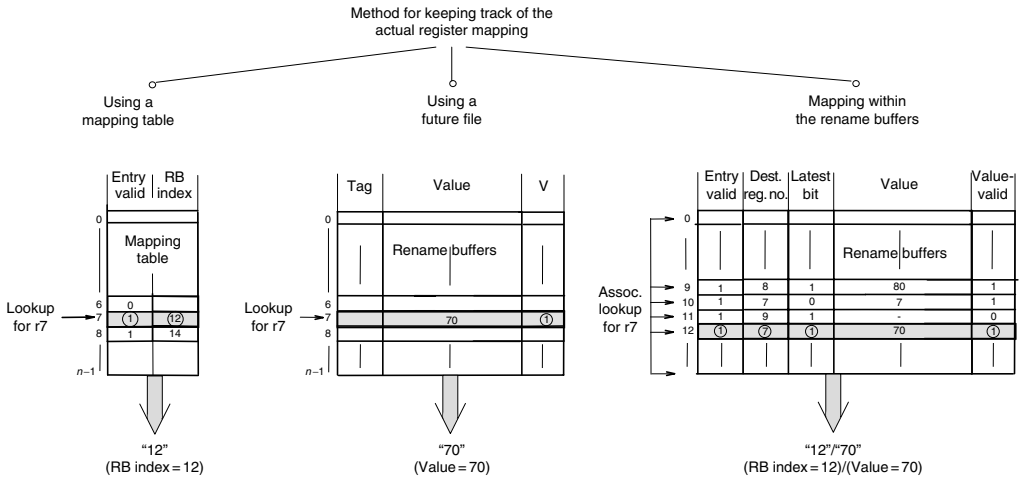



**FIGURE 2.17** Methods for keeping track of the actual mapping of architectural registers to rename buffers.

renamed registers will be accessed from the rename buffers, whereas operands from not renamed registers from the architectural register file. Each valid entry supplies the index of the rename buffer, which is allocated to the architectural register belonging to that entry (called the RB-index). For instance, the left-hand side of Fig. 2.17 shows that the mapping table holds a valid entry for architectural register $r_7$, which contains the RB-index of 12, indicating that the architectural register $r_7$ is actually renamed to rename buffer number 12 or vice versa; rename buffer RB 12 will hold or already holds the generated value for $r_7$, depending on the value of the value-valid bit of the associated rename buffer. To prepare operand access, source registers of dispatched instructions are renamed simply by accessing the mapping table with the register numbers as indices and fetching the associated rename buffer identifiers (RB-indices), assuming that there is a valid renaming for that particular register, indicated by the "entry valid" field of the entry, as shown in Fig. 2.17.

As already mentioned in the previous section, usually each entry is set up during instruction dispatch when new rename buffers are allocated to the destination registers of the dispatched instructions. A new entry is created by setting the "entry valid" bit and writing the index of the allocated rename buffer into the field "RB index." A valid mapping is updated when the architectural register belonging to that entry is renamed again, and it will be invalidated when the instruction associated with the actual renaming completes. In this way, the mapping table continuously holds the latest allocations.

We note that for split architectural register files obviously separate FX- and FP-mapping tables are needed. Mapping tables should provide one read port for each source operand that may be fetched in any one cycle, and one write port for each rename buffer that may be allocated in any one cycle (as discussed earlier in the Section 2.2.4.4).

The second option for tracking register allocations is based on the future file concept. Originally introduced for implementing precise interrupts in pipelined processors in the middle of the 1980s [46], the future file has the same number of entries as the architectural register file and holds the most recent values produced for the architectural registers so far. In connection with renaming, the future file is used for holding the latest values of the renamed (temporarily buffered) register values and delivering those values (if available else their tags) when the operands are accessed. Subsequently, with reference to Fig. 2.18, we describe the operation of the future file in more detail, assuming that the processor makes use of shelving, accesses operands dispatch-bound, and holds instruction results (that is renamed values) in the ROB.

As instructions are dispatched to the reservation stations (RS) in the future file, the processor clears the "value-valid" bits belonging to the destination registers (Rd) of the instructions dispatched in order

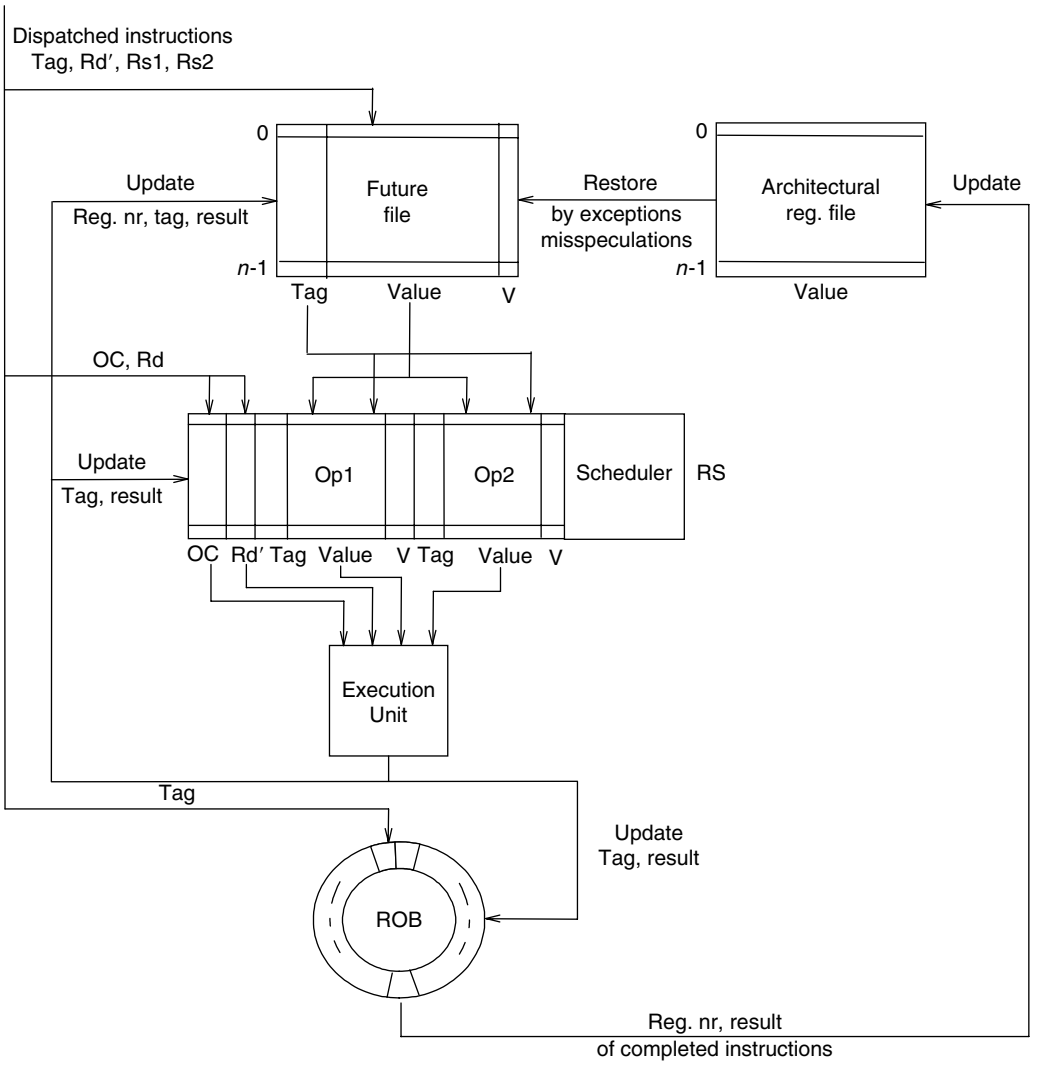



**FIGURE 2.18** Principle of using a future file for keeping track of actual register mappings, assuming shelving with issue-bound operand fetching and a ROB for holding instruction result temporarily.

to indicate a not yet available value and marks a missing value by the tag (Tag) of the instruction to prepare a later updating when one of the available execution units generates the result. Furthermore, the future file is accessed by the referenced source operand identifiers (Rs1, Rs2) and delivers the requested operand values if they are available (i.e., their value-valid bits are set), else is accessed by their tags to the RS. Dispatched instructions remain in the RS and wait for their missing operands. The scheduler checks the instructions held in the RS in each clock cycle and issues the oldest instruction that owes all required operand values to the associated execution unit. The generated result is then written into the ROB into the associated entry, that is, into the entry carrying the same tag as the result. In addition, both the future file and the RS will be updated by the generated result. The RS needs an associative access to update all operands waiting for the particular result that is holding the matching tag. The future file is updated only if the referenced register entry has the same tag as the forwarded result; else the forwarded result is not the latest one belonging to the referenced register since a subsequent instruction has already rewritten its value.

When instructions complete in program order their results are written into the architectural register file to update the program state. In case of mispredicted branches, misspeculated loads, etc., or accepted exceptions, the future file is flushed and restored with the content of the architectural register file.

The third fundamentally different alternative for keeping track of the actual register mappings relies on an associative mechanism (see the right-hand side of Fig. 2.17) and is called mapping within the rename buffers. In this case, no mapping table exists but each rename buffer holds the identifier of the associated architectural register (usually the register number of the renamed destination register) and additional status bits as well. These entries are set up usually during instruction dispatch when a particular rename buffer is allocated to a specified destination register. As Fig. 2.17 shows, in this case each rename buffer holds five pieces of information: (1) a status bit, which indicates that this rename buffer is actually allocated (called the entry valid bit in the figure); (2) the identifier of the associated architectural register (Dest. reg. no.); (3) a further status bit, called the "latest bit," whose role will be explained subsequently; (4) another status bit, called the "value-valid" bit, which shows whether the actual value of the associated architectural register has already been generated; and finally (5) the value itself (value), provided that the value-valid bit signifies an already produced result. The latest bit is needed to mark the last allocation of a given architectural register if it has more than one valid allocation due to repeated renaming. For instance, in our example, architectural register $r_7$ has two subsequent allocations. From these, entry number 12 is the latest one as its latest bit has been set. Thus, in our figure, renaming of the source register $r_7$ would yield the RB-index of 12. We point out that in this method source registers are renamed by an associative lookup for the latest allocation of the given source register.

If operands are fetched dispatch bound, source registers are both renamed and accessed during the dispatch process. Then processors usually integrate renaming and operand accessing, and therefore maintain register mapping within the rename buffers or use a future file. For issue-bound operand fetching, however, these tasks are separated. Source registers are usually renamed during instruction dispatch, whereas the source operands are accessed while the processor issues the instructions to the execution units. Therefore, in this case, processors typically use mapping tables.

#### 2.2.5.4 Deallocation Scheme of Rename Buffers

If rename buffers are no longer needed, they should be reclaimed (deallocated). The actual scheme of reclaiming depends on key aspects of the overall rename process. In particular, it depends on the allocation scheme of the rename buffers, the type of rename buffers used, the method of keeping track of actual allocations, and even whether operands are fetched dispatch bound or issue bound. Here, we do not go into details, but refer to Section 2.2.6.1 for a few examples on how processors reclaim rename registers.

#### 2.2.5.5 Rename Rate

As its name suggests, the rename rate stands for the maximum number of renames that a processor is able to perform in a cycle. Basically, the processor should be able to rename all instructions dispatched in the same cycle to avoid performance degradation. Thus, the rename rate should equal the dispatch rate. This is easier said than done because it is not at all an easy task to implement a high rename rate (four or higher). This is true for two reasons. First, for higher rename rates, the detection and handling of inter-instruction dependencies during renaming (as discussed later in Section 2.2.6.1) becomes a more complex task. Second, higher rename rates require a larger number of read and write ports on register files and on mapping tables. For instance, the four-way superscalar R10000 can dispatch any combination of 4 FX- and FP-instructions. Accordingly, its FX-mapping table needs 12 read and 4 write ports, and its FP-table requires 16 read and 4 write ports. This number of ports is needed since FX-instructions can refer up to three and FP-instructions up to four source operands in this processor.

Another example worth looking at is the PM1, also called SPARC64. This four-way superscalar processor can dispatch any combination of 4 FX- and 2 FP-instructions, up to a maximum of four instructions. In this case, both the FX-mapping table and the merged register file have 10 read and 4 write ports while its FP-counterpart has 6 read and 3 write ports. According to Asato et al. [44],

this 14-port 116 word 64-bit merged register file needs 371 K transistors, far more than the entire Intel 8086 processor (about 30 K transistors) or slightly more than the i386 (about 275 K transistors) [45].

### 2.2.6 Basic Alternatives and Possible Implementation Schemes of Register Renaming

In the design space of register renaming, theoretically each possible combination of the available design choices yields one possible implementation alternative. Instead of considering all possible implementation alternatives, it makes sense to focus only on those, which differ in relevant qualitative aspects from each other. We designate these alternatives the basic alternatives. Possible basic alternatives can be derived from the design space in two steps—first by identifying the relevant qualitative design aspects and then by composing their possible combinations. Concerning the selection of the relevant qualitative design aspects, we recall the design space of renaming, shown in Fig. 2.10. First, we can ignore two main aspects, the scope of register renaming, as recent processors typically implement full renaming, and the rename rate, because of its quantitative character. Thus, two main design aspects remain; the layout of the rename buffers and the implementation of register mapping. Furthermore, as Fig. 2.12 indicates, the layout of the rename buffers itself covers three design aspects: the type of rename buffers, the number of rename buffers, and the number of the read and write ports. Of these, only the type of the rename buffers is of qualitative character. From the design aspect layout of the register mapping (Fig. 2.16), we consider the method of keeping track of actual mappings the only relevant aspect. It follows that the design space of register renaming includes only two relevant qualitative aspects: the type of the rename buffers and the method of keeping track of actual mappings.

The design choices available for these two relevant design aspects result in nine possible combinations, called the basic alternatives for register renaming (as shown in Fig. 2.19), if we neglect the unpromising possibility to implement rename buffers in the shelving buffers. In addition, as the operand fetch policy of the processor, which is a design aspect of shelving, significantly affects how the rename process is carried out, in this figure we also take into account this aspect. This splits the nine basic renaming alternatives into 18 feasible implementation schemes. In this figure, we also indicate the implementation schemes that are used in relevant superscalar processors, as well as give some hints about their origins.

As Fig. 2.19 indicates, out of the nine possible basic alternatives of renaming, relevant superscalar processors make use only of five. Moreover, latest processors employ mostly the following four basic alternatives of renaming:

1. Use of merged architectural and rename register files and of mapping tables (POWER4, POWER5, Pentium 4, as well as K7 (Athlon) and K8 (Hammer) for floating-point processing)
2. Use of separate rename register files and mapping registers within the rename buffers (PA8x00 line, Power3)
3. Renaming within the ROB and using mapping tables (Pentium Pro, Pentium II, Pentium III, Pentium M, Core)
4. Renaming within the ROB and using a future file (K7 (Athlon) and K8 (Hammer) for fixed point processing)

We emphasize that a few processors use different basic alternatives for renaming FX- and FP-instructions, as is manifested for instance in the K7 and K8 processors. These processors use the ROB for renaming FX-instructions and a merged architectural and rename register file for renaming floating-point ones.

#### 2.2.6.1 Implementation of the Rename Process

With reference to Section 2.2.2, we emphasize that the rename process can be broken down into the following subtasks:

1. Renaming the destination registers
2. Renaming the source registers

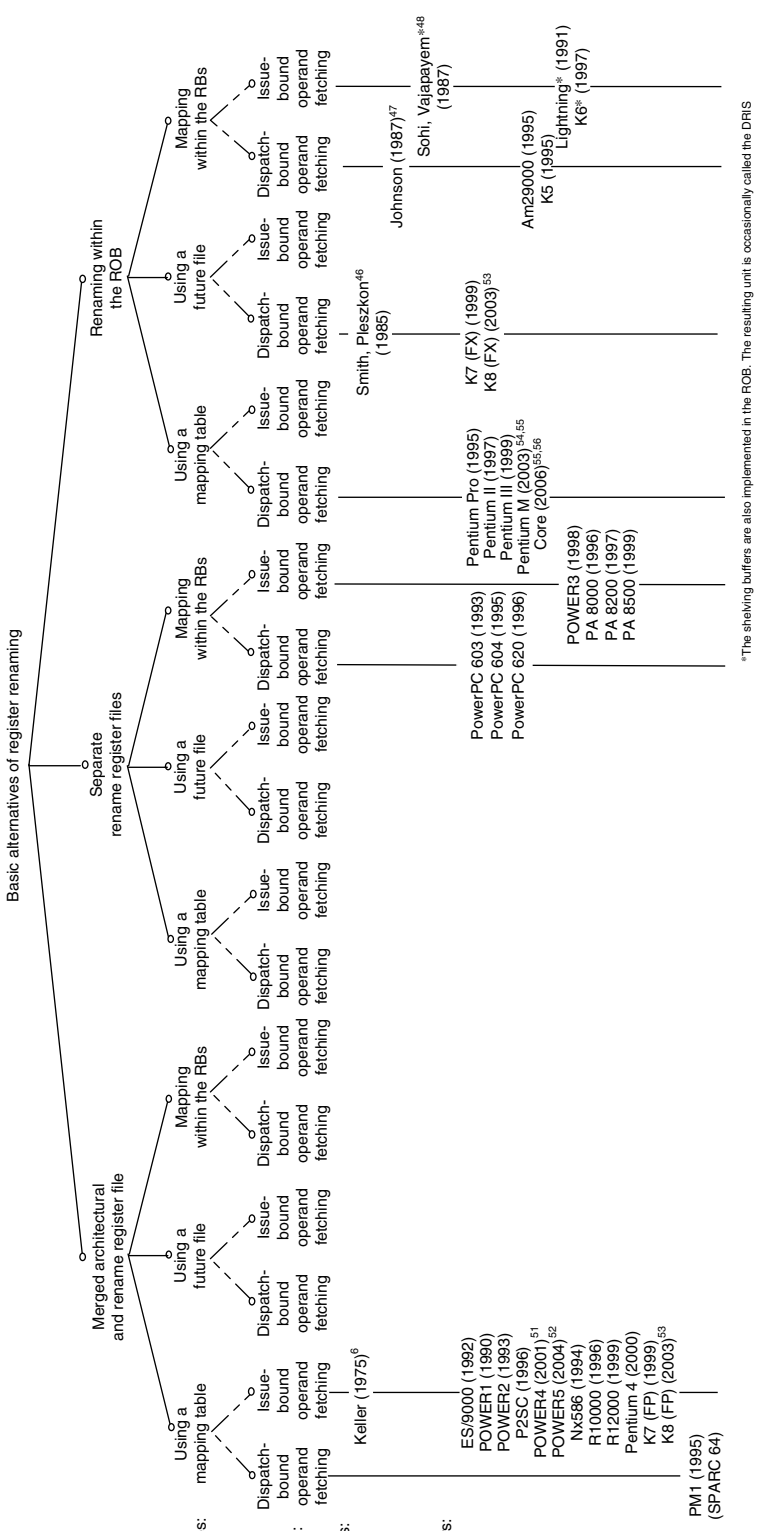



**FIGURE 2.19** Basic implementation alternatives of register renaming (RB designates rename buffer).

3. Fetching the renamed source operands
4. Updating the rename buffers
5. Updating the architectural registers with the content of the rename buffers
6. Reclaiming of the rename buffers
7. Recovery from wrongly performed speculative execution and handling of exceptions

These subtasks are carried out more or less differently in the 18 distinct implementation schemes of renaming.

Of these, in Section 2.2.2 we described the rename process presuming one particular basic alternative (assuming the use of rename register files and mapping tables) in both operand-fetch scenarios that is in two implementation schemes. Below, instead of pointing out all differences in all further implementation schemes of register renaming, we focus only on three particular tasks of renaming and point out significant differences encountered in different implementation schemes. In addition, we briefly discuss how inter-instruction dependencies are dealt with during renaming, how the processor recovers from misspeculations, and how it handles exceptions.

#### *2.2.6.1.1 Remarks on Renaming Destination Registers*

The way how the processor allocates new rename buffers depends on the type of rename buffers used. If rename buffers are realized in the ROB, a new ROB entry, and thereby a new rename buffer will automatically be allocated to each dispatched instruction. Else rename buffers need to be allocated only to those dispatched instructions, which include a destination register.

#### *2.2.6.1.2 Remarks on Updating the Architectural Registers*

As discussed previously, when instructions complete, their results need to be forwarded from the associated rename buffers into the originally addressed architectural registers. In cases where rename buffers are implemented separately from the architectural register file (as a stand-alone rename register file, or they are in the ROB or in the shelving buffer file), this task instructs the processor to physically transfer the contents of the related rename buffers into the referenced architectural registers. By contrast, if the processor uses a merged architectural and rename file, no physical data transfer is required; instead only the status of the related registers needs to be changed, as indicated before and shown in Fig. 2.15.

#### *2.2.6.1.3 Remarks on Reclaiming Rename Buffers*

The conditions for reclaiming no longer used rename buffers vary with the rename scheme employed. Thus, when operands are fetched dispatch bound, associated rename buffers may immediately be reclaimed after an instruction has been completed. On the other hand, if the processor fetches operands issue bound, associated rename buffers may only be reclaimed after the related instruction has been completed and, in addition, if it is also sure that no outstanding operand fetch requests are available to that rename buffer. The latter condition can be checked in different ways. One possibility is to use a counter for each rename buffer for checking outstanding fetch requests, as described in Section 2.2.2. Another option is applicable with merged architectural and rename register files. In this case, however, during instruction execution, a rename buffer becomes an architectural register and reclaiming is related to no longer used architectural registers, as discussed in Section 2.2.4.2. This method relies on keeping track of the most recent earlier instance of the same architectural register, and on reclaiming it when the instruction giving rise to the new instance completes [28].

#### *2.2.6.1.4 Renaming of Destination and Source Registers if Inter-Instruction Dependencies Exist between the Instructions Dispatched in the Same Cycle*

As we know, shelving relieves the processor of the need to check for data and control dependencies as well as for busy EUs during instruction dispatch. Nevertheless, despite shelving, instructions dispatched in the same cycle must still be checked for inter-instruction dependencies, and, in the case of dependencies, the rename logic must be modified accordingly. Let us assume, for instance, that there are RAW dependencies between two subsequent instructions dispatched in the same cycle, as in the following example:

$i_1$: mul r2,...,...
$i_2$: add..., r2,...

Here, $i_2$ needs the result of $i_1$ as r2 is one of its source operands. We will also assume that the destination register of $i_1$ (r2) will be renamed to r33 as follows:

$i'_1$: mul r33,...,...

In this case, the RAW-dependent source operand of $i_2$ (r2) has to be renamed to r33 rather than to the rename buffer allocated before renaming of $i_1$ to r2.

Similarly, if WAW dependencies exist among the instructions dispatched in the same cycle, as for instance, between the instructions

$i_1$: mul r2,...,...
$i_2$: add r2,...,...

Obviously, different rename buffers need to be allocated to the destination registers of $i_1$ and $i_2$, as shown below:

$i'_1$: mul r34,...,...
$i'_2$: add r35,...,...

#### *2.2.6.1.5 Recovery of the Rename Process from Wrongly Executed Speculation and Handling of Exceptions*

If the processor performs speculative execution, for instance, due to branch prediction, it may happen that the speculation turns out to be wrong. In this case, the processor needs to recover from the misspeculation. This involves essentially two tasks: (i) to undo all register mappings setup, and (ii) to reclaim rename buffers allocated, as already discussed. To invalidate established mappings there are two basic methods to choose from, independent of the actual implementation of renaming. The first option is to roll back all register mappings made during speculative execution, by using the identifiers of the faulty instructions, supplied by the ROB. While using this alternative, the recovery process lasts several cycles, since the processor can cancel only a small number of instructions (two to four) per cycle. A second alternative is based on checkpointing. In this method, before the processor begins with speculative execution, it saves the relevant machine state, including also the actual mapping, in shadow registers. If the speculative execution turns out to be wrong, the processor restores the machine state in a single cycle by reloading the saved state. For instance, both the PM1 (SPARC64) and the R10000 use checkpointing for recovery. Both processors incorporate mapping tables for register mapping, while the R10000 provides four sets of shadow registers and the PM1 16 for subsequent speculations.

We note that beyond the two basic methods discussed above, there is also a third option in the case when the processor uses mapping tables and issue-bound operand fetching. This method relies upon shadow mapping tables, which keep track of the actual mappings of the completed instructions. The entries of the shadow tables are set up when instructions complete and are deleted when allocated rename buffers are reclaimed. In the case of misspeculation, the correct state of the mapping table can be restored by loading the content of the shadow table. For example, Cyrix's M3 makes use of this recovery mechanism.

The second task to be done during misspeculation is to reclaim rename buffers, which are allocated to the faulty instructions. This task can easily be performed by changing the state of the rename buffers involved to available, as indicated in Figs. 2.8 and 2.14.

A similar situation to the above described misspeculation arises when exceptions occur. In this case, the exception request must wait until the associated instruction comes to completion to provide precise exceptions [46]. At this time, the processor accepts the exception and cancels all instructions, which have

been dispatched after the failing one. For cancellation of the rename process, the same methods can be used as discussed above. For example, in the event of an exception the R10000 rolls back all younger register mappings made, whereas the PM1 first restores the mapping state to the first checkpoint after the failing instruction in one cycle, and then rolls back the remaining mappings until the failing instruction is reached.

## Appendix A Types of Data Dependencies

Data dependencies [1–3] are precedence requirements between operands of subsequent instructions. Data dependencies may occur in two different situations: either in straight line code segments called inter-instruction dependencies, or between operands of instructions occurring in subsequent loop iterations, designated as recurrences (see Fig. A.1). In both situations, either register operands or memory operands may be involved.

Inter-instruction dependencies may be broken down into read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies, as depicted in Fig. A.2. In the following overview of these types of dependencies, we confine ourselves to register operands, but the given interpretation can be applied to memory operands as well.

RAW dependencies, designated also as flow dependencies, are producer–consumer relations between operands, which can be bisected into load-use and define-use dependencies (see Fig. A.2). Load-use dependencies arise in scenarios when an instruction uses a register operand, which needs to have been loaded by a preceding load instruction from the memory, as shown in the example in Fig. A.2. If, however, the requested operand is produced by a preceding operational instruction, the arising dependency is called define-use dependency, as illustrated in Fig. A.2.

WAR dependencies or anti-dependencies arise between instructions if a given instruction reads an operand from a particular register and a subsequent instruction writes the same register, as depicted in Fig. A.2. If, for any reason, the subsequent instruction ($i_2$) would have written this register before it is read by the previous one ($i_1$), then the subsequent instruction would pick up an erroneous operand value.

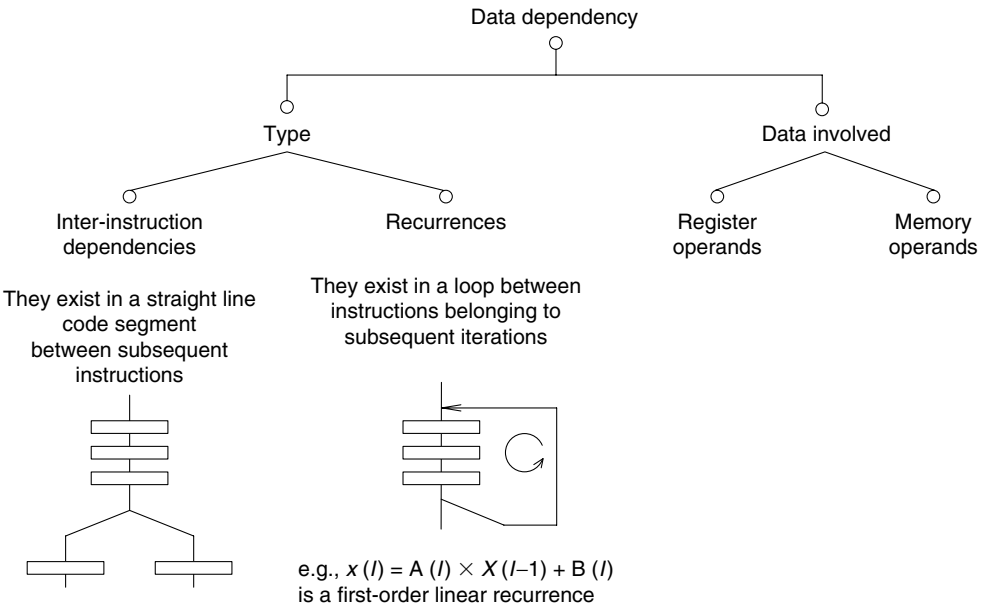



**FIGURE A.1** Main aspects of data dependencies. (In this and in subsequent figures relevant aspects and possible alternatives are illustrated by using DS-trees.) (From Sima, D., Fountain, T., and Kacsuk, P., *Advanced Computer Architectures,* Addison Wesley Longman, Harlow, 1997; Sima, D., *IEEE Micro*, 17, Sept./Oct., 29, 1997.)

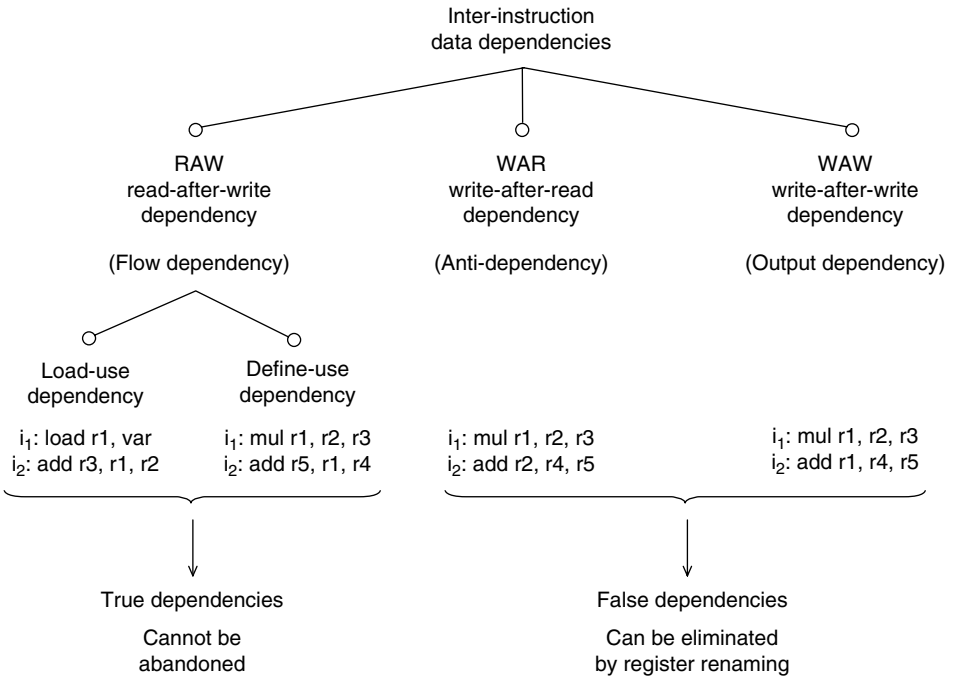



**FIGURE A.2** Terms relating to data dependencies occurring in straight line code. (Instruction semantics is r1 ← r2*r3 etc.)

Finally, two instructions are said to be WAW dependent, or output dependent, if they both write the same destination. WAR and WAW dependencies are designated as false dependencies, since they can be removed by appropriate techniques (i.e., register renaming in the case of register operands). By contrast, RAW dependencies are true dependencies, since they cannot be eliminated.

Data dependencies may also occur in loops. This is the case if an instruction of the loop body is dependent on an instruction belonging to a previous loop iteration, as exemplified in Fig. A.1. This type of dependency is called recurrence, designated also as inter-iteration data dependency or loop-carried dependency. In the above example, the value of $X(I)$ depends on the value that is computed in the previous iteration. The recurrence shown is a first-order linear one.

## Appendix B Principle of Instruction Shelving

Instruction shelving (also known as indirect dispatch or dynamic instruction scheduling) [1–3,49] removes the dispatch bottleneck caused by control and data dependencies and by busy execution units. Its main idea is to shelve dispatched instructions and defer dependency checking until a subsequent processing step, designated as issuing.

Without shelving (see Fig. B.1) the processor dispatchs instruct ions from the so-called dispatch window (instruction window), to the execution units (EU). Actually, the dispatch window comprises the last $n$ entries of the instruction buffer (I-buffer), where $n$ is the dispatch rate. The processor decodes the instructions kept in the window and checks for dependencies between the instructions in the window and those in execution, and also among the instructions held in the window. Dependent instructions are not dispatched; moreover, depending on the dispatch policy of the processor [36], they can even block the dispatch of subsequent not dependent instructions. Occurring blockages heavily restrict the average number of instructions dispatched per cycle and thus also processor performance.

Shelving removes the dispatch bottleneck by decoupling instruction dispatch and dependency checking through buffering dispatched instructions, as indicated in Fig. B.2. There are various possibilities as to how shelving buffers can be implemented [49]. Of these, in Fig. B.2 we show shelving buffers provided in front of each execution unit (EU), also called individual reservation stations or simply reservation stations. With shelving, instructions are dispatched first to the shelving buffers, with no checks for data dependencies or busy execution units. In the second step, instructions held in the shelving buffers are issued for execution. During issuing, instructions are checked for dependencies and not dependent instructions are forwarded to free execution units. Concerning terminology we note that at the time being there is no consensus on the use of the terms instruction dispatch and instruction issue. Both terms are used in both possible interpretations.

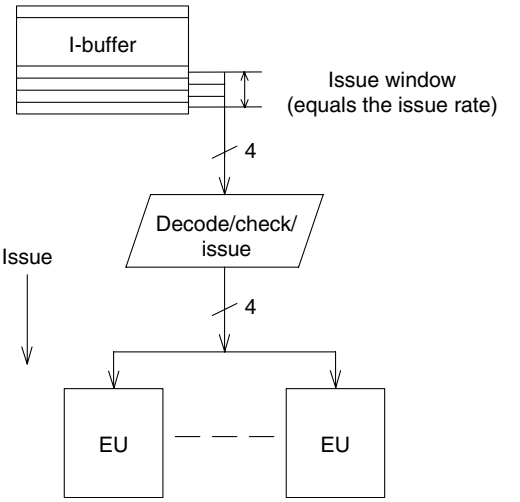



**FIGURE B.1** Principle of direct dispatch.

Shelving not only removes the dispatch bottleneck but substitutes the dispatch window with the much wider issue window (instruction window), which is made up of all shelving buffers. Because the total number of the shelving buffers is usually an order of magnitude higher than the dispatch rate,

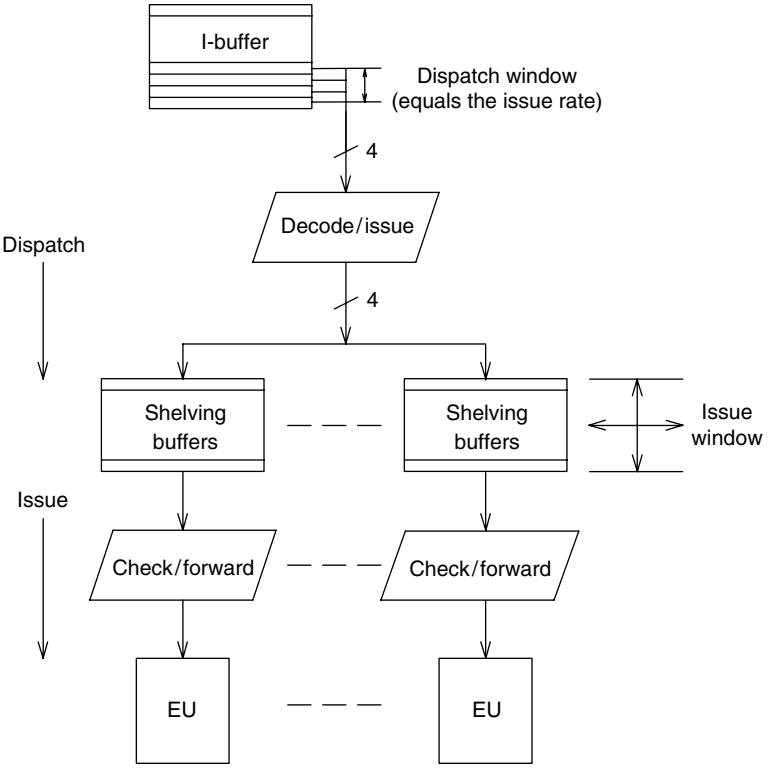



**FIGURE B.2** Principle of shelving (indirect dispatch).

with shelving the processor will find in each clock cycle on the average far more executable instructions than without shelving. Thus, shelving substantially raises the sustained instruction throughput rate of the processor. Although conceived as early as in the middle of the 1960s for the first instruction level parallel (ILP) processors (the CDC6600 [50] and the IBM 360/91) [4], because of the complexity of its implementation, shelving only came into widespread use more than two decades later in superscalars.

## Appendix C Operand Fetch Policies

If the processor uses the dispatch-bound fetch policy [3] it fetches referenced register operands during instruction dispatch, that is while it forwards decoded instructions into the shelving buffers [3,36]. In contrast, the issue-bound fetch policy defers operand fetching until executable instructions are forwarded from the shelving buffers to the execution units. When the processor fetches operands dispatch bound, shelving buffers hold the source operand values. In contrast, in the case of issue-bound operand fetching, shelving buffers have much shorter entries, as they contain only the register identifiers. Here we note that the terms instruction dispatch and issue are not used unanimously in the literature, both terms are used for both tasks.

## Appendix D Principle of the Reorder Buffer (ROB)

It is implemented basically as a circular buffer whose entries are allocated and deallocated by means of two revolving pointers [46]. The ROB operates as follows. When instructions are dispatched, a ROB entry is allocated to each instruction strictly in program order. Each ROB entry keeps track of the execution status of the associated instruction. The ROB allows instructions to complete (commit, retire) only in program order by permitting an instruction to complete only if it has finished its execution and all preceding instructions are already completed. In this way, instructions update the program state in exactly the same way as a sequential processor would have done. After an instruction has completed, the associated ROB entry is deallocated and becomes eligible for reuse.

## References

1. Rau, B.R. and Fisher, J.A., Instruction level parallel processing: History, overview and perspective, *The Journal of Supercomputing,* 7, 9, 1993.
2. Smith, P.E. and Sohi, G.S., The microarchitecture of superscalar processors, *Proceedings of the IEEE,* 83, 1609, 1995.
3. Sima, D., Fountain, T., and Kacsuk, P., *Advanced Computer Architectures,* Addison Wesley Longman, Harlow, 1997.
4. Tomasulo, R.M., An efficient algorithm for exploiting multiple arithmetic units, *IBM Journal of Research and Development,* 11, 1, 25, 1967.
5. Tjaden, G.S. and Flynn, M.J., Detection and parallel execution of independent instructions, *IEEE Transactions on Computers,* C-19, 889, 1970.
6. Keller, R.M., Look-ahead processors, *Computing Surveys,* 7, 177, 1975.
7. Leibholz, D. and Razdan, R., The Alpha 21264: A 500 MIPS out-of-order execution microprocessor, in *Proceedings of COMPCON*, 1997, 28.
8. Kurpanek, G., Chan, K., Zheng, J., CeLano, E., and Bryg, W., PA-7200: A PA-RISC processor with integrated high performance MP bus interface, in *Proceedings of COMPCON*, 1994, 375.
9. Hunt, D., Advanced performance features of the 64-bit PA-8000, in *Proceedings of COMPCON*, 1995, 123.
10. Scott, A.P. et al., Four-way superscalar PA-RISC Processors, *Hewlett-Packard Journal,* Aug., 1, 1997.
11. Lesartre, G. and Hunt, D., PA-8500: The continuing evolution of the PA-8000 Family, PA-8500 Document, Hewlett-Packard Company, 1998.

12. Grohoski, G.F., Machine organization of the IBM RISC System/6000 processor, *IBM Journal of Research and Development,* 34, 1, 37, 1990.
13. White, S. and Reysa, J., PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000, IBM Corporation, 1994.
14. Gwennap, L., IBM crams POWER2 onto single chip, *Microprocessor Report,* 10, 11, 14, 1996.
15. Becker, M. et al., The PowerPC 601 microprocessor, *IEEE Micro,* 13, Oct., 54, 1993.
16. Burgess, B. et al., The PowerPC 603 microprocessor, *Communications of the ACM,* 37, 6, 34, 1994.
17. Song, S.P. et al., The PowerPC 604 RISC microprocessor, *IEEE Micro,* 141, 8, 1994.
18. Ogden, D. et al., A new PowerPC microprocessor for low power computing systems, in *Proceedings of COMPCON*, 1995, 281.
19. Levitan, D. et al., The PowerPC 620 microprocessor: A high performance superscalar RISC microprocessor, in *Proceedings of COMPCON*, 1995, 285.
20. Song, S.P., IBM's POWER3 to replace P2SC, *Microprocessor Report,* 11, 15, 23, 1997.
21. Gwennap, L., MIPS R10000 uses decoupled architecture, *Microprocessor Report,* 8, 18, 14, 1994.
22. Gwennap, L., MIPS R12000 to hit 300 MHz, *Microprocessor Report,* 11, 13, 1, 1997.
23. Patkar, N. et al., Microarchitecture of HaL's CPU, in *Proceedings of COMPCON*, 1995, 259.
24. Gwennap, L., Intel's P6 uses decoupled superscalar design, *Microprocessor Report,* 9, 2, 9, 1995.
25. Gwennap, L., Klamath extends P6 family, *Microprocessor Report,* 11, 2, 1, 1997.
26. Pentium III Processor, Product Overview, Intel Corp., 1999.
27. Hinton, G. et al., The microarchitecture of the Pentium IV processor, *Intel Technology Journal,* 1.Q., 1, 2001.
28. Liptay, J.S., Design of the IBM Enterprise Sytem/9000 high-end processor, *IBM Journal of Research and Development,* 36, 4, 713, 1992.
29. Burkhardt, B., Delivering next-generation performance on today's installed computer base, in *Proceedings of COMPCON*, 1994, 11.
30. "Cyrix 686MX," Cyrix Corporation, Order No. 94329-00, 1997.
31. Gwennap, L., NexGen enters market with 66-MHz Nx586, *Microprocessor Report,* 8, 4, 12, 1994.
32. Slater, M., AMD's K5 designed to outrun Pentium, *Microprocessor Report,* 8, 14, 1, 1994.
33. Shriver, B. and Smith, B., *The Anatomy of a High-Performance Microprocessor,* IEEE Computer Society Press, Los Alamitos, 1998.
34. Golden, M. et al., A seventh-generation x86 microprocessor, *IEEE Journal of Solid-State Circuits*, 34, 11, 1999, 1466.
35. Sima, D., The design space of register renaming techniques, *IEEE Micro,* 20, Sept./Oct., 70, 2000.
36. Sima, D., Superscalar instruction issue, *IEEE Micro,* 17, Sept./Oct., 29, 1997.
37. Popescu, V., Schultz, M., Spracklen, J., Gibson, G., Lightner, B., and Isaman, D., The Metaflow architecture, *IEEE Micro,* 11, June, 10, 1991.
38. Gwennap, L., Nx686 goes toe-to-toe with Pentium Pro, *Microprocessor Report,* 9, 14, 1, 1995.
39. Monreal, T. et al., Delaying physical register allocation through virtual-physical registers, in *Proceedings of MICRO-32*, 1999, 186.
40. Wallace, S. and Bagheryadeh, N., A scalable register file architecture for dynamically scheduled processors, in *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, 1996, 179.
41. González, A. et al., Virtual registers, in *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, IEEE CS Press, 1997, 364.
42. González, A., González, J., and Valero, M., Virtual-physical register, in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, IEEE CS Press, 1998, 175.
43. Jourdan, S. et al., A novel renaming scheme to exploit value temporal locality through physical register reuse and unification, in *Proceedings of MICRO-31*, IEEE CS Press, 1998, 216.
44. Asato, C. et al., A 14-port 3.8 ns 116 word 64b read-renaming register file, in *Proceedings of ISSC*, 1995, 104.
45. Crawford, H., The I486 CPU: executing instructions in one clock cycle, *IEEE Micro,* 10, Feb., 27, 1990.

46. Smith, J.E. and Pleszkon, A., Implementation of precise interrupts in pipelined processors, in *Proceedings of ISCA*, IEEE CS Press, 1985, 36.
47. Johnson, M., *Superscalar Microprocessor Design,* Prentice-Hall, Englewood Cliffs, NJ, 1991.
48. Sohi, G.S. and Vajapayem, S., Instruction dispatch logic for high performance, interruptable pipelined processors, in *Proceedings of the 14th ASCA*, 1987, 27.
49. Sima, D., The design space of shelving, *Journal of Systems Architecture,* 45, 863, 1999.
50. Thornton, J.E., *Design of a Computer: The CDC 6600,* Scott Foresman, Glenview, IL, 1970.
51. Tendler, J.M., POWER4 system microarchitecture, *IBM. Journal of Research and Development* 46, 1, 5, 2002.
52. Sinhaoy, B., POWER5 system microarchitecture, *IBM. Journal of Research and Development* 49, 4/5, 505, 2005.
53. de Vries, H., Understanding the detailed Architecture of AMD's 64 bit Core, www.chip-architect.com, 2003.
54. Torres, G., Inside Pentium M Architecture, www.hardwaresecrets.com/article/270, 2006.
55. Torres, G., Inside Intel Core Microarchitecture, www.hardwaresecrets.com/article/313/1, 2006.
56. Wechsler, O., Inside Intel Core Microarchitecture, White Paper Intel Corp, 2006.
57. de Vries, H., Looking at Intel's Prescott die, part II, www.chip-architect.com, 2003.

## 2.3 Predicting Branches in Computer Programs

*Kevin Skadron and David Tarjan*

### 2.3.1 What Is Branch Prediction and Why Is It Needed?

#### 2.3.1.1 What Is Branch Prediction?

Branch instructions permit a program to control the flow of instruction execution within a program. Examples of high-level program constructs that translate into branches are "if-then" statements and "for" loops. They test some condition, and depending on the outcome, execution proceeds down one of two possible paths. In almost all instruction sets, branch instructions have exactly two possible outcomes: not-taken, the sequential or fall-through case, in which the condition is false and the program continues executing the instructions that immediately follow the branch; and taken, the nonsequential case, in which the condition is true and execution jumps to a target specified in the branch instruction. In the case of an "if" statement, the two outcomes are the "then" clause and the fall-through case, which may correspond to an "else" clause. In the case of a "for" loop, the two outcomes are an iteration of the loop body or the fall-through case, which corresponds to exiting the loop. For example, a typical loop structure in assembly code might look like this (bnez means branch if the condition is not equal to zero):

```
L: (loop body)
   ...
   sub     r1, #1, r1      ; rl is the loop counter
   bnez    r1, L           ; if the loop count is not yet zero, branch back
                           ; to the top of the loop (label "L") and iterate
   (fall-through code)     ; this code gets executed after the loop exits
```

Note that in all the assembly language examples in this chapter, destination registers are listed last.

Branches create a problem because the identity of the proper path can only be known after the branch has finished testing its condition, a process that takes time. Because of the pipelined nature of almost all modern processors, this resolution latency necessitates branch prediction. Figure 2.20 shows the flow of

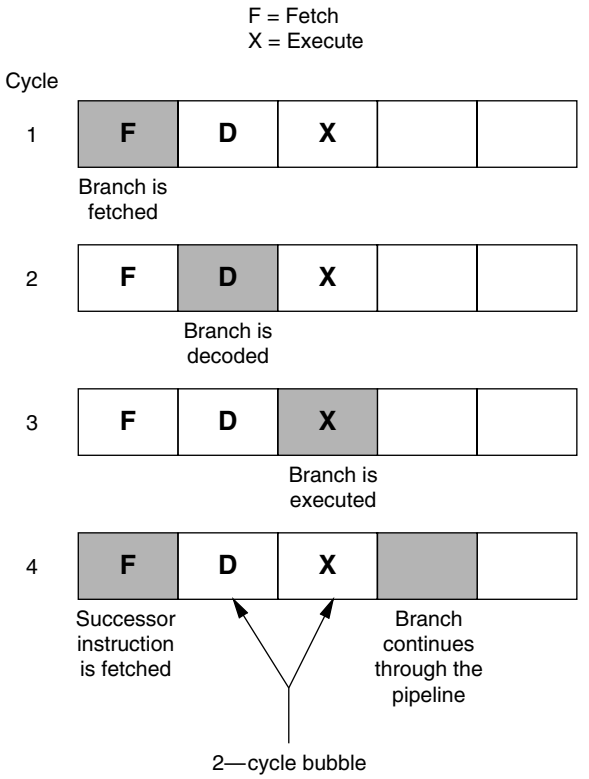



**FIGURE 2.20** A branch flowing through a generic pipeline with no prediction. The branch, the first gray box, flows from left to right. After being fetched, one or more cycles elapse (one cycle in this diagram) while the instruction is decoded and perhaps other manipulation takes place. Once the branch finally completes executing (i.e., testing its condition), the next instruction (the next gray box) can be identified and fetched. This figure shows that the resolution time introduces a delay during which the pipeline stalls. The corresponding bubble here is two cycles long. (From Skadron, K., Characterizing and removing branch mispredictions. Ph.D. Thesis, Princeton University, Princeton, June 1999. With permission.)

a branch through a generic pipeline. Resolving the branch requires waiting until it proceeds through several stages and finally executes. If the fetching of subsequent instructions must wait until the proper path is known with confidence, stall time or a bubble results [1].

If branch outcomes are instead predicted and subsequent instructions are speculatively fetched and executed, this bubble is eliminated whenever the prediction is correct. This is shown in Fig. 2.21. If the prediction is incorrect, these speculative instructions must be squashed—removed from the pipeline—and no time has been wasted compared to the alternative of no prediction. Squashing can be accomplished simply by preventing the misspeculated instructions from modifying any processor state. These squashed instructions, however, represent an opportunity cost: had the branch been correctly predicted, those instructions would have been correct, and would have performed useful work. This wasted time is called the misprediction penalty and is equal to the branch resolution time.

Other control-flow instructions exist that transfer execution to some other program location but are not conditional and do not branch. These jump instructions either jump to the target specified in the instruction (direct jumps), or jump to a target that has been computed and whose address is found in a register (indirect jumps). A procedure call is an example of the former, and a procedure return is an example of the latter. Jumps are also frequently used to jump around the else clause of an if-then-else

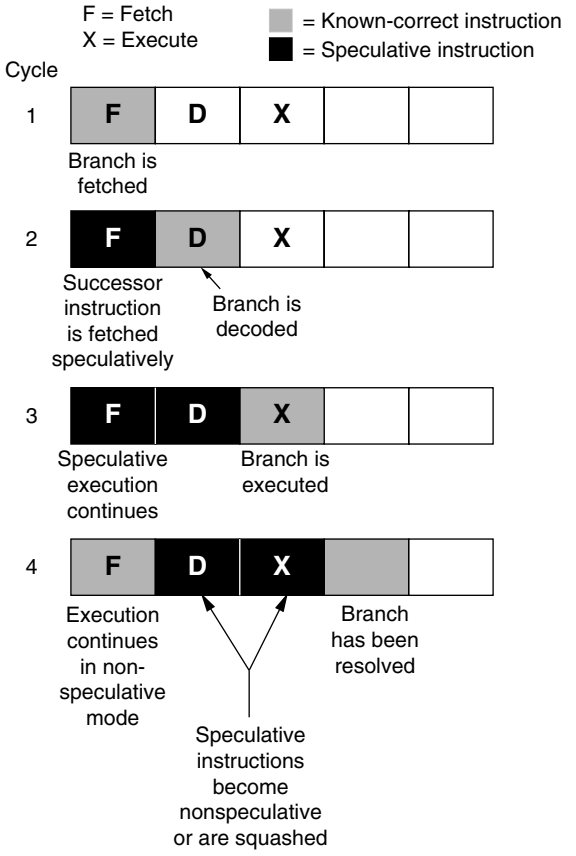



**FIGURE 2.21** Pipeline behavior with branch prediction. In this diagram, the branch's outcome is predicted. Immediately in the next cycle, subsequent instructions are fetched and executed speculatively (black boxes). If the prediction is correct, the speculative instructions do useful work and the bubble has been eliminated. If the prediction is incorrect, the speculative instructions are squashed. (From Skadron, K., Characterizing and removing branch mispredictions. Ph.D. Thesis, Princeton University, Princeton, June 1999. With permission.)

construct. For example, the following C code on the left would be translated into the pseudo-assembly code on the right (bz: branch if zero):

```
                                                    ; r1 holds cond
if (cond)                       bz      r1, L1      ; if cond == 0, do else clause
        procedure1();           call    procedure 1 ; cond != 0
else    /* cond == 0 */         jump    L2          ; skip else clause
        procedure2();       L1: call    procedure2  ; cond == 0
x = x +1;                   L2: add     r20, #1, r20; r20 holds x
```

As with branches, some time is required to determine jump targets. Direct jumps can be resolved early with proper hardware in the fetch stage to extract the jump target from the instruction, or the targets can be predicted (e.g., using a branch target address cache—see Section 2.3.3.2). Indirect jumps generally cannot be resolved early, and instead must proceed through the pipeline in order to read their target from the register file, just like any other instruction. Fortunately, their targets can also be predicted. Prediction of indirect jumps is an active research topic [2–6], but is beyond the scope of this treatment

of branch prediction. Return instructions are a special case of indirect jumps, and are easily predicted using a simple structure known as a return-address stack [7,8].

The term "branch" is often used to refer to any type of control-flow instruction, giving us not only conditional branches but also direct and indirect (unconditional) branches instead of jumps. But the term branch is best reserved for conditional branches, because control truly branches at such instructions, and unconditional control-flow instructions are best called jumps.

#### 2.3.1.2 Why Is It Needed?

Branch prediction is necessary because branches are frequent, 15%–25% of instructions in a typical program. Without prediction, the pipeline would stall for each branch's resolution (refer again to Fig. 2.20) and impose a substantial performance penalty. Even if the processor could issue only one instruction per cycle, and branch resolution stalled the pipeline for only one cycle, this would impose a performance penalty of 15%–25%. But today's pipelines are substantially longer (to permit faster clock speeds) and wider (to exploit instruction-level parallelism or ILP), making the penalties much more severe in terms of wasted instruction-issue opportunities. Every additional stage in the pipeline between fetch and execute adds a cycle to the branch resolution delay. In addition, in today's wide-issue superscalar pipelines, the penalty is equal to the resolution delay times the issue width. The minimum resolution delay in the Compaq* Alpha 21264—a four-wide superscalar processor—is 7 cycles [9], and the minimum resolution delay in the Intel Pentium† Pro—a three-wide superscalar organization—and its successors is 11 cycles [10]. The corresponding penalties are 28 and 33 instruction-issue slots. Of course, programs often do not exhibit enough ILP to use the full issue width all the time, so the actual penalties are not quite so severe. On the other hand, the resolution delays just specified are only the minimum delays. The out-of-order nature of many high-performance processors' execution engines means that instructions may spend an arbitrary amount of time in decoupling buffers, and this makes the pipeline seem longer and exacerbates the branch resolution delays. A correct branch prediction eliminates these stall cycles. A further problem is that mispredictions limit the processor's ability to build up a large window of instructions over which to expose ILP.

With the misprediction penalty so high in terms of wasted instruction-issue opportunities, not only is branch prediction necessary but also the highest possible prediction accuracy is necessary in order to minimize stall cycles and maximize the processor's ability to exploit ILP.

### 2.3.2 Software Techniques

Branches can be predicted or otherwise managed by both software and hardware techniques. This section focuses on software techniques, and Section 2.3.3 focuses on hardware techniques.

#### 2.3.2.1 Branch Delay Slots

One early software technique that was able to eliminate the need for prediction in early processors is the branch delay slot. Instead of predicting the branch's outcome, the instruction-set architecture can be defined so that some number of instructions following a branch execute regardless of the branch's outcome. These instruction positions are called delay slots and must be filled with instructions that are safe to execute regardless of the outcome of the branch, or with nops (but nops do no useful work). Instructions to fill the delay slot might come from positions that preceded the branch in the original code schedule but can safely be reordered, for example. Consider the sequence of code:

1. add r1, r2, r3
2. add r4, r5, r6
3. bnez r6
4. (delay slot)

*Compaq Computer Corp., Houston, Texas.

†Intel Corp., Santa Clara, California.

Instruction 1 can safely be moved into the delay slot, because doing so violates no data dependencies. Instruction 2, of course, cannot be moved into the delay slot, because it computes the value of r6 that the branch then examines. More aggressive techniques can analyze instructions from after the branch, identify a safe instruction, and hoist it into the delay slot. A more thorough treatment of branch delay slots and associated techniques can be found in Ref. [11].

Unfortunately, delay slots have drawbacks. Even the most aggressive techniques still leave some delay slots unfilled, wasting instruction-issue opportunities. Delay slots also have the problem that they expose processor implementation details that might change. Current instruction sets that use delay slots were defined when processors issued instructions in order, one at a time, and pipelines were short. The branch resolution delay was hence just one cycle and the corresponding penalty was only one instruction issue slot, so these instruction sets defined branches to have a single delay slot. Examples include the MIPS* [12] and SPARC† [13] instruction sets. Yet, later implementations made the pipeline longer and issued multiple instructions per cycle. This meant that the resolution delay corresponded to many issue slots, even though the number of delay slots was still fixed by the instruction set at one instruction. In addition, with multiple issue, a bundle of instructions being considered for issue in any particular cycle might consist of several instructions following a branch. Exactly one of these—the delay slot—must be issued unconditionally, while the others are control-dependent on the branch and their execution depends on the branch outcome. For these reasons, later instruction sets like Alpha AXP [14] do not include delay slots.

#### 2.3.2.2 Profiling and Compiler Annotation

An alternative software technique is to profile the program's behavior by gathering data about how individual branches behave. This involves gathering data while the program is running about its branch's behavior. This data can then be fed to a second compilation pass, which annotates the branches to indicate the predominant direction. The hardware then predicts each branch according to the annotation. So for example, a branch that is taken 80% of the time and not taken 20% of the time would be annotated predict-taken. More sophisticated profiling and compiler analysis can even make multiple copies of segments of code so that the branches therein have more consistent behavior, or uncover branches whose behavior is correlated and thus capture some of the same behavior as global-history prediction. This is described by Young and Smith [15].

#### 2.3.2.3 Predication

A third technique is predication or if-conversion, in which the branch is removed and instructions from both the taken and not-taken paths can be executed simultaneously. This eliminates the need to predict the branch, and converts code that was control dependent into code that is data dependent on the branch condition. This defers the dependence to the execution core and permits fetching to continue without risk of rollback due to mispredictions. If done judiciously and execution from the two paths is properly balanced, if-conversion can be done without any performance penalties. Correctness is ensured by modifying the instructions that were once controlled or guarded by the if-converted branch so that they can only commit if the branch condition would have permitted it.

If-conversion is accomplished in one of two ways. In full predication, each individual instruction is guarded by a condition. This predicate value is specified as a third operand register, usually from a dedicated register file. Clearly, this requires instruction-set support in every instruction. In partial predication, on the other hand, there is no support for guarding predicates. Instead, predication is accomplished using conditional move instructions (CMOVs), which can simply be added to retrofit to existing instruction sets. One branch path is executed unconditionally. The results for the other path are

*MIPS Technologies, Mountainview, California.

†SPARC International, Inc., Santa Clara, California.

computed into temporary registers and then moved into their final destination with CMOVs. The CMOV only completes if the specified condition (the branch condition) holds true. The following code sequence gives an example:

```
Original code          Full predication          Partial predication
  if (cond)            pdef   cond, p            add    a, b, x
    x = a + b;         add    a, b, x(p)        cmov   a, x (cond)
  else                 mov    a, x(!p)          mul    x, x, y
    x = a;             mul    x, x, y
  y = x * x;
```

The pdef instruction defines a predicate; the condition is evaluated and the result placed in p. In all cases, $y = x * x$ gets the correct value of $x$ because $y$ is data dependent on $x$ and can only use $x$ once its final value is assigned. The final value of $x$, in turn, is either control dependent (original code) or data dependent (predicated code) on cond. Although in this example, the partially-predicated sequence is shorter, partial predication has two drawbacks. It requires a CMOV instruction for each destination register on the path being predicated, and each destination register requires a temporary register [16].

Research by Mahlke et al. [17] has shown that predication substantially reduces both the number of branches executed as well as the branch misprediction rate. Nevertheless, resource constraints mean that not all branches can be predicated, and so predication still requires the presence of branch prediction. This brings us to the hardware techniques, which can be used alone or in conjunction with the software techniques just described. Note that predication can actually hurt the predictability of the remaining branches. As seen in the next section, many branch prediction algorithms depend on the ability to track the history of earlier branch outcomes. Predication removes branches and hence removes this history from view. Simon et al. [18] explore some ways to rectify this problem.

The literature on predication techniques is a rich body in its own right, and interested readers are encouraged to consult both the architecture and compiler literature.

## 2.3.3 Hardware Techniques

### 2.3.3.1 Static Techniques

The simplest hardware technique is to simply stall after every branch until its outcome is known. As described above, the consequent delays lead to untenable performance penalties. A better yet still simple technique is to statically predict all branches to be either taken or not taken. A static not-taken policy is the easier of the two, because it corresponds to sequential execution. This eliminates the need for the fetch engine to identify the instructions that are branches or to compute branch targets. Unfortunately, in most programs more than half of branches are taken [19], making the performance of static-not-taken usually quite poor. On the other hand, a static taken policy either requires the fetch engine to identify the instructions that are branches and immediately identify their taken targets, or requires some delay while instructions are decoded and the target is computed.

A third policy takes advantage of the fact that backward conditional branches almost always correspond to loops, which tend to iterate multiple times, so these branches are likely to be taken. Non-backward branches, on the other hand, are less biased. Patterson and Hennessy [11] found that 85% of backward branches are taken whereas only 60% of forward branches are taken. This suggests a static policy of backwards-taken, forwards-not-taken, or BTFNT. The problem of computing branch targets remains.

These policies were described by Smith [19] along with the core, bimodal dynamic prediction technique described in Section 2.3.3.4. Another seminal paper from this era is the exploration of branch predictor and branch target address cache (BTAC) design choices by Lee and Smith [20]. Both papers also survey the earliest literature on branch handling.

#### 2.3.3.2 Branch Target Address Caches

Not only static techniques, but in fact all branch-prediction techniques have the problem that on a predicted-taken branch, the branch's target must be computed. This requires extracting the offset field from the branch instruction and adding it to the PC; tasks which typically cannot be performed until the instruction-decode stage. If this is the case, some stall cycles result, called a branch-taken bubble. A second type of predictor—a branch target predictor—can eliminate this problem. In its simplest form, this is simply a small on-chip memory in the fetch stage that serves as a table of recently seen branches, BTAC [21,22]. (The BTAC is also often referred to as a branch target buffer [BTB], but this latter term is too heavily overloaded.) The BTAC is indexed with the branch's address (in other words, the PC—program counter—used to fetch the branch). It may be direct-mapped or associative, and tagged or not tagged. Omitting tags reduces cost, but then a BTAC miss cannot be identified, the predicted-taken branch will use the wrong target, and this will not be discovered until the branch resolves. For this reason, BTACs are best tagged.

The dynamic hardware schemes described later in this section maintain tables in which they track state about conditional branch directions. These direction-prediction tables are often indexed using the branch address. Because the BTAC table is also indexed by branch address, it may be convenient with these dynamic schemes to store the direction-prediction information in the BTAC along with each branch's target. Apart from the convenience of integrating these different sources of information into one table, this confers the advantage that if the BTAC is tagged, any branch prediction state stored in the BTAC is also tagged. Although some processors use this organization, Calder and Grunwald [23] point out that many branches are not taken and hence do not require the BTAC to store a target. Decoupling the direction-prediction state from the target-prediction state therefore permits a smaller BTAC. It also improves flexibility, as some predictors, such as global-history predictors (see Section 2.3.3.5) do not keep a one-to-one mapping between branch addresses and direction-prediction entries.

Instead of a BTAC, the processor might employ a branch target instruction cache, which stores some actual instructions from the branch target rather than merely the target address. This replicates quite a bit of state from the instruction cache, so this organization is rarely seen, although it does appear in the Motorola* PowerPC† G4 [24], for example.

The BTAC can also be integrated with the instruction cache. Each cache line can simply store the target address of one or more of its branches in case that branch is predicted taken. Alternatively, the I-cache can implement a next-line predictor [25]. Each cache line now stores the index of the next cache line to be fetched (and also the set if the cache is associative) [26]. If no branches are taken in the current line, the next-line address will be the next sequential address. If there is a taken branch, the next-line address will be the appropriate target address. As branches change their taken/not-taken behavior, this next-line address is updated accordingly. The next-line predictor is, therefore, a combination of the functionality of a BTB and a bimodal predictor (see Section 2.3.3.4). If a more sophisticated direction predictor is present, it overrides the next-line predictor. One motivation for using such an organization is to permit a larger, slower, but more accurate direction predictor that may not be able to be accessed in a single cycle. The Alpha 21264 takes such an approach [27], using as its slower but more accurate direction predictor the hybrid predictor described in Section 2.3.3.6.

#### 2.3.3.3 Pipeline Issues

In the most efficient organization, both the BTAC and the branch direction predictor are consulted during the fetch stage as shown in Fig. 2.22. In this way, the PC can be updated immediately and the processor can fetch from the appropriate location (taken or not-taken) in the next cycle. This avoids introducing pipeline bubbles unless there is a BTAC miss or a branch misprediction.

*Motorola, Inc., Schaumburg, Illinois.

†International Business Machines Corp., Armonk, New York.

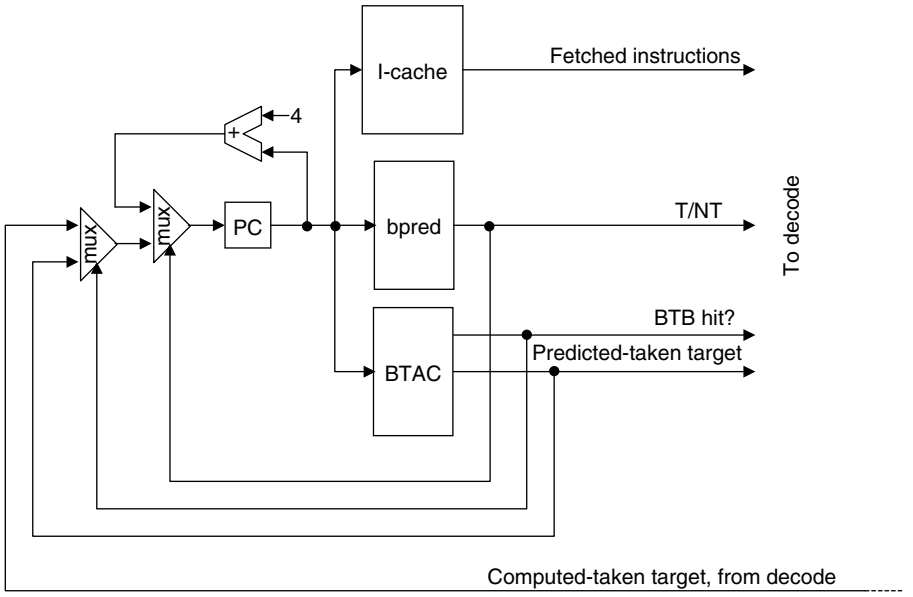



**FIGURE 2.22** The placement of the branch prediction components in the pipeline.

Unfortunately, some problems occur with probing the branch-prediction hardware in the fetch stage. One concern is the branch-predictor and BTAC lookup times. These tables must be fast enough, and hence small enough, to permit the lookup to complete and the PC to be updated within a single cycle. Otherwise, the fetch stage falls behind. Current processors use predictors as big as 32 Kbits, but Jiménez et al. [28] argue that the feasible predictor size for single-cycle access will shrink in the coming years. The reason for this is that even though the feature size on a processor die continues to shrink with Moore's law [29], electrical RC delays are not shrinking accordingly, and hence wire delays are not shrinking as fast as logic delays. As feature size shrinks, large structures therefore seem to be getting relatively slower.

Another problem is that in a typical organization, the fetch stage cannot determine whether the instructions being fetched from the instruction cache contain any branches; that information must wait until the instructions are decoded. Several solutions are available. The first solution is for the instructions to be pre-decoded before they are installed into the instruction cache to indicate the instructions that are branches. The predictor structures can then be indexed using the actual addresses of the branches. Note that this means either that the predictor must be multiported to cope with fetch blocks that contain more than one branch, or the predictor can only predict one branch at a time. This is not necessarily a major restriction, since if the predicted result is not-taken, the remaining instructions in the fetch block after the branch are still valid and can still be passed on to decode. The second solution is for the branch predictor to just predict fetch-block successors instead of specific branches. In this case, the predictor simply predicts whether the next fetch block will be sequential (not-taken) or nonsequential (taken, in which case the target supplied by the BTAC is used). This is slightly better than the first choice, because it eliminates the need for pre-decode bits and can fetch past more than one not-taken branch in a fetch block. It does require the decode stage to identify how each branch in a fetch block was implicitly predicted. The third solution is for the BTAC and branch predictor to be indexed with the address of every instruction in the fetch block. Hits in the BTAC indicate the instructions that are branches, and only the corresponding direction predictions are then used. The problem with this approach is that it requires as many ports into the BTAC and branch-prediction structures as there are instructions in the fetch block. These are the basic choices, although many variations and improvements have been proposed, e.g., [26,30–32].

### 2.3.3.4 Bimodal Prediction

The simplest dynamic technique, introduced by Smith [19], is to maintain a small, on-chip memory with a table of saturating counters that is indexed by branch address. The saturating counters—typically two bits each—simply remember the predominant direction of previous outcomes for that branch. A schematic for a bimodal predictor appears in Fig. 2.23. As mentioned, the table usually called the pattern history table (PHT), although logically a distinct entity might actually be implemented as a unified structure with the BTAC. This prediction scheme goes by different names, often simply two-bit prediction, but recent literature has often referred to it as bimodal prediction to distinguish it from other more sophisticated schemes that also use two-bit saturating counters.

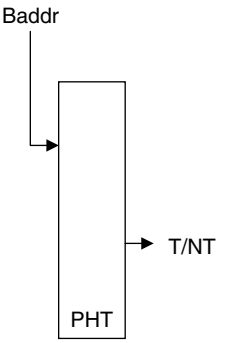



**FIGURE 2.23** A schematic for a bimodal predictor. Baddr is the branch address or PC, which is used to index the PHT (pattern history table), select the corresponding two-bit counter, and make a prediction of taken or not-taken.

Each time a branch resolves, its corresponding counter is incremented if the branch was taken, and decremented if not. Incrementing or decrementing has no effect if the counter is already at its maximum or minimum value, hence the term saturating counter and the name bimodal. In the simplest case of a one-bit counter, the only possibilities are values of 0 and 1 and the predictor simply remembers the last outcome for each branch. In the case of two-bit counters, values of 00 and 01 correspond to strongly not-taken and weakly not-taken, and values of 10 and 11 corresponding to weakly taken and strongly taken. Two-bit counters give better performance because they exhibit some hysteresis that makes them less sensitive to infrequent occurrences of outcomes in the nondominant direction. A state-transition diagram for the most common two-bit counter configuration appears in Fig. 2.24.

Other configurations [20,33] are possible; however, for example, regardless of its current state, the counter might reset to 00 on a not-taken branch.

As an example of how two-bit counters improve over one-bit counters, recall that a loop branch will normally be taken. When the loop exits, a one-bit counter will only remember that most recent direction (not taken), even though the predominant direction is taken. When this same loop is encountered again, and the loop branch will once again be taken until the loop exits, the first prediction with a one-bit counter will be not-taken. A two-bit counter, on the other hand, only changes its state from 11 to 10 upon loop exit, and still predicts taken when it returns to the loop, thus eliminating a misprediction compared to the one-bit counter.

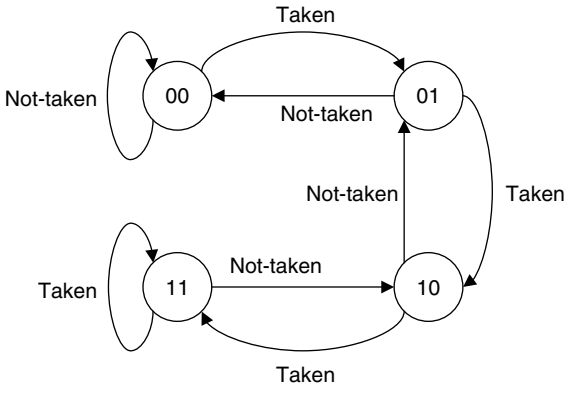



**FIGURE 2.24** The state-transition diagram for a saturating two-bit counter.

Wider counters have been considered [19] but confer little benefit and take longer to adjust to a change in a branch's behavior.

The size of the PHT is of course not infinite, so the ideal of one entry per branch may not be realized. The table is indexed by the branch address modulo the table size, so some branches may collide. If these branches are biased in the same direction this is harmless, but if not, they will interfere with each others' attempts to update the counter, and these destructive PHT conflicts will lead to mispredictions. Sources of mispredictions are discussed in Section 2.3.4.

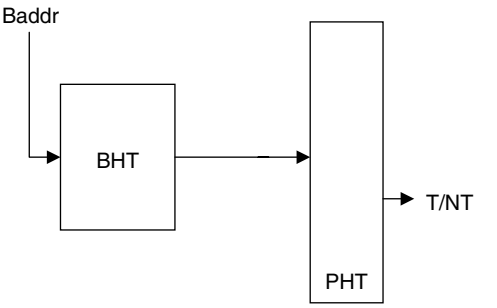



**FIGURE 2.25** A schematic for a PAs local-history predictor. The branch address is used to index the table of per-branch histories (the BHT), select the appropriate history, and then this history is used to index the PHT.

#### 2.3.3.5 Two-Level Prediction

Bimodal prediction can be improved in two ways, both of which explicitly track earlier branch outcomes and were introduced by Yeh and Patt. Local-history prediction [34] maintains a table of per-branch histories. Instead of tracking each branch's predominant direction, this branch history table (BHT) tracks explicit history in order to detect patterns. For example, a local history can detect patterns like TNTN . . . that confound simple saturating counters. The predictor still keeps a PHT of two-bit counters, but these are now indexed using the local history pattern, and the counters now learn outcomes for each history pattern. A schematic of a local history predictor appears in Fig. 2.25. One apparent problem with local-history prediction is that it would seem to require two serial lookups: first the BHT to obtain the history pattern, then the PHT to obtain the actual prediction.

This problem is solved by caching the most recent PHT value for a given BHT entry as an extra field in the BHT. The next time that BHT entry is indexed, it provides both the current history and the cached prediction. Fetching proceeds with that cached prediction while the PHT is probed with the history pattern. The PHT result overrides the cached result, so if the PHT disagrees with the cached prediction, the pipeline is flushed from the point of the mispredicted branch.

Global-history prediction [35] on the other hand, keeps a single history register—the global branch history register or GBHR—into which all branch outcomes are shifted, as seen in Fig. 2.26. It might seem that intermingling outcomes from different branches simply produces noise, but instead global-history prediction is extremely effective. The reason is that global history exposes correlation among branches (and hence these predictors are also called correlating predictors).

Consider the following sequence of code:

```
B1:     if (x)
        ...
B2:     if (y)
        ...
        z = x && y;
B3:     if (z)
        ...
```

Even if B1 and B2 are entirely unpredictable because *x* and *y* have very random behavior, B3 can be predicted with 100% accuracy if the outcomes of B1 and B2 are known, because the outcome of B3 is entirely correlated with the outcomes of B1 and B2. Global history is an admittedly crude way to expose this sort of correlation, because the

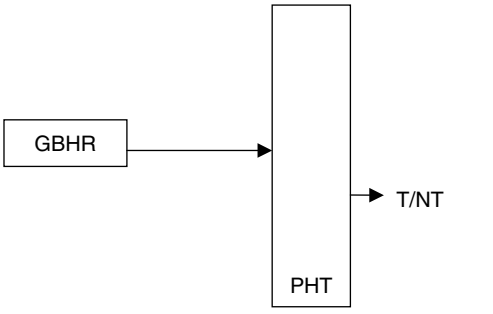



**FIGURE 2.26** A schematic for a GAs global-history predictor. The global history of recent branch outcomes, contained in the global branch history register (GBHR) is used to index the PHT.

GBHR also contains outcomes from other branches that provide no useful information. Yet, as Section 2.3.5 shows, global history is quite effective, and Evers et al. [36] have shown that many programs contain substantial degrees of correlated branch behavior. Unfortunately, no one has come up with a practical hardware technique for exposing correlation while avoiding the noise that unrelated branches introduce into the GBHR.

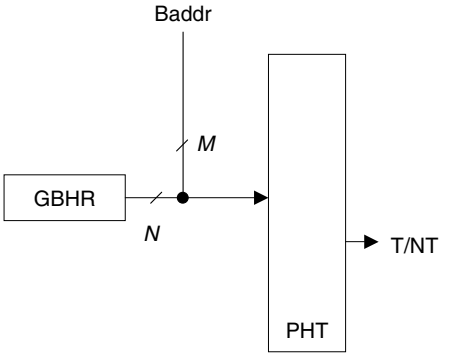



**FIGURE 2.27** A schematic for a GAs global-history predictor. *N* global history bits are concatenated with *M* bits from the branch address to form the PHT index.

Both the local-history and global-history predictors described above have the problem that different branches may see the same history. All branches that see the same history will map to the same PHT entry. Especially with global prediction, equivalent history does not always mean the branches will behave the same way. To reduce the consequent destructive PHT conflicts, Pan et al. [37] point out that bits from the branch address can be combined with the history bits in order to provide some degree of anti-aliasing—see Fig. 2.27 for example. The simplest technique is to concatenate the two bit sources. For *N* bits of history and *M* bits of branch address, this creates a configuration where each *M*-bit address pattern has its own $2^N$-entry PHT.

For a fixed table size and hence a fixed number of bits in the index, this necessitates a reduction in the number of history bits, so a balance must be found between the added prediction capability provided by history bits and the anti-aliasing capability provided by address bits. This balance is sensitive to the table size. In a study of the SPECint95 benchmarks [38], Skadron et al. [39] found that as a general rule of thumb, both global- and local-history predictors should use at least 6–7 bits of branch address, regardless of predictor size. Predictors with more aggressive anti-aliasing techniques, e.g., the bi-mode predictor of Lee et al. [40], will need fewer address bits.

To classify the different possible two-level predictor organizations, Yeh and Patt [35,41] developed a naming scheme that uses three letters to characterize the different organizational choices. The first letter, G, P, or S, indicates the type of history, global, per branch (i.e., local), or per branch set. The last choice refers to a predictor that explicitly allocates groups of branches to particular BHT entries, and is only feasible with extensive profiling or compiler support and hence has received little study. Skadron et al. [39] added a fourth type, M, to this naming scheme to describe predictors that track a combination of global and local history. The second letter, A or S, indicates whether the PHT is adaptive, using a finite state machine based on saturating counters, or fixed, using statically assigned directions (a profiling pass might determine the best PHT value for each entry); almost all predictors proposed or under study, however, are A—adaptive. The third letter, g, s, or p, indicates the PHT organization. The PHT might be indexed purely by history (g); or indexed using some concatenated branch address bits, making it set-associative (s); or the predictor might have a separate PHT for each branch (p, for per-branch). This last choice eliminates aliasing among branches but is prohibitively large for all but small history sizes, and is therefore mainly of theoretical interest. A pure global-history predictor like that in Fig. 2.26 is, therefore, a GAg predictor and a pure local-history predictor like that in Fig. 2.25 is a PAg predictor. If either of these concatenate some address bits into the index, like the global-history predictor in Fig. 2.27, they become GAs or PAs predictors. Note that the GAs predictor has also sometimes been referred to as *gselect* [42]. Finally, a predictor that uses both global and per-branch history, such as the bi-mode predictor, would be an MAg or MAs predictor [39]. As for specifying the specific configuration of a predictor—how many bits, how many entries, etc.—so many notations are involved that it is better to just be explicit.

An alternative anti-aliasing approach is to XOR the history string and address string together; this approach, introduced by McFarling [42], is called *gshare*. This avoids the need to use a shorter history

string—both strings can be as long as the index. Recent data by Sechrest et al. [43], however, suggest that *gshare* confers little benefit over GAs.

Two-level prediction can seem like magic, especially global-history prediction. But it operates on the same principle as compression; a predictable sequence is also compressible. Indeed, two-level prediction is a simplified version of a Markov model, the same principle that underlies the prediction by partial matching (PPM) compression scheme [44].

#### 2.3.3.6 Hybrid Prediction

Because some branches do benefit from global history and others do not, McFarling [42] proposed hybrid branch prediction. Several different organizations have been proposed [45–47], but the common idea is to operate two different predictors in parallel, and for each branch select the predictor's output that is to be actually used in making the prediction. The selector is itself a predictor and can be any of the structures described above, but the selector tracks predictor successes rather than branch outcomes. For each branch, the selector attempts to learn the predictor component that is more effective. Figure 2.28 shows a high-level schematic of a hybrid predictor's organization. Note that both predictor components and the selector can all be accessed in parallel to minimize lookup time.

The Compaq Alpha 21264 [27] uses a hybrid predictor comprised of 12-bit GAg and 10-bit PAg predictors. The PAg component has a 1 K-entry BHT and, because it uses only history bits in indexing the PHT, the PHT is also 1 K entries. An unusual aspect of the PAg component is that it uses three-bit instead of two-bit saturating counters in order to achieve a stronger bias once the predictor has trained. The selector is also a 12-bit GAg predictor, but its PHT tracks the component that has been more successful for each branch, rather than the direction the branch should take.

Hybrid prediction has also been called tournament, competitive, and combining branch prediction.

#### 2.3.3.7 Loop Prediction

As was pointed out above, static schemes such as BTFNT take advantage of the fact that most backward branches are loop branches, which are predominantly taken. Two-level predictors that use local history can accurately predict loop branches (absent aliasing) as long as the loop iteration count is shorter than the local history length. Similarly, predictors using global history can accurately predict loop branches if

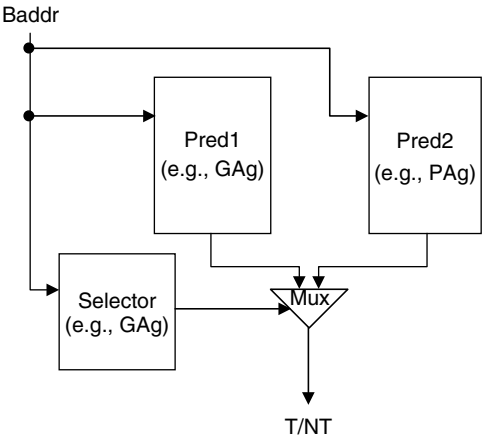



**FIGURE 2.28** A schematic for a hybrid predictor. Pred1 and pred2 are configured as regular, stand-alone branch predictors (they might be a global-history and a local-history component, for example), but both components make a prediction and the selector then chooses one component prediction (via the multiplexor or mux) to use as its final prediction. The selector is a predictor too, but tracks component outcomes rather than branch direction outcomes.

the global history is longer than the loop iteration count, and there are no intervening branches. Another way in which global predictors can predict loops is if they can identify a pattern of branches that is different on the last iteration of the loop.

There are, however, loops that have large iteration counts or have too many intervening branches between iterations of the loop. For these and other loops, a loop predictor can be a useful addition to an existing branch predictor. Loop predictors usually are organized similarly to a tagged BTAC, indexed and tagged by the branch address. Each entry contains a field for the maximum iteration count, a field containing the current committed iteration count, and a confidence field. When a new loop branch is encountered, an entry is allocated, with all fields set to zero. As the usual iteration count is identified, the maximum iteration field is updated. Each time the same iteration count as in the maximum iteration count field is seen, the confidence field is incremented. If the actual count is different from the stored value, the confidence field is reset and the maximum iteration count field is updated. If the confidence field reaches its maximum value, the loop predictor starts making prediction based on whether the current iteration count field is equal to the maximum iteration count field or not.

For tight loops that can have multiple loop iterations flowing through the pipeline at the same time, the committed iteration count will always lag behind the loop and produce many mispredictions. To deal with this problem, a separate speculative current iteration field is often added to each entry. Similar to speculative branch history, the field is updated right after a prediction is made. Predictions are then made based on the state of the speculative current iteration count field. In case of a misprediction, the value of the committed current iteration count field is copied to the speculative field.

A loop predictor similar to the one described has been included in the Intel Pentium M [48].

#### 2.3.3.8 Neural Prediction

A new class of predictors that have recently drawn a lot of attention are the so-called neural predictors. The name derives from the fact that the first neural predictors [49,50] were based of off simple neural networks.

Neural predictors have a core distinguishing feature in that they aggregate the information from different sub-predictors by computing the sum of a small number of weights, one for each sub-predictor. Depending on whether the sum is positive or not, the branch is predicted to be taken or not. For ease of implementation, each weight is usually defined as being a small signed integer; however, other representations are possible. Each sub-predictor determines the weight it contributes to the sum, but a global training mechanism determines the sign and magnitude of each weight. The sub-predictors can be organized similarly to any of the previously described predictors, where counters are replaced by weights, or in other ways.

The training mechanism works as follows. On all mispredicted branches, and all correctly predicted branches where the magnitude of the sum was below a threshold, each weight involved in the prediction is incremented or decremented depending on whether the branch was taken or not.

The most accurate neural predictors [51,52] typically utilize larger hardware budgets more effectively than older, nonneural predictors, primarily through utilizing a small number of adders to calculate the overall sum.

#### 2.3.3.9 Partitioning the Predictor Hardware

Workloads with large instruction working sets suffer the most from destructive aliasing. Since many commercially important workloads, such as web servers, databases, and transaction processing systems, fall into this category, manufacturers have tended to include the largest branch predictors they could afford.

Large branch predictors can however limit the maximum frequency of a processor design [28]. One way to have both large capacities without limiting frequency is to organize branch predictors according to the same principles as multilevel caches. The first level can be small and fast, while covering only a

limited number of static branches (for example those currently in the instruction cache). Additional levels provide large capacity at much higher latencies, without hurting the frequency of a processor.

The division into multiple levels is most natural for bimodal predictors, since each entry (absent aliasing) is associated with one static branch. Entries can be moved between levels in parallel with the movement of instructions from the higher level caches into the instruction cache. This approach was taken in the AMD Opteron [53], where some bimodal information can be stored in the level 2 cache.

Similar divisions into two or more levels for two level predictors using local branch history and loop predictors are also possible. The BTAC can also be organized into multiple levels if needed.

#### 2.3.3.10 Other Issues in Predictor Organization

The preceding sections have described the basic predictor organizations. Because prediction accuracy so strongly underpins processor performance, branch prediction remains an active area of research and a wealth of additional organizations have been proposed, primarily focusing on reducing mispredictions due to destructive aliasing. Interested readers should consult recent proceedings of the symposia and conferences in the list of works cited.

It is worth noting, however, that researchers have also considered how to adapt branch prediction to wider-issue machines. Such a machine must fetch past multiple, possibly taken branches in order to exploit the wider fetch width. Otherwise the processor becomes fetch-bottlenecked and its effective width is restricted by the average basic block size.

Yeh et al. [31] describe a different branch address cache that learns segments of the control-flow graph and, in conjunction with a banked instruction cache, can fetch several blocks from noncontiguous cache lines in a single cycle. Conte et al. [30] describe a collapsing buffer that can also fetch past branches that are taken but whose target is in the same fetch block. Reinman et al. [54] decouple branch prediction from fetch, allowing the branch predictor to run ahead of fetch when possible, and they predict the length of fetch blocks so that not-taken branches do not unnecessarily limit fetching. The most aggressive proposal is the trace cache of Rotenberg et al. [32], which dynamically collapses noncontiguous fetch streams into contiguous traces that are stored in the trace cache, a form of which appears in the Intel Pentium 4 [55] and the HAL Sparc64 V [56]. This collapsing function can improve fetch bandwidth, but Co et al. [57] showed that the main value of trace caches is their ability to perform implicit branch prediction: a trace embodies only one path for each interior branch. On the other hand, trace caches require sophisticated trace predictors. Co et al. also explore the performance and energy-efficiency trade-offs of traditional instruction-cache/branch predictor organizations versus trace caches. Highly accurate branch predictors like neural predictors reduce some of the branch-prediction benefit that trace caches once conferred. A compromise approach, described by Ramirez et al. [58] is to fetch and predict streams instead of traces: instruction sequences that are terminated by a taken branch. This allows a conventional instruction cache to be used but allows a single prediction to guide fetch of longer instruction sequences.

In modern processors, which include simultaneous multithreading (SMT) [59], the choice how to allocate the shared resources in the pipeline between the different threads is critical for overall throughput. Branch prediction is affected if the branch predictor is shared, because threads can interfere with each others' state. Of greatest concern is when threads corrupt the branch history [60] or the return-address stack [61]. The solution is to keep a separate branch history structure and return-address stack for each thread.

### 2.3.4 Sources of Mispredictions

To better understand the behavior of branch predictors, it is helpful to examine some of the reasons a misprediction might occur. These reasons can be broken down into two broad categories: behavioral and structural. Behavioral mispredictions stem from the intrinsic behavior of a branch and are independent of the predictor's organization. Any irregular or random behavior by a branch will inhibit its predictability. Structural mispredictions, on the other hand, stem from properties of the predictor's hardware organization. The major structural sources of mispredictions are described below and come from Ref. [39].

#### 2.3.4.1 Destructive PHT and BHT Conflicts

All predictors that track state can suffer when unrelated branches map to the same predictor entry and interfere with each others' state. In the predictor's PHT or in a hybrid predictor's selector PHT, destructive conflicts arise when branches map to the same two-bit PHT counter and these branches go in opposite directions. In the BHT of a local-history predictor, destructive conflicts arise when branches map to the same history entry and hence the history of one branch displaces that of another branch. Note that constructive conflicts can also occur in all these structures when—for reasons of either luck or correlated behavior—the state of one branch causes a correct prediction by other branches. This means the expected gain from eliminating conflicts would eliminate both destructive and constructive conflicts, but the destructive behavior usually outweighs the constructive behavior.

#### 2.3.4.2 Training Time

Because dynamic predictors work by recognizing patterns of branch behavior, they take time to train. The training time comes from two sources. First, the predictor must see enough branches to observe any patterns that exist. Second, the predictor must reach steady state. Consider the simple pattern TN,TN... in a six-bit history. After this branch has been seen twice, the history will contain xxxxTN, where "x" signifies that these history bits contain a random value. Unless by sheer luck the bits in the "xxxx" portion happen to be TNTN, this pattern is a transient that will not be seen again. This branch must therefore be seen four more times before the history is fully initialized. In addition, for both types of training, a pattern must be seen often enough not only to initialize the branch history but also to put the corresponding counters in the PHT into the proper state. In the TNTN... example and assuming two-bit counters in the PHT, this means that the history TNTNTN must be seen twice to ensure that the two-bit counter has crossed the threshold. (The counter's initial value might have been 00, but the correct prediction is T.) The larger the saturating counters, the longer is this component of the training time.

The predictor must not only train when a program first starts executing but also retrain after every context switch and also when the program's behavior changes, either because it enters a new phase or the nature of its input changes, etc.

#### 2.3.4.3 Wrong Type of History

Mispredictions can also occur because the predictor does not track the most useful type of history—global or local—for the branch in question. This has been called wrong history, even though it does not imply that the actual history bits contain any invalid information. Unfortunately, most programs have some branches that do well with global history and some branches that do well with local history. A predictor that only tracks one or the other type of history therefore penalizes some branches in each program. Evers et al. [36] showed this to be important. Skadron et al. [39] found wrong-history mispredictions are especially severe in global-history predictors, comprising 35%–50% of the total misprediction rate. They advocated alloyed prediction, which combines global and local history in the same structure. Lu et al. [62] further explored alloyed prediction.

#### 2.3.4.4 History Length

Mispredictions might also arise even if the predictor tracks the correct type of history but it uses too short a history. For example, a history length of only two bits may not capture the full behavior of a pattern longer than two bits. Consider the pattern TNNN,TNNN.... A two-bit local history will learn that TN$\in\rightarrow$ N and this is always correct. But the problem arises for the pattern NN. The predictor will first learn NN$\in\rightarrow$ N, but on the fifth occurrence of the branch, this will cause a misprediction. On the other hand, there exist longer patterns for which short history is still sufficient. Consider two bits of history and the pattern TNNTT,TNNTT.... Although the overall pattern is longer than two bits, none of the distinct sub-patterns (TN, NT, and TT) are longer than two bits.

Alternatively, the history can also be too long. The problem here is that the history may contain many bits that are entirely uncorrelated with the behavior of the branch to be predicted. This means that every

time this branch is seen, those bits may have a different value, and the predictor may potentially have to train on all possible combinations of those unrelated bits. This has the effect of smearing a particular branch's predictor state across a large portion of the PHT. In the absence of conflicts, this should, however, only be a problem for global history. This problem might also be called a training-time misprediction and was discussed by Evers et al. [36].

#### 2.3.4.5 Update Timing

Depending on how the predictor is updated, mispredictions can also arise because the predictor contains stale state. If the predictor is not updated until a branch exits the pipeline, information about that branch's behavior does not appear in the predictor while the branch is in flight. Yet, later branches that are fetched and predicted before the first branch retires may depend on that first branch's outcome [63]. Consider again the sequence of correlated branches:

```
B1:     if(x)
            ...
B2:     if(y)
            ...
        z = x && y;
B3:     if(z)
            ...
```

In a global-history predictor, if B1 or B2 has not yet resolved, the predictor will use state and hence possibly incorrect global history when looking up the prediction for B3. A similar problem arises in a local-history predictor for branches with repeating patterns.

The solution is to speculatively update the branch history immediately after the branch has been predicted, using the just-predicted value. If the prediction is correct, all subsequent branches see the correct history. If not, the history must be repaired, or the predictor will accumulate bogus history. Fortunately, because all instructions after a misprediction are squashed and re-fetched, subsequent branches still see the correct history. This speculative-update-with-repair scheme therefore gives the illusion of omniscient history update. These mechanisms were first described by Jourdan et al. [64], who also found that in two-level predictors, it is only early update of the branch history that matters. Barring destructive conflicts, the prediction for a particular PHT index is fairly stable over time, so the two-bit saturating counters can be updated after the branch resolves.

### 2.3.5 Comparison of Hardware Prediction Strategies

Figures 2.29 and 2.30 present the prediction accuracies of conditional-branch directions for static-not-taken, static-taken, BTFNT, bimodal, GAs, PAs, and hybrid predictors for the SPECint95 benchmarks [38] and for two different sizes: a small predictor configuration of 8 Kbits, and a large configuration of 64 Kbits.

The specific configurations are presented in Tables 2.2 and 2.3. Of course, static predictors have no size, so the data for these is simply replicated in both graphs. The configurations for GAs, PAs, and the hybrid predictor are taken from Skadron et al. [39], which explored the different possible combinations of history bits, address bits, and, for the hybrid predictor, different possible sizes of the three structures.

The data was gathered using a modified version of the simple, instruction-level branch-predictor simulator from SimpleScalar version 2.0 [65]. All the benchmarks were compiled using gcc version 2.6.3 for the SimpleScalar research instruction set (PISA), and with optimization set at −O3—funroll-loops (note that −O3 includes inlining).

Simulation captures all user-level behavior, including libraries, but cannot capture any behavior in the kernel due to system calls. Data was gathered using the SPEC reference inputs. Some benchmarks come with multiple inputs, in which case one has been chosen. Go uses a playing level of 50 and a $21 \times 21$

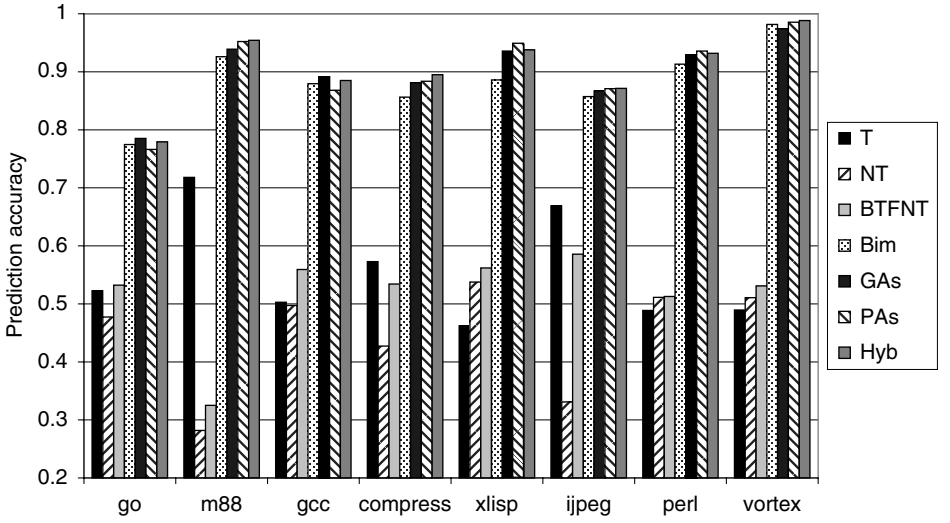



**FIGURE 2.29** Branch prediction accuracies for 8-Kbit predictors for the SPECint95 benchmarks. Bim is the bimodal predictor, and Hyb is the hybrid predictor. Specific configurations appear in Table 2.3.

board with the 9 stone 21 input. M88ksim uses the dhrystone input, gcc the cccp.i input, xlisp the 9-queens problem, ijpeg the vigo.ppm input, and perl the scrabble game.

The reference inputs produce very long simulation times—on the order of days even for simpler instruction-level simulations—so the results here are taken for only a representative, one-billion-instruction segment of each program's execution. A representative segment is reached by fast-forwarding past unrepresentative initial program behavior using a fast-simulation mode that updates the branch predictor state (so that the predictor state is accurate when the full-detail simulation starts) but does not gather branch-prediction statistics [66]. The fast-forward intervals are taken from Ref. [39] and are presented in Table 2.4, along with the observed number of static branch sites and the number of dynamic branches executed for each benchmark.

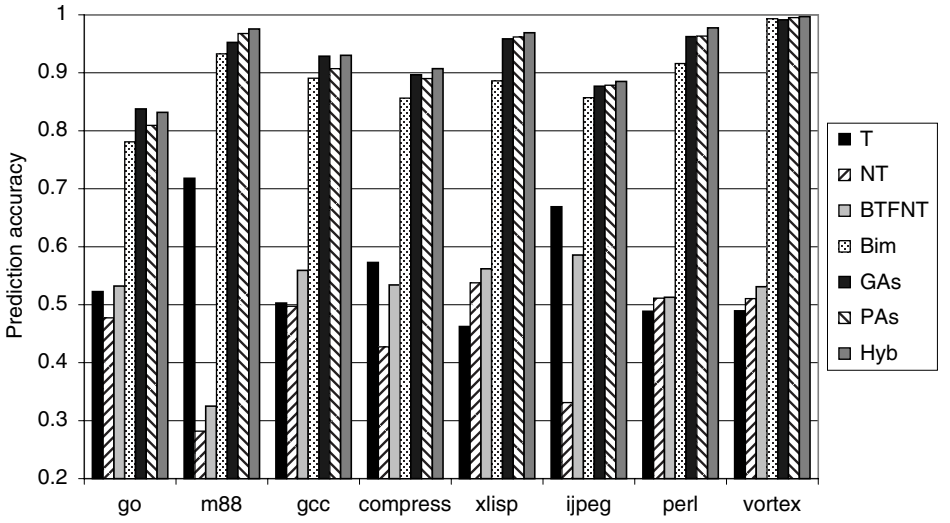



**FIGURE 2.30** Branch prediction accuracies for 64-Kbit predictors for the SPECint95 benchmarks. Specific configurations appear in Table 2.4.

**TABLE 2.2** Predictor Configurations for an 8-Kbit Hardware Budget

| Predictor | Index Bits (h = hist., a = addr.) | BHT Entries | PHT Entries |
|---|---|---|---|
| Static not-taken | — | — | — |
| Static taken | — | — | — |
| BTFNT | — | — | — |
| Bimodal | 12a | — | 4 K |
| Gas | 5h, 7a | — | 4 K |
| Pas | 4h, 7a | 1 K | 2 K |
| Hybrid (selector) | 3h, 7a | — | 1 K |
| (global) | 4h, 7a | — | 2 K |
| (local) | 2h, 7a | 512 | 512 |

As can be seen from the data in Figs. 2.29 and 2.30, the static schemes all perform terribly, and different schemes are better for different benchmarks, usually by a significant margin. BTFNT is the best static scheme for five of the eight benchmarks, but performs terribly for m88ksim. (But bear in mind that BTFNT would look better for floating-point codes, which are heavily loop oriented.) Always-taken is the best static scheme for m88ksim and ijpeg, but is the worst for three other benchmarks. This variability of the best scheme among different benchmarks makes it difficult to choose one static scheme to implement.

Among dynamic schemes, bimodal is worse than the more sophisticated dynamic schemes for all except go and vortex. Unlike the other schemes, however, bimodal is less sensitive to predictor size, with a mean difference between the 8-Kbit and 64-Kbit bimodal predictors of only 0.55%. The reason for this is that the bimodal predictor allocates only one entry to each branch site (i.e., a static branch location in the program), no matter how often that branch executes or how varied its behavior is. Most of the programs have a fairly small number of branch sites, and of course the property of locality means that only a subset of these are active at any one time. A 4 K-entry (8 Kbit) table is, therefore, sufficient to capture most of the static branch locations, and making the table larger has little effect.

Among two-level predictors, GAs are better than PAs about as often as PAs are better than GAs. As with the static schemes, this variability of the best scheme among benchmarks makes it difficult to choose the scheme to implement. This is strong motivation for use of the hybrid predictor, especially given the observation [36,39] that many programs have some branches that are much better predicted using global history, whereas other branches are much better predicted using local history. GAs, PAs, and hybrid, however, are all the more sensitive to predictor size than bimodal is. Go and gcc are particularly sensitive to predictor size, and so are xlisp and perl to some extent. The hybrid predictor is the most sensitive to size, because it must allocate the available hardware budget across four tables: the selector's PHT, the global-history component's PHT, and the local-history component's BHT and PHT.

**TABLE 2.3** Predictor Configurations for a 64-Kbit Hardware Budget

| Predictor | Index Bits (h = hist., a = addr.) | BHT Entries | PHT Entries |
|---|---|---|---|
| Static not-taken | — | — | — |
| Static taken | — | — | — |
| BTFNT | — | — | — |
| Bimodal | 15a | — | 32 K |
| Gas | 8h, 7a | — | 32 K |
| Pas | 8h, 6a | 4 K | 16 K |
| Hybrid (selector) | 6h, 7a | — | 8 K |
| (global) | 7h, 7a | — | 16 K |
| (local) | 8h, 4a | 1 K | 4 K |

**TABLE 2.4** Branch and Fast-Forward Statistics for the SPECint95 Benchmarks

| | Fast-Forward Distance (million) | Static Conditional Branch Sites | Dynamic Conditional Branches Executed (million) |
|---|---|---|---|
| go | 925 | 5331 | 112 |
| m88ksim (m88) | 0 | 968 | 162 |
| gcc | 0 | 20,783 | 190 |
| compress | 1648 | 203 | 151 |
| xlisp | 0 | 676 | 154 |
| ijpeg | 823 | 1,415 | 58 |
| perl | 600 | 614 | 129 |
| vortex | 2450 | 3,203 | 124 |

*Note*: All benchmarks are run for one billion instructions in statistics-gathering mode after the fast-forward interval.

Each of these tables is therefore substantially smaller than in a single two-level predictor and therefore suffers more destructive interference. This especially affects the programs with large static branch footprints, like go and gcc. Yet, a hybrid predictor also has an important advantage: in order to better control destructive conflicts, it can dynamically shift the component it uses to make a prediction for each branch.

Note that these results do not include the effects of predication, context switching, operating system behavior, or any profile-guided feedback. All of these effects might change the results.

### 2.3.6 Summary

Branch prediction is important because otherwise every branch stalls the fetch engine. Some alternatives exist, like delay slots and predication, but delay slots are not compatible with modern, wide-issue superscalar processors, and predication cannot remove all branches. Static prediction techniques that require no hardware support are also possible, but they are either very simple, or in the case of compiler directives, require instruction-set support. Static techniques also have the drawback that they cannot adapt to changing run-time conditions.

Dynamic branch-prediction techniques have evolved from the simple bimodal predictor to more sophisticated two-level and hybrid predictors that exploit patterns in branch behavior and correlation among branches. Refinements to these techniques, as well as new fetch organizations that permit fetching past multiple branches continue to be active areas of research.

The massive effort to find better branch-handling techniques is motivated by the severe penalty imposed by mispredictions. Especially with the long and wide pipelines of modern processors, a very small misprediction rate can severely harm performance. Indeed, the fetch bottleneck remains one of the most severe limitations on faster processing, and Jouppi and Ranganathan [67] argue that it may become the most severe bottleneck in future processors, even more severe than memory latency or memory bandwidth.

With the dramatic shift in focus of recent processors from pure performance to a balance of energy efficiency and performance, the impact of branch prediction accuracy on the overall energy efficiency of a processor has become more important [68]. Since branch prediction accuracy has such a leveraged effect on the performance and energy efficiency (through less wasted work on mispredicted instructions) of a processor, further increases in the size, complexity, and power usage of branch predictors can be justified, as long as the extra power usage of the branch predictor does not exceed the savings in the whole processor [69]. The recent Intel Core Duo [70] and the announced Core 2 Duo processors have shorter pipelines than the previous Intel Pentium 4, yet continue to increase the size and accuracy of their branch prediction hardware, because it allows them to reach a given performance level at a lower power level.

## References


1. Skadron, K., Characterizing and removing branch mispredictions, Ph.D. Thesis, Princeton University, Department of Computer Science, Princeton, NJ, 1999.
2. Calder, B. and Grunwald, D., Reducing indirect function call overhead in C++ programs, in *Proceedings of 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 397–408, Jan. 1994.
3. Chang, P.-Y., Hao, E., and Patt, Y.N., Target prediction for indirect jumps, in *Proceedings of 24th Annual International Symposium on Computer Architecture*, pp. 274–283, June 1997.
4. Driesen, K. and Hölzle, U., Accurate indirect branch prediction, in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 167–178, July 1998.
5. Kalamatianos, J. and Kaeli, D.R., Predicting indirect branches via data compression, in *Proceedings of 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 272–281, Dec. 1998.
6. Seznec, A. and Michaud, P., A case for (partially) tagged geometric history length branch prediction, *The Journal of Instruction Level Parallelism,* 8(1), Feb. 2006, (http://www.jilp.org/vol8).
7. Kaeli, D.R. and Emma, P.G., Branch history table prediction of moving target branches due to subroutine returns, in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 34–41, May 1991.
8. Webb, C.F., Subroutine call/return stack, *IBM Technical Disclosure Bulletin*, Vol. 30, No. 11, April 1988.
9. Gwennap, L., Digital 21264 sets new standard, *Microprocessor Report*, pp. 11–16, Oct. 28, 1996.
10. Gwennap, L., Intel's P6 uses decoupled superscalar design, *Microprocessor Report*, pp. 9–15, Feb. 16, 1995.
11. Patterson, D.A. and Hennessy, J.L., *Computer Architecture: A Quantitative Approach,* 2nd ed., Morgan Kaufmann, San Francisco, CA, 1996.
12. Price, C., *MIPS IV Instruction Set,* Revision 3.1, MIPS Technologies Inc., Mountain View, CA, Jan. 1995.
13. SPARC International Inc., *The SPARC Architecture Manual,* Version 8, Prentice Hall, Englewood Cliffs, NJ, 1992.
14. Digital Equipment Corp., *Alpha AXP Architecture Handbook*, Oct. 1994.
15. Young, C. and Smith, M.D., Static correlated branch prediction, *ACM Transactions Programming Languages and Systems,* 21(5):1028–1075, Sept. 1999.
16. Mahlke, S.A. et al., A comparison of full and partial predicated execution support for ILP processors, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 138–149, June 1995.
17. Mahlke, S.A. et al., Characterizing the impact of predicated execution on branch prediction, in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 217–227, Dec. 1994.
18. B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors in *Proceedings of the 9th Annual International Symposium on High Performance Computer Architecture*, pp. 53–64, Feb. 2003.
19. Smith, J.E., A study of branch prediction strategies, in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135–148, May 1981.
20. Lee, J.K.F. and Smith, A.J., Branch prediction strategies and branch target buffer design, *IEEE Computer,* 17(1):6–22, Jan. 1984.
21. Holgate, R.W. and Ibbett, R.N., An analysis of instruction fetching strategies in pipelined computers, *IEEE Transactions on Computers,* C29(4):325–329, Apr. 1980.
22. Losq, J.J., Generalized history table for branch prediction, *IBM Technical Disclosure Bulletin*, Vol. 25, No. 1, pp. 99–101, June 1982.
23. Calder, B. and Grunwald, D., Fast & accurate instruction fetch and branch prediction, in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 2–11, May 1994.

24. Diefendorff, K., PowerPC G4 gains velocity, *Microprocessor Report*, pp. 10–15, Oct. 25, 1999.
25. W.M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
26. Calder, B. and Grunwald, D., Next cache line and set prediction, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 287–296, June 1995.
27. Kessler, R.E., McLellan, E.J., and Webb, D.A., The Alpha 21264 microprocessor architecture, in *Proceedings 1998 International Conference on Computer Design*, pp. 90–95, Oct. 1998.
28. Jiménez, D.A., Keckler, S.W., and Lin, C., The impact of delay on the design of branch predictors, in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 67–77, Dec. 2000.
29. Moore, G.E., Cramming more components onto integrated circuits, *Electronics,* 38(8):114–117, Apr. 1965.
30. Conte, T. et al., Optimization of instruction fetch mechanisms for high issue rates, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 333–344, June 1995.
31. Yeh, T.-Y., Marr, D.T., and Patt, Y.N., Increasing the instruction fetch rate via multiple branch prediction and a branch address cache, in *Proceedings of the 7th International Conference on Supercomputing*, pp. 67–76, July 1993.
32. Rotenberg, E., Bennett, S., and Smith, J.E., Trace cache: A low latency approach to high bandwidth instruction fetching, in *Proceedings of the 29th Annual. IEEE/ACM International Symposium on Microarchitecture*, pp. 24–34, Dec. 1996.
33. Nair, R., Optimal 2-bit branch predictors, *IEEE Transactions on Computers,* 44(5):698–702, May 1995.
34. Yeh, T.-Y. and Patt, Y.N., Two-level adaptive training branch prediction, in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, Nov. 1991.
35. Yeh, T.-Y. and Patt, Y.N., Alternative implementations of two-level adaptive branch prediction, in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 124–134, May 1992.
36. Evers, M. et al., An analysis of correlation and predictability: What makes two-level branch predictors work, in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 52–61, June 1998.
37. Pan, S.-T., So, K., and Rahmeh, J.T., Improving the accuracy of dynamic branch prediction using branch correlation, in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, Oct. 1992.
38. Standard Performance Evaluation Corporation, SPEC CPU95 benchmarks, http://www.specbench.org/.
39. Skadron, K., Martonosi, M., and Clark, D.W., A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions, in *Proceedings of 2000 International Conference on Parallel Architectures and Compilation Techniques*, pp. 199–206, Oct. 2000.
40. Lee, C.-C., Chen, I.-C.K., and Mudge, T.N., The bi-mode branch predictor, in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 4–13, Dec. 1997.
41. Yeh, T.-Y. and Patt, Y.N., A comparison of dynamic branch predictors that use two levels of branch history, in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, May 1993.
42. McFarling, S., Combining branch predictors, Technical Note TN-36, Digital Equipment Corp. Western Research Laboratory, June 1993.
43. Sechrest, S., Lee, C.-C., and Mudge, T., Correlation and aliasing in dynamic branch predictors, in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 22–32, May 1995.
44. Chen, I.-C., Coffey, J.T., and Mudge, T.N., Analysis of branch prediction via data compression, in *Proceedings on the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 128–137, Oct. 1996.

45. Chang, P.-Y., Hao, E., and Patt, Y.N., Alternative implementations of hybrid branch predictors, in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 252–257, Dec. 1995.
46. Evers, M., Chang, P.-Y., and Patt, Y.N., Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches, in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 3–11, May 1996.
47. Grunwald, D., Lindsay, D., and Zorn, B., Static methods in hybrid branch prediction, in *Proceedings 1998 International Conference on Parallel Architectures and Compilation Techniques*, pp. 222–229, Oct. 1998.
48. Gochman, S. et al., The Intel Pentium M Processor: Microarchitecture and Performance, *Intel Technology Journal*, 7(2):21–59, May 2003.
49. Vintan, L., Towards a high-performance neural branch predictor, in *Proceedings of the International Joint Conference on Neural Networks*, pp. 868–873, July 1999.
50. Jiménez, D.A. and Lin, C., Dynamic branch prediction with perceptrons, in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pp. 197–206, Jan. 2001.
51. Seznec, A., Analysis of the o-geometric historic length branch predictor, in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 394–405, May 2005.
52. Tarjan, D. and Skadron, K., Merging path and gshare indexing in perceptron branch prediction, *ACM Transactions on Architecture and Code Optimization.*, 2(3):280–300, Sept. 2005.
53. Keltcher, C.N., The AMD opteron processor for multiprocessor servers, *IEEE Micro*, 23(2):66–76, Mar.–Apr. 2003.
54. Reinman, G., Austin, T., and Calder, B., A scalable front-end architecture for fast instruction delivery, in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 234–245, May 1999.
55. Glaskowsky, P.N., Pentium 4 (partially) previewed, *Microprocessor Report*, pp. 1, 11–13, Aug. 2000.
56. Diefendorff, K., Hal makes Sparcs fly, *Microprocessor Report*, pp. 1, 6–12, Nov. 15, 1999.
57. Co, M., Weikle, D.A.B., and Skadron, K., Evaluating trace cache energy efficiency, *ACM Transactions on Architecture and Code Optimization,* 3(4):450–476, Dec. 2006.
58. Ramirez, A. et al., Fetching instruction streams, in *Proceedings of the 35th International Symposium on Microarchitecture*, pp. 371–382, Nov. 2002.
59. Tullsen, D.M., Eggers, S.J., and Levy, H.M., Simultaneous multithreading: Maximizing on-chip parallelism, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 392–403, June 1995.
60. Ramsay, M., Feucht, C., and Lipasti, M.H., Exploring efficient SMT branch predictor design, in *Proceedings of the 2003 Workshop on Complexity Effective Design*, June 2003.
61. Skadron, K. et al., Improving prediction for procedure returns with return-address-stack repair mechanisms, in *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pp. 259–271, Dec. 1998.
62. Lu, Z. et al., Alloyed branch history: Combining global and local branch history for robust performance, *International Journal of Parallel Programming*, Kluwer, 31(2):137–177, Apr. 2003.
63. Hao, E., Chang, P.-Y., and Patt, Y., The effect of speculatively updating branch history on branch prediction accuracy, revisited, in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp. 228–232, Nov. 1994.
64. Jourdan, S. et al., Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution, *International Journal of Parallel Programming,* 25(5):363–383, Oct. 1997.
65. Burger, D.C. and Austin, T.M., The SimpleScalar tool set, version 2.0, *Computer Architecture News,* 25(3):13–25, June 1997.
66. Skadron, K. et al., Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques, *IEEE Transactions on Computers,* 48(11):1260–1281, Nov. 1999.

67. Jouppi, N.P. and Ranganathan, P., The relative importance of memory latency, bandwidth, and branch limits to performance, in *Proceedings of the Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.
68. Parikh, D. et al., Power issues related to branch prediction, in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pp. 233–244, Feb. 2002.
69. Co, M., Weikle, D.A.B., and Skadron, K., A break-even formulation for evaluating branch predictor energy efficiency, in *Proceedings of the Workshop on Complexity Effective Design*, June 2005.
70. Gochman, S. et al., The introduction to Intel core duo processor architecture, *Intel Technology Journal*, 10(2):89–97, May 2006.

## 2.4 Network Processor Architecture

*Tzi-cker Chiueh*

### 2.4.1 Introduction

The explosive traffic growth on the Internet comes with an ever more demanding requirement on the available bandwidth and thus on the performance of the network devices that move network packets from sources to destinations. In the most general sense, network processors are those that are specifically designed to transport, order, and manipulate network packets as they move through the network. As network protocols are typically structured as a stack of layers, network processors can be classified into *physical-layer, link-layer,* and *network-layer* processors, depending on the protocol layer at which they operate. Physical-layer processors are responsible for electrical or optical signal generation and interpretation for transporting digital bits, whereas link-layer network processors deal with framing, bit error detection/correction and arbitration of concurrent accesses to shared media. Network-layer processors operate on individual packets and determine how to route packets from their senders to receivers in a particular order, and modify their headers or even payloads along the way if necessary. Because the Internet is largely based on the IP protocol, almost all state-of-the-art network-layer processors are designed to process IP packets only. The conspicuous exception is network-layer processors designed for ATM networks. The focus of this paper, however, is exclusively on network-layer processors, which can operate at from Layer 3 to Layer 7 in the ISO/OSI protocol stack model.

In general, three approaches are used for network processor design, which correspond to different design points in the programmability/performance spectrum. The *ASIC* approach takes a full customization route by dedicating specially-made hardware logic to specific network packet processing functionalities. Although this approach gives the highest performance, it is typically not programmable and therefore not sufficiently flexible to support a wide variety of network devices. Consequently, such processors are more expensive and tend to be outdated sooner because they cannot exploit economies of scale to keep up with technology advances. The *general-purpose CPU* approach either takes an existing processor for PCs or embedded systems as it is, or augments it with a small set of instructions specifically included to improve network packet processing. While this approach admits the most programming flexibility, the throughput of these processors is substantially lower than what modern network devices require. The main reason for this lackluster performance is that network device workloads are data movement-intensive, whereas traditional processors are designed to support computation-intensive tasks. The last approach to network processor design, *programmable network processor,* attempts to strike a balance between programmability and performance and achieves the best of both worlds. Instead of using general-purpose instruction set, a programmable network processor defines the set of instruction set primitives for network packet processing from scratch, and exposes these primitives to system designers so that they can tailor the processor to the requirement of different network devices. In the rest of this paper, we will concentrate only on programmable network processors, as they represent the most promising and commercially popular approach to network processor design.

In addition to the basic network processing function such as packet routing and forwarding, modern network processors are tasked with additional capabilities that support advanced network functionalities, such as differentiated quality of service (QoS), encryption/decryption, etc. For network processors that are to be used in *edge* network devices, they may need to perform even higher-level tasks such as firewalling, virtual private network (VPN) support, load balancing, etc. Given an increasing variety of features that network devices have to support, it is crucial for a network processor architecture to be sufficiently general that system designers can build newer functionalities on these processors without causing serious performance degradation. The challenge for network processor design is thus to identify the set of packet processing primitives that is elastic enough to support as many different types of network devices as possible, and at the same time is sufficiently customized so that the performance overhead due to "impedance mismatch" is minimized.

In the rest of this chapter, we first discuss fundamental design issues related to network processor architecture in Section 2.4.2, and then describe specific network processor architectural features that have been proposed in Section 2.4.3. In Section 2.4.4, we review the design of several commercially available network processors to contrast their underlying approaches. Finally, we outline future network processor research directions in Section 2.4.5.

## 2.4.2 Design Issues

To understand the network processor architecture, let us first look at what a programmable network processor is supposed to do. After receiving an IP packet from an input interface, the network processor first determines the output interface via which the packet should be forwarded toward its destination. In the case that multiple input packets are destined to the same output interface, the network processor also decides the order in which these packets should be sent out on the associated output link, presumably according to certain quality of service (QoS) policy. Finally, before the packet is forwarded to the next-hop router, the network processor modifies the packet's header or even payload according to standard network protocols or application-specific semantics. For IP packets, at least the TTL (time-to-live) field in the header must be decremented at each hop and as a result the IP packet header checksum needs to be re-computed. Other IP header fields such as TOS (type-of-service) may also need to be modified for QoS reasons. In some cases, even the packet body need to be manipulated, e.g., transcoding of a video packet in the presence of congestion. In summary, given an input packet, the network processor needs to identify its output interface, schedule its transmission on the associated output link, and make necessary modifications to its header or payload to satisfy general protocol or application-specific requirements.

Fundamentally, a network processor performs three types of tasks: *packet classification, packet scheduling,* and *packet forwarding*. Given an input IP packet, the packet classification module in the network processor decides how to process this packet, based on the packet's header and sometimes even payload fields. In the simplest case, the result of packet classification is the output interface through which the input packet should be forwarded. To support differentiated QoS, the result of packet classification becomes a specific output connection queue into which the input packet should be buffered. In the most general case, the result of packet classification points to the software routine that is to be invoked to process the input packet; possible processing ranges from forwarding the input packet into an output interface and a buffer queue, to arbitrarily complex packet manipulation. The design challenge of packet classification is that the number of bits used in packet classification is increasing due to IPv6 and/or multiple header fields, and varying because of application-level protocols such as URL in the HTTP protocol.

The packet forwarding module of a network processor physically moves an input packet from an incoming interface to its corresponding outgoing interface. The key design issues on packet forwarding are the topology of the switch fabric and the switch scheduling policy to resolve output contention, i.e., when multiple incoming packets need to be forwarded to the same output interface. State-of-the-art network devices are based on crossbar fabrics, which are more expensive but greatly reduce the implementation complexity of the switch scheduler. Given a crossbar fabric, the switch scheduler

finds a match between the incoming packets and the output interfaces so that the switch fabric is utilized with maximum efficiency and the resulting matching is consistent with the output link's scheduling policy, which in turn depends on the QoS requirement. Algorithmically, this is a constrained bipartite graph matching problem, which is known to be NP-complete. The design challenge of switch scheduling is to find a solution that approximates the optimal solution as closely as possible and that is simple enough for efficient hardware implementation. One such algorithm is iterative random matching [1] and its optimized variant [2].

Traditionally, a FIFO queue is associated with each output link of a network device to buffer all outgoing packets through that link. To support fine-grained QoS, such as per-network-connection bandwidth guarantee, one buffer queue is required for each network connection whose QoS is to be protected from the rest of the traffic. After classification, packets that belong to a specific connection are buffered in the connection's corresponding queue. A link scheduler then schedules the packets in the per-connection queues that share the same output link in an order that is consistent with each connection's QoS requirement. A general framework of link scheduling is packetized fair queuing (PFQ) [3], which performs the following two operations for each incoming packet, *virtual finish time computation,* which is *O(N)* computation, and *priority queue sorting,* which is $O(\log N)$ computation, where $N$ is the number of active connections associated with an output interface. Intuitively, a packet's virtual finish time corresponds to the logical time at which that packet should be sent if the output link is scheduled according to the fluid fair queuing model. After the virtual finish time for each packet is computed, packets are sent out in an ascending order of their virtual finish time. A nice property of virtual finish time is that an earlier packet's virtual finish time is unaffected by the arrival of subsequent packets. With per-connection queuing and output link scheduling, traffic shaping is automatic if packets are dropped when they reach a queue that is full. As the complexity of both operations in link scheduling depends on $N$, they cannot readily scale to a large number of QoS-sensitive connections. Although there are various attempts to simplify PFQ, most hardware link schedulers use a much simpler weighted round-robin algorithm, which multiplexes connections on an output link according to weights that are proportional to the connections' QoS requirements.

In addition to the above three types of tasks, a network processor may need to support such higher-level functionalities as security, multicast, congestion control, etc., because more and more intelligence is moved from end hosts to the network. It is not clear what common primitives these high-level functions can share, as comprehensive workload characterization along this line is almost nonexistent. The entire *active network* field [4] is about the development of operating system and network procotols support for *programmable* network devices so that they can perform application-specific operations on an application's packets as they go through the network; however, until now the research on architectural support for active networking is almost nonexistent in the literature.

Figure 2.31 shows the generic architecture of an Internet router, which serves as an instance of a network device that is built on network processor. The line cards are input/output interfaces and are connected to external network links. Line cards typically include hardware for physical-layer and link-layer protocol processing, and a network processor that performs packet classification, queuing and output-link scheduling. The switch fabric controller determines when input packets should be forwarded to their corresponding output interface. The control processor is typically a standard RISC processor and is responsible for non-time-critical tasks such as routing table maintenance and traffic statistics collection and reporting.

### 2.4.3 Architectural Support for Network Packet Processing

Given the network device system architecture in Fig. 2.31, in this section we present architectural features that were proposed previously to improve the performance of network processors. But first let's consider the performance requirements of a very high-speed router. Assuming the worst-case scenario, i.e., each packet is 64-byte long, an OC-768 or 40 Gbps network processor needs to handle 80 million packets per second, or one packet every 12.5 ns. Assume further that a packet is processed in a

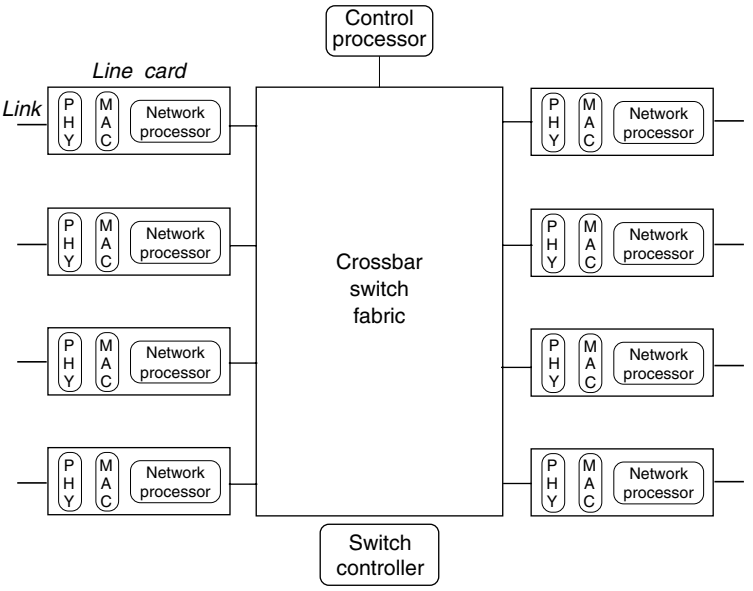



**FIGURE 2.31** The system architecture of a generic network device such as an IP packet router.

pipeline fashion, i.e., packet classification, packet forwarding, and packet scheduling. Therefore, this 12.5 ns corresponds to the pipeline cycle time, rather than the total packet latency within the network device. In packet classification, multiple memory accesses to the routing/classification table data structure are needed. For a static RAM with 2-ns cycle time, a 12.5-ns cycle time means that the network processor cannot access more than six memory words if the memory system is 32-bit wide. In packet forwarding, the output buffer queue memory should run at 1,250 MHz using a 128-bit-wide interface, because the rule of the thumb is that packet buffers need to operate at four times as fast the line rate. In packet scheduling, even assuming a modest number of active connections, say 1,000, the link scheduler logic needs to perform each primitive operation in virtual finish time calculation within 12.5 ps, or about one CMOS transistor delay. The above analysis demonstrates that in all three cases, new architecture-level and circuit-level innovations are required to develop a network processor that meets the OC-768 performance goal.

Two basic approaches to speeding up network packet processing inside a network processor are pipelining and parallelization. A deeper pipeline reduces the cycle time of the pipeline and, therefore, improves the system throughput. However, because packet classification, packet forwarding, and packet scheduling each exhibit complicated internal dependencies and sometimes iterative structures, it is difficult to pipeline these functions effectively. Parallelization can be applied at different granularities. Finer granularity parallelism is more difficult to exploit but potentially leads to higher performance gain. In the case of network packet processing, because processing of one packet is independent of processing of another packet, packet-level parallelism appears to be the right granularity that strikes a good balance between performance gain and implementation complexity. Typically, a thread is dedicated to the processing of one packet, and different threads can run in parallel on distinct hardware engines. To reap further performance improvement by exploiting instruction-level parallelism, researchers and companies have proposed to run concurrent packet-processing threads on a simultaneous multithreading processor [5,11] to mask as many pipeline stalls as possible. Such multithreading processors require the support of multiple hardware contexts and fast context switching.

Compared with generic CPU workloads, network packet processing requires much more frequent bit-level manipulation, such as header field extraction and header checksum computation. In a standard RISC processor, extracting an arbitrary range of bits from a 32-bit word requires at least three

instructions, and performing a byte-wide summing of the four bytes within a word takes at least 13 instructions. Therefore, commercial network processors [6] include special bit-level manipulation and 1's complement instructions to speed up header field extraction and replacement, as well as packet checksumming computation.

Caching is arguably the most effective and most often used technique in modern computer system design. One place in network processor design to which caching can be effectively applied is packet classification. Since multiple packets travel on a network connection in its lifetime, in theory each intermediate network device only needs to perform packet classification once for the first packet and reuses the resulting classification decision for all subsequent packets. This corresponds to temporal locality if one treats the set of all possible values of the header fields used in packet classification as an address space. Empirical studies [7,8] show that network packet streams indeed exhibit substantial temporal locality but very little spatial locality. In addition, unlike CPU cache, the classification results for neighboring points in this address space tend to be identical. Therefore, network processor cache can be designed to cache address ranges rather than just address space points, as in standard cache. Chiueh and Pradhan [8] showed that caching ranges of classification address space can increase the effective coverage of a network processor cache by several orders of magnitude as compared to conventional caches that cache individual addresses.

Another alternative to speed up packet classification is through special content-addressable memory (CAM) [13]. Commercial CAMs support ternary comparison logic (0, 1, and X or don't-care). Classification rules are pre-stored in the CAMs. Given an input packet, the selective portion of its packet header is compared against all the stored classification patterns in parallel, and a priority decoder picks the highest priority among the matched rules if there are multiple of them. Although CAM can identify relevant packet classification rules at wire speed, two problems are associated with applying CAM to the packet classification problem. First, to support range match, e.g., source port number 130–202, one has to break a range rule to multiple range rules, each covering a range whose size is a multiple of 2. This is because CAMs only support don't-care match but not arbitrary arithmetic comparison. For example, the range 129–200 needs to be broken down into eight ranges: 130–130, 131–132, 133–136, 137–144, 145–160, 161–192, 193–200, and 201–202. For classification rules with multiple range fields, the need for range decomposition can significantly increase the number of CAM entries required. Second, because CAMs are hardwired memory with built-in width, it cannot easily support matching of variable-length fields such as URL, or accommodate changing packet classification rules after network devices are put into field use.

Finally, because the main task of network devices is to move packets from one interface to another, efficient data movement is of paramount importance. Because most packet buffer memory is implemented in DRAM, it is essential to exploit the fast access mode in modern DRAM chips to keep up with the line rate. In addition, it should support multiple DMA channels to allow multiple data transfer transactions to proceed in parallel without the attention of network processors.

### 2.4.4 Example Network Processors

Intel's Internet exchange architecture (IXA) [6] includes an IXE component as the switching fabric, an IXF component for framing and formatting, an LXT component for physical-layer processing, and an Internet exchange processor (IXP) for packet processing. The IXP consists of a StrongARM core, six microengines and interfaces with the SRAM, SDRAM, the PCI bus, and a proprietary bus, the IX bus. The StrongARM core performs such supervisory processing as maintaining the routing table. Each of six microengines is a RISC core augmented with special instructions optimized for network processing such as bit extraction, table lookup, and single-cycle shifting, and with support for hardware multithreading. Each microengine has four program counters that allow four parallel threads to time-share a microengine's data path. There are two banks of single-ported general-purpose registers for ALU operations, and four single-ported transfer registers to read/write SRAM and SDRAM. The IX bus allows the IXPs to interface with IXFs and IXEs, and supports 5 Gbps at 80 MHz.

Agere's PayloadPlus architecture [9] includes a fast pattern processor (FPP), a routing switch processing (RSP), an agere system interface (ASI), and a functional programming language (FPL) for programming the FPP and RSP. The FPP sits between the physical interface and the RSP, and performs packet re-assembly, protocol recognition and associated computation, and calculation of checksums and CRC. The FPP is based on a pipelined and multithreaded architecture. It allocates a thread and a context to process each incoming packet, and operates on one 64-byte block at a time, each in the associated packet's context. To program the FPP, system designers use a declarative programming language, FPL, to specify the set of protocols to recognize and the set of actions to take for each specified protocol. Programs for the FPP are represented as trees, where nodes correspond to pattern recognition functions and leaves as actions. The RSP sits between the FPP and the switch fabric controller, and consists of three VLIW engines: Traffic Management Compute engine that enforces packet discarding policies and maintains queue statistics, Traffic Shaper Compute engine that ensures QoS and CoS for each connection queue, and Stream Editor Compute engine that performs necessary packet modifications. These three engines work on each packet together as a linear pipeline. The ASI interfaces with the host processor for configuration and program download, and in addition coordinates the data movement between the FPP and RSP.

C-Port's digital communications processor (DCP) [10] includes 16 channel processors (CP), five specialized processors, and a 160 Gpbs internal bus. Each CP interfaces with the physical link interface, and consists of a RISC core and two serial data processors (SDP). SDPs perform low-level bit manipulation task whereas the RISC core performs such high-level task as packet scheduling and traffic statistics collection. The five specialized processors perform classification table access, packet buffering, routing table lookup, interfacing with the switch fabric, and supervisory processing. C-Port supports a special communications programming interface called C-Ware to simplify system designers' task of programming DCP.

### 2.4.5 Conclusion

In this chapter, we present the set of tasks that a modern network processor needs to perform, describe a set of architectural features specifically designed for network packet processing, and survey several commercial network processor architectures as examples. Most of existing network processors include special instructions to speed up packet processing, and use a parallel multithreaded architecture to exploit multiple levels of parallelism; however, these architectures cannot scale to OC768 link rate and beyond, and, therefore, further research into network processor architecture is warranted. Here are several research directions that we believe are worth exploring:

- Scalable packet classification mechanism that supports variable-length application-level classification patterns
- Integrated packet scheduling for both switch fabric and output links to achieve per-connection QoS in an input queuing network device architecture
- Novel memory management scheme that exploits the abundant internal bandwidth of intelligent RAM architecture [12] to cost-effectively satisfy the memory bandwidth requirements of terabit links
- Architectural support for active networking and other high-level network functionalities

## References

1. Nick McKeown and Thomas E. Anderson. "A quantitative comparis on of scheduling algorithms for input-queued switches." *Computer Networks and ISDN Systems*, vol. 30, no. 24, pp. 2309–2326, December 1998.
2. Nick McKeown. "iSLIP: A scheduling algorithm for input-queued switches." *IEEE Transactions on Networking*, vol. 7, no. 2, April 1999.

3. Keshav, S., "On the efficient implementation of fair queueing." *Journal of Internetworking: Research and Experience*, Vol. 2, no. 3, September 1991.
4. Tennenhouse, D. and D. Wetherall. "Towards an active network architecture." *Computer Communication Review*, vol. 26, no. 2, p. 5–18, April 1996.
5. Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad. "Characterizing processor architectures for programmable network interfaces." In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, N.M., May 2000.
6. Intel Internet Exchange Architecture, http://developer.intel.com/design/ixa/whitepapers/ixa.htm.
7. Tzi-cker Chiueh and Prashant Pradhan. "High-performance IP routing table lookup using CPU caching." In *Proceedings of IEEE INFOCOM 1999*, New York City, April 1999.
8. Tzi-cker Chiueh and Prashant Pradhan. "Cache memory design for internet processors." In *Proceedings of Sixth Symposium on High-Performance Computer Architecture (HPCA-6)*, Toulouse, France, January 2000.
9. Agere Systems. The PayloadPlus Architecture. http://www.lucent.com/micro/netcom/docs/fppproductbrief.pdf.
10. David Husak and Robert Gohn. "Network processor programming models: the key to achieving faster time-to-market and extending product life." http://www.cportcorp.com/products/pdf/net_proc_prog_models.pdf.
11. Xtream Logic Corporation. "Xstream logic packet processor core." http://www.xstreamlogic.com/architectural_files/v3_document.htm.
12. David Patterson et al. "A case for intelligent DRAM: IRAM," *IEEE Micro*, April 1997.
13. Anthony J. McAuley, Paul F. Tsuchiya, and Daniel V. Wilson. "Fast multilevel hierarchical routing table using content-addressable memory." U.S. Patent serial number 034444. Assignee Bell Communications Research, Inc., Livingston, NJ, January 1995.

## 2.5 Stream Processors and Their Applications for the Wireless Domain

*Binu Mathew and Ali Ibrahim*

### 2.5.1 Introduction

Many embedded media and digital signal processing applications involve simple repeated computations on a very long or never-ending sequence of data. There is limited access or no access at all to past data. A good example is an image processing system such as the simplified version of a face recognition system shown in Fig. 2.32. This surveillance system accepts a video stream from a camera, identifies the pixels that have human skin color, segments the image into regions that contain skin or no-skin, uses a neural network-based algorithm to identify regions that may contain a face, uses another neural network-based algorithm to locate the eyes, and then tries to match the face against a database of known faces to obtain a person's identity. Details of the system may be found in Ref. [1]. The application represents a well-structured assembly of simple compute intensive algorithms. Data flow between the component blocks is regular and predictable. The whole computation may be abstracted as a data-flow graph consisting of a few key procedures and an input stream and an output stream. We say that applications with such simple regular structures are "stream-able" and the style of computation is called stream processing. Other examples include link-level encryption in networks, video transcoding, video compression, cellular telephony as well as the image and speech processing. Even though stream optimized processor hardware is a relatively new area, stream oriented techniques are ubiquitous in the software world with UNIX pipes being a prime example.

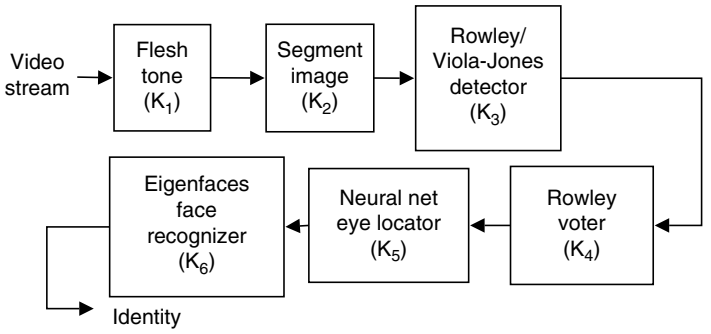



**FIGURE 2.32** Face recognizer: an example stream application.

#### 2.5.1.1 Rationale

Most current research in processor architecture revolves around optimizing four criteria: energy (or power), delay, area, and reliability. For scientific applications, memory bandwidth is also a precious commodity that critically affects delay. One or more of these criteria can often be traded off for the sake of another. For processors, it is often the case that the product of energy and delay required to process a given work load is relatively constant across different architectures after normalizing for the CMOS process [2]. To achieve a significant improvement in the energy-delay product, the architecture needs to be optimized to exploit the characteristics of the target application. Stream applications exhibit three forms of parallelism that can be taken advantage of: instruction level parallelism (ILP) (execute independent instructions in parallel), data parallelism (operate on multiple data elements at once, often using SIMD), and task parallelism (execute different tasks in parallel on different processors).

Current high-performance dynamically scheduled out-of-order processors are optimized for applications that have a limited amount of ILP. They do not depend heavily on task level parallelism as evidenced by the fact that most processors support only one- or two-way SMT and one or two cores per chip. Deep submicron CMOS processes offer the opportunity to fabricate several thousands of 32-bit adders or multipliers on a $10 \times 10$ mm microprocessor die. Yet, because of the limited ILP and irregular nature of typical applications most microprocessors have six or fewer function units. The bulk of the area and power is consumed by caches, branch predictors, instruction windows, and other structures associated with identifying and exploiting ILP and speculation. Stream applications on the other hand have very regular structures, are loop oriented, and exhibit very high levels of ILP. In fact, Kapasi et al. report being able to achieve 28 to 53 instructions per cycle for a set of stream applications on the Imagine stream processor [3]. To achieve such high levels of parallelism, stream processors typically utilize a large number of function units that are fed by a hierarchy of local register files (LRF), stream register files (SRF), and stream caches that decouple memory access from program execution. The resulting bandwidth hierarchy can exploit data parallelism and main memory bandwidth much more efficiently than traditional processors resulting in better performance per unit area or unit power. In addition, it is possible to construct multiple stream processors on the same chip and use stream communication mechanisms to ensure high bandwidth data flow and exploit task level parallelism. The trade-off when compared to general-purpose processors is a much more restrictive programming model and applicability that is limited to the streaming domain.

#### 2.5.1.2 Terminology

Definitions of common stream programming terms follow:

| | |
|---|---|
| Record | A record is either an atomic data type such as an integer or a floating point number or an aggregate of records. |
| Stream | A stream is a directed sequence of records. Depending on their direction, streams may be either input streams or output streams. While most |

streams are very long or never ending, stream programming systems also support some specialized types of streams. Constant streams have a fixed length and are often repeatedly reused. Conditional streams access a record from the stream only when a condition code that is computed in a processor or a function unit cluster of a processor is true [4]. They may be input or output streams depending on the direction of access. Most streams are sequential, i.e., for input streams the algorithm can read only the next record in the stream and write the sequentially next record to an output stream. Gather streams allow arbitrary indexing to fetch records [5]. Indexed streams allow accessing elements within a constrained range of one stream to be accessed using indices from another stream [6].

Derivation — A new stream defined to contain a subset of the records from another stream is called a derived stream. For example, if the variable $x$ refers to an existing input stream, $y$ is a constant stream containing the integers (0,2,4,8) an indexed stream $z$ may be derived from $x$ and $y$ that is constrained to have the range (0,8) relative to the current position of $x$. If $S[i]$ denotes the $i$th record relative to the current position of stream $S$, then $z[0] = x[0]$, $z[1] = x[2]$, and so on. Derivations may be done using other mechanisms such as strided or bit-reversed access. The key point is that the stream compiler should be able to reason about the relationship of records of the derived and base streams.

Kernel — A kernel is a function that transforms input streams to output streams. Kernel functions are typically loop-oriented computations with few or nonexistent conditional branches in the loop body. Simple conditional operations in the original algorithm may often be converted into forms such as predicated execution/if-conversion, conditional moves, and input/output to conditional streams.

Stream Graph — A stream graph is a directed graph where the nodes are kernels and the edges correspond to streams that convey the data between the kernels.

SRF — SRF are specialized storage structures used to hold streaming data. They are essentially large blocks of SRAM under the control of hardware that know how to fetch and store the sequentially next element of a particular stream, perform operations for gather streams, indexed streams, etc.

LRF — The complexity of multiported register files is one of the critical factors that limit the issue width in a traditional microprocessor. Rather than use a single central register file, stream architectures attach LRF to execution clusters or function units. These LRFs typically serve only the units they are attached to. Intercluster communication is handled by a compiler-controlled communication network.

Producer–consumer locality — It is common in stream applications for a producer kernel to generate and save some results to an SRF and the results are immediately used by a consumer kernel straight out of the SRF without saving intermediate results to lower levels of the memory hierarchy. Such reuse of intermediate results is termed producer–consumer locality.

### 2.5.2 Stream Virtual Machine

The stream virtual machine (SVM) is an abstract machine model that has been proposed by Labonte et al. to represent the important characteristics of stream architectures and to develop techniques to compile applications and analyze their performance across different implementation architectures [7]. Their compilation technique proceeds in two stages. First a high level compiler (HLC) reads a stream application written in a stream programming language such as StreamIt or ArrayC. The HLC also reads an abstract SVM model for a stream architecture such as the MIT RAW machine or Stanford Imagine. It then uses the abstract machine model to partition the application into kernels that will execute on particular processing resources and into data transfers between the kernels. This mapping may be described in terms of functions available in the SVM API. API functions provide for initializing local memory, scheduling kernels for execution, declaring dependence between kernels, coordinating DMA transfers between different units, etc. [7]. The kernels are then compiled into binary form by a low level compiler (LLC) that is specific to the particular architecture.

An SVM model for a stream architecture consists of three types of components: processors, memories, and links. Processors in turn come in three varieties. Control processors decide the sequence of operations performed by the entire machine. Control processors offload the compute intensive task of stream kernel execution to kernel processors (stream processors). Lastly, DMA engines are considered as processors that execute specialized kernels that transfer data between the many different memories in the system. The parameters that describe an SVM processor are its type (control, kernel or DMA), its operating frequency, function unit mix, degree of SIMD parallelism, the number and capacity of register files, etc. The memories in an SVM system may be classified depending on their access mechanism into RAMs (random access allowed), FIFOs (only sequential access allowed), and caches (associative lookup allowed). Since stream processors use a hierarchy of memories that capture producer–consumer locality to economize main memory bandwidth, a natural characterization parameter for SVM memories is the bandwidth and latency they offer to entities that are above and below them in the bandwidth hierarchy. Links allow processors and memories to communicate with each other and are characterized by their bandwidth and latency.

Figure 2.33 shows an SVM model to which we can map our face recognizer from Fig. 2.32. It consists of three stream processors, a control processor and a multichannel DMA engine that can move data between the SRFs and main memory. Solid lines indicate data paths and dotted lines indicate control paths. We next describe different types of task to resource mappings for such an application.

### 2.5.3 Time and Space Multiplexing

Consider a set of $n$ kernels named $K = \{K_1, K_2, \ldots, K_n\}$ and $m$ processors $P = \{P_1, P_2, \ldots, P_m\}$. Let $D(K_i, P_j)$ denote the execution time (delay) of kernel $K_i$ when executed on processor (or processor set) $P_j$. When the set of processors is understood from the context, we will denote this as $D(K_i)$. Let $P(P)$ denote

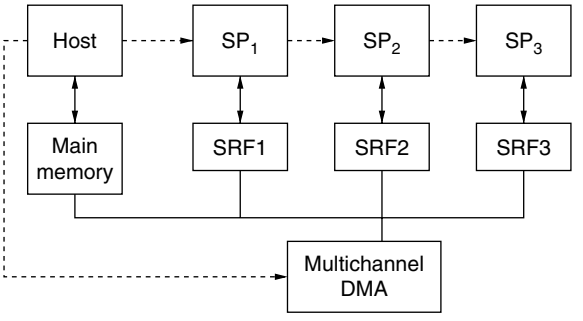



**FIGURE 2.33** SVM machine model example.

the power set of $P$. A schedule $S$ is a mapping $S(K) \rightarrow P(P) \times t$ where $t$ is the start time of the kernel. We will hereafter use the notation $K_i \cdot_p$ and $K_i \cdot_t$ to denote the set of processors and start time assigned to kernel $K_i$. Legal mappings obey the following rules:

1. $K_j \cdot t \geq K_i \cdot t + D(K_i)$ when ever $K_j$ depends on $K_i$, i.e., dependencies are not violated.
2. For all $i, j$ such that $i \neq j$, and $K_i \cdot t \leq K_j \cdot t < K_i \cdot t + D(K_i, K_i \cdot p)$, it should be the case that $K_i \cdot p \cap K_j \cdot p = \phi$, i.e., only one kernel may be executing on a processor at any time.
3. For all $i$, $K_i \cdot p \neq \phi$, i.e., every kernel should be allocated at least one processor.

A stream processing system whose schedules follow the rule, *for all* $i$, $K_i \cdot p = P$ is called a time multiplexed system. In such a system, all available processing resources are allocated to one kernel and then to the next kernel and so on. A stream processing system whose schedules follow the rule, *for all* $i$, $|K_i \cdot p| = 1$ is called a space multiplexed system. In such a system, only one processor is ever allocated to one kernel. If there is some $i, j$ such that $i \neq j$, and $K_i \cdot t \leq K_j \cdot t < K_i \cdot t + D(K_i, K_i \cdot p)$ and $|K_i \cdot p| > 1$, the system is said to be space–time multiplexed. In that case, multiple kernels execute simultaneously and some kernels are given more than one processor. Figure 2.34 shows three different mappings for the application from Fig. 2.32 on to the architecture from Fig. 2.33. By our definitions, Fig. 2.34a is a space multiplexed schedule, Fig. 2.34b is a time multiplexed schedule, and Fig. 2.34c is a space–time multiplexed schedule. To simplify the example, DMA transfers are not shown.

To consider the relative benefits of time and space multiplexing we present a simplified analysis that considers only two kernels $K_1$ and $K_2$ where $K_2$ depends on $K_1$. A more detailed theoretical framework has been developed by the authors, but the details will be deferred to a later publication. Consider the case where the amount of data parallelism available is much larger than the number of processors $m$. For the space multiplexed case, we follow the schedule:

$$S(K_1) = (0, \{P_1, P_2, \ldots, P_{m/2-1}\}) \quad \text{and} \quad S(K_2) = (D(K_1), \{P_{m/2}, P_{m/2+1}, \ldots, P_m\}).$$

For the time multiplexed case, we follow the schedule:

$$S(K_1) = (0, \{P_1, P_2, \ldots, P_m\}) \quad \text{and} \quad S(K_2) = (D(K_1), \{P_1, P_2, \ldots, P_m\}).$$

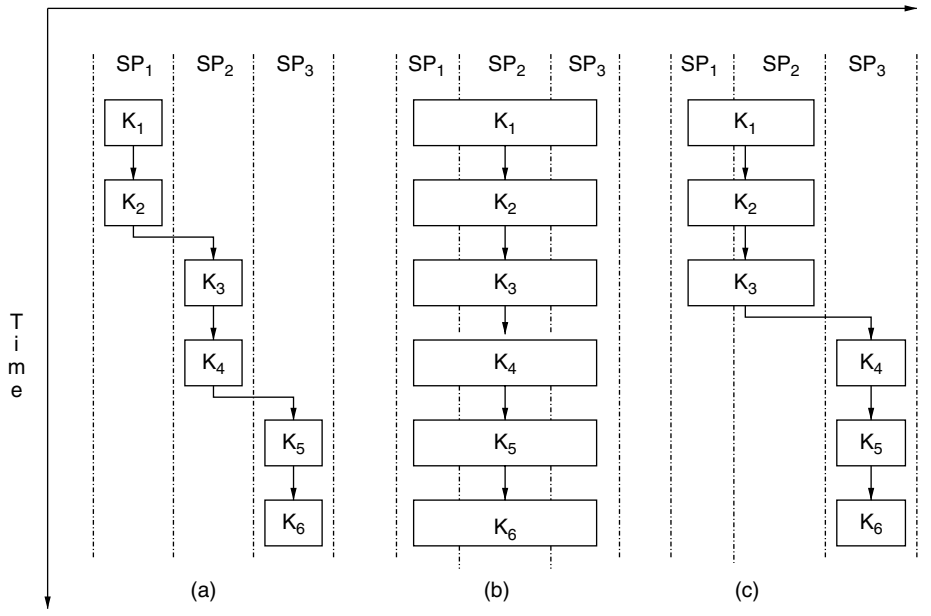



**FIGURE 2.34** Example schedules for the application from Fig. 2.32.

Because of abundant data parallelism, we can assume that execution time of a kernel is inversely proportional to the number of processors allocated to it. Then, $\text{Throughput}_{\text{time mux}} = \frac{1}{D(K_1)+D(K_2)}$ and $\text{Throughput}_{\text{space mux}} = \frac{1}{MAX(2\times D(K_1),\ 2\times D(K_2))} = \frac{1}{2\times D(K_1)}$ (arbitrarily picking $K_1$ as the larger term).

Therefore, $\text{Throughput ratio}_{\text{time mux/space mux}} = \frac{MAX(2\times D(K_1),\ 2\times D(K_2))}{D(K_1)+D(K_2)}$. Since this ratio is greater than or equal to one, time multiplexing works better than space multiplexing in this case. Intuitively, the space multiplexed version works like a pipeline where the stage delays are unbalanced and the pipeline shifts at the speed of the slowest stage while the space multiplexed version is perfectly load balanced. In addition, it is possible to convert the time multiplexed system into an $m$-way SIMD version where all $m$ copies share the same control logic and instruction memory leading to lower area and higher power efficiency. When the stages are balanced $MAX(2\times D(K_1),\ 2\times D(K_2)) = 2\times D(K_1) = 2\times D(K_2)$. Then the throughput ratio becomes one, but time multiplexing is still better because of higher area and power efficiency.

The situation is quite different when the data/instruction level parallelism is limited. Let kernels $K_1$ and $K_2$ each require $N_1$ and $N_2$ instructions worth of computation, respectively. Assume that all instructions require one cycle. Further, assume that dependencies limit the peak IPC possible to $I_1$ and $I_2$ where $I_1, I_2 \leq m/2$. Then, $D(K_1) = N_1/I_1$ and $D(K_2) = N_2/I_2$. Then, $\text{Throughput}_{\text{time mux}} = \frac{1}{\frac{N_0}{I_0}+\frac{N_1}{I_1}}$ and $\text{Throughput}_{\text{space mux}} = \frac{1}{MAX\left(\frac{N_0}{I_0},\frac{N_1}{I_1}\right)}$. Therefore, $\text{Throughput ratio}_{\text{time mux/space mux}} = \frac{MAX\left(\frac{N_0}{I_0},\frac{N_1}{I_1}\right)}{\frac{N_0}{I_0}+\frac{N_1}{I_1}}$.

Since this quantity is less than one, space multiplexing is the better alternative in this case. Intuitively, time multiplexing lets execution resources go waste while space multiplexing shares the resources leading to better performance. As before, when the load is perfectly balanced they have equivalent throughput, but time multiplexing is the better option in that case. On a practical note, when time multiplexed architectures are based on very wide SIMD execution, it is often cumbersome to reformulate algorithms to match the wide SIMD model. Programmers might find it much more convenient to express an algorithm as a pipeline of tasks in a form suitable for space multiplexing. Programming languages like StreamIt attempt to automatically compile space multiplexed code to space–time multiplexed binaries, but further research in this area is required to take advantage of the efficiency of time multiplexed architectures.

## 2.5.4 Stream Processor Implementations

Now that we have introduced a minimal framework to reason about various styles of stream processing, we present an overview of three specific implementations: Stanford Imagine, MIT RAW, and IBM's cell processor.

### 2.5.4.1 Imagine

The Imagine image and signal processor developed by Prof. William Dally and the Concurrent VLSI Architecture Group (CVA) at Stanford University was the pioneering project in stream processing [3]. Figure 2.35 shows the internal structure of an Imagine processor. The Imagine processor consists of eight execution clusters where each cluster contains six ALUs resulting in a peak execution rate of 48 arithmetic operations per second. Each cluster executes a VLIW instruction under the control of the microcontroller. The same VLIW instruction is issued to all clusters resulting in the instruction fetch and control overhead being amortized over eight-way SIMD execution. The bandwidth hierarchy consists of LRF attached to each function unit that provide 435 GB/s, an SRF that feeds the LRFs at 25.6 GB/s, and a streaming memory system that feeds the SRF at 2.1 GB/s. The LRFs within each cluster are connected directly to function units but can accept results from other function units over an intra-cluster switching network. Each cluster also contains a communication unit that can send and receive results from other units over an intercluster switching network. The SRF is internally split into banks that serve each

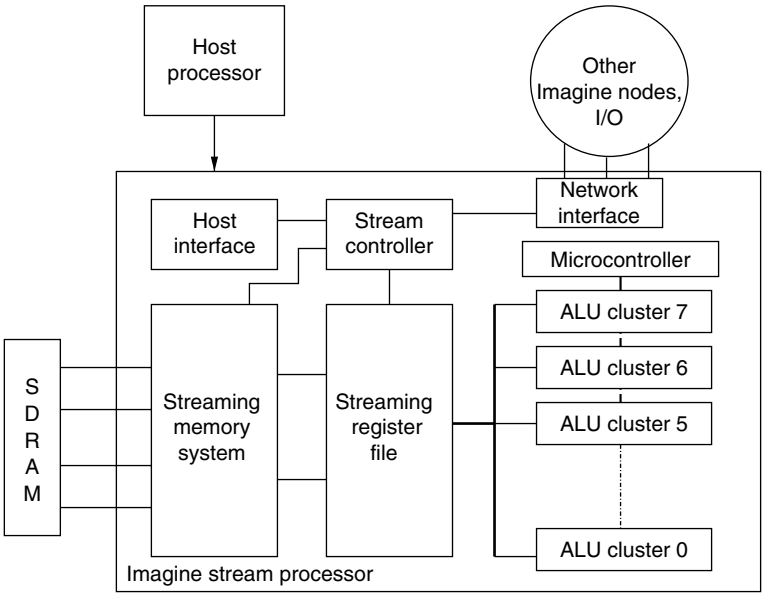



**FIGURE 2.35** Imagine stream processor.

cluster. In addition, each cluster also has a 256 word 32-bit scratchpad memory. The host processor queues kernels for execution with the stream controller via the host interface. The stream controller initiates stream loads and stores via the stream memory system, uses an internal scoreboard to ensure that dependencies are satisfied, and then lets the microcontroller sequence the execution of the next kernel function whose dependencies have all been satisfied. The Imagine processor was fabricated in a 0.18 μ CMOS process and achieved 7.96 GFLOPS and 25.4 GOPS at 200 MHz. Imagine was succeeded by the Merrimac project where the focus was on developing a streaming supercomputer for scientific computing.

Both Imagine and Merrimac were developed primarily around the concept of time multiplexing. Thus they are optimized for applications with very high levels of data parallelism. In a multichip configuration, it is possible to space–time multiplex these systems by making kernels on each node communicate with their counterparts on other nodes over a network interface.

### 2.5.4.2 RAW

The RAW processor is a wire delay exposed tiled architecture developed by Prof. Anant Agarwal and the Computer Architecture Group (CAG) at MIT as a part of the oxygen ubiquitous computing project [8]. Increasing wire delays in submicron CMOS processes and the demand for high clock rates have created a need to decentralize control and resources and distribute resources as semiautonomous clusters that avoid the need for single-cycle global communication. The RAW processor approaches this problem by splitting the die area into a square array of identical tiles and the tiles communicate with each other over a mesh network. Each tile contains an eight-stage in-order single issue MIPS-like processor with a pipelined FPU, 32 KB of instruction cache, 32 KB of data cache, and routers for two static and two dynamic networks that transport 32-bit data. The routers have another 64 KB of instruction cache. Point-to-point transport of scalar values is done over the high performance static network that is fully compiler controlled and guarantees in-order operand delivery. The dynamic network routes operations such as I/O, main memory traffic, and inter-tile message passing that are difficult to fully schedule statically. The static router controls two cross bars each with seven inputs namely the four neighboring tiles in the square array, the router pipeline itself, the other crossbar, and the processor. For tiles on the periphery of the chip, some of the links connect to external interfaces. The tiles and the static router are

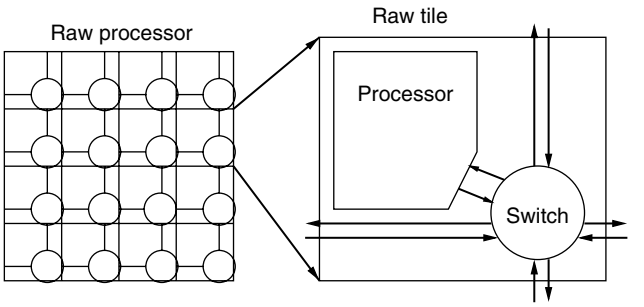



**FIGURE 2.36** RAW processor.

designed for single cycle latency between hops. The compiler encodes the routing decisions for the crossbars into a 64-bit instruction that is fetched from a 64 KB instruction cache and executed by the static router. Inter-tile communication latency is reduced by integrating the network with the bypass paths of the processor. A 225 MHz implementation of a 16 tile RAW processor was fabricated in a 0.18 μ CMOS process and achieved speedups of 4.9–15.4 over a 600 MHz Pentium 3 for a set of stream-oriented benchmarks written in the StreamIt language.

Because of its independent threads of execution in each tile, the RAW processor is capable of performing time, space, and space–time multiplexing. The Streamlt language mostly exposes a space multiplexed programming model even though the compiler is capable of partitioning kernels and load balancing them for space–time multiplexing (Fig. 2.36).

#### 2.5.4.3 Cell

Stream processors made their commercial debut in 2005 with the cell broadband engine architecture (CBEA) from IBM that was developed under collaboration with Toshiba and Sony [9]. This architecture is optimized for a range of compute intensive applications varying from computer games, cryptography, graphics transformations, and lighting to scientific workloads. The cell consists of a 64-bit power processor that serves a similar function to the host processor of Imagine and eight streaming units named synergistic processing elements (SPE). Each SPE is capable of 128-bit SIMD operations that may be two 64-bit, four 32-bit, eight 16-bit, or 16 byte-wide operations. An SPE consists of two pipelines. The even pipeline executes floating point and integer arithmetic while the odd pipeline handles branches, memory accesses, and permutations. Up to two instructions may be issued in-order per cycle to a set of seven function units. The bandwidth hierarchy consists of a 128 word 128-bit LRF in each cluster that is filled from a 256 KB local store (SRF) that is in turn serviced by a globally coherent DMA engine. Interestingly, the SRF also serves as the instruction store for an SPE. Like in the case of the RAW processor, each SPE has its own thread of execution and the system is capable of performing time, space, and space–time multiplexing. The cell processor was fabricated in a 90 nm CMOS process, has a peak operating frequency of 4 GHz, and achieves an SIMD speedup of 9.9 times on a set of compiled benchmarks.

### 2.5.5 Stream Processing for Wireless Systems

Until now, most research in stream processing has addressed media and scientific applications. Wireless communication in forms such as cellular telephony systems and local and personal area networks has become a ubiquitous part of modern life. Mobile wireless systems have relatively high demands for processing and power efficiency and represent a new domain in which stream processors could be quite useful. With around one billion cell phones sold annually, the volume in this market provides economic justification for the development of specialized processor architectures. We present a brief overview of cellular communication technology and indicate avenues for the application of stream processors.

First generation (1G) cellular systems using analog technology were introduced in Scandinavia in 1981 and were followed by similar systems in the United States. They provided only voice transmission. The first digital cellular systems that made their appearance in 1990 and were termed second generation wireless (2G) systems. They provided better voice quality and added data services support with transmission rates up to 9.6 Kbits/s. To support high data rates and to be able to provide multimedia services anytime and anywhere, the International Telecommunications Union defined a family of systems for the third generation (3G) mobile telecommunications called IMT-2000. The 3G system provides data rates up to 2 Mbits/s for stationary users, 384 Kbits/s for pedestrians, and 144 Kbits/s for vehicular users. The services offered by 3G systems can be divided into different classes depending on their delay sensitivity. Voice, video telephony, and video games are delay sensitive. E-mail, short message service, and data downloads are not delay sensitive. In this subsection, we explain the wide-band code division multiple access (WCDMA) system that is commonly used in 3G systems.

The quest to improve data rates and quality of service and to provide seamless roaming and global mobility for voice and data services, a new wireless standard (4G) is currently being formulated. The technologies that would most likely play an important role in 4G are software defined radio (SDR) and multiple input multiple output (MIMO) antenna systems. 3G and 4G systems consist of several layers each providing a specific function. The following sections will only focus on the physical layer where all the compute intensive algorithms are located.

### 2.5.6 WCDMA Physical Layer

Figure 2.37 shows the block diagram of a receiver that uses WCDMA physical layer technology. We focus on the receiver rather than the transmitter, since the latter has a much lower computational complexity than the former.

At the output of the A/D converter, the signal is first filtered using a root-raised cosine (RRC) filter that reduces inter-symbol interference. Subsequently, the signal is fed to a rake receiver and a searcher. Because of the possibility of multipath propagation, the rake receiver has a number of fingers called correlators that individually process several multipath components. In each of these fingers, the signal is unspread by multiplying it by a unique PN code used by the transmitter. This operation separates the desired data signal from interfering signals. The outputs from different correlators are then combined to achieve improved reliability and performance. Depending on the number of multipath components, the number of correlators varies between two and six. The searcher provides an estimate of the number of multipaths and their relative delays. The rake receiver uses this information to control the number of correlators and the delay between them. Finally, the signal is passed through a turbo decoder for error correction. Turbo coding is used in 3G wireless cellular systems because of its outstanding error correction capabilities.

All of these algorithms are dominated by inner loops of low-to-moderate complexity that are applied to real-time data streams. The loop bodies tend to have a high degree of parallelism and have operations

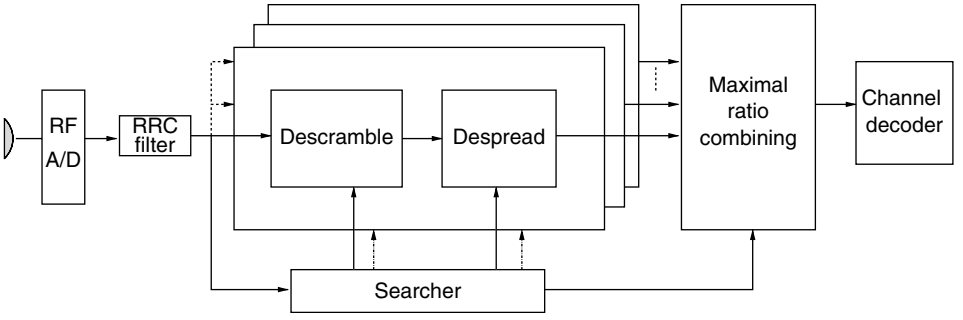



**FIGURE 2.37** Functional diagram of the most compute intensive receiver algorithms.

such as complex-correlation arithmetic that could benefit from processor specialization. The simple regular data-flow and real-time requirements make this system a good candidate for acceleration using customized stream processors.

### 2.5.7 4G

As explained before, it is anticipated that MIMO and software radio technologies may play a key role in 4G systems. The use of a MIMO system could significantly increase the data rate but when compared with 3G it could also increase the computational complexity by one to two orders of magnitude, depending on the number of antennas. There are several different flavors of MIMO-based systems such as MIMO–OFDM (orthogonal frequency division multiplexing) and MIMO MC-CDMA (multi-carrier code division multiple access). A simplified block diagram of a MIMO–OFDM receiver is shown in Fig. 2.38. MIMO–OFDM combines OFDM and MIMO techniques to realize good spectral efficiency and high throughout. It can transmit OFDM modulated data from multiple antennae simultaneously. The receiver first performs OFDM demodulation then does MIMO decoding to extract data from all the transmit antennae and sub-channels.

In software-defined radio technology, a large portion of the radio frequency functionality such as the intermediate frequency (IF) stage, bit-stream processing, modulation/demodulation, and source processing are performed in software running as opposed to the traditional approach of using RF circuits. SDR provides the opportunity to implement multimode terminals that can operate at several different frequency ranges. New services and advanced signal processing techniques can be easily implemented and tested using the SDR approach. Its reprogrammability allows it to download algorithms on demand, and intelligently adapt radio interfaces to different applications and environments. Both MIMO and SDR are computationally intensive techniques with well-structured data-flow and are amenable to stream processing.

### 2.5.8 Computational Complexity and Power Consumption

The computational requirements imposed by cellular standards have increased exponentially from 1G to 4G. This is because of the increased complexity of algorithms introduced to reduce the bit-error rate use of the wireless spectrum more efficiently. These algorithms have been shown to require more performance than is currently available in embedded processors. Table 2.5, shows the computation requirements to handle 384 Kbits/s transmission rate on a 3G WCDMA receiver [10]. The problem will persist in the future since the computation requirements are growing faster than Moore's law [10]. Power dissipation is also a major problem in battery-powered mobile computing and communication devices. Although the computational requirements are increasing exponentially, battery capacity is only improving at the

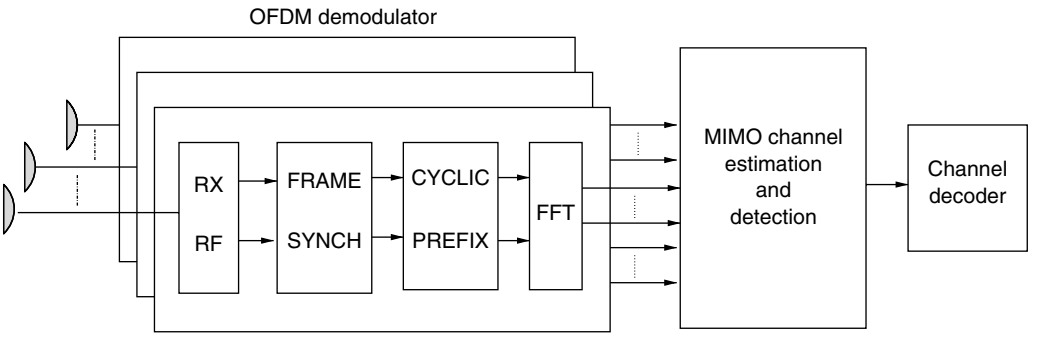



**FIGURE 2.38** MIMO–OFDM receiver.

**TABLE 2.5** Computation Requirements for 3G WCDMA Receiver

| Algorithm | Approximate MIPS |
|---|---|
| Digital filter (RRC, channelization) | 3000 |
| Searcher | 1500 |
| Rake | 650 |
| Maximal-ratio combining | 24 |
| Channel estimator | 12 |
| AGC, AFC | 10 |
| De-interleaving, rate matching | 14 |
| Turbo decoding | 52 |
| Sum | 5262 |

rate of 1.03 times per year [11]. Flexibility and low time to market require the use of programmable processors for the implementation of the increasingly sophisticated digital signal processing algorithms. Power efficiency on the other hand, requires the use of a customized solution. To make 4G systems usable, the performance per unit power consumption of processors needs to improve significantly—an area in which stream processors have a significant advantage.

## 2.5.9 Current Solutions

As explained in the previous sections, wireless applications can be partitioned into different components that can be space–time multiplexed efficiently. This structural simplicity has given rise to several multiprocessor system-on-chip (SoC) architectures with the OMAP from TI, SB3010 from Sandbridge technologies, and the EVP processor from Philips being prominent examples.

### 2.5.9.1 Texas Instruments OMAP

The OMAPV2230 is one of the interesting SoCs from TI's OMAP-VOX product family [12]. It is an integrated universal mobile telecommunications system (UMTS) solution for 3G handsets. It integrates a digital base-band system and an applications processor. The digital base-band system consists of an ARM processor for control purposes, a TI TMS320C55x processor for DSP algorithms, and a custom ASIC module that handles the compute intensive portions of 3G base-band processing. The applications processor includes a dedicated 2D/3D graphics accelerator and an image video audio accelerator (IVA).

### 2.5.9.2 Philips EVP

The embedded vector processor (EVP) developed by Philips is a VLIW-based scalable SIMD architecture designed to support multiple 3G standards [13]. The EVP includes several specialized function units such as a shuffle unit, an intra-vector unit that supports intra-vector operations, and a code generation unit that supports CDMA code generation. It also includes an address calculation unit that supports an extensive set of addressing modes. The EVP exploits VLIW parallelism by providing the ability to issue five vector operations, four scalar operations, and address updates and loop control operations simultaneously.

### 2.5.9.3 Sandbridge Technologies SB3010

Sandbridge Technologies has implemented a 3G multimedia handset design using their SB3010 base-band processor [14]. The processor integrates an ARM 9 RISC core, four of Sandbridge's own Sandblaster DSP cores, on-chip instruction caches and data memories, and a programmable RF interface. In most broadband communication systems, data is streamed from an A/D converter. To accommodate for this, their design uses scratchpad memories rather than data caches. Each Sandblaster core delivers two billion MAC operations per second and supports eight hardware threads. With a total performance of the order of 10 billion MACs per second, the SB3010 is able to run different wireless protocols such as WCDMA, as well as multimedia codecs such as MPEG-4 H.264 and MP-3.

The DSP architecture can be partitioned into an instruction fetch and branch unit, an integer and load store unit, and an SIMD vector unit. This SIMD unit consists of four vector processing elements (VPE), an accumulator register file, a shuffle unit, and a reduction unit. Integer operations 16, 32, and 40-bit fixed-point data types is supported. The Sandblaster DSP implements an unorthodox multi-threading method to avoid the hassles posed by data dependencies and hazards in pipelined processors. Eight hardware threads are supported, but they issue instructions round-robin. Effectively, each thread issues an instruction every eighth cycle and its dependencies resolve while it waits for other threads to take their turn.

### 2.5.10 Stream Processor Based Wireless SoCs

As shown in Fig. 2.37 and explained in previous sections, wireless systems usually consist of multiple DSP algorithm kernels connected in feed forward pipelines and data is streamed between the kernels. These kernels are typically compute bound, exhibit high levels of data parallelism, typically require low precision fixed-point arithmetic, and contain bit manipulation operations that could benefit from customized instructions and function units.

The ACT stream processor developed at the University of Utah has demonstrated that compute intensive 3G base-band algorithms stream architectures perform very well on stream architectures [15–17]. The energy-delay product of this stream processor was within one to two orders of magnitude of that of an ASIC. This research is a first step toward the goal of creating high-efficiency wireless SoCs based on space–time multiplexed implementations of base-band algorithms on a network of customized stream processors. Unlike the general mesh network of the RAW processor, the on-chip interconnect between the stream processors can be customized to account for the data-flows observed in 3G and 4G systems. It is known that the input from the A/D converter stage to a WCDMA system requires at most 7.68 MB/s bandwidth [18]. Communication between later kernels in a WCDMA system requires even less bandwidth. This makes it possible to use low throughput interconnects between different stream processors. Once the data is received by a particular core, it may need to be buffered and accumulated. In the case of MIMO-OFDM, this takes the form of a FIFO that is required between the OFDM modulator and MIMO detection. In the case of turbo codes, this is because the algorithm operates only on blocks of data. In either case, an SRF structure that supports sequential and indexed streams would be adequate to handle the buffering requirements. Some parts of wireless processing may not be amenable to space multiplexing because of load imbalance between stages, variability in data arrival times. A mixture of distributed control, space–time multiplexing, data rearrangement units, and programmable interconnect may be required to solve all the complex challenges posed by 4G algorithms.

### 2.5.11 Conclusions

Stream processors have been extensively studied in academia by projects such as Stanford Imagine and MIT RAW. They made their commercial debut with the cell, a broadband processor from IBM that is the computing engine for the Sony PlayStation 3. Research has demonstrated that stream processors can achieve very high levels of energy efficiency and performance on a variety of speech and image processing as well as wireless communication tasks [1,17]. The cellular telephony market has experienced rapid growth around the world and represents a significant opportunity for stream processors because this domain requires very high computation rates to reduce the bit error rate and to support high data rates, full motion video and multimedia applications, and a variety of wireless standards. Simultaneously, they must also be energy efficient and flexible, have a low time to market, and be low cost. The stringent power requirements of mobile wireless systems calls for different patterns of stream processor customization than previously studied for scientific stream processors. The possibility of customizing the instruction set architecture and on-chip interconnect and performing energy-delay trade-offs for wireless optimized stream processors merits further study.

## References


1. B. Mathew. *The perception processor*. Ph.D. Thesis, School of Computing, University of Utah, Salt Lake City, UT, Aug. 2004.
2. R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, Sept. 1996.
3. U. Kapasi, W.J. Dally, S. Rixner, J.D. Owens, and B. Khailany. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design*, pp. 282–288, Sept. 2002.
4. U.J. Kapasi, W.J. Dally, S. Rixner, P.R. Mattson, J.D. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In *MICRO 33: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 159–170, New York, 2000. ACM Press.
5. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Transaction Graph*, 23(3):777–786, 2004.
6. P. Mattson, U. Kapasi, J. Owens, and S. Rixner. Imagine programming system user's guide. http://cva.stanford.edu/classes/ee482s/docs/ips_user.pdf, 2002.
7. F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (Pact, 04)*, pp. 267–277, 2004.
8. M.B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, p. 2, Washington, DC, 2004. IEEE Computer Society.
9. D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, 2006.
10. D. Greifendorf, J. Stammen, and P. Jung. The evolution of hardware platforms for mobile "software defined radio" terminals. *Proceeding IEEE Personal, Indoor, and Mobile Radio Conference*, Sept. 2002.
11. J.M. Rabaey, M. Potkonjak, F. Koushanfar, S. Li, and T. Tuan. Challenges and opportunities in broadband and wireless communication designs. In *Proceedings IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 76–83, Nov. 2000.
12. T. Instruments. Omapv2230. Technical Report, http://focus.ti.com/pdfs/wtbu/ti_omapv2230.pdf.
13. K. van Berkel, F. Heinle, P.P. Meuwissen, K. Moerman, and M. Weiss. Vector processor as an enabler for software defined radio in handheld devices. *EURASIP Journal on Applied Signal Processing*, 16:2613–2625, 2005.
14. M. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, and S. Vassiliadis. A low-power multithreaded processor for software defined radio. *Journal of VLSI Signal Processing Systems*, 43 (2/3):143–159, 2006.
15. A. Ibrahim and A. Davis. Address acceleration mechanisms for an adaptive cellular telephony processor. International Conference on Multimedia and Expo (ICME), 2005.
16. A. Ibrahim and A. Davis. Exploiting data context switching in a low power VLIW coprocessor. *Workshop on Optimizations for DSP and Embedded Systems (ODES), in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2005.
17. A. Ibrahim, M. Parker, and A. Davis. Energy efficient cluster coprocessors. *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2004.
18. H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access for Third Generation Mobile Communications*, 2nd edn., John Wiley & Sons, 2002.

# 3

# Architectures for Low Power




Pradip Bose
*IBM T.J. Watson Research Center*


## 3.1 Introduction

Power dissipation limits have emerged as a key constraint in the design of microprocessors, even for those targeted for the high-end server product space. At the low end of the performance spectrum, power has always dominated over performance as the primary design constraint. However, although battery life expectancies have shown modest increases, the larger demand for increased functionality and speed has increased the severity of the power constraint in the world of handheld and mobile systems. At the high end, where performance was always the primary driver, we are witnessing a trend where increasingly, energy and power limits are dictating the high-level processing paradigms, as well as the lower-level issues related to clocking and circuit design.

Figure 3.1 shows the expected maximum chip power (for high-performance processors) through the year 2020. The data plotted is based on the updated 2006 projections made by the International Technology Roadmap for Semiconductors (ITRS) (http://public.itrs.net). The projection indicates that beyond the continued growth period (through the year 2007) for high-end microprocessors, there will be a saturation in the maximum chip power (with a projected cap of around 200 W from years 2008 all the way through 2020). This is ostensibly because of thermal/packaging and die size limits that have already started to kick in. Beyond a certain power regime, air cooling is not sufficient to dissipate the heat generated; on the other hand, use of liquid cooling and refrigeration causes a sharp increase of the cost–performance ratio. Thus, power-aware design techniques, methodologies, and tools are of the essence at all levels of design.

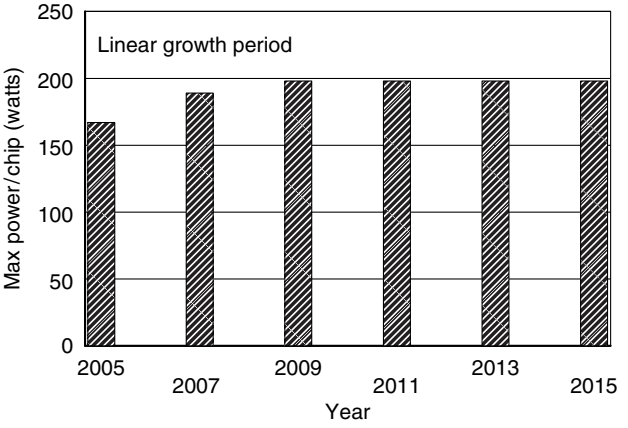



**FIGURE 3.1** Maximun chip power projection (ITRS roadmap).

In this chapter, we first present a survey of some of the most promising ideas in power-aware design at the microarchitecture level. We base this review on the currently available literature, with special emphasis on relevant work presented at recent architecture conferences and workshops. Where useful, we also refer to earlier papers that deal with fundamental issues related to the performance, cost, and scalability of concurrent machine architectures. We show how the fundamentals of machine performance relate to the modern problem of architecting processors in a way that allows them to scale well (over time) in terms of joint power–performance metrics. In this context, we comment on and compare the viability and future promise of several microarchitectural paradigms that have already been picked up by industrial designs, e.g., clustered superscalars, various flavors of multithreading (e.g., simultaneous multithreading (SMT)), and chip multiprocessors (CMP).

In Section 3.2, we review the fundamentals of pipelined and vector/parallel computation as they relate to performance and energy characteristics. We also dwell briefly on the topic of defining a suitable set of metrics to measure power–performance efficiency at the microarchitecture level. In Section 3.3, we provide a survey of the most promising ideas and approaches in power-aware design at the microarchitecture level, with references to circuit design and clocking issues that are relevant in the discussion. This review is presented in the context of workloads and benchmarks that represent different markets. In Section 3.4, we compare the power–performance outlook of three emerging microarchitectural paradigms in the general-purpose processor space: multicluster superscalars, multithreaded processors, and CMP. In Section 3.5, we conclude by summarizing the main issues addressed in this chapter. We also point to a list of future research items that the architecture community needs to pursue in collaboration with the circuit design community to achieve the targets dictated by future cost and performance pressures.

## 3.2 Fundamentals of Performance and Power: An Architect's View

### 3.2.1 Performance Fundamentals

The most straightforward metric for measuring performance is the execution time of a given program mix on the target processor [1,2]. The execution time can be written as

$$T = \mathrm{PL} * \mathrm{CPI} * \mathrm{CT} = \mathrm{PL} * \mathrm{CPI} * (1/f) \tag{3.1}$$

where

PL is the dynamic path length of the program mix, measured as the number of machine instructions executed

CPI is the average processor cycles per instruction incurred in executing the program mix

CT is the processor cycle time (measured in seconds per cycle) whose inverse determines the clock frequency $f$

Since performance increases with decreasing $T$, one may formulate performance, Perf, as

$$\text{Perf}_{\text{chip}} \sim = K_{\text{pf}} f \sim = K_{\text{pv}} V \tag{3.2}$$

where, the $K$'s are constants for a given microarchitecture-compiler implementation and for a specific workload mix. The $K_{\text{pf}}$ value stands for the average number of machine instructions that are executed per cycle on the machine being measured; this is usually referred to as IPC, the inverse of CPI. Performance, $\text{Perf}_{\text{chip}}$, in this case is measured in units like millions of instructions per second (mips). In Equation 3.2, $V$ is the chip supply voltage (often written as $V_{\text{dd}}$). As stated below, and in Refs. [2,3], the operating frequency is often assumed to be roughly proportional to the supply voltage.

Selecting a suite of publicly available benchmark programs that everybody accepts as being representative of real-world workloads is difficult to begin with. Adopting a noncontroversial weighted mix is also not easy. For the commonly used SPEC benchmark suite (see http://www.specbench.org), the SPEC marks rating (for each class, e.g., integer or floating point) is derived as a geometric mean of execution time ratios for the programs within that class. Each ratio is calculated as the speedup with respect to execution time on a specified reference machine. An advantage of this method is that different machines can be ranked unambiguously from a performance viewpoint, if one believes in the particular benchmark suite. That is, the ranking can be shown to be independent of the reference machine used in such a formulation.

Even if one is able to fix the input workload mix to some known average characteristics, there is usually a large variation in workload behavior across different applications in the mix and in some cases, within even a single application. Thus, even though one can compute an average IPC (or $K_{\text{pf}}$ in Equation 3.2), it is possible to exploit the variations in IPC to reduce average power in architectures where the resources are dynamically adapted to match the IPC requirements. (See Section 3.3.1.6).

Let us now discuss the basics of power dissipation in a processor chip.

### 3.2.2 Power Fundamentals

At the elementary transistor gate (e.g., an inverter) level, total power dissipation can be formulated as the sum of three major components: switching loss, leakage, and short-circuit loss [2–5].

$$\text{Power}_{\text{device}} = (1/2) C V_{\text{dd}} V_{\text{swing}} af + I_{\text{leakage}} V_{\text{dd}} + I_{\text{sc}} V_{\text{dd}} \tag{3.3}$$

where

$C$ is the output capacitance

$V_{\text{dd}}$ is the supply voltage

$f$ is the chip clock frequency

$a$ is the activity factor ($0 < a \leq 1$), which determines the device switching frequency

$V_{\text{swing}}$ is the maximum voltage swing across the output capacitor, which in general can be less than $V_{\text{dd}}$

$I_{\text{leakage}}$ is the leakage current

$I_{\text{sc}}$ is the short-circuit current

In the literature, $V_{\text{swing}}$ is often approximated to be equal to $V_{\text{dd}}$ (or simply $V$ for short) making the switching loss $\sim (1/2) CV^2 af$. Also, as discussed in Ref. [3], for a prior generation range of $V_{\text{dd}}$ (say 1–3 V)

switching loss, $(1/2)CV^2af$, was the dominant component, assuming the activity factor to be above a reasonable minimum. So, as a first-order approximation, for the whole chip of the previous generation (e.g., CMOS 180 nm, and before) we may formulate the power dissipation to be

$$\text{Power}_{\text{chip}} = (1/2)\left[\sum_i C_i V_i^2 a_i f_i\right] \tag{3.4}$$

where, $C_i$, $V_i$, $a_i$, and $f_i$ are unit- or block-specific average values in the most general case; the summation is taken over all blocks or units $i$, at the microarchitecture level (e.g., icache, dcache, integer unit, floating-point unit (FPU), load-store unit, register files, and buses [if not included in individual units], etc). Also, for the voltage range considered, the operating frequency is roughly proportional to the supply voltage, and the capacitance $C$ remains roughly the same if we keep the same design but scale the voltage. If a single voltage and clock frequency are used for the whole chip, the above reduces to

$$\text{Power}_{\text{chip}} = V^3\left(\sum_i K_i^v a_i\right) = f^3\left(\sum_i K_i^f a_i\right) \tag{3.5}$$

If we consider the very worst-case activity factor for each unit $i$, i.e., if $a_i = 1$ for all $i$, then, an upper bound on the maximum chip power may be formulated as

$$\text{Max power}_{\text{chip}} = K_{\text{V}} V^3 = K_{\text{F}} f^3 \tag{3.6}$$

where $K_{\text{V}}$ and $K_{\text{F}}$ are design-specific constants. Note that an estimation of peak or maximum power is important, for the purposes of determining the packaging and cooling solution required. The larger the maximum power, the more expensive is the net cooling solution. Note also that the formulation in Equation 3.6 is overly conservative, as stated. In practice, it is possible to estimate the worst-case achievable maximum for the activity factors. This allows the designers to come up with a tighter bound on maximum power before the packaging decision is made.

The last equation (Equation 3.6) is what leads to the so-called cube root rule [3], where redesigning a chip to operate at half the voltage (and frequency) results in the power dissipation being lowered to $(1/2)^3$ or one-eighth the original. This implies the single-most efficient method for reducing power dissipation for a processor that has already been designed to operate at high frequency—reduce the voltage (and hence the frequency). There is a limit, however, of how low $V_{\text{dd}}$ can be reduced (for a given technology), which has to do with manufacturability and circuit reliability issues. Thus, a combination of microarchitecture and circuit techniques to reduce power consumption, without necessarily employing multiple or variable supply voltages is of special relevance in the design of robust systems.

In post-180 nm technologies, static (i.e., leakage or standby) power has increasingly become a major, if not the dominating, component of chip power. As discussed in Ref. [6], the three major types of leakage effects are (1) subthreshold, (2) gate, and (3) reverse-biased, drain- and source-substrate junction band-to-band tunneling (BTBT). With technology scaling, each of these leakage components tends to increase drastically. For example, as technology scales downward, the supply voltage ($V_{\text{dd}}$) must also scale down to reduce dynamic power and maintain device reliability. However, this requires the scaling down of the threshold voltage ($V_{\text{th}}$) to maintain reasonable gate overdrive and therefore, performance, which is a function of ($V_{\text{dd}} - V_{\text{th}}$). However, lowering the $V_{\text{th}}$ causes substantial increases in leakage current, and therefore, standby power, in spite of the lower $V_{\text{dd}}$. The subthreshold channel leakage current in an MOS device is governed by the following equation [7]:

$$I_{\text{leakage}} = K_{\text{w}} * W \times 10^{-V_{\text{th}}/S} \tag{3.7}$$

where

$K_w$, measured in units of microamps per micron ($\mu A/\mu m$) can be thought of as the width-sensitivity coefficient

$W$ is the device width

$S$ is the subthreshold swing, measured in mV like the $V_{th}$

In Ref. [7], the value of $K_w$ is quoted to be 10. $S$ is a parameter that is defined to characterize the efficiency of a device in turning on or off. It can be shown that the turn-off characteristic of a device is proportional to the thermal voltage ($kT/q$) and the ratio of junction capacitance ($C_j$) to oxide capacitance ($C_{ox}$) [8]. The parameter $S$ can be formulated as

$$S = 2.3(kT/q)(1 + C_j/C_{ox}) \tag{3.8}$$

This parameter is usually specified in units of millivolt (mV) per decade and it defines how many millivolts the gate voltage must drop before the drain current is reduced by one decade. The thermal voltage $kT/q$ is equal to 26 mV at room temperature. Thus, at room temperature, the minimum value of $S$ is about 60 mV per decade. This means that an ideal device at room temperature would experience a 10× reduction in drain current for every 60 mV reduction of the gate voltage $V_{gs}$ in the subthreshold region. In the deep submicron era, a typical transistor device has an $S$ value in the range of 85–90 mV per decade.

Also note that $V_{th}$ (Equation 3.7) is itself a function of temperature ($T$); in fact $V_{th}$ decreases by 2.5 mV/K as temperature increases. Also, $K_w$ itself is a strong function of temperature ($\sim T^2$). Thus, as $T$ increases, leakage current increases dramatically, both because of its dependence on $T$ and decrease in $V_{th}$. The delay of an inverter gate is given by the alpha-power model [9] as

$$T_g \sim \frac{L_{eff} V_{dd}}{\mu(T)(V_{dd} - V_{th})^{\alpha}} \tag{3.9}$$

where, $\alpha$ is typically around 1.3 and $\mu$ is the mobility of carriers (which is a function of temperature $T$, $\mu(T) \sim T^{-1.5}$). As $V_{th}$ decreases, $(V_{dd} - V_{th})$ increases so the inverter becomes faster. As $T$ increases, $(V_{dd} - V_{th})$ increases, but $\mu(T)$ decreases [10]. This latter effect dominates; so, with higher temperatures, the logic gates in a processor generally become slower.

### 3.2.3 Power–Performance Efficiency Metrics

The most common (and perhaps obvious) metric to characterize the power–performance efficiency of a microprocessor is a simple ratio, like mips/watt. This attempts to quantify the efficiency by projecting the performance achieved or gained (measured in millions of instructions per second or mips) for every watt of power consumed. Clearly, the higher the number, the better the machine is. Dimensionally, mips/watt equates to the inverse of the average energy consumed per instruction. This seems a reasonable choice for some domains where battery life is important. However, there are strong arguments against it in many cases, especially when it comes to characterizing higher end processors. Performance has typically been the key driver of such server-class designs and cost or efficiency issues have been of secondary importance. Specifically, a design team may well choose a higher frequency design point (which meets maximum power budget constraints) even if it operates at a much lower mips/watt efficiency compared to one that operates at better efficiency, but at a lower performance level. As such, $(\text{mips})^2$/watt or even $(\text{mips})^3$/watt may be the metric of choice at the high end. On the other hand, at the lowest end, where battery life (or energy consumption) is the primary driver, one may want to put an even greater weight on the power aspect than the simplest mips/watt metric; i.e., one may just be interested in minimizing the watts for a given workload run, irrespective of the execution time performance, provided the latter does not exceed some specified upper limit.

The mips metric for performance and the watts value for power may refer to average or peak values, derived from the chip specifications. For example, for a 1 GHz ($= 10^9$ cycles/s) processor, which can

complete up to four instructions per cycle, the theoretical peak performance is 4000 mips. If the average completion rate for a given workload mix is $p$ instructions per cycle, then the average mips would equal 1000 times $p$. However, when it comes to workload-driven evaluation and characterization of processors, metrics are often controversial. Apart from the problem of deciding on a representative set of benchmark applications, there are fundamental questions that persist about how to boil down performance into a single (average) rating that is meaningful in comparing a set of machines. Since power consumption varies, depending on the program being executed, the issue of benchmarking is also relevant in assigning an average power rating. In measuring power and performance together for a given program execution, one may use a fused metric like power-delay product (PDP) or energy-delay product (EDP) [5,11]. In general, the PDP-based formulations are more appropriate for low-power, portable systems, where battery life is the primary index of energy efficiency. The mips/watt metric is an inverse PDP formulation, where delay refers to average execution time per instruction. The PDP, being dimensionally equal to energy, is the natural metric for such systems. For higher end systems (e.g., workstations), the EDP-based formulations are deemed to be more appropriate, since the extra delay factor ensures a greater emphasis on performance. The $(\text{mips})^2$/watt metric is an inverse EDP formulation. For the highest performance, server-class machines, it may be appropriate to weight the delay part even more. This would point to the use of $(\text{mips})^3$/watt, which is an inverse $ED^2P$ formulation. Alternatively, one may use $(\text{cpi})^3 \cdot \text{watt}$ as a direct $ED^2P$ metric, applicable on a "per instruction" basis (see [2]).

The energy*$(\text{delay})^2$ metric, or $\text{perf}^3$/power formula, is analogous to the cube root rule [3], which follows from constant voltage scaling arguments (see previous discussion, Equation 3.6). Clearly, to formulate a voltage-invariant power–performance characterization metric, we need to think in terms of $\text{perf}^3$/power. When we are dealing with the SPEC benchmarks, one may therefore evaluate efficiency as $(\text{SPECrating})^x$/watt, or $(\text{SPEC})^x$/watt for short, where the exponent value $x$ (= 1, 2, or 3) may depend on the class of processors being compared.

Brooks et al. [2] discuss the power–performance efficiency data for a range of commercial processors of approximately the same generation (circa year 2000). SPEC/watt, $\text{SPEC}^2$/watt, and $\text{SPEC}^3$/watt are used as the alternative metrics, where SPEC stands for the processor's SPEC rating (see definition principles, in Section 3.2.1 or in Ref. [1]). The data validates our assertion that depending on the metric of choice and the target market (determined by workload class and the power/cost), the conclusion drawn about efficiency can be quite different. For performance-optimized, high-end processors, the $\text{SPEC}^3$/watt metric seems to be fairest. For power-first processors targeted toward integer workloads, SPEC/watt seems to be the fairest.

## 3.3 A Review of Key Ideas in Power-Aware Microarchitectures

In our review of power-efficient design concepts at the microarchitecture level, our primary attention will be on dynamic (also known as active or switching) power governed by the formula, $CV^2af$. Recall that $C$ refers to the switching capacitance, $V$ is the supply voltage, $a$ is the activity factor ($0 < a < 1$), and $f$ is the operating clock frequency. Power reduction ideas must therefore focus on one or more of these basic parameters. Reducing active power generally results in reduction of on-chip temperatures, and this indirectly causes leakage power to go down as well. Similarly, any increase in efficiency directed at lowering the latch count (e.g., by reducing the basic pipeline depth, or by reducing the number of back-end execution pipes within a given functional unit) also results in area and leakage reduction as a side benefit. However, later in this section, we also deal with the problem of mitigating leakage power directly, by providing microarchitectural support to what are primarily circuit-level mechanisms.

### 3.3.1 Power Efficiency at the Processor Core Level

In this section, we examine the key ideas that have been proposed in terms of microarchitectural support for power efficiency, at the level of a single processor core.

The effective (average) value of $C$ can be reduced by using (1) area-efficient designs for various macros; (2) adaptive structures, which change in effective size, latency, or communication bandwidth depending on the needs of the input workload, (3) selectively "gating off" the clock for unused or idle units; (4) reducing or eliminating "speculative waste" resulting from executing instructions in mis-speculated branch paths or prefetching useless instructions and data into caches, based on wrong guesses.

The average value of $V$ can be reduced via dynamic voltage scaling, i.e., by reducing the voltage as and when required or possible. Microarchitectural support, in this case, is not required, unless the mechanisms to detect "idle" periods or temperature overruns are detected using counter-based "proxies," specially architected for this purpose. Hence, in this chapter, we do not dwell on dynamic voltage scaling methods. (Note again, however, that since reducing $V$ also requires (or results in) reduction of the operating frequency, $f$, net power reduction has a cubic effect; thus, dynamic voltage and frequency scaling (DVFS), though arguably not a microarchitectural technique per se, is the most effective way of power reduction). Deciding when and how to apply DVFS, as a function of the input workload characteristics and overall operating environment, on the other hand, is very much a microarchitectural issue. It is a problem that is increasingly relevant in the era of variability-tolerant, power-efficient multicore chip design, described briefly in Section 3.4.

The average value of the activity factor, $a$, can be reduced by (1) the use of clock-gating, where the normally free-running, synchronous clock is disabled in selected units or subunits within the system based on information or predictions about current or future activity in those regions; (2) the use of data representations and instruction schedules that result in reduced switching. Microarchitectural support is provided in the form of added mechanisms to (1) detect, predict, and control the generation of the applied gating signals or (2) aid in power-efficient data and instruction encodings. Compiler support for generating power-efficient instruction scheduling and data partitioning or special instructions for "nap/doze/sleep" control, if applicable, must also be considered under this category.

While clock-gating helps eliminate (or drastically reduce) active or switching power when a given macro, subunit, or unit is idle, power-gating can be used to also eliminate the residual leakage power of that idle entity. In this case, as described in detail later on, the power supply voltage $V$ is itself gated off from the target circuit block, with the help of a header or footer transistor. Here, the need for microarchitectural support in the form of predictive control of the gating signal is even stronger because of the relatively large performance overheads that would be incurred without such support. There are other techniques like adaptive body biasing that are also targeted at leakage power control, and these too require some degree of microarchitectural support. However, these techniques are most relevant to bulk-CMOS designs (as opposed to SOI–CMOS technology), and are predominantly device- and circuit-level methods. As such, we do not dwell on them in this chapter.

Lastly, the average value of the design frequency, $f$, can be controlled or reduced by using (1) variable, multiple, or locally asynchronous (self-timed) clocks, e.g., in GALS [12] designs; (2) clock throttling, where the frequency is reduced dynamically in response to power or temperature overrun indicators; or (3) reduced pipeline depths in the baseline microarchitecture definition.

We consider power-aware microarchitectural constructs that use $C$, $a$, or $f$ as the primary lever for reducing active power; and those that use the supply voltage $V$ as the lever for reducing leakage power. In any such proposed processor architecture, the efficacy of the particular power reduction method that is used must be assessed by understanding the net performance impact. Here, depending on the application domain (or market), a PDP, EDP, or $ED^2P$ metric for evaluating and comparing power–performance efficiencies must be used. (See earlier discussion in Section 3.2).

#### 3.3.1.1 Optimal Pipeline Depth

A fundamental question that is asked has to do with pipeline depth. Is a deeply pipelined, high frequency (speed demon) design better than an IPC-centric low frequency (braniac) design? In the context of the topic of this chapter, "better" must be judged in terms of power–performance efficiency.

Let us consider, first, a simple, hazard-free, linear pipeline flow process, with $k$ stages. Let the time for the total logic (without latches) to compute one answer be $T$. Assuming that the $k$ stages into which the

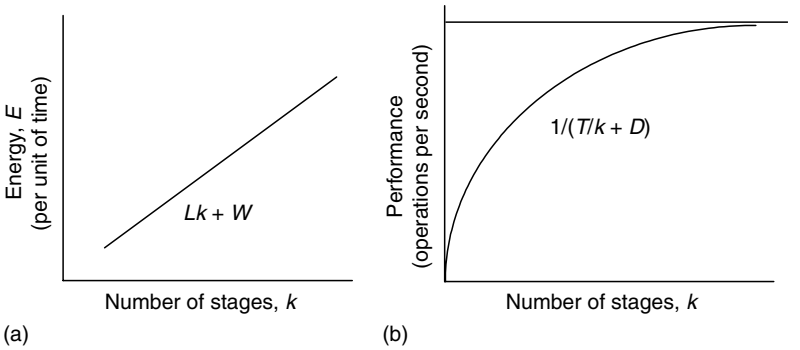



**FIGURE 3.2** Power and performance curves for idealized pipeline flow.

logic is partitioned are of equal delay, the time per stage and thus the time per computation becomes (see Chapter 2 in Ref. [13])

$$t = T/k + D \tag{3.10}$$

where $D$ is the delay added due to the staging latch. The inverse of $t$ determines the clocking rate or frequency of operation. Similarly, if the energy spent (per cycle, per second, or over the duration of the program run) in the logic is $W$ and the corresponding energy spent per level of staging latches is $L$ then the total energy equation for the $k$-stage pipelined version is roughly given as the follows:

$$E = Lk + W \tag{3.11}$$

The energy equation assumes that the clock is free-running; i.e., on every cycle, each level of staging latches is clocked to enable the advancement of operations along the pipeline. (Later, we shall consider the effect of clock-gating.) Equations 3.10 and 3.11, when plotted as a function of $k$, are depicted in Figure 3.2a and b, respectively.

As the number of stages increases, the energy or power consumed increases linearly; while, the performance also increases, but not as fast. In order to consider the PDP-based power–performance efficiency, we compute the ratio

$$\begin{aligned}\frac{\text{Power}}{\text{Performance}} &= (Lk + W)(T/k + D) \\ &= LT + WD + (LDk^2 \\ &\quad + WT)/k\end{aligned} \tag{3.12}$$

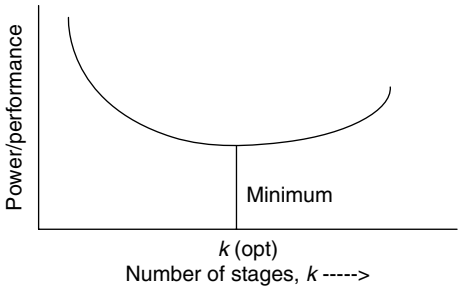



**FIGURE 3.3** Power performance ratio curve for idealized pipeline flow.

Figure 3.3 shows the general shape of this curve as a function of $k$. Differentiating the right-hand side expression in Equation 3.12 and setting it to zero, one can solve for the optimum value of $k$ for which the power–performance efficiency is maximized; i.e., the minimum of the curve in Figure 3.2b can be shown to occur when

$$k(\text{opt}) = \sqrt{(WT)/(LD)} \tag{3.13}$$

Larson [14] first published the above analysis, albeit from a cost/performance perspective. This analysis shows that, at least, for the simplest, hazard-free pipeline flow, the highest frequency operating point achievable in a given technology may not be the most energy-efficient. Rather, the optimal number of stages (and hence operating frequency) is expected to be at a point which increases for greater $W$ or $T$ and decreases for greater $L$ or $D$. For a prior generation POWER4-class ($\sim$0.18 $\mu$) superscalar processor operating at around 1 GHz [15], the floating-point arithmetic unit is estimated to yield values of $T = 7.5$ ns, $D = 0.15$ ns, $W = 0.15$ W, and $L = 0.1$ W. This yields a $k$(opt) $\sim$8 (rounded off from 8.67), if we use the idealized formalism (Equation 3.13).

For real superscalar machines, the number of latches in the overall design tends to go up much more sharply with $k$ than the linear assumption in the above model. This tends to make $k$(opt) even smaller. Also, in real pipeline flow with hazards, e.g., in the presence of branch-related stalls and disruptions, performance actually peaks at a certain value of $k$ before decreasing [3,16,17] (instead of the asymptotically increasing behavior shown in Figure 3.2b). This effect would also lead to decreasing the effective value of $k$(opt). (However, $k$(opt) increases if we use EDP or $ED^2P$ metrics instead of the PDP metric used.). In a detailed simulation-based analysis of a POWER4-class superscalar machine, it has been shown [18,19] that the optimal pipeline depth using a $ED^2P$ metric like $(BIPS)^3$/watt (where BIPS is the standard performance metric of billions of instructions completed per second) is around 18 FO4* per pipe stage for SPEC2000 workloads. For commercial workloads like TPC-C, the optimal point is shown to shift to shallower pipelines (25–28 FO4). In contrast, note that if one considered a power-unaware performance-only metric, like BIPS, the optimal pipeline depth for SPEC2000 is around 10 FO4 per stage. For TPC-C, the performance-only optimal point is reported to be pretty flat across the 10–14 FO4 points.

### 3.3.1.2 Vector/SIMD Processing Support

Vector/SIMD modes of parallelism present in current architectures afford a power-efficient method of extending performance for vectorizable codes. Fundamentally, this is because for doing the work offetching and processing a single (vector) instruction, a large amount of data is processed in a parallel or pipelined manner. If we consider an SIMD machine, with $p$ $k$-stage functional pipelines (see Figure 3.4) then looking at the pipelines alone, one sees a $p$-fold increase of performance, with a $p$-fold increase in power, assuming full utilization and hazard-free flow, as before. Thus, an SIMD pipeline unit offers the potential of scalable growth in performance, with commensurate growth in power; i.e., at constant power–performance efficiency. If, however, one includes the front-end instruction cache and fetch/dispatch unit that are shared across the $p$ SIMD pipelines, then power–performance efficiency can actually grow with p. This is because, the energy behavior of the instruction cache (memory) and the fetch/decode path remains essentially invariant with $p$, while net performance grows linearly with $p$.

In a superscalar machine with a vector/SIMD extension, the overall power-efficiency increase is limited by the fraction of code that runs in vector/SIMD-mode (per Amdahl's law).

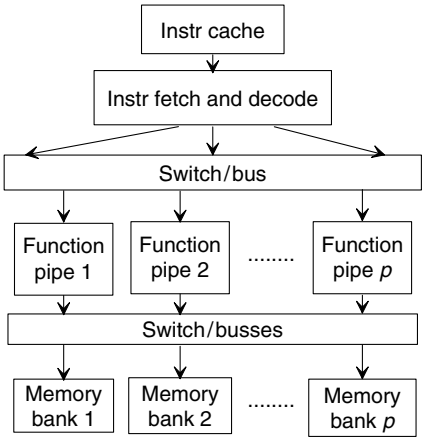



**FIGURE 3.4** Parallel SIMD architecture.

*Fan-out-of-four (FO4) delay is defined as the delay of one inverter driving four copies of an equally sized inverter. The amount of logic and latch overhead per pipeline stage is often measured in terms of FO4 delay.

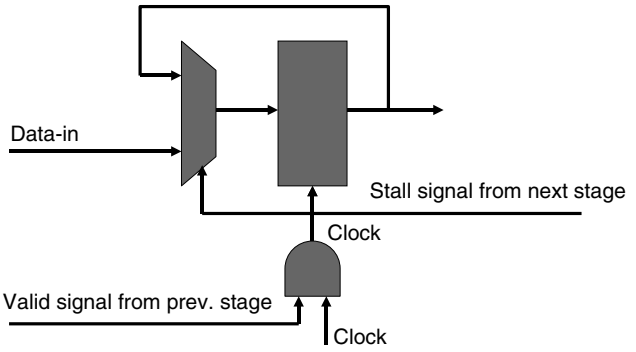



**FIGURE 3.5** Clock gating mechanism.

### 3.3.1.3 Clock-Gating: Power Reduction Potential and Microarchitectural Support

Clock-gating refers to circuit-level control [21,22] for disabling the clock to a given set of latches, a macro, a bus, to a cache or register file access path, or an entire unit on a particular machine cycle.

Figure 3.5 depicts a typical clocking arrangement used in pipelined dataflow logic within a high-end microprocessor. The bank of latches is clocked via an AND gate that is enabled by a valid-bit signal from the previous pipeline stage. A stall-bit from the next pipeline stage is used to recirculate the current data during a pipeline stall. The latches are clocked only when there is valid data available from the previous stage or when the data needs to be held. In alternate designs, the stall-bit can also be used to gate the clock to further improve clock-gating efficiency.

In current generation server-class microprocessors, about 70% of the active (switching) power is consumed by the clock distribution network and its latch load alone. As reported in Ref. [23], the major part of the clock power is dissipated close to the leaf nodes of the clock tree that drive latch banks. Since a clock-gated latch keeps its current data value stable, clock-gating prevents signal transitions of invalid data from propagating down the pipeline thereby reducing switching power in the combinational logic between latches. In addition to reducing dynamic power, clock-gating can also reduce static (leakage) power. As already explained, leakage current in CMOS devices is exponentially dependent on temperature. The temperature reduction brought on by clock-gating can therefore significantly reduce the leakage power as well.

We define clock-gating efficiency (CGE) for a given input workload as follows: CGE $-$ [1 $-$ (average clock-gated power)/(maximum unconstrained power)] $\times$ 100%, so that higher numbers imply greater levels of power reduction.

Figure 4 in Ref. [24] shows the computed CGE across various workloads (SPEC2000 suite) using the Turandot/PowerTimer [23] power–performance simulator. Such microarchitecture-level analysis points to opportunities of power savings in a processor, since idle periods of a particular resource (e.g., a pipeline stage) can be identified and quantified.

Figure 3.6a and b show the opportunities available within several units (and in particular, the instruction fetch unit, IFU) of the same example processor in the context of the TPC-C trace segment referred to above. Figure 3.6a depicts the instruction frequency mix of the trace segment used. This shows that the FPU operations are a very tiny fraction of the total number of instructions in the trace. Therefore, with proper detection and control mechanisms architected in hardware, the FPU could essentially be "gated off" in terms of the clock delivery for the most part of such an execution. Figure 3.6b shows the fraction of total cycles spent in various modes within the IFU. I-fetch was on hold for about 48% of the cycles, and the fraction of useful fetch cycles was only 28%. Again, this points to great opportunities either in terms of clock-gating or dynamic ifetch throttling (see Section 3.3.1.8).

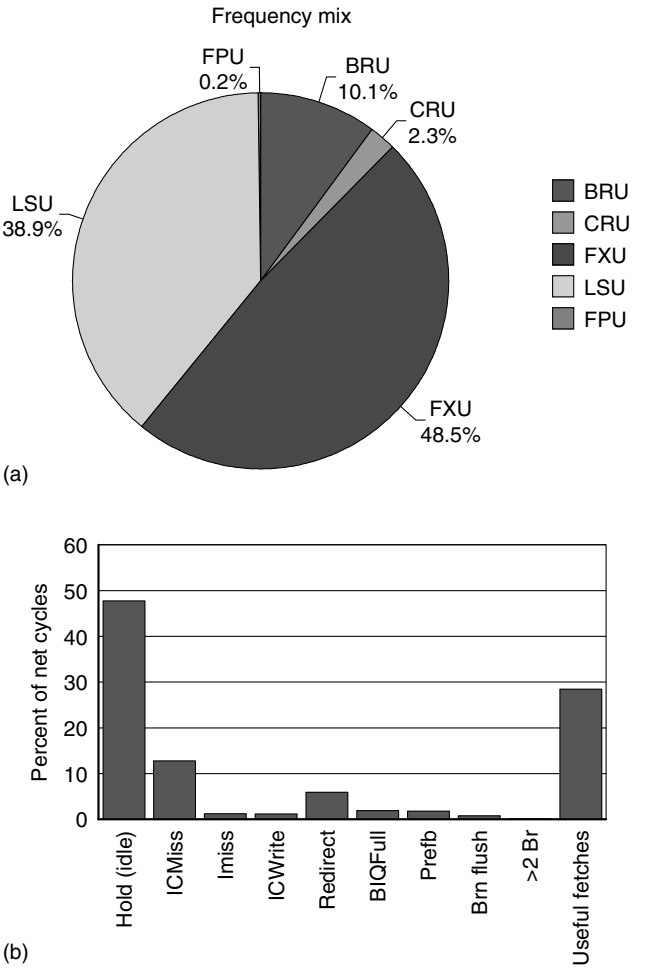



**FIGURE 3.6** (a) Instruction frequency mix for a typical commercial workload trace segment. (b) Stall profile in the instruction fetch unit (IFU) for the commercial workload.

Microarchitectural support for conventional clock-gating can be provided in at least three ways: (1) dynamic detection of idle modes in various clocked units or regions within a processor or system, (2) static or dynamic prediction of such idle modes, and (3) using "data valid" bits within a pipeline flow path to selectively enable/disable the clock applied to the pipeline stage latches. If static prediction is used, the compiler inserts special nap/doze/sleep/wake-type instructions where appropriate, to aid the hardware in generating the necessary gating signals. Methods 1 and 2 result in coarse-grain clock-gating, where entire units, macros, or regions can be gated off to save power, whereas method 3 results in fine-grain clock-gating, where unutilized pipe segments can be gated off during normal execution within a particular unit, like the FPU. The detailed circuit-level implementation of gated-clocks, the potential performance degradation, inductive noise problems, etc., are not discussed in this chapter. However, these are very important issues that must be dealt with adequately in an actual design.

Referring back to Figures 3.2 and 3.3, note that since (fine-grain) clock-gating effectively causes a fraction of the latches to be gated off, we may model this by assuming that the effective value of $L$ decreases when such clock-gating is applied. This has the effect of increasing $k$ (opt); i.e., the operating frequency for the most power-efficient pipeline operation can be increased in the presence of clock-gating. This is an added benefit.

In a recently reported work [24], the limits of CGE has been examined and then stretched by adding a couple of new advances: transparent pipeline clock-gating (TCG) [25] and elastic pipeline clock-gating (ECG) [26]. TCG introduces a new way of clock-gating pipelines. In traditional clock-gating, latches are held opaque to avoid data races between adjacent latch stages; this $N$ clock pulses are needed to propagate a single data item through an $N$-stage pipeline, even if at a given clock cycle all other (i.e., $N-1$) stages have invalid input data. In a transparent clock-gated pipeline, latches are held transparent by default. TCG is based on the concept of data separation. Assume that a pair of data items A and B simultaneously moves through a TCG pipeline. A data race between A and B is avoided by separating the two data items by clocking or gating a latch stage opaque, such that the opaque latch stage acts as a barrier separating the two data items from each other. The number of clock pulses required for a data item item A to move through an $N$-stage pipeline is no longer only dependent on $N$, but also on the number of clock cycles that separate A from the closest upstream data item B. For an $N$-stage pipeline, where B follows $n$ clock cycles behind A, only floor $(N/n)$ clock pulses have to be generated to move A safely through the pipeline. ECG is a different technique that achieves further efficiency by exploiting the inherent storage redundancy afforded by a traditional master–slave latch pair. ECG allows the designer to allow stall signals to propagate backward in pipeline flow logic in a stage-by-stage fashion, without incurring the leakage power and area overhead of explicitly inserted stall buffers. Logic-level details of TCG and ECG are available in the originally published papers [24–26]. As reported there, TCG enables clock power reduction to the tune of 50% over traditional stage-level clock-gating under commercial (TPC-C) class workloads. Even under heavy floating-point workloads, where fewer bubbles are available in the pipeline, the clock power in the floating-point pipeline can be reduced by 34%. The significant reduction in dynamic stall power (27%) and leakage power (44%) afforded by ECG in a FPU design have also been reported in the published literature.

#### 3.3.1.4 Predictive Power-Gating

As previously indicated, leakage power is a major (if not dominant) component of total power dissipation in current and future CMOS microprocessors. Cutting off the power supply ($V_{dd}$) to major circuit blocks to conserve idle power (sleep mode) is not a new concept, especially for battery-powered mobile systems. However, dynamically effecting such gating, on a unit-by-unit basis, as function of input workload demand is not a design technique that has seen widespread usage yet, especially in server-class microprocessors. The main reasons have been the perceived risks or negative effects arising from (1) performance and area overheads, (2) inductive noise on the power supply grid, and (3) potential design tools and verification concerns. Advances in circuit design have minimized the area and cycle-time delay overhead concerns in recent industrial practices. Microarchitectural predictive techniques [27–29] have recently been so perfected that now they equip designers with the tools needed to minimize any architectural performance overheads as well. The inductive noise concerns do persist, but there are known solution approaches in the realm of power distribution networks and package design that will no doubt mature to help mitigate those concerns. The design tools and verification challenge, of course will prevail as a difficult roadblock—but again, solutions will eventually emerge to get rid of that concern.

#### 3.3.1.5 Variable Bit-Width Operands

One of the techniques proposed for reducing dynamic power consists of exploiting the behavior of data in programs, which is characterized by the frequent presence of small values. Such values can be represented as and operated upon as short bit-vectors. Thus, by using only a small part of the processing datapath, power can be reduced without loss of performance. Brooks and Martonosi [30] analyzed the potential of this approach in the context of 64-bit processor implementations (e.g., the Compaq Alpha architecture). Their results show that roughly 50% of the instructions executed had both operands whose length was less than or equal to 16 bits. Brooks and Martonosi proposed an implementation that exploits this by dynamically detecting the presence of narrow-width operands on a cycle-by-cycle basis.

#### 3.3.1.6 Adaptive Microarchitectures

Another method of reducing power is to adjust the size of various storage resources within a processor or system, with changing the needs of the workload. Albonesi [31] proposed a dynamically reconfigurable caching mechanism that reduces the cache size (and hence power) when the workload is in a phase that exhibits reduced cache footprint. Such downsizing also results in improved latency, which can be exploited (from a performance viewpoint) by increasing the cache cycling frequency on a local clocking or self-timed basis. Maro et al. [32] have suggested the use of adapting the functional unit configuration within a processor in tune with changing workload requirements. Reconfiguration is limited to shutting down certain functional pipes or clusters, based on utilization history or IPC performance. In that sense, the work by Maro et al. is not too different from coarse-grain clock-gating support, as discussed earlier. In early work done at IBM Watson, Buyuktosunoglu et al. [33] designed an adaptive issue queue that can result in (up to) 75% power reduction when the queue is sized down to its minimum size. This is achieved with a very small IPC performance hit. Another example is the idea of adaptive register files (see Ref. [34]) where the size and configuration of the active size of the storage is changed via a banked design, or through hierarchical partitioning techniques. A recent tutorial article by Albonesi et al. [35] provides an excellent coverage of advances in the field of adaptive architectures.

#### 3.3.1.7 Dynamic Thermal Management

Most clock-gating techniques are geared toward the goal of reducing average chip power. As such, these methods do not guarantee that the worst-case (maximum) power consumption will not exceed safe limits. The processor's maximum power consumption dictates the choice of its packaging and cooling solution. In fact, as discussed in Ref. [22], the net cooling solution cost increases in a piecewise linear manner with respect to the maximum power, and the cost gradient increases rather sharply in the higher power regimes. This necessitates the use of mechanisms to limit the maximum power to a controllable ceiling, defined by the cost profile of the market for which the processor is targeted. Most recently, in the high-performance world, Intel's Pentium 4 processor is reported to use an elaborate on-chip thermal management system to ensure reliable operation [22]. At the lower end, the G3 and G4 PowerPC microprocessors [36,37] include a thermal assist unit (TAU) to provide dynamic thermal management. In a recently reported academic work, Brooks and Martonosi [38] discuss and analyze the potential reduction in maximum power ratings without significant loss of performance, by the use of specific dynamic thermal management (DTM) schemes. The use of DTM requires the inclusion of on-chip sensors to monitor actual temperature, or proxies of temperature [38] estimated from on-chip counters of various events and rates.

#### 3.3.1.8 Dynamic Throttling of Communication Bandwidths

This idea has to do with reducing the width of a communication bus dynamically, in response to reduced needs or in response to temperature overruns. Examples of on-chip buses that can be throttled are instruction fetch bandwidth, instruction dispatch/issue bandwidths, register renaming bandwidth, instruction completion bandwidths, memory address bandwidth, etc. In the G3 and G4 PowerPC microprocessors [36,37], the TAU invokes a form of instruction cache throttling as a means to lower the temperature when a thermal emergency is encountered.

#### 3.3.1.9 Speculation Control

In current generation, high-performance microprocessors, branch mispredictions, and mis-speculative prefetches end up wasting a lot of power. Manne et al. [39,40] have described means of detecting or anticipating an impending mispredict and using that information to prevent mis-speculated instructions from entering the pipeline. These methods have been shown to reduce power by up to 38% with less than a 1% performance loss.

## 3.4 Power-Efficient Microarchitecture Paradigms

Now that we have examined specific microarchitectural constructs that aid power-efficient design, let us examine the inherent power–performance scalability and efficiency of selected paradigms that are currently established or are emerging in the high-end processor roadmap. In particular, we consider (1) wide-issue, speculative superscalar processors, (2) multicluster superscalars, (3) SMT processors, and (4) chip multiprocessors (CMPs). those that use single program speculative multithreading, as well as those that are general multicore symmetric multiprocessing (SMP) or throughput engines.

In illustrating the efficiency advantages or deficiencies, we use the following running example. It shows one iteration of a loop trace that we consider in simulating the performance and power characteristics across the above computing platforms.

Let us consider the following floating-point loop kernel, shown below (coded using the PowerPC instruction set architecture):

*Example loop test case*

```
[P]  [A] fadd fp3, fp1, fp0
[Q]  [B] lfdu fp5, 8(r1)
[R]  [C] lfdu fp4, 8(r3)
[S]  [D] fadd fp4, fp5, fp4
[T]  [E] fadd fp1, fp4, fp3
[U]  [F] stfdu fp1, 8(r2)
[V]  [G] bc loop_top
```

The loop body consists of seven instructions, the final one being a conditional branch that causes control to loop back to the top of the loop body. The instructions are labeled A through G. (The labels P through V are used to tag the corresponding instructions for a parallel thread—when we consider SMT and CMP). The lfdu/stfdu instructions are load/store instructions with update, where the base address register (e.g., r1, r2, or r3) is updated after execution by holding the newly computed address.

### 3.4.1 Single-Core Superscalar Processor Paradigm

One school of thought anticipates a continued progression along the path of wider, aggressively superscalar paradigms. Researchers continue to innovate in an attempt to extract the last ounce of IPC-level performance from a single-thread instruction-level parallelism (ILP) model. Value prediction advances (pioneered by Lipasti et al. in Ref. [41]) promise to break the limits imposed by true data dependencies. Trace caches (Smith et al. in Ref. [41]) ease the fetch bandwidth bottleneck, which can otherwise impede scalability. However, increasing the superscalar width beyond a certain limit tends to yield diminishing gains in net performance (i.e., the inverse of CPI * CT; see Equation 3.1). At the same time, the power–performance efficiency metric (e.g., performance per watt or $(\text{performance})^2$ per watt, etc.) tends to degrade beyond a certain complexity point in the single-core superscalar design paradigm. This is illustrated below in the context of our example loop test case.

Let us consider a base machine that is a four-wide superscalar, with two load-store units supporting two floating-point pipes (see Figure 3.7). The data cache has two load ports and a separate store port. Two load-store unit pipes (LSU0 and LSU1) are fed by a single issue queue, LSQ; similarly, the two FPU unit pipes (FPU0 and FPU1) are fed by a single issue queue, FPQ. In the context of the loop above, we essentially focus on the LSU–FPU subengine of the whole processor.

Let us assume the following high-level parameters (latency and bandwidth) characterizing the base superscalar machine model of width $W = 4$:

- Instruction fetch bandwidth, fetch_bw $= 2 * W = 8$ instructions per cycle
- Dispatch/decode/rename bandwidth, disp_bw $= W = 4$ instructions per cycle; dispatch is assumed to stall beyond the first branch scanned in the instruction fetch buffer

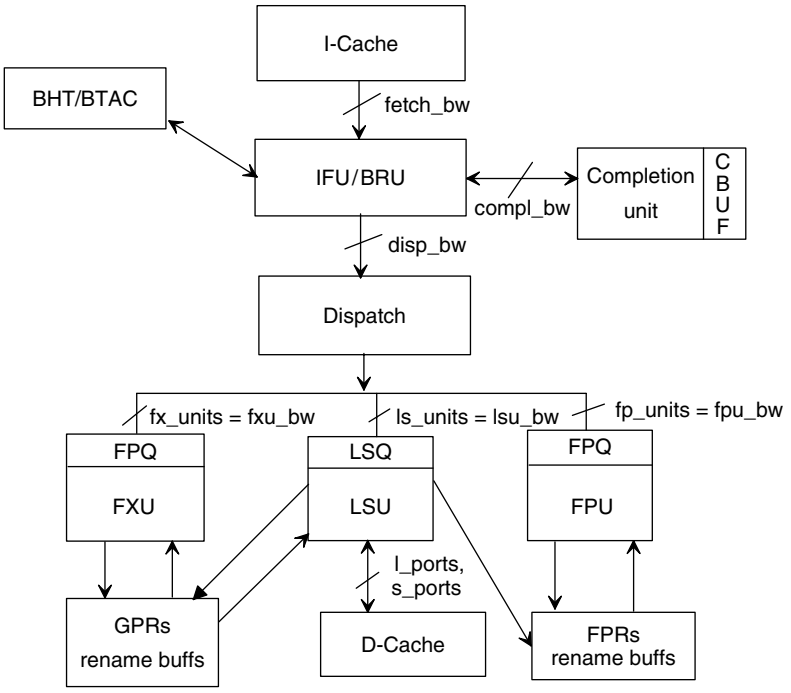



**FIGURE 3.7** High-level block-diagram of machine organization modeled in the eliot/elpaso tool.

- Issue_bandwidth from LSQ (reservation station), lsu_bw = $W/2 = 2$ instructions per cycle
- Issue_bandwidth from FPQ, fpu_bw = $W/2 = 2$ instructions per cycle
- Completion bandwidth, compl_bw = $W = 4$ instructions per cycle
- Back-to-back-dependent floating-point operation issue delay, fp_delay = 1 cycle
- The best-case load latency, from fetch to writeback = 5 cycles
- The best-case store latency, from fetch to writing in the pending store queue = 4 cycles; (a store is eligible to complete the cycle after the address–data pair is valid in the store queue)
- The best-case floating-point operation latency, from fetch to writeback = 7 cycles (when the issue queue, FPQ is bypassed, because it is empty)

Loads and floating-point operations are eligible for completion (retirement) of the cycle after writeback into rename buffers. For simplicity of analysis let us assume that the processor uses in-order issue from the issue queues (LSQ and FPQ). In our simulation model, the superscalar width $W$ is a ganged parameter, defined as follows:

$W$ = (fetch_bw/2) = disp_bw = compl_bw.

The number of LSU units, ls_units, FPU units, fp_units, data cache load ports, l_ports, and data cache store ports are varied as follows as $W$ is changed:

ls_units = fp_units = l_ports = max [floor($W/2$), 1]
s_ports = max [floor(l_ports/2), 1].

For illustrative purposes, a simple (and decidedly naive) analytical energy model is assumed, where the power consumed is a function of the following parameters: $W$, ls_units, fp_units, l_ports, and s_ports. In particular, the power, PW, in watts is computed as PW = $K * [(W)^y + \text{ls\_units} + \text{fp\_units} + \text{l\_ports} + \text{s\_ports}]$, where $y$ $(0 < y < 1)$ is an exponent that may be varied to see the effect on power–performance efficiency and $K$ is a constant. Figure 3.8 shows the performance and performance/power

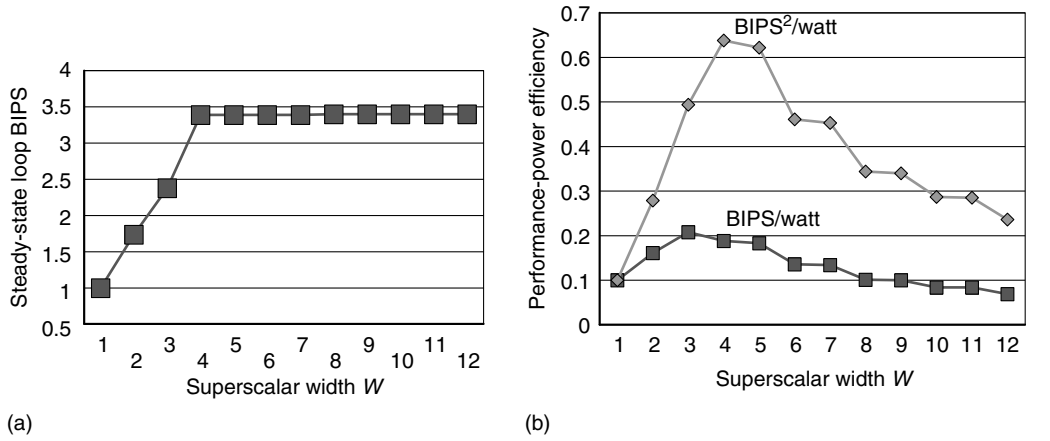



**FIGURE 3.8** Loop performance and performance/power variation with issue width.

ratio variation with superscalar width, $W$; for this graph, $y$ has been set to 0.5 and the scaling constant $K$ is 2. The BIPS values are computed from the IPC (instruction per cycle) values, assuming a clock frequency of 1 GHz.

The graph in Figure 3.8a shows that a maximum issue width of $W = 4$ could be used to achieve the best (idealized) BIPS performance. This idealized plot is obtained using a tool called eliot [42]. This is a parameterized, PowerPC superscalar model, that can operate either in cycle-by-cycle simulation mode, or, it can generate idealized bounds, based on static analysis of a loop code segment. The eliot model has now been updated to include parameterized, analytical energy models for each unit or storage resource within the processor. This new tool, called elpaso can be used to generate power–performance efficiency data for loop test cases. As shown in Figure 3.8b, from a power–performance efficiency viewpoint (measured as a performance over power ratio), the best-case design is achieved for $W < 4$. Depending on the sophistication and accuracy of the energy model (i.e., how power varies with microarchitectural complexity), and the exact choice of the power–performance efficiency metric, the inflexion point in the curve in Figure 3.9b changes; however, it should be obvious that beyond a certain superscalar width, the power–performance efficiency diminishes continuously. Fundamentally, this is because of the single-thread ILP limit of the loop trace being considered (as apparent from Figure 3.8a).

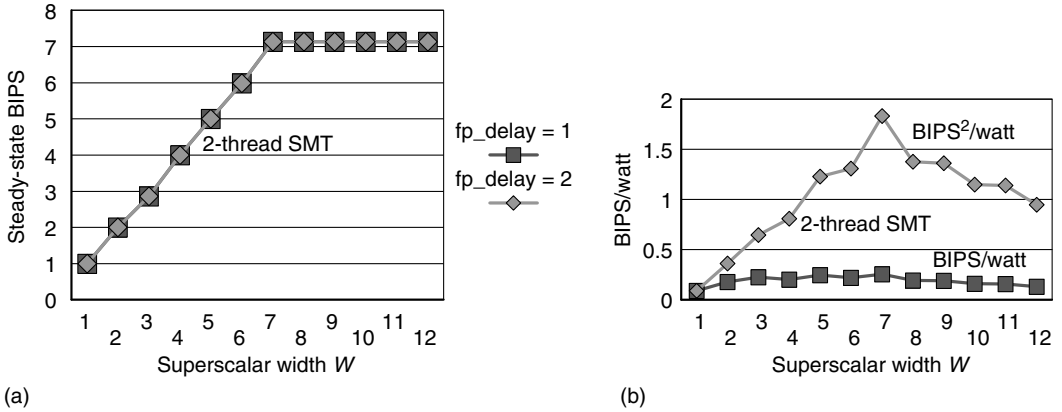



**FIGURE 3.9** Performance and power–performance variation width $W$ for 2-thread symmetric multiprocessing (SMT).

Note, by the way, that the resource sizes are assumed to be large enough, so that they are effectively infinite for the purposes of our running example above. Some of the actual sizes assumed for the base case ($W = 4$) are as follows: completion (reorder) buffer size, cbuf_size = 32; load-store queue size, lsq_size = 6; floating-point queue size, fpq_size = 8; pending store queue size, psq_size = 16.

The microarchitectural trends beyond the current superscalar regime are effectively targeted toward the goal of extending the power–performance efficiency factors. That is, the complexity growth must ideally scale at a slower rate than the growth in performance. Power consumption is one index of complexity; it also determines packaging and cooling costs. (Verification cost and effort is another important index). In that sense, a microarchitecture paradigm that ensures that the power–performance efficiency measure of choice is a nondecreasing function of time is the ideal, complexity-effective design paradigm for the future. Of course, it is hard to keep scaling a given paradigm beyond a few processor generations. Whenever we reach a maximum in the power–performance efficiency curve, it is time to invoke the next paradigm shift.

Next, we examine some of the promising new trends in microarchitecture that can serve as the next platform for designing power–performance scalable machines.

### 3.4.2 Multicluster Superscalar Processors

As described in our earlier article [2], Zyuban et al. [43,44] studied the class of multicluster superscalar processors as a means of extending the power-efficient growth of the basic superscalar paradigm. One way to address the energy growth problem at the microarchitectural level is to replace a classical superscalar CPU with a set of clusters, so that all key energy consumers are split among clusters. Then, instead of accessing centralized structures in the traditional superscalar design, instructions scheduled to an individual cluster would access local structures most of the time. The main advantage of accessing a collection of local structures instead of a centralized one is that the number of ports and entries in each local structure is much smaller. This reduces the latency and energy per access. If the nonaccessed substructures of a resource can be gated off (e.g., in terms of the clock), then the net energy savings can be substantial.

According to the results obtained in Zyuban's work, the energy dissipated per cycle in every unit or subunit within a superscalar processor can be modeled to vary (approximately) as $\text{IPC}_{\text{unit}} * (\text{IW})^g$, where IW is the issue width, $\text{IPC}_{\text{unit}}$ is the average IPC performance at the level of the unit or structure under consideration, and $g$ is the energy growth parameter for that unit. Then, EDP for the particular unit would vary as

$$\text{EDP}_{\text{unit}} = \frac{\text{IPC}_{\text{unit}} * (\text{IW})^g}{\text{IPC}_{\text{overall}}} \tag{3.14}$$

Zyuban shows that for real machines, where the overall IPC always increases with issue width in a sublinear manner, the overall EDP of the processor can be bounded as

$$(\text{IPC})^{2g-1} \leq \text{EDP} \leq (\text{IPC})^{2g} \tag{3.15}$$

where $g$ is the energy-growth factor of a given unit and IPC refers to the overall IPC of the processor that is assumed to vary as $(\text{IW})^{0.5}$. Thus, according to this formulation, superscalar implementations that minimize $g$ for each unit or structure will result in energy-efficient designs. The eliot/elpaso tool does not model the effects of multiclustering in detail; however, from Zyuban's work, we can infer that a carefully designed multicluster architecture has the potential of extending the power–performance efficiency scaling beyond what is possible using the classical superscalar paradigm. Of course, such extended scalability is achieved at the expense of reduced IPC performance for a given superscalar machine width. This IPC degradation is caused by the added intercluster communication delays and other power

management overhead in a real design. Some of the IPC loss (if not all) can be offset by a clock frequency boost, which may be possible in such a design, due to the reduced resource latencies and bandwidths.

High-performance processors (e.g., the Compaq Alpha 21264 and the IBM POWER4) certainly have elements of multiclustering, especially in terms of duplicated register files and distributed issue queues. Zyuban proposed and modeled a specific multicluster organization in his work. This simulation-based study determined the optimal number of clusters and their configurations, for the EDP metric.

### 3.4.3 Simultaneous Multithreading

Let us examine the SMT paradigm [45] to understand how this may affect our notion of power–performance efficiency. The data in Table 3.1 shows the steady-state utilization of some of the resources in our base superscalar machine in response to the input loop test case discussed earlier. Since, because of fundamental ILP limits the IPC will not increase beyond $W = 4$, it is clear why power–performance efficiency will be on a downward trend beyond a certain width of the machine. (Of course, here we assume maximum processor power numbers, without any clock-gating or dynamic adaptation to bring down the power).

With SMT, assume that we can fetch from two threads (simultaneously, if the icache is dual-ported, or in alternate cycles if the icache remains single-ported). Suppose two copies of the same loop program (see example at the beginning of Section 3.4) are executing as two different threads. So, thread-1 instructions A-B-C-D-E-F-G and thread-2 instructions P-Q-R-S-T-U-V are simultaneously available for dispatch and subsequent execution on the machine. This facility allows the utilization factors, and the net throughput performance to go up, without a significant increase in the maximum clocked power. This is because, the issue width $W$ is not increased but the execution and resource stages or slots can be filled up simultaneously from both threads. The added complexity in the front end, of maintaining two program counters (fetch streams) and the global register space increase alluded to before, adds to the power a bit. On the other hand, the core execution complexity can be relaxed a bit without a performance hit. For example, the fp_delay parameter can be increased, to reduce core complexity, without any performance degradation. Figure 3.9 shows the expected performance and power–performance variation with superscalar width $W$ for the 2-thread SMT processor. The power model assumed for the SMT machine is the same as that of the underlying superscalar, except that a fixed fraction of the net power is added to account for the SMT overhead. (The fraction added is assumed to be linear in the number of threads, in an $n$-thread SMT processor.) Figure 3.9 shows that under the assumed model, the performance–power efficiency scales better with $W$, compared with the base superscalar (Figures 3.7 and 3.8).

Seng and Tullsen [46] presented analysis to show that using a suitably architected SMT processor, the per-thread speculative waste can be reduced, while increasing the utilization of the machine resources by

**TABLE 3.1** Steady-State Resource Utilization Profile for Base ($W = 4$) Superscalar Machine

| Resource Name | Steady-State Utilization (%) |
|---|---|
| Completion (reorder) buffer, CBUF | 53 |
| Load-store issue queue, LSQ | 0 |
| Load-store unit pipe 0, LSU-0 | 100 |
| Load-store unit pipe 1, LSU-1 | 50 |
| Floating point issue queue, FPQ | 0 |
| Floating point unit pipe 0, FPU-0 | 100 |
| Floating point unit pipe 1, FPU-1 | 50 |
| Data cache read port 0, C0 | 50 |
| Data cache read port 1, C1 | 50 |
| Data cache store port, C2 | 50 |
| Pending store queue, PSQ | 12.5 |

executing simultaneously from multiple threads. This was shown to reduce the average energy per instruction by 22%.

### 3.4.4 Chip Multiprocessing

In a multiscalar-like CMP machine [47], different iterations of a single-loop program could be initiated as separate tasks or threads on different core processors on the same chip. Thus, the threads A-B-C-D-E-F-G and P-Q-R-S-T-U-V derived from the same user program would be issued in sequence by a global task sequencer to two cores, in a two-way multiscalar CMP. Register values set in one task are forwarded in sequence to dependent instructions in subsequent tasks. For example, the register value in fp1 set by instruction E in task 1 must be communicated to instruction T in task 2; so instruction T must stall in the second processor until the value communication has occurred from task 1. Execution on each processor proceeds speculatively, assuming the absence of load-store address conflicts between tasks; dynamic memory address disambiguation hardware is required to detect violations and restart task executions as needed. In this paradigm also, if the performance can be shown to scale well with the number of tasks, and if each processor is designed as a limited-issue, limited-speculation (low complexity) core, it is possible to achieve better overall scalability of performance–power efficiency.

Another trend in high-end microprocessors is true CMP, where multiple (distinct) user programs execute separately on different processors on the same chip. A commonly used paradigm in this case is that of (shared memory) symmetric multiprocessing (SMP) on a chip (see Hammond et al. in Ref. [41]). Larger SMP server nodes can be built from such chips. Server product groups such as IBM's high-end PowerPC division have relied on such CMP paradigms as the scalable paradigm for the immediate future. IBM's POWER4 and POWER5 designs [15,48,49] are the first examples of this industry-wide trend, more recent examples being Intel's Montecito [50] and Sun's Niagara [51]. Such CMP designs offer the potential of convenient coarse-grain clock-gating and "power-down" modes, where one or more processors on the chip may be turned off or slowed down to save power when needed.

In general, in a design era where technological constraints like power dissipation have caused a significant slow down in the growth of single-thread execution clock frequency (and performance), parallelism via multiple cores on a die, each operating at lower than achievable single-core frequency (i.e., at larger FO4 per stage than earlier generation processors), is the natural paradigm of choice for power-efficient, scalable performance growth through the next several generations of processor design. Heterogeneity in the form of different types of cores, accelerators, and mixed signal components [52–54] or variable per-core frequency support are on the horizon. Such heterogeneity is needed to support the diverse set of workloads that will enable the next generation of computing systems in a power-constrained design era.

## 3.5 Conclusions

In this chapter, we first discussed issues related to power–performance efficiency and metrics from an architect's viewpoint. We showed that depending on the application domain (or market), it may be necessary to adopt one metric over another in comparing processors within that segment. Next, we described some of the promising new ideas in power-aware microarchitecture design. This discussion included circuit-centric solutions like clock-gating and power-gating, where microarchitectural support is needed to make the right decisions at run time. Later, we used a simple loop test case to illustrate the limits of power–performance scalability in some popular paradigms that are being developed by various vendors within the high-end processor domain. In particular, we show that scalability of current generation superscalars may be extended effectively through multiclustering, SMT, and CMP. Our experience in simulating these structures points to the need of keeping a single core (or the uni-threaded core) simple enough to ensure scalability in the power, performance, and verification cost of future systems. Detailed simulation results with benchmarks to support these conclusions were not provided in this tutorial-style paper. We limited our focus to a few key ideas and paradigms of interest in future power-aware processor design. Many new ideas to address various aspects of power reduction have been presented in recent

workshops [55–57]. All of these could not be discussed in this chapter, but the interested reader should certainly refer to the cited references for further detailed study.

## Acknowledgments

The author is indebted to his fellow researchers (and summer interns) who either have contributed or still contributing to the power-aware microprocessor project at IBM Watson. In particular, he is thankful to Victor Zyuban, Alper Buyuktosunoglu, Hans Jacobson, Stanley Schuster, Peter Cook, Jude Rivers, Zhigang Hu, Prabhakar Kudva, Chen-Yong Cher, Eren Kursun, Yingmin Li, Canturk Isci, Hendrik Hamann, Alan Weger, Hubertus Franke, Joenghwan Choi, Rakesh Kumar, and Phil Emma. The author is also thankful to other colleagues in his organization's management chain, including Jaime Moreno and Michael Rosenfield for their constant support and encouragement. In addition, the author would like to express his gratitude to Prof. David Brooks at Harvard University, Prof. Margaret Martonosi at Princeton University, Prof. Kevin Skadron at University of Virginia, Prof. Vojin Oklobdzija at University of California, Davis, and Prof. David Albonesi at University of Rochester for their active collaboration with our group at IBM T.J. Watson Research Center.

## References

1. Hennessey, J.L. and Patterson, D.A., *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.
2. Brooks, D.M., Bose, P., Schuster, S.E., Jacobson, H., Kudva, P.N., Buyuktosunoglu, A., Wellman, J.-D., Zyuban, V., Gupta, M., and Cook, P.W., Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors, *Proc. IEEE Micro*, 20(6): 26–44, November/December 2000.
3. Flynn, M.J., Hung, P., and Rudd, K., Deep-submicron microprocessor design issues, *Proc. IEEE Micro*, 19(4): 11–22, July/August 1999.
4. Borkar, S., Design challenges of technology scaling, *Proc. IEEE Micro*, 19(4): 23–29, July/August 1999.
5. Gonzalez, R. and Horowitz, M., Energy dissipation in general purpose microprocessors, *IEEE J. Solid-State Circuits*, 31(9): 1277–1284, September 1996.
6. Agrawal, A., Mukhopadhyay, S., Raychowdhury, A., Roy, K., and Kim, C.H., Leakage power analysis and reduction in nanoscale circuits, *IEEE Micro*, 26(2): 68–80, March 2006.
7. Hu, C., Device and technology impact on low power electronics, in *Low Power Design Methodologies*, Jan Rabaey (Ed.), Kluwer Publishing, Dordrecht, 1996, pp. 21–35.
8. Brews, J.R., Subthreshold behavior of uniformly and non-uniformly doped long-channel MOSFET, *IEEE Trans. Electron Devices*, 26(9): 1282–1291, September 1979.
9. Sakurai, T. and Newton, R., Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas, *IEEE J. Solid State Circuits*, 25(2): 584–594, 1990.
10. Kanda, K., et al., Design impact of positive temperature dependence on drain current in sub-1-V CMOS VLSIs, *IEEE J. Solid State Circuits*, 36(10): 1559–1564, 2001.
11. Oklobdzija, V.G., Architectural tradeoffs for low power, *Proceedings of the International Symposium on Computer Architecture (ISCA) Workshop on Power-Driven Microarchitectures*, Barcelona, Spain, June 1998.
12. Iyer, A. and Marculescu, D., Power–performance evaluation of globally asynchronous, locally synchronous processors, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Anchorage, AK, May 2002, pp. 158–168.
13. Kogge, P.M., *The Architecture of Pipelined Computers*, Hemisphere Publishing Corporation, New York, 1981.
14. Larson, A.G., Cost-effective processor design with an application to fast Fourier transform computers, Digital Systems Laboratory Report SU-SEL-73-037, Stanford University, Stanford, CA, August 1973; see also, Larson and Davidson, Cost-effective design of special purpose processors:

A fast Fourier transform case study, *Proceedings of the 11th Annual Allerton Conference on Circuits and System Theory*, University of Illinois, Champaign-Urbana, IL, 1973, pp. 547–557.
15. Tendler, J.M., Dodson, J.S., Fields, J.S., Jr., Le, H., and Sinharoy, B., POWER4 system microarchitecture, *IBM J. Res. Dev.*, 46(1): 1–116, January 2002.
16. Dubey, P.K. and Flynn, M.J., Optimal pipelining, *J. Parallel and Distributed Computing*, 8(1): 10–19, January 1990.
17. Hartstein, A. and Puzak, T.R., The optimum pipeline depth for a microprocessor, *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, Anchorage, AK, May 2002, pp. 7–13.
18. Srinivasan, V., Brooks, D., Gschwind, M., Bose, P., Zyuban, V., Strenski, P.N., and Emma, P.G., Optimizing pipelines for power and performance, *Proceedings of the 35th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-35)*, Istanbul, Turkey, November 2002, pp. 333–344.
19. Zyuban, V., Brooks, D., Srinivasan, V., Gschwind, M., Bose, P., Strenski, P., and Emma, P., Integrated analysis of power and performance for pipelined microprocessors, *IEEE Trans. Comput.*, 53(8): 1004–1016, August 2004.
20. Borkar, S., VLSI design challenges for gigascale integration, Keynote speech given at 18th International Symposium on VLSI Design, Kolkata, India, January 2005.
21. Tiwari, V. et al., Reducing power in high-performance microprocessors, *Proceedings of the IEEE/ACM Design Automation Conference*, ACM, New York, 1998, pp. 732–737.
22. Gunther, S.H., Binns, F., Carmean, D.M., and Hall, J.C., Managing the impact of increasing microprocessor power consumption, *Intel Technology Journal Q1*, 5(1): 1–9, February 2001.
23. Brooks, D., Bose, P., Srinivasan, V., Gschwind, M.K., Emma, P.G., and Rosenfield, M.G., New methodology for early-stage, microarchitecture-level power–performance analysis of microprocessors, *IBM J. Res. Dev.*, 47(5/6): 653–662, September/November 2003.
24. Jacobson, H., Bose, P., Hu, Z., Eickemeyer, R., Eisen, L., and Griswell, J., Stretching the limits of clock-gating efficiency in server-class processors, *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, San Francisco, CA, February 2005, pp. 238–242.
25. Jacobson, H.M., Improved clock-gating through transparent pipelining, *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, Newport Beach, CA, August 2004, pp. 26–31.
26. Jacobson, H.M. et al., Synchronous interlocked pipelines, *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, Manchester, UK, April 2002, pp. 3–12.
27. Kaxiras, S., Hu, Z., and Martonosi, M., Cache decay: Exploiting generational behavior to reduce cache leakage power, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Goteborg, Sweden, June–July 2001, pp. 240–251.
28. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., and Mudge, T., Drowsy caches: Simple techniques for reducing leakage power, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.
29. Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H., and Bose, P., Microarchitectural techniques for power gating of execution units, *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, Newport Beach, CA, August 2004, pp. 32–37.
30. Brooks, D. and Martonosi, M., Dynamically exploiting narrow width operands to improve processor power and performance, *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA-5)*, Orlando, FL, January 1999.
31. Albonesi, D., Dynamic IPC/clock rate optimization, *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, Barcelona, Spain, 1998, pp. 282–292.
32. Maro, R., Bai, Y., and Bahar, R.I., Dynamically reconfiguring processor resources to reduce power-consumption in high-performance processors, *Proceedings of Power Aware Computer Systems (PACS) Workshop*, held in conjunction with ASPLOS, Cambridge, MA, November 2000.

33. Buyuktosunoglu, A. et al., An adaptive issue queue for reduced power at high performance, *Proceedings of ISCA Workshop on Complexity-Effective Design (WCED)*, Vancouver, Canada, June 2000.
34. Cruz, J.-L., Gonzalez, A., Valero, M., and Topham, N.P., Multiple-banked register file architectures, *Proceedings of the International Symposium on Computer Architecture (ISCA),* Vancouver, Canada, June 2000, pp. 316–325.
35. Albonesi, D.H., Balasubramonian, R., Dropsho, S.G., Dwarkadas, S., Friedman, E.G., Huang, M.C., Kursun, V., Magklis, G., Scott, M.L., Semeraro, G., Bose, P., Buyuktosunoglu, A., Cook, P.W., and Schuster, S.E., Dynamically tuning processor resources with adaptive processing, *IEEE Computer*, Special Issue on Power-Aware Computing, 36(12): 49–58, December 2003.
36. Reed, P. et al., 250 MHz 5 W RISC microprocessor with on-chip L2 cache controller, in *Digest of Technical Papers, IEEE Journal of Solid-State Circuits (JSSC)*, 32(11): 1635–1649, 1997.
37. Sanchez, H. et al., Thermal management system for high performance PowerPC microprocessors, in *Digest of papers, IEEE COMPCON*, p. 325, 1997.
38. Brooks, D. and Martonosi, M., Dynamic thermal management for high-performance microprocessors, *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, Nuevo Leon, Mexico, January 2001, pp. 20–24.
39. Manne, S., Klauser, A., and Grunwald, D., Pipeline gating: Speculation control for energy reduction, *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, Barcelona, Spain, 1998, pp. 132–141.
40. Grunwald, D., Klauser, A., Manne, S., and Pleszkun, A., Confidence estimation for speculation control, *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, Barcelona, Spain, 1998, pp. 122–131.
41. Theme issue, The future of processors, *IEEE Comput.*, 30(9): 37–93, September 1997.
42. Bose, P., Kim, S., O'Connell, F.P., and Ciarfella, W.A., Bounds modeling and compiler optimizations for superscalar performance tuning, *J. Syst. Architecture*, 45: 1111–1137, 1999, Elsevier Press.
43. Zyuban, V., Inherently lower-power high performance super scalar architectures, Ph.D Thesis, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, 2000.
44. Zyuban, V. and Kogge, P., Optimization of high-performance superscalar architectures for energy efficiency, *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED)*, Rapallo, Italy, July 2000, pp. 84–89.
45. Tullsen, D.M., Eggers, S.J., and Levy, H.M., Simultaneous multithreading: Maximizing on-chip parallelism, *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, Santa Margherita Ligure, Italy, June 1995, pp. 292–403.
46. Seng, J.S., Tullsen, D.M., and Cai, G., The power efficiency of multithreaded architectures, Invited talk presented at ISCA Workshop on Complexity-Effective Design (WCED), Vancouver, Canada, June 2000.
47. Sohi, G., Breach, S.E., and Vijaykumar, T.N., Multiscalar processors, *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, IEEE CS Press, Los Alamitos, CA, 1995, pp. 414–425.
48. Diefendorff, K., POWER4 focuses on memory bandwidth, *Microprocessor Report*, October 6, 1999, pp. 11–17.
49. Kalla, R., Sinharoy, B., and Tendler, J., IBM POWER5 chip: A dual-core multithreaded processor, *IEEE Micro*, 24(2): 40–47, March/April 2004.
50. McNairy, C. and Bhatia, R., Montecito: A dual-core, dual-thread Itanium processor, *IEEE Micro*, 25(2): 10–20, March/April 2005 (see also, Hot Chips 2004).
51. Kongetira, P., A 32-way multithreaded SPARC® processor, presented at Hot Chips, August 2004.
52. Rakesh, Kumar, Tullsen, D., Jouppi, N., and Ranganathan, P., Heterogeneous chip multiprocessors, *IEEE Computer*, November 2005.
53. Kahle, J.A. et al., Introduction to the cell multiprocessor, *IBM J. Res. Dev.*, 49(4/5), 2005.
54. Gara, A. et al., Overview of the Blue Gene/L system architecture, *IBM J. Res. Dev.*, 49(2/3), 2005.

55. Talks presented at the Kool Chips Workshops: http://www.cs.colorado.edu/~grunwald/LowPowerWorkshop.
56. Talks presented at the ISCA Workshops on Complexity Effective Design (WCED-2000 through WCED-2006), http://www.csl.cornell.edu/~albonesi/wced.html.
57. Talks presented at the Power Aware Computer Systems (PACS) Workshops; e.g., the 2004 offering: http://www.ece.cmu.edu/~pacs04/.

# 4

# Performance Evaluation




Jozo J. Dujmović
Daniel N. Tomasevich
Ming Au-Yeung
*San Francisco State University*

Lizy Kurian John
*University of Texas at Austin*

Eric Rotenberg
*North Carolina State University*


## 4.1 Measurement and Modeling of Disk Subsystem Performance

*Jozo J. Dujmović, Daniel N. Tomasevich, and Ming Au-Yeung*

### 4.1.1 Introduction

In queuing theory literature, many models describe the dynamic behavior of computer systems. Good sources of such information [1,2] usually include stochastic models based on birth–death formulas, the convolution algorithm [3], load-independent and load-dependent mean value analysis (MVA) models [4], and BCMP networks [5]. Theoretical queuing models presented in computer literature easily explains phenomena such as bottlenecks, saturation, resource utilization, etc.; however, it is very difficult to find sources that show a second level of modeling, which focuses on the ability of models to also

achieve good numerical accuracy when modeling real computer systems running real workloads. Although the phenomenology is important in the classroom, it is the numerical accuracy that counts in engineering practice. The usual task of performance analysts is to measure system performance and then derive models that can describe and predict the behavior of analyzed systems with reasonable accuracy. Trying to model the dynamic behavior of real computer systems running real workloads frequently is found to be a difficult task.

Our first goal is to develop load-dependent models of disk units that have a moderate level of complexity suitable for engineering practice. The presented disk unit models describe disk access times, head movement optimization, and disk caching. These models are then used for creating MVA models of disk subsystems. Our second goal is to investigate and exemplify the limits of numerical accuracy of presented queuing models, and to propose indicators of predictive power of analyzed models. We present case studies of disk subsystem modeling of a VAX under VMS and a PC under Windows NT. They provide a good insight into the level of difficulty encountered in practical disk subsystem modeling and help establish realistic expectations of modeling errors. The relative simplicity of MVA models makes them attractive for practice. They can be easily combined with our disk unit models. However, our experiments with MVA models show that only the load-dependent version of MVA generates results with reasonable accuracy.

### 4.1.2 Description Errors and Prediction Errors of Disk Subsystem Models

Modeling errors are defined as differences between queuing theory results and experimental measurements. Simple queuing models of disk subsystems (such as load-independent MVA) frequently generate modeling errors of 30%–50% or more. In the majority of practical cases, such low accuracy is not acceptable. Usually, modeling errors below 5% are acceptable, but require detailed and more sophisticated models.

The accuracy of models can only be evaluated with respect to measurements performed for a specific system running a specific workload. Our approach to modeling and analysis of disk subsystems includes the following main steps:

- Specification of drive workload that can be used to create various levels of disk subsystem load
- Measurement of system performance under a strictly increasing disk subsystem load
- Development of an analytic model with adjustable parameters that describes the dynamics of a disk subsystem
- Calibration of the analytic model by adjusting all model parameters to minimize the difference between the measured values and the values computed from the analytic model
- Assessment of the predictive power of the analytic model
- The use of the calibrated model for performance prediction of systems with different parameters or workload

It is useful to identify two types of modeling errors: description errors and prediction errors. We define a description error as the mean relative error between the measured performance indicators of a real system and the performance indicators of a calibrated model. The description error is defined only within the range of measurements. By contrast, the prediction error is defined as the error in predicting the values of performance indicators outside the range of measurements or for different configurations of the analyzed system or for a different workload. For example, if we measure the response time for the degree of multiprogramming from 1 to 8 then the description error is the mean error between eight measured values and eight computed values. The prediction error is the error between the computed response time for 20 jobs and the actual response time, if it were measured. The prediction error is also the error between predicted time and actual response time for a system with a different number of disks, processors, or for a different workload.

The basic problem of modeling is that the ratio between prediction errors and description errors can frequently be large, e.g., 2–10. Consequently, to provide reasonable prediction power, the description errors of analytic models must be small, typically just a few percentage.

### 4.1.3 A Simple Acceleration/Deceleration Model of Disk Access Time

The movement of the disk input/output (I/O) mechanism is usually modeled assuming that movement is caused by applying a constant force for both acceleration and deceleration. This is a simple and usually a realistic assumption illustrated in Fig. 4.1. If the mass of the I/O mechanism is $m$, then force $f$ causes the acceleration $a = f/m$. After applying acceleration $a$ with force $f$, we apply deceleration with force $-f$. The I/O heads travel the total distance $x$ and the corresponding time $T$ is called the seek time. After accelerating for time $T/2$, the I/O heads attain the speed $v = aT/2$ and travel the distance $x/2 = a(T/2)^2/2$. Therefore, the seek time as a function of distance is $T(x) = 2\sqrt{x/a}$. If the maximum distance is $x_{\max}$ then the maximum seek time is $T_{\max}$; the acceleration is $a = 4x_{\max}/T_{\max}^2$, and $T(x) = T_{\max}\sqrt{x/x_{\max}}$. The distance $x$ can be expressed as a physical length or as a number of cylinders traveled (in this case, the dimension of acceleration is cylinder/s$^2$). If the head moves from cylinder $y$ to cylinder $z$ then the traveled distance is $x = |y - z|$. The seek time is $t(y,z) = T_{\max}\sqrt{|y-z|/x_{\max}}$. The initial position $y$ and the final position $z$ can be anywhere in the interval $[0, x_{\max}]$. Assuming the uniform distribution of accesses to all cylinders, the average seek time, $T_{\text{seek}}$, can be computed as follows:

$$T_{\text{seek}} = \frac{1}{x_{\max}^2}\int_0^{x_{\max}}\int_0^{x_{\max}} t(y,z)\,\mathrm{d}y\mathrm{d}z = \frac{T_{\max}}{x_{\max}^{5/2}}\int_0^{x_{\max}}\int_0^{x_{\max}} \sqrt{|y-z|}\,\mathrm{d}y\mathrm{d}z = \frac{8}{15}T_{\max} = \frac{16}{15}\sqrt{\frac{x_{\max}}{a}}$$

Consider the case where each cylinder has a fixed data capacity of $b$ bytes. For a large contiguous file of size $F$, the maximum distance the I/O mechanism can travel is $x_{\max} = F/b$ cylinders. Consequently, the average seek time for this file is $T_{\text{seek}} = c\sqrt{F}$, where $c = 16/(15\sqrt{ab}) =$ constant. If we access data in the range $0 \le F \le F_{\max}$, the average seek time is the following function of the file size:

$$T_{\text{seek}}(F) = T_{\max\ \text{seek}}\sqrt{F/F_{\max}}$$

$$T_{\max\ \text{seek}} = \frac{16}{15}\sqrt{\frac{F_{\max}}{ab}}$$

Now the constant $T_{\max\ \text{seek}}$ has a suitable interpretation of the maximum value of the average seek time for the file of size $F_{\max}$.

When the I/O mechanism reaches a destination cylinder, it is necessary to wait for the latency time until the desired sector reaches the read/write head. For a disk that rotates at $N_{\text{rev}}$ revolutions/min, the total revolution time is $60/N_{\text{rev}}$. The latency time is uniformly distributed between 0 and $60/N_{\text{rev}}$. Therefore, the average latency is half of the revolution time, i.e., $T_{\text{latency}} = 30/N_{\text{rev}}$. The disk data transfer time for one sector is $T_{\text{transfer}} = 60/N_{\text{rev}}N_{\text{sector}} = 2T_{\text{latency}}/N_{\text{sector}}$. Therefore, the mean time to access and transfer disk data from a file of size $F$ is

$$\begin{aligned} T_{\text{access}} &= T_{\text{seek}} + T_{\text{latency}} + T_{\text{transfer}} \\ &= T_{\max\ \text{seek}}\sqrt{F/F_{\max}} + 30(1 + 2/N_{\text{sector}})/N_{\text{rev}} \\ &\cong T_{\max\ \text{seek}}\sqrt{F/F_{\max}} + 30/N_{\text{rev}} \end{aligned}$$

This is the simplest and idealized model of the mean disk access time. This model neglects phenomena such as differences between disk acceleration and deceleration, head settle time [6] (dominating short seeks), limited maximum velocity, nonuniformity of rotational latency for nonindependent requests, zoning, bus interface and contention [7], disk command overhead [8], as well as effects of caching and head movement optimization.

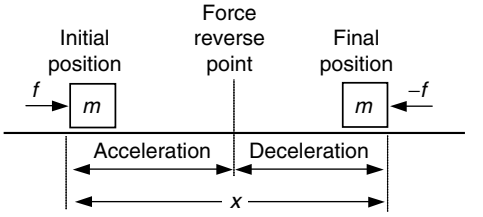



**FIGURE 4.1** Movement of the disk I/O mechanism (seek). (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

### 4.1.4 A Fixed Maximum Velocity Model of Seek Time

A more realistic model of seek time can be derived if we assume that the maximum velocity of the I/O heads has a constant maximum value, as shown in Fig. 4.2. For small distances (exemplified by seek times $t_1$, $t_2$, and all other seek times $t \leq T^*$) we assume that the heads first linearly accelerate for some time and then decelerate exactly the same amount of time. For larger distances (seek times $t \geq T^*$), the heads linearly accelerate until they achieve the maximum speed $v_{\max}$, then travel at this constant speed, and eventually decelerate linearly.

The maximum time $T^*$ that the disk mechanism can travel without being limited by the maximum velocity is called the critical seek time. $T^*$ is the time to accelerate disk heads to the maximum speed and then to decelerate them to a standstill. In cases where disk heads travel at the constant maximum speed, the critical seek time $T^*$ denotes the acceleration plus deceleration time, i.e., the total seek time minus the time heads spent traveling at the constant maximum speed. Assuming constant acceleration/deceleration, the critical seek time is $T^* = 2v_{\max}/a$. During the critical seek time interval, the mechanism travels the distance $x^*$, called the critical distance. In the case of constant acceleration/deceleration, we have $v = at$, $x(t) = at^2/2$, and $x^* = 2x(T^*/2) = v_{\max}^2/a$.

Let us now investigate a general symmetrical case where the function $v(t)$ has the property that the acceleration time equals the deceleration time. We differentiate the small distance model ($T \leq T^*$) and the large distance model ($T \geq T^*$) as follows:

$$x(T) = \int_0^T v(t)\,dt$$

$$= \begin{cases} \underbrace{\int_0^{T/2} v(t)\,dt}_{\text{acceleration}} + \underbrace{\int_{T/2}^{T} v(t)\,dt}_{\text{deceleration}}, & T \leq T^* \\ \underbrace{\int_0^{T^*/2} v(t)\,dt}_{\text{acceleration}} + \underbrace{\int_{T^*/2}^{T-T^*/2} v_{\max}\,dt}_{\text{max velocity}} + \underbrace{\int_{T-T^*/2}^{T} v(t)\,dt}_{\text{deceleration}} = x^* + v_{\max}(T - T^*), & T \geq T^* \end{cases}$$

$$x^* = \underbrace{\int_0^{T^*/2} v(t)\,dt}_{\text{acceleration}} + \underbrace{\int_{T-T^*/2}^{T} v(t)\,dt}_{\text{deceleration}}$$

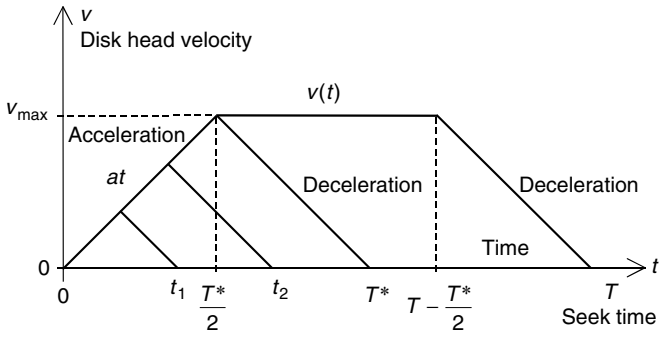



**FIGURE 4.2** A simplified disk head velocity diagram. (From Dujmovic, J.J. and Tomasevich, D., *Calibration and comparison of disk unit models*, 27th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 2001 Proceedings, Vol. 1, pp. 315–325, 2001. With permission.)

The distance $x^*$ is the total distance for acceleration from 0 to $v_{max}$ and deceleration from $v_{max}$ to 0. In the constant acceleration case ($v(t) = at$), this model yields

$$x(T) = \begin{cases} \dfrac{aT^2}{4} = \dfrac{v_{max}}{2T^*}T^2, \; T \le T^* = \dfrac{2v_{max}}{a} \\ v_{max}\left(T - \dfrac{v_{max}}{a}\right) = v_{max}\left(T - \dfrac{T^*}{2}\right), \quad T \ge T^* \end{cases}$$

The critical distance $x^*$ is given by the following expression:

$$x^* = x(T^*) = \frac{a(T^*)^2}{4} = \frac{v_{max}T^*}{2} = \frac{v_{max}^2}{a}$$

Therefore, the acceleration and the maximum speed depend on the critical values $x^*$ and $T^*$:

$$a = 4x^*/(T^*)^2, \quad v_{max} = 2x^*/T^*$$

They can also be numerically determined from the linear segment of the seek time characteristic, using arbitrary points $(x_1, T_1)$ and $(x_2, T_2)$:

$$v_{max} = \frac{x_2 - x_i}{T_2 - T_1}, \quad a = \frac{(x_2 - x_1)^2}{(T_1x_2 - T_2x_1)(T_2 - T_1)}$$

The seek time characteristic follows from $x(T)$:

$$T(x) = \begin{cases} 2\sqrt{\dfrac{x}{a}} = 2\sqrt{\dfrac{xx^*}{v_{max}}} = T^*\sqrt{\dfrac{x}{x^*}}, \quad x \le x^* \\ \dfrac{x + x^*}{v_{max}} = \dfrac{T^*}{2} + \dfrac{x}{v_{max}} = \dfrac{T^*}{2}\left(\dfrac{x}{x^*} + 1\right), \quad x \ge x^* \end{cases}$$

This function satisfies the following properties:

- The initial nonlinear segment, for $x \le x^*$, is a square root function.
- The second segment, for $x > x^*$, is a linear function.
- At the critical point $x = x^*$, the first derivative of the square root function equals the first derivative of the linear function ($dT/dx = 1/v_{max}$).

From $T = T^*(1 + x/x^*)/2$, we can easily see that both $x^*$ and $T^*$ can be determined from the linear segment of the measured seek time characteristic using the following four steps, illustrated in Fig. 4.3:

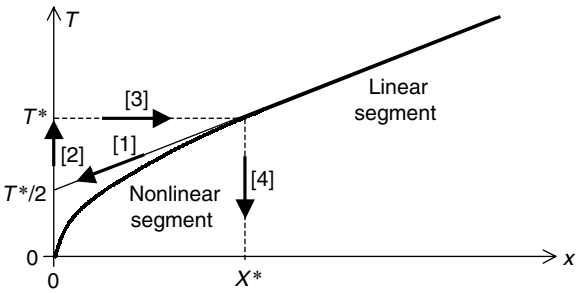



**FIGURE 4.3** Graphical method for determining $x^*$ and $T^*$. (From Dujmovic, J.J. and Tomasevich, D., *Calibration and comparison of disk unit models*, 27th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 2001 Proceedings, Vol. 1, pp. 315–325, 2001. With permission.)

1. Extend the linear segment of the measured seek time characteristic to the vertical axis. For $x = 0$, we get the point $T^*/2$ on the vertical axis.
2. Move vertically up to the point $T^*$ (this value is twice the distance between the origin and the intersection of the linear segment and the vertical axis).
3. Move horizontally to the intersection with the linear segment.
4. Move down to the horizontal axis and determine the point $x^*$.

Because the linear segment of the seek time characteristic determines the values of $x^*$ and $T^*$ (as well as $a$ and $v_{max}$), it follows that the nonlinear segment must terminate exactly in the $(x^*, T^*)$ point. A numerical method for computing $x^*$ and $T^*$ can be based on any two points $(x_1, T_1)$ and $(x_2, T_2)$ taken on the linear segment. From the linear segment function $T = T_1 + (x - x_1)(T_2 - T_1)/(x_2 - x_1)$, we obtain the following expression:

$$x^* = \frac{T_1 x_2 - T_2 x_1}{T_2 - T_1}, \quad T^* = 2\,\frac{T_1 x_2 - T_2 x_1}{x_2 - x_1}$$

The presented model is suitable for a qualitative description of the seek time behavior and for providing insight into disk characteristics, but its flexibility is limited because it has only two parameters. In addition to limited flexibility, this model is not appropriate for those disk units that do not satisfy the assumptions of constant acceleration/deceleration, and for units where the seek time for short distances is significantly affected by the head settling time. The accuracy of modeling can be improved using numerical models that fit the measured characteristic and use more than two adjustable parameters.

### 4.1.5 Numerical Computation of the Average Seek Time

A simple numerical model can be based on three points: $(x_1, T_1)$, $(x^*, T^*)$, and $(x_2, T_2)$, where $x_1 < x^* < x_2$. Here, $(x_1, T_1)$ is the point from the initial nonlinear segment. Contrary to the approach presented in Section 4.1.4, the middle point $(x^*, T^*)$ is not computed from the linear part of the measured characteristic, but directly selected from the seek time graph as the beginning of the linear segment. The point $(x_2, T_2)$ denotes the end of the linear segment (the maximum distance heads can travel). The corresponding model is as follows:

$$T(x) = \begin{cases} T^*\left(\dfrac{x}{x^*}\right)^r = tx^r, & t = \dfrac{T^*}{(x^*)^r}, \quad x \le x^* \\ T^* + \dfrac{T_2 - T^*}{x_2 - x^*}(x - x^*) = Ax + B, & x \ge x^* \end{cases}$$

From $T_1 = tx_1^r$ and $T^* = t(x^*)^r$, it follows that $T^*/T_1 = (x^*/x_1)^r$. Therefore, the parameters are

$$r = \frac{\log(T^*/T_1)}{\log(x^*/x_1)}, \quad t = T_1/x_1^r = T^*/(x^*)^r$$
$$A = \frac{T_2 - T^*}{x_2 - x^*}, \quad B = T^* - Ax^*$$

Using this model, it is possible to numerically compute the mean seek time. Suppose that a file occupies $N$ cylinders. The probability that the seek distance $x$ is less than or equal to a given value $z$ is

$$P_x(z) = P[x \le z] = (2Nz - z^2)/N^2$$

This probability distribution function yields the following probability density function:

$$p_x(z) = \mathrm{d}P/\mathrm{d}z = 2(N - z)/N^2$$

Hence, the average seek time for the $N$ cylinders file can be computed as follows:

$$\bar{T}_{\text{seek}} = \int_0^N T(z)p_x(z)\mathrm{d}z = \frac{2}{N^2}\int_0^N T(z)(N - z)\mathrm{d}z$$

Using the presented seek time model for $N \leq x^*$, we have

$$\bar{T}_{\text{seek}} = \frac{2t}{N^2}\int_0^N z^r(N - z)\mathrm{d}z = \frac{2tN^r}{(r + 1)(r + 2)}$$

Similarly, for $N \geq x^*$, we get

$$\begin{aligned}
\bar{T}_{\text{seek}} &= \frac{2}{N^2}\left[\int_0^{x^*} T(z)(N - z)\mathrm{d}z + \int_{x^*}^N T(z)(N - z)\mathrm{d}z\right] \\
&= \frac{2}{N^2}\left[\int_0^{x^*} tz^r(N - z)\mathrm{d}z + \int_{x^*}^N (Az + B)(N - z)\mathrm{d}z\right] \\
&= \frac{2t(x^*)^{r+1}}{N^2}\left(\frac{N}{r + 1} - \frac{x^*}{r + 2}\right) + \frac{(N - x^*)[N(AN + 3B + Ax^*) - x^*(3B + 2Ax^*)]}{3N^2}
\end{aligned}$$

Therefore, the average seek time as a function of file size is given by the following function:

$$\bar{T}_{\text{seek}}(N) = \begin{cases} \dfrac{2tN^r}{(r + 1)(r + 2)}, & N \leq x^* \\ \dfrac{2t(x^*)^{r+1}}{N^2}\left(\dfrac{N}{r + 1} - \dfrac{x^*}{r + 2}\right) + \dfrac{(N - x^*)[N(AN + 3B + Ax^*(3B + 2Ax^*)]}{3N^2}, & N \geq x^* \end{cases}$$

Advantages of the presented exponential model are (1) parameters can be quickly computed from three selected points of the characteristic and (2) parameter $r$ enables modeling of disk characteristics that are different from the square root model. From $T(1) = t$, it follows that $t$ is interpreted as the single cylinder seek time. The limitation of this model is that by determining $t$ and $r$ from points $(x_1, T_1)$ and $(x^*, T^*)$, it is not possible to have the exact value of $t$ and optimum modeling of the curvature. To improve this model we can introduce one more parameter as follows:

$$T(x) = \begin{cases} 0, & x = 0 \\ t + c(x - 1)^r & 1 \leq x \leq x^* \\ Ax + B, & x \geq x^* \end{cases}$$

This new model has a nonlinear part (for $x \leq x^*$) and a linear part (for $x \geq x^*$). Since $T(1) = t$ and $T(2) = t + c$, the parameter $t$ is the single cylinder seek time and the parameter $c$ is the difference $c = T(2) - T(1)$. The parameters of this model ($t$, $c$, $r$, $A$ , $B$, $x^*$) are not independent. First, for $x = x^*$ the nonlinear part must be connected to the linear part:

$$t + c(x^* - 1)^r = Ax^* + B$$

In addition, to assure perfect continuity of this model, for $x = x^*$ the nonlinear and linear model must have the same first derivatives:

$$\left.\frac{dT}{dX}\right|_{x=x^*} = A = cr(x^* - 1)^{r-1} \tag{4.1}$$

Inserting this value of $A$ in the connection relation, we have the expression for $B$:

$$B = t + c(x^* - 1)^r - cr(x^* - 1)^{r-1}x^* \tag{4.2}$$

The linear function can now be written as

$$Ax + B = \frac{cr(x - x^*)}{(x^* - 1)^{1-r}} + t + c(x^* - 1)^r$$

Therefore, using Eqs. 4.1 and 4.2, our exponential model is now

$$T(x) = \begin{cases} t + c(x-1)^r, & 1 \le x \le x^* \\ \dfrac{cr(x - x^*)}{(x^* - 1)^{1-r}} + t + c(x^* - 1)^r, & x \ge x^* \end{cases}$$

The model has four independent parameters—$t$, $c$, $r$, $x^*$ that can be determined using a calibration procedure. The objective of calibration is to make the model as close as possible to the measured values $(x_1,T_1),\ldots,(x_n,T_n)$. Optimum values of parameters can be computed from the measured values $(x_1,T_1),\ldots,(x_n,T_n)$ by minimizing one of the following criterion functions:

$$E_1(t,c,r,x^*) = \frac{1}{n}\sum_{i=1}^{n}(T(x_i) - T_i)^2$$

$$E_2(t,c,r,x^*) = \frac{1}{n}\sum_{i=1}^{n}\left|T(x_i) - T_i\right|$$

$$E_3(t,c,r,x^*) = \max_{1\le i\le n}\left|T(x_i) - T_i\right|$$

$E_1$ is a traditional mean square error
$E_2$ is the mean absolute error
$E_3$ is used to minimize the maximum error (minimax)

These criterion functions yield consistent or similar results and in this section we primarily used $E_1$. For all the above criteria, the most suitable minimization method is the Nelder–Mead simplex algorithm [9]. The resulting calibrated (optimum) values of parameters $t$, $c$, $r$, $x^*$ are those that yield the minimum value of the selected criterion function.

Experiments with modern disk units show that the 4-parameter exponential model regularly achieves high accuracy. Typical average relative errors are between 2% and 3%. The quality of this model is illustrated in Fig. 4.4 for the Quantum Atlas III disk that has 8057 cylinders, and capacity of 9.1 GB. The optimum parameters of the model are $t = 1.55$ ms, $c = 0.32$ ms, $r = 0.387$, and $x^* = 1686$ cylinders. Note that the optimum value of the exponent $r$ is not $1/2$ as expected from constant acceleration/deceleration models, and frequently used in disk performance literature.

A detailed comparison of the accuracy of various disk seek time models can be found in Ref. [17]. The analyzed modes include a best-fit version of the square root model [7], the polynomial square root

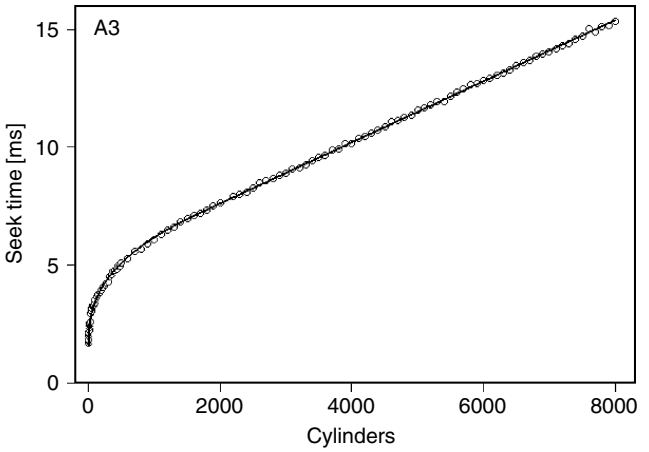



**FIGURE 4.4** The 4-parameter exponential seek time model of the Quantum Atlas III disk (based on DiskSim measurements of G.R. Ganger available at http://www.ece.cmu.edu/∼ganger). (From Dujmovic, J.J. and Tomasevich, D., *Calibration and comparison of disk unit models*, 27th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 2001 Proceedings, Vol. 1, pp. 315–325, 2001. With permission.)

model [12], and our exponential model. Typical average and maximum relative errors for seek time characteristics of 10 sample disk units are 4.5% and 66% for the best-fit square root model, 5.8% and 66% for the polynomial square root model, and 2.6% and 24% for the exponential model. The maximum errors occur in the initial nonlinear segment of the seek time characteristic.

The mean seek time for accessing a file (or database) occupying $N$ adjacent cylinders is

$$\bar{T}_{\text{seek}}(N) = \begin{cases} \dfrac{2}{N^2}\displaystyle\int_0^N T(z)(N-z)\mathrm{d}z, & N \le x^* \\ \dfrac{2}{N^2}\left[\displaystyle\int_0^{x^*} T(z)(N-z)\mathrm{d}z + \int_{x^*}^N T(z)(N-z)\mathrm{d}z\right], & N \ge x^* \end{cases}$$

The final result is

$$\bar{T}_{\text{seek}}(N) = \begin{cases} \dfrac{2(N-1)^2}{N^2}\left[\dfrac{t}{2} + \dfrac{c(N-1)^r}{(r+1)(r+2)}\right], & N \le x^* \\ \dfrac{2}{N^2}\left[\dfrac{t(x^*-1)(2N-x^*-1)}{2} + \dfrac{c(x^*-1)^{r+1}[N(r+2)-x^*(r+1)-1]}{(r+1)(r+2)} + \dfrac{(N-x^*)^2(AN+2Ax^*+3B)}{6}\right], & N \ge x^* \end{cases}$$

Here $A$ and $B$ are defined by Eqs. 4.1 and 4.2.

Numerical models based on best fit do not provide a direct correspondence between disk performance and physical attributes such as mass, force, acceleration, maximum speed, etc.; however, they are popular with many authors because of their low numerical errors. For example, Ruemmler and Wilkes [7] used the following simple model:

$$T(x) = \begin{cases} a + b\sqrt{x}, & \text{for short seeks } (a,b = \text{constant}) \\ Ax + B, & \text{for long seeks } (A,B = \text{constant}) \end{cases}$$

Similarly, Ng [11] uses the model

$$T(x) = a + b\sqrt{x} + c\log(d), \quad a,b,c = \text{constant}$$

where $d$ denotes the recording areal density in tracks/in. Lee and Katz [12] propose a model without a linear segment:

$$T(x) = \begin{cases} 0, & \text{x = 0} \\ A\sqrt{x-1} + B(x-1) + T_{\text{min seek}}, & \text{x > 0} \end{cases}$$

In this model, $A$ and $B$ are constants and $T_{\text{min seek}}$ is the minimum seek time corresponding to the minimum nonzero number of cylinders $x = 1$. Finally, Shriver [13] suggests slightly higher model granularity:

$$T(x) = \begin{cases} 0, & x = 0 \\ a + b\sqrt{x}, & 0 \le x \le x_1 \\ c + d\sqrt{x}, & x_1 \le x \le x_2 \\ Ax + B, & x > x_2 \end{cases} \quad (a,b,c,d,A,B = \text{constant})$$

The accuracy of these models is less than the accuracy of the 4-parameter exponential model.

### 4.1.6 A Simple Model of Cached Disk Access Time

Modern operating systems use the main memory as a disk cache. In such cases, disk access is illustrated as shown in Fig. 4.5. If data are not in the cache, it is necessary to fetch data from the disk. This causes one or more disk accesses, taking time $T_{\text{disk}}$. If data are already in the cache, access is very fast, $T_{\text{cache}} \ll T_{\text{disk}}$. The mean access time $T_a$ of a cached disk depends on the cache hit probability $p$:

$$T_a = pT_{\text{cache}} + (t-p)(T_{\text{cache}} + T_{\text{disk}}) = T_{\text{cache}} + (1-p)T_{\text{disk}}$$

Probability $p$ depends on the data access locality properties and for a large number of accesses, assuming $C < F$, it satisfies the inequality $p \ge C/F$. The lowest hit ratio, $p = C/F$, is obtained in the case of minimum locality, i.e., in the case of uniform distribution of disk accesses. If $T_{\text{disk}} = T_{\text{access}}$, the upper bound of the cached disk access time is a function of the disk file size $F$, which can be modeled as follows:

$$T_a(F) = \begin{cases} T_{\text{cache}}, F \le C \\ T_{\text{cache}} + \left(1 - \frac{C}{F}\right)\left[T_{\text{max seek}}\left(\frac{F}{F_{\max}}\right)^r + \frac{30}{N_{\text{rev}}}\right], F \ge C \end{cases}$$

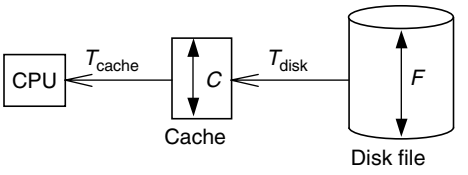



**FIGURE 4.5** File access using a disk cache. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

This model has four adjustable parameters: $T_{\text{cache}}$, $C$, $r$, and $T_{\max\text{ seek}}$. Two adjustable parameters ($T_{\text{cache}}$ and $C$) represent the disk cache, and two parameters ($T_{\max\text{ seek}}$ and $r$) represent the seek time model. The simple 2-parameter seek time model $T = T_{\max\text{ seek}}(F/F_{\max})^r$ yields sufficiently good accuracy for the most frequent cases of relatively small databases where the seek time is nonlinear. For example, in the case of the Quantum Atlas III disk, the nonlinear segment corresponds to data sizes up to (1686/8057)9.1 GB = 1.9 GB. For larger databases (that require the linear segment of the seek time characteristic), we can apply the 3- and 4-parameter models described in the previous section.

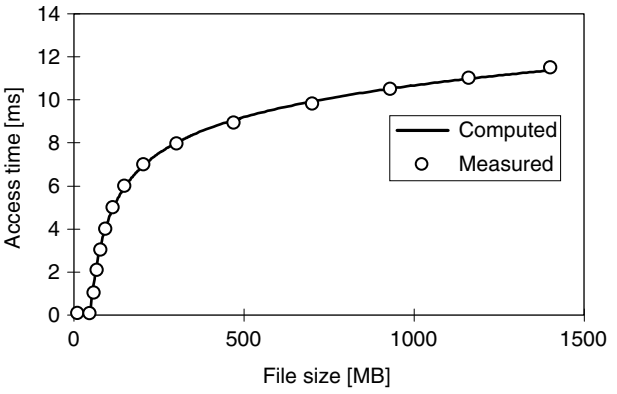



**FIGURE 4.6** Measured and computed access times. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol.2, pp. 670–679, 1999. With permission.)

If we measure disk access times $T_1, \ldots, T_n$ for a sequence of file sizes $F_1, \ldots, F_n$, the calibration of the above model can be performed by selecting the optimum values of $T_{\text{cache}}$, $C$, $r$, and $T_{\text{max seek}}$, which minimize the criterion function

$$E(T_{\text{cache}}, C, R, T_{\text{max seek}}) = \sum_{i=1}^{n} \left| T_i - T_a(F_i, T_{\text{cache}}, C, r, T_{\text{max seek}}) \right|^q$$

The exponent $q$ is usually selected in the range $1 \leq q \leq 4$, where larger values are selected in cases where the primary goal is to minimize large errors. The minimization can be performed using the traditional Nelder–Mead simplex method [9].

A verification of this model is shown in Figs. 4.6 and 4.7 for a 300 MHz PC with 64 MB of memory, Windows NT 4.5, and a 4 GB SCSI disk that has $N_{\text{nev}} = 7200$ rev/min. The resulting parameters are $T_{\text{cache}} = 96$ μs, $C = 48$ MB, $r = 0.234$, and $T_{\text{max seek}} = 7.51$ ms for $F_{\text{max}} = 1400$ MB. In the majority

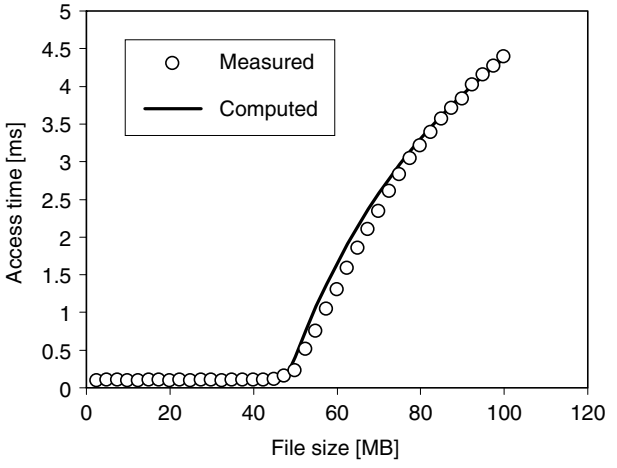



**FIGURE 4.7** Measured and computed access times showing the cache size of 48 MB (magnified detail of Fig. 4.5). (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

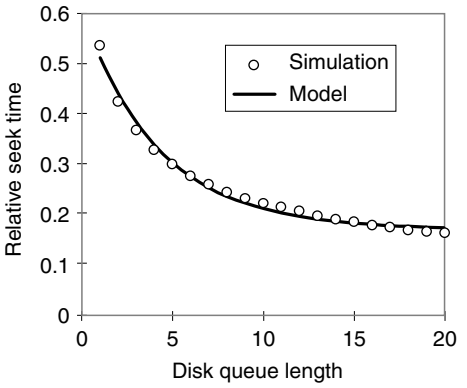



**FIGURE 4.8** Comparing simulation and analytic models of seek time. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

of 245 measured points (only some of them shown in Fig. 4.6), the mean relative error of this model is less than 1%.

#### 4.1.6.1 Disk Access Optimization Model

The disk access time model described in the previous section assumes a single program that generates disk access requests. In such a case, all requests are served strictly in the order they are submitted and disk access optimization is not possible; however, in the case of multiprogramming, the disk queue contains multiple requests independently generated by various programs. It is possible to use a disk access optimization algorithm that increases the global disk throughput by minimizing the movement of I/O head mechanism. The simplest of such an algorithm is the shortest seek time first (SSTF) [10,14], which can be easily analyzed.

A simulator for the SSTF technique can be easily written according to the following algorithm:

1. Create a disk queue with $n$ random requests.
2. Select an arbitrary initial position of disk heads.
3. Find the request that yields the shortest distance $x$.
4. Compute the relative seek time $t = \sqrt{x/x_{\max}}$ (this is a normalized time with a maximum value of 1).
5. The position of the processed request becomes the new position of the disk head mechanism.
6. Replace the processed request with a new random request.
7. Repeat steps 3–6 many times and compute the average relative (normalized) seek time.

The results of the simulation yield a decreasing relative seek time function presented in Fig. 4.8. Taking this function into account, the optimized disk access time can be well approximated, using the following model:

$$T_a(n) = t_{\min} + (t_{\max} - t_{\min})\ \mathrm{e}^{\alpha(n-1)}$$

The parameter $t_{\min}$ is introduced to reflect the mean latency time. The value of $t_{\max}$ corresponds to the maximum seek distance. The quality of this approximation is rather good, as shown in Fig. 4.8. The exponent $\alpha$ has a negative value, which reflects the quality of the optimization algorithm (as the quality of optimization increases, so does the absolute value of the exponent $\alpha$).

#### 4.1.6.2 Disk Service Time Model

Generally, the mean disk access time is a decreasing function of the disk queue length and is an increasing function of file size. If we want to combine the disk cache access model and the disk access optimization model, the result can be the following formula for load-dependent and cached disks:

$$T_a(F,n) = \begin{cases} T_{\text{cache}}, & F \le C \\ T_{\text{cache}} + \dfrac{F-C}{F}\left[T_{\max\ \text{seek}}\left(\dfrac{F}{F_{\max}}\right)^r \mathrm{e}^{\alpha(n-1)} + \dfrac{30}{N_{\text{rev}}}\right], & F \ge C \end{cases}$$

The disk service time is different from the mean disk access time. Disk service time is affected by caching and access optimization, but it includes only the cases of actual disk access, excluding the cases of fetching data from the cache without disk access. Consequently, we propose the following model of the load-dependent cached disk service time:

$$S_d(F,n) = T_{\max\ \text{seek}}\left(\frac{F}{F_{\max}}\right)^r e^{\alpha(n-1)} + \frac{30}{N_{\text{rev}}} + t_0, \quad F > C$$

The first term in this model reflects the optimized seek time, the second term is the latency time, and the third term reflects the data transfer time and related cache I/O operations. This model is successfully applied for the analysis of a cached system presented in Section 4.1.7. The presented load-dependent service time model can also be developed using other seek time models described in Section 4.1.5.

#### 4.1.6.3 Disk Subsystem Benchmark Workload

Disk subsystem workloads always lie between two obvious extremes: sequential access and uniformly distributed random access. Sequential access is more frequent, but random access is more general because it includes seek operations. Similarly, benchmark workloads must balance write and read operations. This balance is based on two facts: (1) read operations are generally more frequent than write operations and (2) write operations are less desirable in benchmarking because performance results are rather sensitive to tuning of disk formats or blocking factors, which frequently yields questionable results. Finally, the benchmark workload can be symmetric (i.e., balanced load, where all disks have the same load) and asymmetric (usually with bottleneck disks). Symmetric loads are more desirable in benchmarking because they expose the capabilities of disk controllers and central processors better.

A simple classification of disk workloads is presented in Table 4.1. Workload type 0 or 1 can be used for creating files that are then processed by other workloads. Workload types 2 and 3 are used for benchmarking systems using sequential access. Similarly, workload types 6 and 7 are used for benchmarking, using random access. We use workload type 6 as the basic workload for measurement of disk subsystem performance. It consists of $n$ copies of a disk random access program (DRAN), which uniformly accesses files that are uniformly distributed over all disk units [15]. This is a simple balanced workload that should properly reflect disk subsystem performance and be suitable for both performance measurement and modeling. Typical results of running the DRAN workload type 6 are presented in Fig. 4.9.

**TABLE 4.1** Classification of Eight Basic Disk Workloads

| | Access Method | Operation | Load Balance |
|---|---|---|---|
| Workload Type | (S = Sequential, R = Random) | (W = Write, R = Read) | (S = Symmetric, A = Asymmetric) |
| 0 | S | W | S |
| 1 | S | W | A |
| 2 | S | R | S |
| 3 | S | R | A |
| 4 | R | W | S |
| 5 | R | W | A |
| 6 | R | R | S |
| 7 | R | R | A |

*Source*: Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.

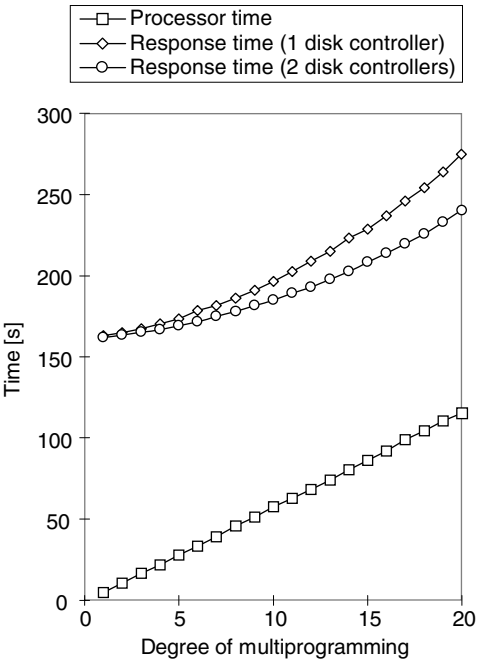



**FIGURE 4.9** Measured response times for VAX 8650. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

The analyzed computer, VAX 8650, has 14 disk units (RA 81) and one or two disk controllers (HSC 50). Each program generates 7000 visits to the central processor, and 7000 visits to disks. Since the disk load is uniformly distributed, each program creates 500 accesses to each disk. In the case of a single program, the measured processor time for a single disk controller is 4.99 s and the total elapsed time is 163.16 s. The processor time can be interpreted as processor demand $D_p = 4.99$ s. A queuing model of this system, in the case of two disk controllers, is shown in Fig. 4.10. The next step is to develop an analytic model for the analysis of this system.

### 4.1.7 MVA Models and Their Limitations

Let us introduce the following variables:

$N$ = degree of multiprogramming (number of jobs in the system)
$K$ = number of service centers (processors, disks, and disk controllers)
$P_k(j|n)$ = probability that there are $j$ jobs at the $k$th service center, if the total number of jobs in the system (degree of multiprogramming) is $n$
$R_k(n)$ = response time of the $k$th service center in the case of $n$ jobs in the system ($n = 1, \ldots, N$)
$R(n)$ = response time of the whole system
$U_k(n)$ = utilization of the $k$th server
$Q_k(n)$ = queue length at the $k$th server
$S_k(j)$ = service time of the $k$th server if $j$ jobs are in the queue; if the $k$th server is load independent then $S_k(j) = S_k =$ constant, $k \in \{1, \ldots, K\}$
$V_k$ = number of visits to the $k$th server per job
$D_k$ = service demand for the $k$th server; in the case of load-independent servers, $D_k = V_k S_k$

$X(n)$ = system throughput (completed jobs per time unit)
$X_k(n)$ = throughput of the $k$th service center when there are $n$ jobs in the system, $X_k(n) = V_k\, X(n)$

The traditional load-independent mean value analysis program (LIMVA) is based on assumption that the service times of all servers are constant. The goal of MVA models is to compute system response times, utilizations, and throughputs. Following is the traditional LIMVA model:

$$Q_k(0) = 0, \quad k = 1, \ldots, K$$

**for** $n = 1$ **to** $N$ **do**

$$R_k(n) = S_k[1 + Q_k(n-1)], \quad k = 1, \ldots, K$$
$$R(n) = \sum_{k=1}^{K} V_k R_k(n)$$
$$X(n) = n/R(n)$$
$$X_k(n) = V_k X(n), \quad k = 1, \ldots, K$$
$$U_k(n) = S_k X_k(n) = D_k X(n), \quad k = 1, \ldots, K$$
$$Q_k(n) = V_k X(n) R_k(n), \quad k = 1, \ldots, K$$

**end_for**

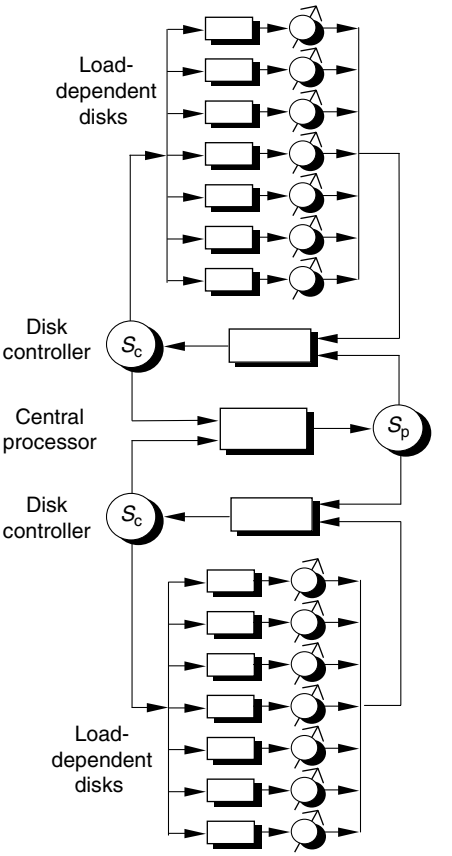



**FIGURE 4.10** A queuing model of VAX 8650. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

For perfectly balanced systems, where all demands are equal ($D = V_1S_1 = V_2S_2 = \cdots = V_KS_K$), this algorithm yields equal distribution of jobs in service centers

$$Q_k(n) = n/K, \quad k = 1, \ldots, K$$

This is a consequence of equal residence times

$$V_k R_k(n) = D_k[1 + Q_k(n-1)] = D[1 + (n-1)/K], \quad k = 1, \ldots, K$$

and their use for computing $Q_k(n) = V_k X(n) R_k(n)$. Furthermore, this yields linear response times and other relations:

$$R(n) = KV_k R_k(n) = (n - 1 + K)D$$
$$R_k(n) = (n - 1 + K)S_k/K, \quad k = 1, \ldots, K$$
$$X(n) = n/(n - 1 + K)D$$
$$X_k(n) = n/(n - 1 + K)S_k, \quad k = 1, \ldots, K$$
$$U_k(n) = n/(n - 1 + K), \quad k = 1, \ldots, K$$

Of course, in a general case we have different demands, and the response time is no longer linear. Unfortunately, the nature of the LIMVA model is essentially quasi-linear. Even for different demands, the response time curves remain similar to straight lines.

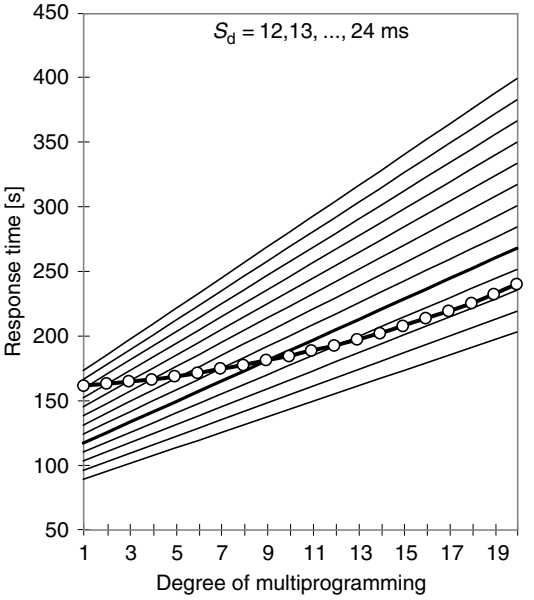



**FIGURE 4.11** A family of LIMVA models and measured VAX 8650 response time. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

Limitations of the LIMVA model are exemplified in Fig. 4.11. In this case the processor service time obtained from measurements is $S_p = 825$ μs, the number of processor visits per job is $V_p = 7000$, yielding processor demand $D_p = 5.775$ s. The number of disks = 14 and the number of disk visits per job is $V_d = 7000/14 = 500$. The available resources include the central processor and 14 equally loaded disk units. A spectrum of LIMVA models can be obtained for disk service times varying in the range 12 ms $\leq S_d \leq$ 24 ms, and yielding disk demands in the range, 6 s $\leq D_d \leq$ 12 s. The corresponding response times presented in Fig. 4.11 are, in the whole range, practically straight lines and obviously inadequate for representing the measured response time function. The best approximation would be obtained for $S_d = 16$ ms, but this approximation is equally poor as the attempt to use a straight line to approximate a parabola. The nature of the dynamic behavior of VAX 8650 is quite different from what can be modeled by LIMVA regardless how well we adjust its parameters. Thus, a more flexible model is needed. Taking into account that disks are never load independent, because they regularly use either access optimization or access optimization and caching, we hope that better results should be expected from load-dependent MVA models.

For batch systems we apply the load-dependent mean value analysis model (LDMVA) introduced in Ref. [5] (see also [1,2]):

$$P_k(0|0) = 1, \quad k = 1, \ldots, K$$

**for** $n = 1$ **to** $N$ **do**

$$R_k(n) = \sum_{j=1}^{n} jS_k(j)P_k(j-1|n-1), \quad k = 1, \ldots, K$$

$$R(n) = \sum_{k=1}^{K} V_k R_k(n)$$

$$X(n) = n/R(n)$$

$$P_k(j|n) = \left\{ \begin{array}{ll} V_k S_k(j) X(n) P_k(j-1|n-1), & j = 1, \ldots, n \\ 1 - \sum_{i=1}^{n} P_k(i|n), & j = 0 \end{array} \right\}, \quad k = 1, \ldots, K$$

$$Q_k(n) = \sum_{j=1}^{n} j P_k(j|n), \quad k = 1, \ldots, K$$

$$U_k(n) = 1 - P_k(0|n), \quad k = 1, \ldots, K$$

**end_for**

### 4.1.8 Experimental Results for LDMVA Model of a Disk Subsystem with Access Optimization

Let us now apply the LDMVA model for the VAX 8650 disk subsystem modeling. The measured response and processor times can be used to adjust parameters of the LDMVA model. This is called calibration of the queuing model. The calibration process is based on measured response times $T_{m1}(n)$ and $T_{m2}(n)$, which correspond to systems with one and two disk controllers. Suppose that the load-dependent disk service time is approximated by $S_d(n) = t_{min} + (t_{max} - t_{min}) e^{\alpha(n-1)}$. The parameters of the LDMVA model are $t_{min}$, $t_{max}$, $\alpha$, $S_p$, $S_c$. Let $R_{c1}(n, t_{min}, t_{max}, \alpha, S_p, S_c)$ and $R_{c2}(n, t_{min}, t_{max}, \alpha, S_p, S_c)$ be the response times computed from the LDMVA model, respectively using one and two controllers. The model calibration procedure is based on the minimization of the compound criterion function

$$C(t_{min}, t_{max}, \alpha, S_p, S_c) = \max \left\{ \sum_{n=1}^{n_{max}} [T_{m1}(n) - R_{c1}(n, t_{min}, t_{max}, \alpha, S_p, S_c)]^2, \sum_{n=1}^{n_{max}} [T_{m2}(n) - R_{c2}(n, t_{min}, t_{max}, \alpha, S_p, S_c)]^2 \right\}$$

The results of calibration performed using the Nelder–Mead simplex method [9] are the following values of parameters: $t_{min} = 11.5$ ms, $t_{max} = 20$ ms, $\alpha = -4$, $S_p = 0.822$ ms, and $S_c = 0.89$ ms. The mean relative error for all presented points is 1% (Fig. 4.12). At this level of description error, it is realistic to expect good prediction results.

### 4.1.9 Experimental Results for LDMVA Model of a Disk Subsystem with Caching and Access Optimization

In this experiment, we use the same 300 MHz PC presented in Section 4.1.5 and Figs. 4.6 and 4.7. Its memory capacity is 64 MB with two disk units, and the disk cache size under Windows NT 4.5 is $C = 48$ MB.

The measured response times for DRAN benchmarks accessing a 100 MB file and 1 GB file are presented in Fig. 4.13. The LDMVA model we used is based on measured disk parameters reported in Section 4.1.5 and on the disk service time model $S_d(F,n)$ proposed in Section 4.1.6.1. Disk accesses to a file of size $F$ occur with the probability $(F - C)/F$ and cache accesses with the probability $C/F$. The number of disk visits $V_d$ now depends on the number of processor visits $V_p$ and the size of file. If the number of disks is $k$ then $V_d(F) = V_p(F - C)/kF$. Therefore, the disk cache causes both the disk service time and the number of visits to be functions of the file size. Processor service time is not constant. For cache accesses, it can be expressed as $S_p = t_p^{prog} + t_p^{cache}$ and for disk accesses as $S_p = t_p^{prog} + t_p^{cache} + t_p^{disk}$, where the three components correspond to the processor activity for the benchmark program, cache access, and serving the file management system during the disk access. The mean service

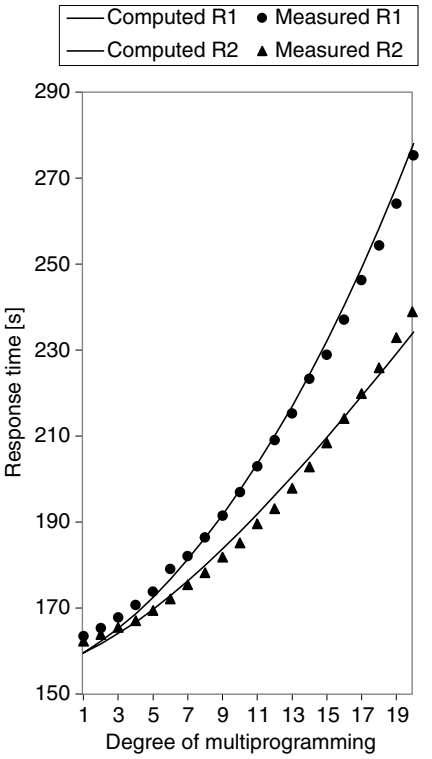



**FIGURE 4.12** The results of LDMVA model calibration. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

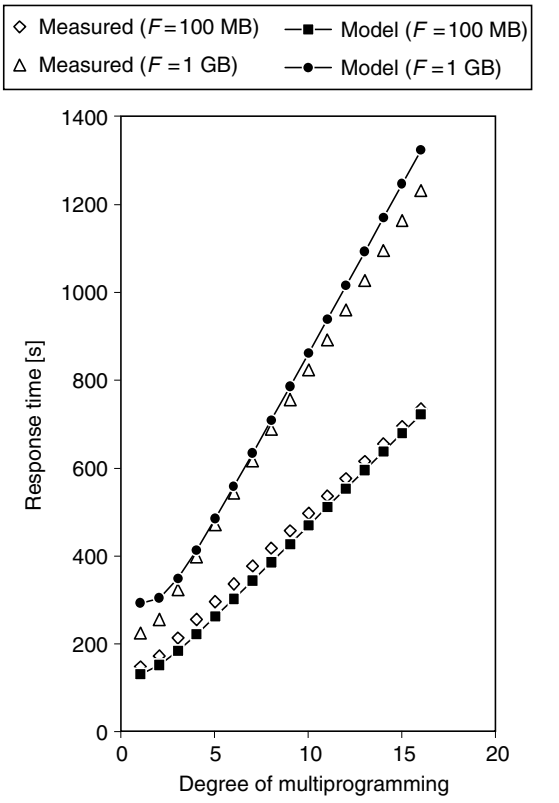



**FIGURE 4.13** Measured response times and the results of the LDMVA model. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

time is $S_p = t_p^{prog} + t_p^{cache} + t_p^{disk}(F - C)/F$. The calibrated model in Fig. 4.13 has the mean modeling error of 7.4%.

### 4.1.10 Predictive Power of Queuing Models

All queuing models have adjustable parameters (e.g., the processor and disk service times [16]). The default values of these parameters, taken from manufacturer specifications, regularly yield rather large prediction errors. These errors can be reduced by corrections of parameters in a calibration process. During the model calibration, the parameters are adjusted to minimize the difference between the measured values and computed values from the model [17]. In essence, this is just a standard curve fitting process, and the resulting low description error is not necessarily a proof of the quality of the model. The only way to assess the quality of the model is through the analysis of the prediction errors.

Let $n$ be the number of measured response times $R_1, \ldots, R_n$ and let us use the first $m$ measured values, $R_1, \ldots, R_m$, $m \leq n$, for the model calibration. The indicator $p = 100\ m/n$ shows the percent of values used for calibration. Let $e(p)$ denote the average relative error between the measured vales and the values of the calibrated model for the whole range of $n$ data points. Generally, $e(p)$ is expected to be a decreasing function, and three typical such functions are presented in Figs. 4.14 and 4.15.

In all cases, we measured the processor time and used this value in our LDMVA model. The calibration process included three parameters of the disk subsystem (minimum service time $t_{\min}$, maximum service time $t_{\max}$, and the exponent $\alpha$ introduced in Section 4.1.6). Consequently, the $e(p)$ function starts with $m = 3$ points.

In the ideal theoretical case of a perfect model, we expect $e(p) = 0$ for all $m \geq 3$. Good predictive power of a model is indicated by $e(p)$, approaching a constant low value as soon as possible. Consequently, the predictive power is related to the smallest value of $p$ after which $e(p)$ remains in the $\delta$ neighborhood of the minimum $e_{\min} = \min_{p \leq 100\%} e(p)$. Let us denote this value as $p^*(\delta)$. In other words

$$e(p) > (1 + \delta)e_{\min}, \quad \forall p \leq p^*(\delta)$$

Now, we can define the following predictive power quality indicator:

$$q(\delta) = \frac{1 - p^*(\delta)}{1 - p_{\min}} 100\%$$

where $p_{\min}$ denotes the minimum value of $p$ necessary for calibration (in our examples $p_{\min}$ corresponds to $m = 3$).

The case in Fig. 4.14 reaches the minimum error $e(100\%) = 2.16\%$. Let us take $\delta = 0.2$. Then we have $(1 + \delta)e_{\min} = 2.59\%$, $p^*(0.2) = 0.875(87.5\%)$, and the resulting predictive power is rather low:

$$q(0.2) = \frac{1 - 0.875}{1 - 0.375} \times 100 = 20\%$$

In the case of Fig. 4.15, the model with 1 HSC50 disk controller yields $e_{\min} = 1.53\%$, and for $\delta = 0.2$ we have

$$(1 + \delta)e_{\min} = 1.84\%, \quad p^*(0.2) = 0.7,$$
$$q(0.2) = 35.3\%$$

Finally, the predictive power in the case with two HSC50 disk controllers is much better: we have $e_{\min} = 0.64\%$ yielding

$$(1 + \delta)e_{\min} = 0.77\%,$$
$$p^*(0.2) = 0.35, \quad q(0.2) = 76.5\%$$

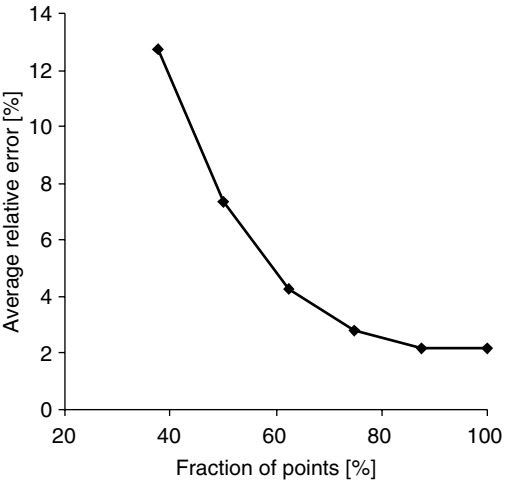



**FIGURE 4.14** Prediction errors $e(p)$ for VAX 11/785. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

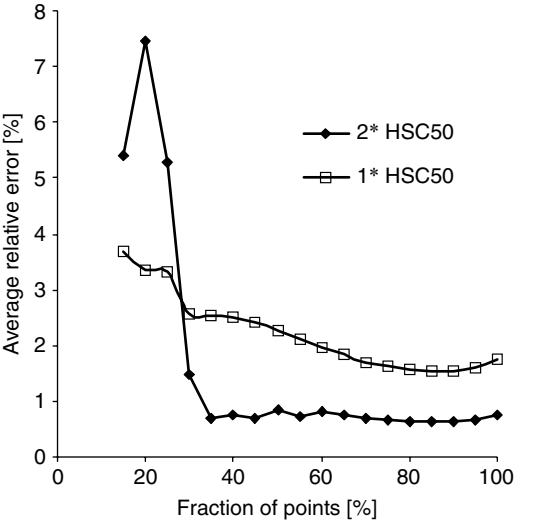



**FIGURE 4.15** Prediction errors $e(p)$ for VAX 8650. (From Dujmovic, J.J., Tomasevich, D., and Au-Yeung, M., *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Vol. 2, pp. 670–679, 1999. With permission.)

### 4.1.11 Conclusions

Even in cases of fully controlled simple synthetic workloads, the performance modeling of disk subsystems is not a simple task. Basic popular load-independent queuing models (load-independent convolution algorithm or MVA) cannot be used for modeling modern disk subsystems, which use caching and disk access optimization. The use of load-dependent MVA models is rather efficient and we proposed models for optimized disk access, cached disk access, and combined optimized and cached disk access. Presented models need a careful calibration procedure. In the majority of cases based on symmetric random disk accesses, the modeling errors for optimized disk access were less than 2%. In more complex cases with optimized disk accesses and caching, our models regularly achieve errors below 10%. Experimental verification of our models has been successfully performed in both VAX/VMS and PC/NT environments.

## References


1. Jain, R., *The Art of Computer System Performance Analysis*. John Wiley & Sons, New York, 1991.
2. Menasce, D., V. Almeida, and L. Dowdy, *Capacity Planning and Performance Modeling*. Prentice-Hall, Englewood Cliffs, NJ, 1994.
3. Buzen, J.P., Computational algorithms for closed queuing networks with exponential servers. *Communications of the ACM*, 16(9): 527–531, 1973.
4. Reiser, M. and S.S. Lavenberg, Mean-value analysis of closed multichain queuing networks. *Journal of the ACM*, 27(2): 313–322, 1980.
5. Basket, F., K. Chandy, R. Munz, and F. Palacios, Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2): 248–260, 1975.
6. Worthington, B.L., G.R. Ganger, Y.N. Patt, and J. Wilkes, *On-line extraction of SCSI disk drive parameters*. Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Performance Evaluation Review, Ottawa, Canada, Vol. 23, pp. 146–156, May 1995.
7. Ruemmler, C. and J. Wilkes, An introduction to disk drive modeling. *IEEE Computer*, 27(3): 17–28, 1994.
8. IBM, *Hard disk drive specifications—Deskstar 60GPX*. S07N-4780–02, Publication #2818, February 1, 2001.
9. Nelder, J.A. and R. Mead, A simplex method for function minimization. *Computer Journal*, 7(4): 308–313, 1965.
10. Silbershatz, A. and P.B. Galvin, *Operating System Concepts*. Addison–Wesley, Reading, MA 1994.
11. Ng, S.W., Advances in disk technology: Performance issues. *IEEE Computer*, 40(1): 75–81, 1998.
12. Lee, E.K. and R.H. Katz, *An analytic performance model of disk arrays*. Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, pp. 98–109, May 1993.
13. Shriver, E., *Performance modeling for realistic storage devices*. Ph.D. Thesis, Department of Computer Science, New York University, New York, May 1997.
14. Worthington, B.L., G.R. Ganger, and Y.N. Patt, *Scheduling algorithms for modern disk drives*. Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Performance Evaluation Review, Nashville, TN, Vol. 22, pp. 241–251, May 1994.
15. Dujmovic, J.J., *Multiprogramming efficiency analysis for computer evaluation and selection studies*. Proceedings of the 11th International Symposium Computer at the University, Cavtat, Croatia, 1989.
16. Dujmovic, J.J., D. Tomasevich, and M. Au-Yeung, *Measurement and modeling of disk subsystem performance*, 25th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 1999 Proceedings, Reno, NV, Vol. 2, pp. 670–679, 1999.
17. Dujmovic, J.J. and D. Tomasevich, *Calibration and comparison of disk unit models*, 27th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems, CMG 2001 Proceedings, Anaheim, CA, Vol. 1, pp. 315–325, 2001.

# 4.2 Performance Evaluation: Techniques, Tools, and Benchmarks

*Lizy Kurian John*

## 4.2.1 Introduction

State-of-the-art high performance microprocessors contain tens of millions of transistors and operate at frequencies close to 2 GHz. These processors perform several tasks in overlap, employ significant amounts of speculation and out-of-order execution, and other microarchitectural techniques, and are true marvels of engineering. Designing and evaluating these microprocessors is a major challenge, especially considering the fact that one second of program execution on these processors involves several billion instructions and analyzing one second of execution may involve dealing with tens of billion pieces of information.

In general, design of microprocessors and computer systems involves several steps (i) understanding applications and workloads that the systems will be running, (ii) innovating potential designs, (iii) evaluating performance of the candidate designs, and (iv) selecting the best design. The large number of potential designs and the constantly evolving nature of workloads have resulted in designs being largely adhoc. In this article, we investigate major techniques used in the performance evaluation process.

It should be noted that performance evaluation is needed at several stages of the design. In early stages, when the design is being conceived, performance evaluation is used to make early design tradeoffs. Usually, this is accomplished by simulation models, because building prototypes of state-of-the-art microprocessors is expensive and time consuming. Several design decisions are made before any prototyping is done. Once the design is finalized and is being implemented, simulation is used to evaluate functionality and performance of subsystems. Later, performance measurement is done after the product is available in order to understand the performance of the actual system to various real world workloads and to identify modifications to incorporate in future designs.

Performance evaluation can be classified into performance modeling and performance measurement, as illustrated in Table 4.2. Performance measurement is possible only if the system of interest is available for measurement and only if one has access to the parameters of interest. Performance measurement may further be classified into on-chip hardware monitoring, off-chip hardware monitoring, software monitoring, and microcoded instrumentation. Performance modeling is typically used when actual systems are not available for measurement or, if the actual systems do not have test points to measure every detail of interest. Performance modeling may further be classified into simulation modeling and analytical modeling. Simulation models may further be classified into numerous categories depending

**TABLE 4.2** A Classification of Performance Evaluation Techniques

| | |
|---|---|
| Performance measurement | Microprocessor on-chip performance monitoring counters |
| | Off-chip hardware monitoring |
| | Software monitoring |
| | Micro-coded instrumentation |
| Performance modeling | Simulation |
| | Trace driven simulation |
| | Execution driven simulation |
| | Complete system simulation |
| | Event driven simulation |
| | Software profiling |
| | Analytical modeling |
| | Probabilistic models |
| | Queuing models |
| | Markov models |
| | Petri net models |

on the mode/level of detail of simulation. Analytical models use probabilistic models, queueing theory, Markov models or Petri nets.

Performance modeling/measurement techniques and tools should possess several desirable features.

- They must be accurate. It is easy to build models that are heavily sanitized, however, such models will not be accurate.
- They must be noninvasive. The measurement process must not alter the system or degrade the system's performance.
- They must not be expensive. Building the performance measurement facility should not cost significant amount of time or money.
- They must be easy to change or extend. Microprocessors and computer systems constantly undergo changes and it must be easy to extend the modeling/measurement facility to include the upgraded system.
- They must not need source code of applications. If tools and techniques necessitate source code, it will not be possible to evaluate commercial applications where source is not often available.
- They should measure all activity including kernel and user activity. Often it is easy to build tools that measure only user activity. This was acceptable in traditional scientific and engineering workloads; however, in database, Web server, and Java workloads, significant operating system activity exists, and it is important to build tools that measure operating system activity as well.
- They should be capable of measuring a wide variety of applications including those that use signals, exceptions, and DLLs (dynamically linked libraries).
- They should be user-friendly. Hard-to-use tools often are underutilized. Hard-to-use tools also result in more user error.
- They should be fast. If a performance model is very slow, long-running workloads, which take hours to run, may take days or weeks to run on the model. If an instrumentation tool is slow, it can be invasive.
- Models should provide control over aspects that are measured. It should be possible to selectively measure what is required.
- Models and tools should handle multiprocessor systems and multithreaded applications. Dual and quad-processor systems are very common nowadays. Applications are becoming increasingly multithreaded especially with the advent of Java, and it is important that the tool handles these.
- It will be desirable for a performance evaluation technique to be able to evaluate the performance of systems that are not yet built.

Many of these requirements are often conflicting. For instance, it is difficult for a mechanism to be fast and accurate. Consider mathematical models. They are fast, although several simplifying assumptions go into their creation, and often they are not accurate. Similarly, it is difficult for a tool to be noninvasive and user-friendly. Many users like graphical user interfaces (GUIs), however, most instrumentation and simulation tools with GUIs are slow and invasive.

Benchmarks and metrics to be used for performance evaluation have always been interesting and controversial issues. There has been a lot of improvement in benchmark suites since 1988. Before that computer performance evaluation has been largely with small benchmarks such as kernels extracted from applications (e.g., Lawrence Livermore Loops), Dhrystone and Whetstone benchmarks, Linpack, Sorting, Sieve of Eratosthenes, 8-queens problem, Tower of Hanoi, etc. [1]. The Standard Performance Evaluation Cooperative (SPEC) consortium and the Transactions Processing Council (TPC) formed in 1988 have made available several benchmark suites and benchmarking guidelines to improve the quality of benchmarking. Several state-of-the-art benchmark suites are described in Section 4.2.4.

Another important issue in performance evaluation is the choice of performance metric. For a system level designer, execution time and throughput are two important performance metrics. Execution time is generally the most important measure of performance. Execution time is the product of the number of

instructions, cycles per instruction (CPI), and the clock period. Throughput of an application is a more important metric, especially in server systems. In servers that serve the banking industry, airline industry, or other similar businesses, what is important is the number of transactions that could be completed in unit time. Such servers, typically called transaction processing systems use transactions per minute (tpm) as a performance metric. Millions of instructions per second (MIPS) and million of floating point operations per second (MFLOPS) were very popular measures of performance in the past. Both of these are very simple and straightforward to understand and hence have been used often, however, they do not contain all three components of program execution time and hence are incomplete measures of performance. Several low level metrics are of interest to microprocessor designers, in order to help them identify performance bottlenecks and tune their designs. Cache hit ratios, branch misprediction ratios, number of off-chip memory accesses, etc. are examples of such measures.

Another major problem is the issue of reporting performance with a single number. A single number is easy to understand and easy to be used by the trade press. Use of several benchmarks also makes it necessary to find some kind of a mean. Arithmetic mean, geometric mean, and harmonic mean are three ways of finding the central tendency of a group of numbers; however, it should be noted that each of these means should be used in appropriate conditions depending on the nature of the numbers which need to be averaged. Simple arithmetic mean can be used to find average execution time from a set of execution times. Geometric mean can be used to find the central tendency of metrics that are in the form of ratios (e.g., speedup) and harmonic mean can be used to find the central tendency of measures that are in the form of a rate (e.g., throughput). Cragon [2] and Smith [3] discuss the use of the appropriate mean for a given set of data. Cragon [2] and Patterson and Hennessy [4] illustrate several mistakes one could possibly make while finding a single performance number.

The rest of this chapter section is organized as follows. Section 4.2.2 describes performance measurement techniques including hardware on-chip performance monitoring counters on microprocessors. Section 4.2.3 describes simulation and analytical modeling of microprocessors and computer systems. Section 4.2.4 presents several state-of-the-art benchmark suites for a variety of workloads. Due to space limitations, we describe some typical examples of tools and techniques and provide the reader with pointers for more information.

## 4.2.2 Performance Measurement

Performance measurement is used for understanding systems that are already built or prototyped. Performance measurement can serve two major purposes: tune this system or systems to be built and tune the application if source code and algorithms can still be changed. Essentially, the process involves (i) understanding the bottlenecks in the system that has been built, (ii) understanding the applications that are running on the system and the match between the features of the system and the characteristics of the workload, and (iii) innovating design features that will exploit the workload features. Performance measurement can be done via the following means:

- Microprocessor on-chip performance monitoring counters
- Off-chip hardware monitoring
- Software monitoring
- Microcoded instrumentation

### 4.2.2.1 On-Chip Performance Monitoring Counters

All state-of-the-art high performance microprocessors including Intel's Pentium III and Pentium IV, IBM's POWER 3 and POWER 4 processors, AMD's Athlon, Compaq's Alpha, and Sun's UltraSPARC processors incorporate on-chip performance monitoring counters, which can be used to understand performance of these microprocessors, while they run complex, real-world workloads. This ability has overcome a serious limitation of simulators, that they often could not execute complex workloads. Now, complex run-time systems involving multiple software applications can be evaluated and monitored

**TABLE 4.3** Examples of Events That Can Be Measured Using Performance Monitoring Counters on an Intel Pentium III Processor

| Event | Description of Event | Event Number in Hex |
|---|---|---|
| DATA_MEM_REFS | All loads and stores from/to memory | 43H |
| DCU_LINES_IN | Total lines allocated in the data cache unit | 45H |
| IFU_IFETCH | Number of instruction fetches (cacheable and uncacheable) | 80H |
| IFU_IFETCH_MISS | Number of instruction fetch misses | 81H |
| ITLB_MISS | Number of Instruction TLB misses | 85H |
| IFU_MEM_STALL | Number of cycles instruction fetch is stalled for any reason | 86H |
| L2_IFETCH | Number of L2 instruction fetches | 28H |
| L2_LD | Number of L2 data loads | 29H |
| L2_ST | Number of L2 data stores | 2AH |
| L2_LINES_IN | Number of lines allocated in the L2 | 24H |
| L2_RQSTS | Total number of L2 requests | 2EH |
| INST_RETIRED | Number of instructions retired | C0H |
| UOPS_RETIRED | Number of micro-operations retired | C2H |
| INST_DECODED | Number of instructions decoded | D0H |
| RESOURCE_STALLS | Number of cycles in which there is a resource related stall | A2H |
| MMX_INSTR_EXEC | Number of MMX Instructions Executed | B0H |
| BR_INST_RETIRED | Number of branch instructions retired | C4H |
| BR_MISS_PRED_RETIRED | Number of mispredicted branches retired | C5H |
| BR_TAKEN_RETIRED | Number of taken branches retired | C9H |
| BR_INST_DECODED | Number of branch instructions decoded | E0H |
| BTB_MISSES | Number of branches for which BTB did not predict | E2H |

very closely. All microprocessor vendors nowadays release information on their performance monitoring counters, although they are not part of the architecture.

For illustration of on-chip performance monitoring, we use the Intel Pentium processors. The microprocessors in the Intel Pentium contain two performance monitoring counters. These counters can be read with special instructions (e.g., RDPMC) on the processor. The counters can be made to measure user and kernel activity in combination or in isolation. A variety of performance events can be measured using the counters [50]. For illustration of the nature of the events that can be measured, Table 4.3 lists a small subset of the events that can be measured on the Pentium III. Although more than 200 distinct events can be measured on the Pentium III, only two events can be measured simultaneously. For design simplicity, most microprocessors limit the number of events that can be simultaneously measured to 4 or 5. At times, certain events are restricted to be accessible only through a particular counter. These steps are necessary to reduce the overhead associated with on-chip performance monitoring. Performance counters do consume on-chip real estate. Unless carefully implemented, they can also impact the processor cycle time.

Several tools are available to measure performance using performance monitoring counters. Table 4.4 lists some of the available tools. Intel's *Vtune* software may be used to perform measurements using the Intel processor performance counters [5]. The *P6Perf* utility is a plug-in for Windows NT performance

**TABLE 4.4** Software Packages for Performance Counter Measurement

| Tool | Platform | Reference |
|---|---|---|
| VTune | IA-32 | http://developer.intel.com/software/products/vtune/vtune_oview.htm |
| P6Perf | IA-32 | http://developer.intel.com/vtune/p6perf/index.htm |
| PMON | IA-32 | http://www.ece.utexas.edu/projects/ece/lca/pmon |
| DCPI | Alpha | http://www.research.digital.com/SRC/dcpi/<br>http://www.research.compaq.com/SRC/dcpi/ |
| Perf-mon | UltraSPARC | http://www.sics.se/~mch/perf-monitor/index.html |

monitoring [6]. The Compaq DIGITAL Continuous Profiling Infrastructure (DCPI) is a very powerful tool to profile programs on the Alpha processors [7,8]. The performance monitor *perf-mon* is a small hack that uses the on-chip counters on UltraSPARC-I/II processors to gather statistics [9]. Packages like Vtune perform extensive post-processing and present data in graphical forms; however, extensive post-processing can sometimes result in tools that are somewhat invasive. *PMON* [10] is a counter reading software written by Juan Rubio of the Laboratory for Computer Architecture at the University of Texas. It provides a mechanism to read specified counters with minimal or no perceivable overhead. All these tools measure user and operating system activity. Since everything on a processor is counted, effort should be made to have minimal or no other undesired process running during experimentation. This type of performance measurement can be done on binaries, and no source code is desired.

#### 4.2.2.2 Off-Chip Hardware Measurement

Instrumentation using hardware means can also be done by attaching off-chip hardware, two examples of which are described in this section.

##### *4.2.2.2.1 SpeedTracer from AMD*

AMD developed this hardware tracing platform to aid in the design of its X86 microprocessors. When an application is being traced, the tracer interrupts the processor on each instruction boundary. The state of the CPU is captured on each interrupt and then transferred to a separate control machine where the trace is stored. The trace contains virtually all valuable pieces of information for each instruction that executes on the processor. Operating system activity can also be traced; however, tracing in this manner can be invasive, and may slow down the processor. Although the processor is running slower, external events such as disk and memory accesses still happen in real time, thus looking very fast to the slowed-down processor. Usually, this issue is addressed by adjusting the timer interrupt frequency. Use of this performance monitoring facility can be seen in Merten [11] and Bhargava [12].

##### *4.2.2.2.2 Logic Analyzers*

Poursepanj and Christie [13] use a Tektronix TLA 700 logic analyzer to analyze 3D graphics workloads on AMD-K6-2-based systems. Detailed logic analyzer traces are limited by restrictions on sizes and are typically used for the most important sections of the program under analysis. Preliminary coarse level analysis can be done by performance monitoring counters and software instrumentation. Poursepanj and Christie used logic analyzer traces for a few tens of frames, which covered a second or two of smooth motion [13].

#### 4.2.2.3 Software Monitoring

Software monitoring is often performed by utilizing architectural features such as a trap instruction or a breakpoint instruction on an actual system, or on a prototype. The VAX processor from Digital (now Compaq) had a T-bit that caused an exception after every instruction. Software monitoring used to be an important mode of performance evaluation before the advent of on-chip performance monitoring counters. The primary advantage of software monitoring is that it is easy to do. The primary disadvantage is that the instrumentation can slow down the application. The overhead of servicing the exception, switching to a data collection process, and performing the necessary tracing can slow down a program by more than 1000 times. Another disadvantage is that software monitoring systems, typically, only handle the user activity.

#### 4.2.2.4 Microcoded Instrumentation

Digital used microcoded instrumentation to obtain traces of VAX and Alpha architectures. The ATUM tool [14] used extensively by Digital in the late 1980s and early 1990s uses microcoded instrumentation. This is a technique lying between trapping information on each instruction, using hardware interrupts (traps) or software traps. The tracing system essentially modified the VAX microcode to record all instruction and data references in a reserved portion of memory. Unlike software monitoring, ATUM could trace all processes including the operating system, but this kind of tracing is invasive, and can slow down the system by a factor of 10 without including the time to write the trace to the disk.

### 4.2.3 Performance Modeling

Performance measurement as described in the previous section can be done only if the actual system or a prototype exists. It is expensive to build prototypes for early stage evaluation. Hence, one needs to resort to some kind of modeling in order to study systems yet to be built. Performance modeling can be done using simulation models or analytical models.

#### 4.2.3.1 Simulation

Simulation has become the de facto performance modeling method in the evaluation of microprocessor architectures for several reasons. The accuracy of analytical models in the past has been insufficient for the type of design decisions computer architects wish to make (for instance, what kind of caches or branch predictors are needed), therefore, cycle accurate simulation has been used extensively by architects. Simulators model existing or future machines or microprocessors. They are essentially a model of the system being simulated, written in a high-level computer language such as C or Java, and running on some existing machine. The machine on which the simulator runs is called the host machine and the machine being modeled is called the target machine. Such simulators can be constructed in many ways.

Simulators can be functional simulators or timing simulators. They can be trace driven or execution driven simulators. They can be simulators of components or that of the complete system. Functional simulators simulate functionality of the target processor, and in essence provide a component similar to the one being modeled. The register values of the simulated machine are available in the equivalent registers of the simulator. In addition to the values, the simulators also provide performance information in terms of cycles of execution, cache hit ratios, branch prediction rates, etc. Thus, the simulator is a virtual component representing the microprocessor or subsystem being modeled plus a variety of performance information.

If performance evaluation is the only objective, one does not need to model the functionality. For instance, a cache performance simulator does not need to actually store values in the cache; it only needs to store information related to the address of the value being cached. That information is sufficient to determine a future hit or miss. Although it is nice to have the values as well, a simulator that models functionality in addition to performance is bound to be slower than a pure performance simulator. Register transfer language (RTL) models used for functional verification may also be used for performance simulations, however, these models are very slow for performance estimation with real-world workloads and are not discussed in this article.

##### *4.2.3.1.1 Trace Driven Simulation*

Trace driven simulation consists of a simulator model whose input is modeled as a trace or sequence of information representing the instruction sequence that would have actually executed on the target machine. A simple trace driven cache simulator needs a trace consisting of address values. Depending on whether the simulator is modeling a unified instruction or data cache, the address trace should contain addresses of instruction and data references.

Cachesim5 and Dinero IV are examples of cache simulators for memory reference traces. Cachesim5 comes from Sun Microsystems along with its Shade package [15]. Dinero IV [16] is available from the University of Wisconsin, Madison. These simulators are not timing simulators. There is no notion of simulated time or cycles, only references. They are not functional simulators. Data and instructions do not move in and out of the caches. The primary result of simulation is hit and miss information. The basic idea is to simulate a memory hierarchy consisting of various caches. The various parameters of each cache can be set separately (architecture, mapping policies, replacement policies, write policy, statistics). During initialization, the configuration to be simulated is built up, one cache at a time, starting with each memory as a special case. After initialization, each reference is fed to the appropriate top-level cache by a single simple function call. Lower levels of the hierarchy are handled automatically. One does not need to store a trace while using Cachesim5, because Shade can directly feed the trace into Cachesim5.

Trace driven simulation is simple and easy to understand. The simulators are easy to debug. Experiments are repeatable because the input information is not changing from run to run; however, trace driven simulation has two major problems:

1. Traces can be prohibitively long if entire executions of some real-world applications are considered. The storage needed by the traces may be prohibitively large. Trace size is proportional to the dynamic instruction count of the benchmark.
2. The traces do not represent the actual stream of processors with branch predictions. Most trace generators generate traces of only completed or retired instructions in speculative processors. Hence, they do not contain instructions from the mispredicted path.

The first problem is typically solved using trace sampling and trace reduction techniques. Trace sampling is a method to achieve reduced traces; however, the sampling should be performed in such a way that the resulting trace is representative of the original trace. It may not be sufficient to periodically sample a program execution. Locality properties of the resulting sequence may be widely different from that of the original sequence. Another technique is to skip tracing for a certain interval, then collect for a fixed interval, and then skip again. It may also be needed to leave a warm-up period after the skip interval, to let the caches and other such structures to warm up [17]. Several trace sampling techniques are discussed by Crowley and Baer [18]. The QPT trace collection system [19] solves the trace size issue by splitting the tracing process into a trace record generation step and a trace regeneration process. The trace record has a size similar to the static code size, and the trace regeneration expands it to the actual full trace upon demand.

The second problem can be solved by reconstructing the mispredicted path [20]. An image of the instruction memory space of the application is created by one pass through the trace, and, thereafter, fetching from this image as opposed to the trace. Although 100% of the mispredicted branch targets may not be in the recreated image, studies show that more than 95% of the targets can be located.

#### *4.2.3.1.2 Execution Driven Simulation*

Researchers and practitioners assign two meanings to this term. Some refer to simulators that take program executables as input as execution driven simulators. These simulators utilize the actual input executable and not a trace. Hence, the size of the input is proportional to the static instruction count and not the dynamic instruction count. Mispredicted branches can be accurately simulated as well. Thus, these simulators solve the two major problems faced by trace driven simulators. The widely used Simplescalar simulator [21] is an example of such an execution driven simulator. With this tool set, the user can simulate real programs on a range of modern processors and systems, using fast execution driven simulation. There is a fast functional simulator and a detailed, out-of-order issue processor that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction.

Some others consider execution driven simulators to be simulators that rely on actual execution of parts of code on the host machine (hardware acceleration by the host instead of simulation) [22]. These execution driven simulators do not simulate every individual instruction in the application. Only the instructions that are of interest are simulated. The remaining instructions are directly executed by the host computer. This can be done when the instruction set of the host is the same as that of the machine being simulated. Such simulation involves two stages. In the first stage or preprocessing, the application program is modified by inserting calls to the simulator routines at events of interest. For instance, for a memory system simulator, only memory access instructions need to be instrumented. For other instructions, the only important thing is to make sure that they get performed and that their execution time is properly accounted for. The advantage of execution driven simulation is speed. By directly executing most instructions at the machine's execution rate, the simulator can operate orders of magnitude faster than cycle by cycle simulators that emulate each individual instruction. Tango, Proteus, and FAST are examples of such simulators [22].

#### 4.2.3.1.3 Complete System Simulation

Many execution and trace driven simulators only simulate the processor and memory subsystem. Neither I/O activity nor operating system activity is handled in simulators such as Simplescalar. But in many work-loads, it is extremely important to consider I/O and operating system activity. Complete system simulators are complete simulation environments that model hardware components with enough detail to boot and run a full-blown commercial operating system. The functionality of the processors, memory subsystem, disks, buses, SCSI/IDE/FC controllers, network controllers, graphics controllers, CD-ROM, serial devices, timers, etc. are modeled accurately in order to achieve this. Although functionality stays the same, different microarchitectures in the processing component can lead to different performance. Most of the complete system simulators use microarchitectural models that can be plugged in and out. For instance, SimOS [23], a popular complete system simulator, provides a simple pipelined processor model and an aggressive superscalar processor model. SimOS and SIMICS [24,25] can simulate uniprocessor and multiprocessor systems. Table 4.5 lists popular complete system simulators.

#### 4.2.3.1.4 Stochastic Discrete Event Driven Simulation

It is possible to simulate systems in such a way that the input is derived stochastically rather than as a trace/executable from an actual execution. For instance, one can construct a memory system simulator in which the inputs are assumed to arrive according to a Gaussian distribution. Such models can be written in general purpose languages such as C, or using special simulation languages such as SIMSCRIPT. Languages such as SIMSCRIPT have several built-in primitives to allow quick simulation of most kinds of common systems. Built-in input profiles including resource templates, process templates, queue structures, etc., facilitate easy simulation of common systems. An example of the use of event driven simulators using SIMSCRIPT may be seen in the performance evaluation of multiple-bus multiprocessor systems in Kurian et al. [26,27].

#### 4.2.3.1.5 Program Profilers

Software profiling tools is a class of tools that is similar to simulators and performance measurement tools. These tools are used to generate traces, to obtain instruction mix, and a variety of instruction statistics. They can be thought of as software monitoring on a simulator. They input an executable and decode and analyze each instruction in the executable. These program profilers can be used as the front end of simulators. A popular program profiling tool is Shade for the UltraSparc [15].

*Shade*—Shade is a fast instruction-set simulator for execution profiling. It is a simulation and tracing tool that provides features of simulators and tracers in one tool. Shade analyzes the original program instructions and cross-compiles them to sequences of instructions that simulate or trace the original code. Static cross-compilation can produce fast code, but purely static translators cannot simulate and trace all details of dynamically linked code. One can develop a variety of analyzers to process the information generated by Shade and create the performance metrics of interest. For instance, one can use shade to generate address traces to feed into a cache analyzer to compute hit-rates and miss rates of cache configurations. The Shade analyzer Cachesim5 does exactly this.

**TABLE 4.5** Examples of Complete System Simulators

| Simulator | Information Site | Instruction Set | Operating System |
|---|---|---|---|
| SimOS | Stanford University<br>http://simos.stanford.edu/ | MIPS | SGI IRIX |
| SIMICS | Virtutech<br>http://www.simics.com<br>http://www.virtutech.com | PC, SPARC, and Alpha | Solaris 7 and 8, Red Hat Linux 6.2 (both x86, SPARC V9, and Alpha versions), Tru64 (Digital Unix 4.0F), and Windows NT 4.0 |
| Bochs | http://bochs.sourceforge.net | x86 | Windows 95, Windows NT, Linux, FreeBSD |

*Jaba*—Jaba [46] is a Java Bytecode Analyzer developed at the University of Texas for tracing Java programs. Although Java programs can be traced using shade to obtain profiles of native execution, Jaba can yield profiles at the bytecode level. It uses JVM specification 1.1. It allows the user to gather information about the dynamic execution of a Java application at the Java bytecode level. It provides information on bytecodes executed, load operations, branches executed, branch outcomes, etc. Use of this tool can be found in [47].

A variety of profiling tools exist for different platforms. In addition to describing the working of Shade, Cmelik et al. [15] also compares Shade to several other profiling tools for other platforms. A popular one for the x86 platform is Etch [51]. Conte and Gimarc [52] is a good source of information to those interested in creating profiling tools.

#### 4.2.3.2 Analytical Modeling

Analytical performance models, while not popular for microprocessors, are suitable for evaluation of large computer systems. In large systems, where details cannot be modeled accurately for cycle accurate simulation, analytical modeling is an appropriate way to obtain approximate performance metrics. Computer systems can generally be considered as a set of hardware and software resources and a set of tasks or jobs competing for using the resources. Multicomputer systems and multiprogrammed systems are examples.

Analytical models rely on probabilistic methods, queuing theory, Markov models, or Petri nets to create a model of the computer system. A large body of literature on analytical models of computer exists from the 1970s and early 1980s. Heidelberger and Lavenberg [28] published an article summarizing research on computer performance evaluation models. This article contains 205 references, which cover all important work on performance evaluation until 1984. Readers interested in analytical modeling should read this article.

Analytical models are cost-effective because they are based on efficient solutions to mathematical equations; however, in order to be able to have tractable solutions, often, simplifying assumptions are made regarding the structure of the model. As a result, analytical models do not capture all the detail typically built into simulation models. It is generally thought that carefully constructed analytical models can provide estimates of average job throughputs and device utilizations to within 10% accuracy and average response times within 30% accuracy. This level of accuracy, while insufficient for microarchitectural enhancement studies, is sufficient for capacity planning in multicomputer systems, I/O subsystem performance evaluation in large server farms, and in early design evaluations of multiprocessor systems.

Only a small amount of work has been done on analytical modeling of microprocessors. The level of accuracy needed in trade off analysis for microprocessor structures is more than what typical analytical models can provide; however, some effort into this arena came from Noonburg and Shen [29] and Sorin et al. [30]. Those interested in modeling superscalar processors using analytical models should read Noonburg et al.'s work [29] and Sorin et al.'s work [30]. Noonburg et al. used a Markov model to model a pipelined processor. Sorin et al. used probabilistic techniques to processor a multiprocessor composed of superscalar processors. Queuing theory is also applicable to superscalar processor modeling, as modern superscalar processors contain instruction queues in which instructions wait to be issued to one among a group of functional units.

### 4.2.4 Workloads and Benchmarks

Benchmarks used for performance evaluation of computers should be representative of applications that are run on actual systems. Contemporary computer applications include a variety of applications, and different benchmarks are appropriate for systems targeted for different purposes. Table 4.6 lists several popular benchmarks for different classes of workloads.

#### 4.2.4.1 CPU Benchmarks

**SPEC CPU2000** is the industry-standardized CPU-intensive benchmark suite. The System Performance Evaluation Cooperative (SPEC) was founded in 1988 by a small number of workstation vendors who

**TABLE 4.6** Popular Benchmarks for Different Categories of Workloads

| Workload Category | Example Benchmark Suite |
|---|---|
| CPU benchmarks | |
| Uniprocessor | SPEC CPU 2000 [31] |
| | Java Grande Forum Benchmarks [32] |
| | SciMark [33] |
| | ASCI [34] |
| Parallel processor | SPLASH [35] |
| | NASPAR [36] |
| Multimedia | MediaBench [37] |
| Embedded | EEMBC benchmarks [38] |
| Digital signal processing | BDTI benchmarks [39] |
| Java | |
| Client side | SPECjvm98 [31] |
| | CaffeineMark [40] |
| Server side | SPECjBB2000 [31] |
| | VolanoMark [41] |
| Scientific | Java Grande Forum Benchmarks [32] |
| | SciMark [33] |
| Transaction processing | |
| OLTP (On-line transaction processing) | TPC-C [42] |
| | TPC-W [42] |
| DSS (Decision support systems) | TPC-H [42] |
| | TPC-R [42] |
| Web server | SPEC web99 [31] |
| | TPC-W [42] |
| | VolanoMark [41] |
| E-commerce | |
| With commercial database | TPC-W [42] |
| Without commercial database | SPECjBB2000 [31] |
| Mail-server | SPECmail2000 [31] |
| Network file system | SPEC SFS 2.0 [31] |
| Personal computer | SYSMARK [43] |
| | Ziff Davis WinBench [44] |
| | 3DMarkMAX99 [45] |

realized that the marketplace was in desperate need of realistic, standardized performance tests. The basic SPEC methodology is to provide the benchmarker with a standardized suite of source code based upon existing applications that has already been ported to a wide variety of platforms by its membership. The benchmarker then takes this source code, compiles it for the system in question. The use of already accepted and ported source code greatly reduces the problem of making apples-to-oranges comparisons. SPEC designed CPU2000 to provide a comparative measure of compute intensive performance across the widest practical range of hardware. The implementation resulted in source code benchmarks developed from real user applications. These benchmarks measure the performance of the processor, memory, and compiler on the tested system. The suite contains 14 floating point programs written in C/Fortran and 11 integer programs (10 written in C and 1 in C++). The SPEC CPU2000 benchmarks replace the SPEC89, SPEC92, and SPEC95 benchmarks.

**The Java Grande Forum Benchmark suite** consists of three groups of benchmarks—microbenchmarks that test individual low-level operations (e.g., arithmetic, cast, create), Kernel benchmarks which are the heart of the algorithms of commonly used applications (e.g., heapsort, encryption/decryption, FFT, Sparse matrix multiplication, etc.), and applications (e.g., Raytracer, Monte Carlo simulation, Euler equation solution, molecular dynamics, etc.) [48]. These are compute intensive benchmarks available in Java.

**SciMark** is a composite Java benchmark measuring the performance of numerical codes occurring in scientific and engineering applications. It consists of five computational kernels: FFT, Gauss-Seidel relaxation, Sparse matrix-multiply, Monte Carlo integration, and dense LU factorization. These kernels are chosen to provide an indication of how well the underlying Java Virtual Machines perform on applications utilizing these types of algorithms. The problems sizes are purposely chosen to be small in order to isolate the effects of memory hierarchy and focus on internal JVM/JIT and CPU issues. A larger version of the benchmark (SciMark 2.0 LARGE) addresses performance of the memory subsystem with out-of-cache problem sizes.

**ASCI**, the Accelerated Strategic Computing Initiative (ASCI) of the Lawrence Livermore laboratories contains several numeric codes suitable for evaluation of compute intensive systems. The programs are available from [34].

**SPLASH**, the SPLASH suite was created by Stanford researchers [35]. The suite contains six scientific and engineering applications, all of which are parallel applications.

**NAS Parallel** Benchmarks (NPB) are a set of eight programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics (CFD) applications, consist of five kernels and three pseudo-applications.

#### 4.2.4.2 Embedded and Media Benchmarks

##### *4.2.4.2.1 EEMBC Benchmarks*

The EDN Embedded Microprocessor Benchmark Consortium (EEMBC—pronounced "embassy") was formed in April 1997 to develop meaningful performance benchmarks for processors in embedded applications. EEMBC is backed by the majority of the processor industry and has therefore established itself as the industry-standard embedded processor benchmarking forum. EEMBC establishes benchmark standards and provides certified benchmarking results through the EEMBC Certification Labs (ECL) in Texas and California. The EEMBC's benchmarks comprise a suite of benchmarks designed to reflect real-world applications, while it also includes some synthetic benchmarks. These benchmarks target the automotive/industrial, consumer, networking, office automation, and telecommunications markets. More specifically, these benchmarks target specific applications that include engine control, digital cameras, printers, cellular phones, modems, and similar devices with embedded microprocessors. The EEMBC consortium dissected these applications and derived 37 individual algorithms that constitutes the EEMBC's Version 1.0 suite of benchmarks.

##### *4.2.4.2.2 BDTI Benchmarks*

Berkeley Design Technology, Inc. (BDTI) is a technical services company that has focused exclusively on digital signal processing (DSP) since 1991. BDTI provides the industry standard BDTI Benchmarks™, a proprietary suite of DSP benchmarks. BDTI also develops custom benchmarks to determine performance on specific applications The benchmarks contain DSP routines such as FIR filter, IIR filter, FFT, dot product, and Viterbi decoder.

##### *4.2.4.2.3 MediaBench*

The MediaBench benchmark suite consists of several applications belonging to the image processing, communications and DSP applications. Examples of applications that are included are JPEG, MPEG, GSM, G.721 Voice compression, Ghostscript, ADPCM, etc. JPEG is the compression program for images, MPEG involves encoding/decoding for video transmission, Ghostscript is an interpreter for the Postscript language, and ADPCM is adaptive differential pulse code modulation. The MediaBench is an academic effort to assemble several media processing related benchmarks. An example of the use of these benchmarks may be found in [49].

#### 4.2.4.3 Java Benchmarks

**SPECjvm98**, the SPECjvm98 suite consists of a set of programs intended to evaluate performance for the combined hardware (CPU, cache, memory, and other platform-specific performance) and software aspects (efficiency of JVM, the JIT compiler, and OS implementations) of the JVM client platform [31].

The SPECjvm98 uses common computing features such as integer and floating point operations, library calls and I/O, but does not include AWT (window), networking, and graphics. Each benchmark can be run with three different input sizes referred to as S1, S10, and S100. The 7 programs are compression/decompression (compress), expert system (jess), database (db), Java compiler (javac), mpeg3 decoder (mpegaudio), raytracer (mtrt), and a parser (jack).

**SPECjbb2000** (Java Business Benchmark) is SPEC's first benchmark for evaluating the performance of server-side Java. The benchmark emulates an electronic commerce workload in a 3-tier system. The benchmark contains business logic and object manipulation, primarily representing the activities of the middle tier in an actual business server. It models a wholesale company with warehouses serving a number of districts. Customers initiate a set of operations such as placing new orders and checking the status of existing orders. It is written in Java, adapting a portable business oriented benchmark called pBOB written by IBM. Although it is a benchmark that emulates business transactions, it is very different from the TPC benchmarks. There are no actual clients, but they are replaced by driver threads. Similarly, there is no actual database access. Data is stored as binary trees of objects.

**CaffeineMark 2.5** is the latest in the series of CaffeineMark benchmarks. The benchmark suite analyses Java system performance in eleven different areas, nine of which can be run directly over the internet. It is almost the industry standard Java benchmark. The CaffeineMark can be used for comparing applet-viewers, interpreters and JIT compilers from different vendors. The CaffeineMark benchmarks can also be used as a measure of Java applet/application performance across platforms.

**VolanoMark** is a pure Java server benchmark with long-lasting network connections and high thread counts. It can be divided into two parts: server and client, although they are provided in one package. It is based on a commercial chat server application, the VolanoChat, which is used in several countries worldwide. The server accepts connections from the chat client. The chat client simulates many chat rooms and many users in each chat room. The client continuously sends messages to the server and waits for the server to broadcast the messages to the users in the same chat room. VolanoMark creates two threads for each client connection. VolanoMark can be used to test both speed and scalability of a system. In speed test, it is run in an iterative fashion on a single machine. In scalability test, the server and client are run on separate machines with high-speed network connections.

**SciMark**, see Section 4.2.4.1.

**Java Grande Forum Benchmarks**, see Section 4.2.4.1.

#### 4.2.4.4 Transaction Processing Benchmarks

The TPC is a nonprofit corporation founded in 1988 to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry. The term transaction is often applied to a wide variety of business and computer functions. Looked at it as a computer function, a transaction could refer to a set of operations including disk read/writes, operating system calls, or some form of data transfer from one subsystem to another. TPC regards a transaction as it is commonly understood in the business world: a commercial exchange of goods, services, or money. A typical transaction, as defined by the TPC, would include the updating to a database system for such things as inventory control (goods), airline reservations (services), or banking (money). In these environments, a number of customers or service representatives input and manage their transactions via a terminal or desktop computer connected to a database. Typically, the TPC produces benchmarks that measure transaction processing (TP) and database (DB) performance in terms of how many transactions a given system and database can perform per unit of time, e.g., transactions per second or transactions per minute. The TPC benchmarks can be classified into two categories, online transaction processing (OLTP) and decision support systems (DSS). OLTP systems are used in day-to-day business operations (airline reservations, banks), and are characterized by large number of clients who continually access and update small portions of the database through short running transactions. Decision support systems are primarily used for business analysis purposes, to understand business

trends, and for guiding future business directions. Information from the OLTP side of the business is periodically fed into the DSS database and analyzed. DSS workloads are characterized by long running queries that are primarily read-only and may span a large fraction of the database. Four benchmarks are active: TPC-C, TPC-W, TPC-R, and TPC-H. These benchmarks can be run with different data sizes, or scale factors. In the smallest case (or scale factor = 1), the data size is approximately 1 GB. The earlier TPC benchmarks, namely TPC-A, TPC-B, and TPC-D have become obsolete.

#### *4.2.4.4.1 TPC-C*

TPC-C is an OLTP benchmark. It simulates a complete computing environment where a population of users executes transactions against a database. The benchmark is centered around the principal activities (transactions) of a business similar to that of a worldwide wholesale supplier. The transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. Although the benchmark portrays the activity of a wholesale supplier, TPC-C is not limited to the activity of any particular business segment, but rather, represents any industry that must manage, sell, or distribute a product or service. TPC-C involves a mix of five concurrent transactions of different types and complexity either executed on-line or queued for deferred execution. There are multiple on-line terminal sessions. The benchmark can be configured to use any commercial database system such as Oracle, DB2 (IBM), or Informix. Significant disk input and output are involved. The databases consist of many tables with a wide variety of sizes, attributes, and relationships. The queries result in contention on data accesses and updates. TPC-C performance is measured in new-order transactions per minute. The primary metrics are the transaction rate (tpmC) and price per transaction ($/tpmC).

#### *4.2.4.4.2 TPC-H*

The TPC Benchmark™ H (TPC-H) is a decision support system (DSS) benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark is modeled after decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The benchmark contains 22 queries. The performance metric reported by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric (QphH@Size), and the TPC-H Price/Performance Metric, $/QphH@Size. One may not perform optimizations based on apriori knowledge of queries in TPC-H.

#### *4.2.4.4.3 TPC-R*

The TPC Benchmark™ R (TPC-R) is a decision support benchmark similar to TPC-H, but which allows additional optimizations based on advance knowledge of the queries. It consists of a suite of business oriented queries and concurrent data modifications. As in TPC-H, there are 22 queries. The performance metric reported by TPC-R is called the TPC-R Composite Query-per-Hour Performance Metric (QphR@Size), and the TPC-R Price/Performance Metric, $/QphR@Size.

#### *4.2.4.4.4 TPC-W*

TPC Benchmark™ W (TPC-W) is a transactional web benchmark. The workload simulates the activities of a business oriented transactional Web server in an electronic commerce environment. It supports many of the features of the TPC-C benchmark and has several additional features related to dynamic page generation with database access and updates. Multiple on-line browser sessions and on-line transaction processing are supported. Contention on data accesses and updates are modeled. The performance metric reported by TPC-W is the number of Web interactions processed per second (WIPS). Multiple Web interactions are used to simulate the activity of a retail store, and each interaction is subject to a response time constraint. Different profiles can be simulated by varying the ratio of browsing and buying i.e., simulating customers who are primarily browsing and those who are primarily shopping.

### 4.2.4.5 Web Server Benchmarks

**SPECweb99** is the SPEC benchmark for evaluating the performance of World Wide Web servers. It measures a system's ability to act as a Web server. The initial effort from SPEC in this direction was

**TABLE 4.7** Popular Personal Computer Benchmarks

| Benchmark | Description |
|---|---|
| Business Winstone [44] | A system-level, application-based benchmark that measures a PC's overall performance when running today's top-selling Windows-based, 32-bit applications. It runs real 32-bit business applications through a series of scripted activities and uses the time a PC takes to complete those activities to produce its performance scores. The suite includes five Microsoft Office 2000 applications (Access, Excel, FrontPage, PowerPoint, and Word), Microsoft Project 98, Lotus Notes R5, NicoMak WinZip, Norton AntiVirus, and Netscape Communicator. |
| WinBench99 [44] | A subsystem-level benchmark that measures the performance of a PC's graphics, disk, and video subsystems in a Windows environment. |
| 3DwinBench [44] | Tests the bus used to carry information between the graphics adapter and the processor subsystem. Hardware graphics adapters, drivers, and enhancing technologies such as MMX/SSE are tested. |
| CD WinBench99 [44] | Measures the performance of a PC's CD-ROM subsystem, which includes the CD drive, controller, and driver, and the system processor. |
| Audio WinBench 99 [44] | Measures the performance of a PC's audio subsystem, which includes the sound card and its driver, the processor, the DirectSound and DirectSound 3D software, and the speakers. |
| Battery Mark [44] | Measures battery life on notebook computers. |
| I-bench [44] | A comprehensive, cross-platform benchmark that tests the performance and capability of Web clients. The benchmark provides a series of tests that measure both how well the client handles features and the degree to which network access speed affects performance. |
| Web Bench [44] | Measures Web server software performance by running different Web server packages on the same server hardware or by running a given Web server package on different hardware platforms. |
| NetBench [44] | A portable benchmark program that measures how well a file server handles file I/O requests from clients. NetBench reports throughput and client response time measurements. |
| 3Dmark MAX 99 [45] | From Futuremark Corporation. Is a nice 3D benchmark that measures 3D gaming performance. Results are dependent on CPU, memory architecture, and the 3D accelerator employed. |
| SYSMARK [43] | Measures a system's real-world performance when running typical business applications. This benchmark suite comprises the retail versions of eight application programs and measures the speed with which the system under test executes predetermined scripts of user tasks typically performed when using these applications. The performance times of the individual applications are weighted and combined into both category-based performance scores as well as a single overall score. The application programs employed by SYSmark 32 are: Microsoft Word 7.0 and Lotus WordPro 96 for word processing, Microsoft Excel 7.0 (for spreadsheet), Borland Paradox 7.0 (for database), CorelDraw 6.0 (for desktop graphics), Lotus Freelance Graphics 96 and Microsoft Powerpoint 7.0 (for desktop presentation), and Adobe Pagemaker 6.0 (for desktop publishing). |

SPECweb96, but it contained only static workloads, meaning that the requests were for simply downloading web pages that do not involve any computation. But if one examines the use of the web, it is clear that many downloads involve computation to generate the information the client is requesting. Such Web pages are referred to as dynamic web pages. SPECweb99 includes dynamic Web pages. The file accesses are made to closely match today's real-world Web server access patterns. The pages also contain dynamic ad rotation using cookies and table lookups.

**VolanoMark,** see Section 4.2.4.3.

**TPC-W,** see Section 4.2.4.4.

#### 4.2.4.6 E-commerce Benchmarks

See SPECjbb2000 in Section 4.2.4.3 and TPC-W in the Section 4.2.4.4.

#### 4.2.4.7 Mail Server Benchmarks

**SPECmail2001** is a standardized mail server benchmark designed to measure a system's ability to act as a mail server servicing e-mail requests. The benchmark characterizes throughput and response time of a mail server system under test with realistic network connections, disk storage, and client workloads. The benchmark focuses on the ISP as opposed to enterprise class of mail servers, with an overall user count in the range of approximately 10,000–1,000,000 users. The goal is to enable objective comparisons of mail server products.

#### 4.2.4.8 File Server Benchmarks

**System File Server Version 2.0** (SFS 2.0) is SPEC's benchmark for measuring NFS (network file system) file server performance across different vendor platforms. It contains a workload that was developed based on a survey of more than 1,000 file servers in different application environments.

#### 4.2.4.9 PC Benchmarks

A variety of benchmarks are available, primarily from Ziff Davis and Bapco to benchmark the Windows-based personal computer. Table 4.7 lists the most common PC benchmarks. Ziff Davis Winstone and Bapco SYSMARK are benchmarks that measure overall performance while the other

**TABLE 4.8** Benchmark Web Sites

| Example Benchmark Suite | Web Site for More Information |
|---|---|
| SPEC CPU 2000 | http://www.spec.org |
| Java Grande Forum Benchmarks | http://www.epcc.ed.ac.uk/javagrande/ |
| SciMark | http://math.nist.gov/scimark2 |
| ASCI | http://www.llnl.gov/asci_benchmarks/asci/asci_code_list.html |
| NASPAR | http://www.nas.nasa.gov/Software/NPB/ |
| MediaBench | http://www.cs.ucla.edu/~leec/mediabench/ |
| EEMBC benchmarks | http://www.eembc.org |
| BDTI benchmarks | http://www.bdti.com/ |
| SPECjvm98 | http://www.spec.org |
| CaffeineMark | http://www.pendragon-software.com/pendragon/cm3 |
| SPECjBB2000 | http://www.spec.org |
| VolanoMark | http://www.volano.com/benchmarks.html |
| TPC-C | http://www.tpc.org |
| TPC-W | http://www.tpc.org |
| TPC-H | http://www.tpc.org |
| TPC-R | http://www.tpc.org |
| SPECweb99 | http://www.spec.org |
| SPECmail2000 | http://www.spec.org |
| SPEC SFS 2.0 | http://www.spec.org |
| SYSMARK | http://www.bapco.com/ |
| Ziff Davis Benchmarks | http://www.zdnet.com/etestinglabs/filters/benchmarks |
| 3DMarkMAX99 | http://www.pcbenchmarks.com |

benchmarks are intended to measure performance of one subsystem such as video or audio or one aspect such as power.

Techniques and tools for performance evaluation improve year by year. For instance, performance monitoring counters were not available to the public until 1997. Benchmarks get updated almost every year. Those interested in experimental performance evaluation should continuously monitor the state of the art. Table 4.8 provides sources for the benchmarks described in this article. The references at the end can provide new information on tools and benchmarks. Microprocessor vendors are inclined to show off their products in the best light, to projecting results for benchmarks that run well on their system, developing special optimizations within their compilers just for the sake of improving benchmark scores, and stretching the benchmark's behavior while staying within the "legal" limits of the benchmark guidelines. It is extremely important to understand benchmarks, their features and metrics used for performance evaluation to really understand the performance results.

## References

1. Reinhold P. Weicker, "An Overview of Common Benchmarks," *IEEE Computer,* pp. 65–75, Dec. 1990.
2. H. Cragon, *Computer Architecture and Implementation,* Cambridge University Press, Cambridge, 2000.
3. J.E. Smith, "Characterizing Computer Performance with a Single Number," *Communications of the ACM,* Oct. 1988.
4. Patterson and Hennessy, *Computer Architecture: The Hardware/Software Approach,* by Hennessy and Patterson, Morgan Kaufman Publishers, 2nd ed., 1998.
5. Vtune profiling software, http://developer.intel.com/software/products/vtune/vtune_oview.htm.
6. P6perf utility, http://developer.intel.com/vtune/p6perf/index.htm.
7. DCPI Tool home page, http://www.research.digital.com/SRC/dcpi/) and http://www.research.compaq.com/SRC/dcpi/.
8. J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos, "Profile Me: Hardware Support for Instruction Level Profiling on Out of Order Processors," MICRO-30 proceedings, pp. 292–302, 1997.
9. Perf-monitor for UltraSparc, http://www.sics.se/∼mch/perf-monitor/index.html.
10. PMON http://www.ece.utexas.edu/projects/ece/lca/pmon.
11. M.C. Merten, A.R. Trick, E.M. Nystrom, R.D. Barnes, and W.W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Hot Spots to Support Runtime Optimization," *Proceedings of the 26th International Symposium on Computer Architecture,* pp. 136–147, May 1999.
12. R. Bhargava, J. Rubio, S. Kannan, L.K. John, D. Christie, and L. Klaes, "Understanding the Impact of x86/NT Computing on Microarchitecture," in *Characterization of Contemporary Workloads*, pp. 203–228, Kluwer Academic Publishers, Dordrecht, the Netherlands, 2001.
13. Ali Poursepanj and David Christie, "Generation of 3D Graphics Workload for System Performance Analysis," *Presented at the First Workshop on Workload Characterization*, Also in *Workload Characterization: Methodology and Case Studies*, edited by John and Maynard, IEEE CS Press, 1999.
14. A. Agarwal, R.L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 119–127, June 1986.
15. B. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," Chapter 2 in *Fast Simulation of Computer Architectures*, by T.M. Conte and C.E. Gimarc, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1995.
16. Dinero IV cache simulator, www.cs.wisc.edu/∼markhill/DineroIV.
17. P. Bose and T.M. Conte, "Performance Analysis and Its Impact on Design," *IEEE Computer*, pp. 41–49, May 1998.

18. P. Crowley and J.-L. Baer, "On the Use of Trace Sampling for Architectural Studies of Desktop Applications," Presented at the First Workshop on Workload Characterization, Also in *Workload Characterization: Methodology and Case Studies*, edited by John and Maynard, IEEE CS Press, pp. 15–24, 1999.
19. J.R. Larus, "Efficient Program Tracing," *IEEE Computer*, pp. 52–61, May 1993.
20. Ravi Bhargava, Lizy K. John, and Francisco Matus, "Accurately Modeling Speculative Instruction Fetching in Trace-Driven Simulation," *Proceedings of the IEEE Performance, Computers and Communications Conference (IPCCC)*, pp. 65–71, Feb. 1999.
21. The Simplescalar simulator suite, http://www.simplescalar.org or http://www.cs.wisc.edu/∼mscalar/simplescalar.html.
22. B. Boothe, "Execution Driven Simulation of Shared Memory Multiprocessors," Chapter 6 in *Fast Simulation of Computer Architectures*, by T.M. Conte and C.E. Gimarc, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1995.
23. The SimOS complete system simulator, http://simos.stanford.edu/.
24. SIMICS www.simics.com.
25. SIMICS, VIRTUTECH http://www.virtutech.com.
26. L. Kurian, *Performance Evaluation of Prioritized Multiple-Bus Multiprocessor Systems*, M.S. Thesis, University of Texas at El Paso, Dec. 1989.
27. L.K. John and Yu-cheng Liu, "A Performance Model for Prioritized Multiple-Bus Multiprocessor Systems," *IEEE Transactions on Computers*, Vol. 45, No. 5, pp. 580–588, May 1996.
28. P. Heidelberger and S.S. Lavenberg, "Computer Performance Evaluation Methodology," *IEEE Transactions on Computers*, pp. 1195–1220, Dec. 1984.
29. D.B. Noonburg and J.P. Shen, "A Framework for Statistical Modeling of Superscalar Processor Performance," *Proceedings of the 3rd International Symposium on High Performance Computer Architecture (HPCA)*, pp. 298–309, 1997.
30. D.J. Sorin, V.S. Pai, S.V. Adve, M.K. Vernon, and D.A. Wood, "Analytic Evaluation of Shared Memory Systems with ILP Processors," *Proceedings of the International Symposium on Computer Architecture*, pp. 380–391, 1998.
31. SPEC Benchmarks, www.spec.org.
32. Java Grande Benchmarks, http://www.epcc.ed.ac.uk/javagrande/.
33. SciMark, http://math.nist.gov/scimark2.
34. ASCI Benchmarks, http://www.llnl.gov/asci_benchmarks/asci/asci_code_list.html.
35. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24–36, June 1995.
36. NAS Parallel Benchmarks, http://www.nas.nasa.gov/Software/NPB/.
37. MediaBench benchmarks, http://www.cs.ucla.edu/∼leec/mediabench/.
38. EEMBC, www.eembc.org.
39. BDTI, http://www.bdti.com/.
40. The Caffeine benchmarks, http://www.pendragon-software.com/pendragon/cm3.
41. VolanoMark, http://www.volano.com/benchmarks.html.
42. Transactions Processing Council, www.tpc.org.
43. SYSMARK, http://www.bapco.com/.
44. Ziff Davis Benchmarks, www.zdbop.com or www.zdnet.com/etestinglabs/filters/benchmarks.
45. PC Benchmarks, www.pcbenchmarks.com.
46. The Jaba profiling tool, http://www.ece.utexas.edu/projects/ece/lca/jaba.html.
47. R. Radhakrishnan, J. Rubio, and L.K. John, "Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels," *Proceedings of IEEE International Conference on Computer Design*, pp. 281–284.
48. J.A. Mathew, P.D. Coddington, and K.A. Hawick, "Analysis and Development of the Java Grande Benchmarks," *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

49. C. Lee, M. Potkonjak, and W.H.M. Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 330–335.
50. D. Bhandarkar and J. Ding, "Performance Characterization of the Pentium Pro Processor," *Proceedings of the 3rd High Performance Computer Architecture Symposium*, pp. 288–297, 1997.
51. Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen, "Instrumentation and Optimization of Win32/Intel Executables Using Etch," USENIX, 1997.
52. T.M. Conte and C.E. Gimarc, *Fast Simulation of Computer Architectures*, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1995.

## 4.3 Trace Caching and Trace Processors

*Eric Rotenberg*

A superscalar processor executes multiple instructions in parallel each cycle. Because there are data dependences among instructions, finding multiple independent instructions that can execute in parallel requires examining an even larger group of instructions, called the *instruction window*. Figure 4.16 shows a high-level view of a superscalar processor, including instruction buffers that make up the window and the decoupled fetch and execution engines. The fetch engine predicts branches, fetches and renames instructions, and dispatches them into the window. Meanwhile, each cycle, the execution engine identifies instructions in the window whose operands are available, and issues them to parallel functional units (FUs).

Peak performance is increased by adding more parallel functional units. But adding more functional units has ramifications for other parts of the processor. First, instruction fetch bandwidth must be commensurate with peak execution bandwidth. Second, the window must be correspondingly larger.

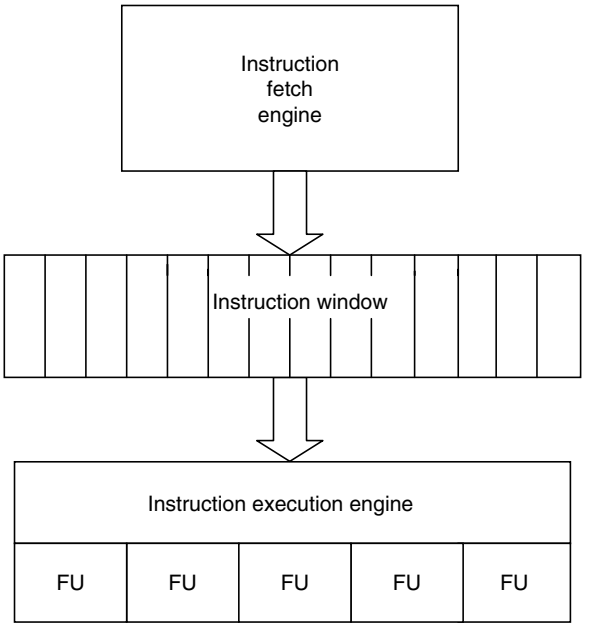



**FIGURE 4.16** High-level view of a superscalar processor: Instruction window and decoupled fetch and execute engines.

A larger window enables the processor to probe deeper into the dynamic instruction stream, increasing the chance of finding enough independent instructions each cycle to keep functional units operating at peak efficiency.

Next-generation, high-performance processors will need to issue 8, 12, or even 16 instructions per cycle. Unfortunately, at high issue rates, supporting mechanisms—instruction supply and the instruction window—are difficult to scale. This chapter section deals with *the instruction fetch bottleneck* and *inefficient execution mechanisms*, and surveys a next-generation microarchitecture, the *trace processor* [21,24,27,29,31], that attacks these problems. A third problem, *control and data dependence bottlenecks*, is also covered; however, because this aspect is more involved, it is left to the reader to investigate the trace processor literature [24,25].

### 4.3.1 Instruction Fetch Bottleneck

Taken branches in the dynamic instruction stream cause frequent disruptions in the flow of instructions into the window. The best conventional instruction cache and next-program-counter logic incurs a single-cycle disruption when a taken branch is encountered. At best, sustained fetch bandwidth is equal to the average number of instructions between taken branches, which is typically from 6 to 8 instructions per cycle for integer programs [2,19,32]. Moreover, conventional branch predictors predict at most one branch per cycle, limiting fetch bandwidth to a single basic block per cycle.

It is possible to modify conventional instruction caches and the next-program-counter logic to remove taken-branch disruptions, however, that approach is typically complex. Low latency is sacrificed for high bandwidth. A *trace cache* [8,14,18,20] changes the way instructions are stored to optimize instruction fetching for both high bandwidth and low latency.

### 4.3.2 Inefficient High-Bandwidth Execution Mechanisms

The scheduling mechanism in a superscalar processor converts an artificially sequential program into an instruction-level parallel program. The scheduling mechanism is composed of register rename logic (identifies true dependences among instructions and removes artificial dependences), the scheduling window (resolves dependences near-optimally by issuing instructions out-of-order), and the register file with result bypasses (moves data to and from the functional units as instructions issue and complete, respectively). All of the circuits are *monolithic* and their speed does not scale well for 8 or more instructions per cycle [13].

*Trace processors* [21,24,27,29,31] use a more efficient, *hierarchical* scheduling mechanism to optimize for both high-bandwidth execution and a fast clock.

### 4.3.3 Control and Data Dependence Bottlenecks

Most control dependences are removed by branch prediction, but branch mispredictions incur large performance penalties because all instructions after a mispredicted branch are flushed from the window, even control- and data-independent instructions. Exploiting *control independence* preserves useful instructions and their results [9], which would otherwise be thrown away due to branch mispredictions, but control independence mechanisms have numerous difficult implementation issues [22]. Moreover, a large instruction window does nothing to reduce the execution time of long data dependence chains, which ultimately limit performance if branch mispredictions do not. Value prediction and other forms of *data speculation* break data dependence chains [10], but difficult implementation issues must be resolved, such as providing high-bandwidth value prediction and high-performance recovery mechanisms.

The hierarchical organization of trace processors can be leveraged to overcome implementation barriers to data speculation and control independence. The interested reader may learn more about trace processor control independence mechanisms and data speculation from other sources [21,24,25].

### 4.3.4 Trace Cache and Trace Predictor: Efficient High-Bandwidth Instruction Fetching

Conventional instruction caches are unable to meet future fetch bandwidth requirements because of taken branches in the dynamic instruction stream. A taken branch instruction and its target instruction reside in different cache lines, or in the same cache line with unwanted instructions in between, as shown in Fig. 4.17a. Figure 4.17a shows a long dynamic sequence of instructions made up of four fetch blocks separated by taken branches. Ideally, to keep a 16-issue machine well-supplied, the entire sequence needs to be fetched in a single cycle. But, because the fetch blocks are noncontiguous, it takes at least four cycles to fetch and assemble the desired sequence.

The fundamental problem is instruction caches store instructions in their static order. A *trace cache* [8,14,18,20] stores instructions the way they appear in the dynamic instruction stream. Figure 4.17(b) shows the same sequence of four fetch blocks stored contiguously in one trace cache line. The trace cache allows multiple, otherwise noncontiguous fetch blocks to be fetched in a single cycle. A *trace* in this context is a dynamic sequence of instructions with a hardware-defined length limit (e.g., 16 or 32 instructions), containing any number of embedded taken and not-taken branches.

A trace cache can be incorporated in the fetch mechanism in several ways. One possibility is to replace the conventional instruction cache with a trace cache. More likely, both a trace cache and instruction cache are used. In trace processors, described in the next section, the trace cache is accessed first and, if it does not have the desired trace, the trace is quickly constructed from the back-up instruction cache. Early trace cache fetch units [14,20] access the trace cache and instruction cache in parallel, as shown in Fig. 4.18. If the trace exists in the trace cache, it supplies instructions and the instruction cache's instructions are discarded since they are subsumed by the trace. Otherwise, the instruction cache supplies a smaller fetch block.

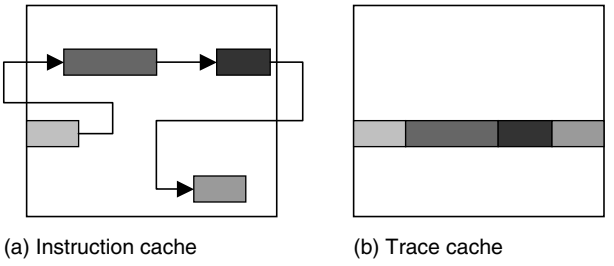



**FIGURE 4.17** Example dynamic sequence stored in an instruction cache and trace cache.

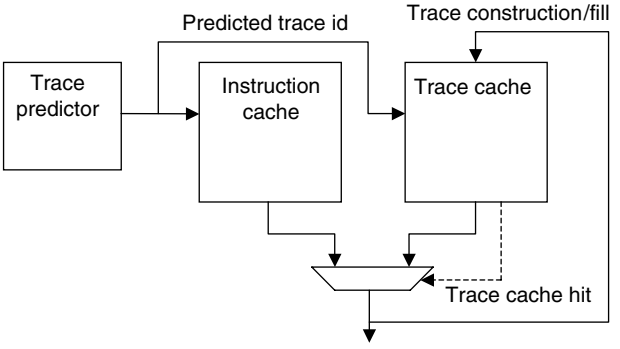



**FIGURE 4.18** Instruction fetch unit with trace cache.

A trace is uniquely identified by the program counter of the first instruction in the trace (start PC) and embedded branch outcomes (taken/not-taken bit for every branch; this assumes indirect branches terminate a trace, since taken/not-taken is insufficient for indirect branches). The start PC and branch outcomes are collectively called the *trace identifier*, or *trace id*. Looking up a trace in the trace cache is similar to looking up instructions/data in conventional caches, except the trace id is used instead of an address. A subset of the trace id forms an index into the trace cache and the remaining bits form a tag. One or more traces and their identifying tags are accessed at that index (the number of traces depends on the trace cache's set-associativity). If one of the tags matches the tag of the supplied trace id, there is a trace cache hit. Otherwise, there is a trace cache miss. New traces are constructed and written into the trace cache either speculatively, as instructions are fetched from the instruction cache (as shown in Fig. 4.18), or non-speculatively, as instructions retire from the processor.

A new *predicted trace id* is supplied to the trace cache each cycle. Conventional branch prediction, with a throughput of only one branch prediction per cycle, is not designed to produce trace ids. Multiple-branch predictor counterparts of single-branch predictors have been proposed but they tend to be unwieldy [19]. A conceptually simpler approach is to not predict branches directly. *Explicit trace prediction* [6] predicts trace ids directly and, in doing so, implicitly predicts any number of embedded branches in a single cycle. The *trace predictor* shown in Fig. 4.18 supplies a predicted trace id—start PC and multiple branch predictions—to both the trace cache and instruction cache.

The trace cache design space is extensive. In addition to typical parameters such as size, set-associativity, and replacement policy, the design space includes: indexing methods (which PC bits and which, if any, branch prediction bits are used), path associativity (ability to simultaneously store different traces with the same start PC), partial matching (ability to use prefix of a trace if the trace id only partially matches), trace selection (policies for beginning and ending traces), trace cache fill policy, parallel or sequential accessing of the trace and instruction caches, and other aspects. The interested reader is referred to trace cache literature to gain an appreciation for the trace cache design space [4,5,7,14–20,23].

A problem of trace caches is they necessarily use storage less efficiently than instruction caches. A given static instruction appears exactly once in the instruction cache. In a trace cache, however, there may be multiple copies of the same static instruction. Redundancy within a trace is caused by dynamic unrolling of small loops. Redundancy among different traces is caused by partial overlap of different paths through a region. For example, two traces may start at the same program counter but diverge at a common branch, such that the two traces share a common prefix; the paths may reconverge before both traces end, causing even more redundancy.

Trace cache redundancy is the price paid for simplicity and a direct approach to high-bandwidth instruction fetching. There are other high-bandwidth fetch mechanisms that work solely out of the conventional instruction cache [2,3,26,32]. They all use the same basic approach. First, the branch predictor is modified to generate pointers to multiple noncontiguous fetch blocks. Second, the instruction cache is highly multiported so that pointers can access noncontiguous cache lines in parallel. Finally, a sophisticated instruction alignment network re-orders the blocks fetched in the previous step to construct the desired dynamic sequence of instructions. The approach is certainly high-bandwidth but the number of stages in the fetch pipeline is increased, and additional fetch unit latency impacts performance negatively [19]. The trace cache approach is less efficient in terms of storage. But other approaches are inefficient in terms of repeatedly constructing dynamic traces on-the-fly from the static instruction cache. The trace cache incurs the latency to construct a trace once and then efficiently reuses it many times.

### 4.3.5 Trace Processor: Efficient High-Bandwidth Instruction Execution

Instruction execution is inefficient in wide-issue superscalar processors because all data dependences are handled *uniformly*. When an instruction issues, its data dependent instructions wakeup with uniform latency, usually a single cycle, regardless of their location in the window. Resolving all dependences in a single cycle optimizes parallelism, but cycle time is extended to accommodate the full length of the window. Increasing processor cycle time penalizes the entire pipeline. A better alternative is to increase the number of cycles to resolve data dependences, e.g., two cycles instead of one, so other pipeline stages are unaffected. However, it still remains the case that *all data dependences are slow to resolve*.

Fortunately, there is a compromise between optimizing for parallelism and optimizing for cycle time if data dependences are handled *nonuniformly*. A *trace processor* [21,24,27–29,31] hierarchically divides the processor into smaller processing elements (PEs), as shown in Fig. 4.19. The approach preserves a fast clock and resolves many data dependences in one clock cycle (data dependences *within* PEs), at the expense of resolving some data dependences in two or more clock cycles (data

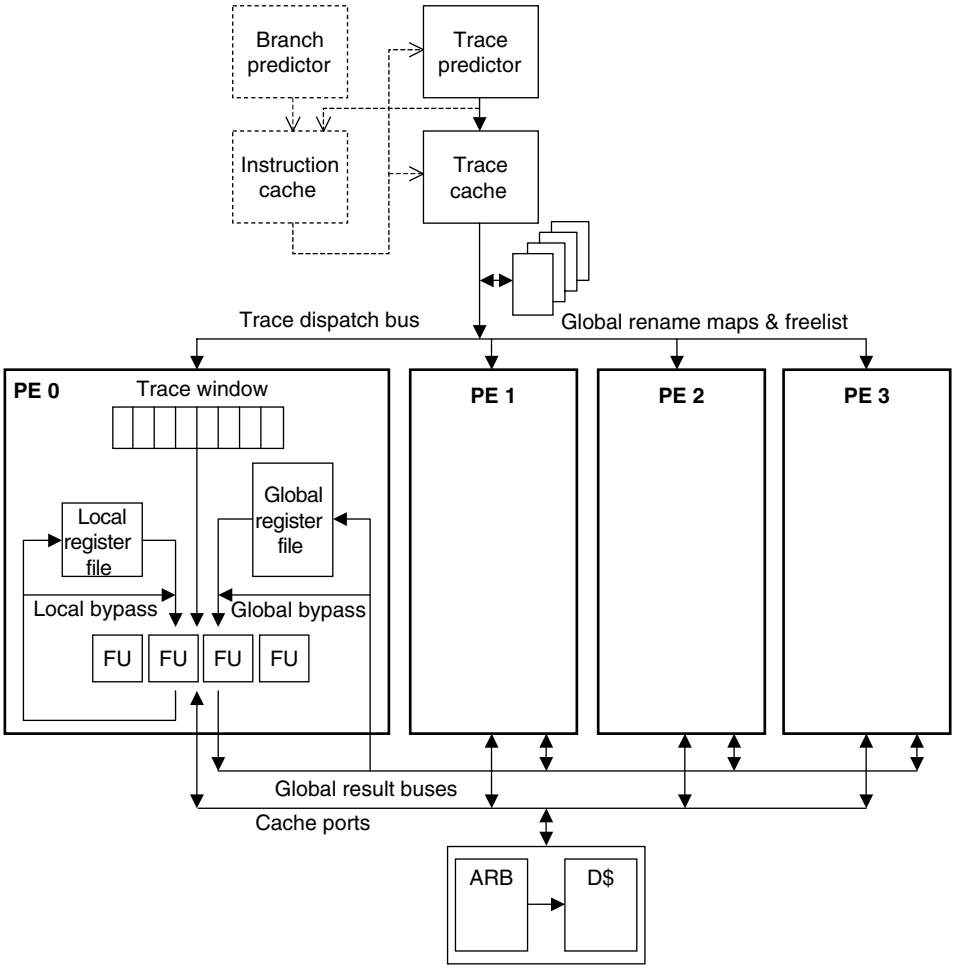



**FIGURE 4.19** Trace processor.

dependences *among* PEs). The microarchitecture shown in Fig. 4.19 is described in the remainder of this chapter.

#### 4.3.5.1 Instruction Supply

The trace predictor and trace cache supply a single trace per cycle. The conventional branch predictor and instruction cache shown in Fig. 4.19 are secondary, back-up mechanisms for constructing traces that miss in the cache or that were mispredicted [21,23,24].

#### 4.3.5.2 Register Renaming

Register renaming determines data dependences among all newly-fetched instructions, and between newly-fetched instructions and other instructions already in the window. The first aspect—determining data dependences among 16 or 32 fetched instructions—almost certainly takes more than a single clock cycle. Each instruction compares its source registers to the destination registers of all logically preceding instructions. The second aspect—linking incoming instructions with previous and future instructions in the window—requires an impractical number of read and write ports to the register rename map table and high bandwidth to the register freelist.

The efficiency of register renaming and later execution stages is improved by hierarchically dividing data flow into intra-trace and inter-trace values [31], as shown in Fig. 4.20. *Local values* are produced

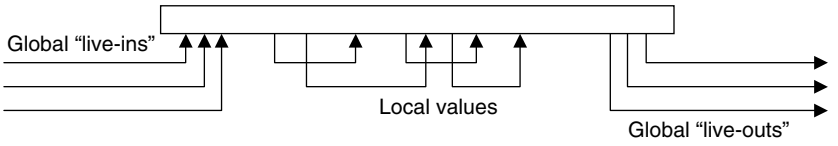



**FIGURE 4.20** Data flow hierarchy of traces.

and consumed solely within a trace and are not visible to other traces. *Global values* are communicated among traces. Global input values to a trace are called *live-ins* and global output values of a trace are called *live-outs*.

Local dependences in a trace are static because control flow is pre-determined. Thus, intra-trace dependence checking is performed only once, when the trace is first constructed and written into the trace cache. Furthermore, the local values that correspond to intra-trace dependences can be statically bound to registers in a local register file, a process called *pre-renaming* [31]. Each PE has a small private local register file, large enough to hold all local values produced by a trace. Local register files are private because their values do not need to be communicated to other PEs. Pre-rename information is computed once and stored along with the trace in the trace cache. Pre-renaming eliminates the first aspect of register renaming from the rename stage—dependence checking among newly fetched instructions.

The second aspect of renaming—linking fetched instructions with other instructions in the window—is still performed, but the hierarchical treatment of values makes this aspect efficient. The only linkages are inter-trace dependences. Live-in and live-out values are dynamically renamed to what is logically a single shared global register file. The global register file communicates values among traces.

Although the global register file, its map table, and its freelist are similar to a superscalar processor's monolithic register file and renaming structures, the trace processor's register file is more efficient because fewer values are processed. Reduced register file complexity is described below in the context of instruction issue logic. Global renaming structures are simplified in three ways. First, fewer read and write ports lead to the global rename map table because only live-ins/live-outs are renamed, and not local values. Second, bandwidth to the global register freelist is reduced since only live-outs consume free registers, and not local values. Third, to support trace misprediction recovery, the global rename map table is checkpointed only at trace boundaries instead of at every branch, so fewer shadow maps are required.

#### 4.3.5.3 Instruction Dispatch

Merging instructions into the window is also simplified. A single trace is routed to a single PE. A conventional processor routes multiple instructions to as many, possibly noncontiguous instruction buffers.

#### 4.3.5.4 Instruction Issue Logic, Register File, and Result Bypasses

The instruction issue mechanism is possibly the most complex aspect of current dynamically scheduled superscalar processors [13]. Each cycle, the processor examines the instruction window for instructions whose input values are available and are ready to issue (*wakeup logic*). Of the ready instructions, a number of them are selected for issue based on resource constraints (*select logic*). The selected instructions read values from the register file and are routed to functional units, where they execute and write results to the register file. Each result must also be quickly bypassed to functional units to be consumed by pipelined, data dependent instructions (*result bypasses*). The wakeup logic, select logic, register file, and result bypasses all grow in complexity as the size of the instruction window and the number of parallel execution units are increased [13].

The large trace processor instruction window is distributed among multiple smaller processing elements (PEs), as shown in Fig. 4.19. Each PE resembles a small superscalar processor and at any given time is allocated a single trace to process. A PE has (1) enough instruction issue buffers to hold an

entire trace, (2) multiple dedicated functional units, and (3) a dedicated local register file for storing and communicating local values.

Logically, a single global register file stores and communicates global values. Each PE contains a copy of the global register file for private read ports. Write ports to the global register file are shared, however. All PEs are connected to shared global result buses, which write values simultaneously into all copies of the global register file.

A hierarchical instruction window simplifies the wakeup logic, select logic, register file, and result bypasses. Each aspect is described below.

Waiting instructions monitor fewer "tags" to determine when to wakeup. Tags are broadcast by producer instructions soon after issuing, to wakeup dependent instructions. Although each PE monitors both its own local tags and all global tags, overall, fewer tags are monitored than in an equivalent, nonhierarchical processor. The number of local tags is small, e.g., four tags for a four-issue PE. Even the number of global tags is small due to reduced global register traffic, e.g., typically two to four tags are sufficient [24]. Also, tags are broadcast on shorter wires—the length of a PE trace window instead of the length of the entire window (of course, global tags and values first incur one cycle of delay on the global result buses, as discussed below). The combination of fewer tags and a smaller wakeup window greatly reduces wakeup circuit delay, allowing a faster clock.

Instruction select logic is fully distributed. Each PE independently selects ready instructions from its trace and routes them to dedicated functional units. Here, fewer instruction candidates and fewer functional units reduce select circuit delay, allowing a faster clock.

The local register file is quite fast because it contains few registers (e.g., typically eight registers [24]) and has relatively few read and write ports, comparable to today's four-issue superscalar processors. The complexity of the global register file is reduced because much of the register traffic is off-loaded to the local register files. For an equivalent instruction window, the global register file requires far fewer registers and read/write ports than the monolithic file of nonhierarchical processors.

Finally, result bypasses, which are primarily long interconnect, are receiving much attention lately due to technology trends. In deep sub-micron technologies, interconnect delay improves less with technology scaling than logic delay does [1,13]. This trend highlights the importance of purposeful, nonuniform bypass latencies. Local values are bypassed quickly among functional units in a PE. Global values incur an extra cycle (or more) on the global result buses, but at least not all values are broadcast on global interconnect. In a conventional superscalar processor, all bypasses are effectively global.

### 4.3.6 Summary via Analogy

Prior to superscalar processors, comparatively simple out-of-order processors fetched, dispatched, issued, and executed one instruction per cycle, as shown in the left-hand side of Fig. 4.21. The branch predictor predicts up to one branch each cycle and a single PC fetches one instruction from a simple instruction cache. The renaming mechanism, e.g., Tomasulo's algorithm [30], performs simple dependence checking by looking up a couple of source tags in the register file. And at most one instruction is steered to the reservation station of a functional unit each cycle. After completing, instructions arbitrate for a common data bus, and the winner writes its result and tag onto the bus and into the register file.

The superscalar paradigm "widens" each of these pipeline stages and increases complexity with each additional instruction per cycle. This is clearly manageable up to a point: high-speed, dynamically scheduled 4-issue superscalar processors currently set the performance standard in microprocessors. But there is a crossover point beyond which it becomes more efficient to manage instructions in groups, that is, hierarchically.

A trace processor manages instructions hierarchically. In the right-hand side of Fig. 4.21, the top-most level of the trace processor hierarchy is shown (the trace-level). The picture is virtually identical to the single-issue, out-of-order processor on the left-hand side. The unit of operation has changed from one instruction to one trace, but the pipeline bandwidth remains 1 unit per cycle.

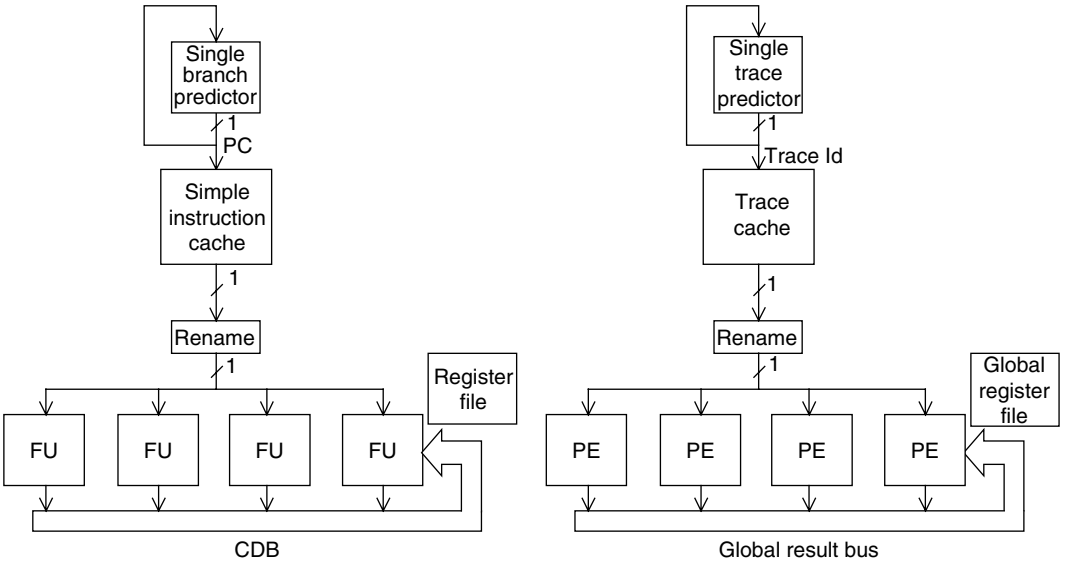



**FIGURE 4.21** Analogy between a single instruction and a single trace.

In essence, grouping instructions within traces is a reprieve. Complexity (cycle time) does not necessarily increase with each additional instruction added to a trace. Additional branches are absorbed by the trace cache and trace predictor, and additional source and destination operands are absorbed by handling data flow hierarchically. Also, complexity (cycle time) does not necessarily increase with one or two additional PEs. Hardware parallelism is allowed to expand incrementally—up to a point, at which time perhaps another level of hierarchy, and another reprieve, is needed.

Perhaps the most important thing to remember about trace processors is that the whole processor contributes to parallelism, but cycle time is influenced more by an individual processing element than the whole processor.

## References

1. M. Bohr. Interconnect Scaling—The Real Limiter to High Performance ULSI. *1995 International Electron Devices Meeting Technical Digest*, pp. 241–244, 1995.
2. T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. *22nd International Symposium on Computer Architecture,* pp. 333–344, June 1995.
3. S. Dutta and M. Franklin. Control Flow Prediction with Tree-like Subgraphs for Superscalar Processors. *28th International Symposium on Microarchitecture,* pp. 258–263, November 1995.
4. D. Friendly, S. Patel, and Y. Patt. Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism. *30th International Symposium on Microarchitecture,* pp. 24–33, December 1997.
5. D. Friendly, S. Patel, and Y. Patt. Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors. *31st International Symposium on Microarchitecture,* pp. 173–181, December 1998.
6. Q. Jacobson, E. Rotenberg, and J.E. Smith. Path-Based Next Trace Prediction. *30th International Symposium on Microarchitecture,* pp. 14–23, December 1997.
7. Q. Jacobson and J.E. Smith. Instruction Pre-Processing in Trace Processors. *5th International Symposium on High-Performance Computer Architecture,* January 1999.
8. J. Johnson. Expansion Caches for Superscalar Processors. Technical Report CSL-TR-94-630, Computer Systems Laboratory, Stanford University, June 1994.
9. M.S. Lam and R.P. Wilson. Limits of Control Flow on Parallelism. *19th International Symposium on Computer Architecture,* pp. 46–57, May 1992.

10. M. Lipasti and J. Shen. Exceeding the Dataflow Limit via Value Prediction. *29th International Symposium on Microarchitecture,* December 1996.
11. S. Melvin, M. Shebanow, and Y. Patt. Hardware Support for Large Atomic Units in Dynamically Scheduled Machines. *21st International Symposium on Microarchitecture,* pp. 60–66, December 1988.
12. S. Melvin and Y. Patt. Performance Benefits of Large Execution Atomic Units in Dynamically Scheduled Machines. *3rd International Conference on Supercomputing,* pp. 427–432, June 1989.
13. S. Palacharla, N. Jouppi, and J.E. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-96-1328, Computer Sciences Department, University of Wisconsin-Madison, November 1996.
14. S. Patel, D. Friendly, and Y. Patt. Critical Issues Regarding the Trace Cache Fetch Mechanism. Technical Report CSE-TR-335-97, Department of Electrical Engineering and Computer Science, University of Michigan-Ann Arbor, 1997.
15. S. Patel, M. Evers, and Y. Patt. Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing. *25th International Symposium on Computer Architecture,* pp. 262–271, June 1998.
16. S. Patel, D. Friendly, and Y. Patt. Evaluation of Design Options for the Trace Cache Fetch Mechanism. *IEEE Transactions on Computers* (special issue on cache memory), 48(2):193–204, February 1999.
17. S. Patel. Trace Cache Design for Wide-Issue Superscalar Processors. PhD Thesis, University of Michigan-Ann Arbor, 1999.
18. A. Peleg and U. Weiser. Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line. U.S. Patent Number 5,381,533, January 1995.
19. E. Rotenberg, S. Bennett, and J.E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. Technical Report CS-TR-96-1310, Computer Sciences Department, University of Wisconsin-Madison, April 1996.
20. E. Rotenberg, S. Bennett, and J.E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. *29th International Symposium on Microarchitecture,* pp. 24–34, December 1996.
21. E. Rotenberg, Q. Jacobson, Y. Sazeides, and J.E. Smith. Trace Processors. *30th International Symposium on Microarchitecture,* pp. 138–148, December 1997.
22. E. Rotenberg, Q. Jacobson, and J.E. Smith. A Study of Control Independence in Superscalar Processors. *5th International Symposium on High-Performance Computer Architecture,* January 1999.
23. E. Rotenberg, S. Bennett, and J.E. Smith. A Trace Cache Microarchitecture and Evaluation. *IEEE Transactions on Computers* (special issue on cache memory), 48(2):111–120, February 1999.
24. E. Rotenberg. Trace Processors: Exploiting Hierarchy and Speculation. PhD Thesis, University of Wisconsin-Madison, August 1999.
25. E. Rotenberg and J.E. Smith. Control Independence in Trace Processors. *32nd International Symposium on Microarchitecture,* November 1999.
26. A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block Ahead Branch Predictors. *7th International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1996.
27. J.E. Smith and S. Vajapeyam. Trace Processors: Moving to Fourth-Generation Microarchitectures. *IEEE Computer* (special issue on Billion-Transistor Processors), September 1997.
28. G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. *22nd International Symposium on Computer Architecture,* pp. 414–425, June 1995.
29. K. Sundararaman and M. Franklin. Multiscalar Execution along a Single Flow of Control. *International Conference on Parallel Processing,* August 1997.
30. R. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development,* 11(1):25–33, January 1967.
31. S. Vajapeyam and T. Mitra. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. *24th International Symposium on Computer Architecture,* pp. 1–12, June 1997.
32. T.-Y. Yeh, D.T. Marr, and Y.N. Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. *7th International Conference on Supercomputing,* pp. 67–76, July 1993.

# II

# Embedded Applications

# 5

# Embedded Systems-on-Chips




Wayne Wolf
*Princeton University*


## 5.1 Introduction

Advances in VLSI technology now allow us to build systems-on-chips (SoCs), also known as systems-on-silicon (SoS). SoCs are complex at all levels of abstraction; they contain hundreds of millions of transistors; they also provide sophisticated functionality, unlike earlier generations of commodity memory parts. As a result, SoCs present a major productivity challenge.

One solution to the SoC productivity problem is to use embedded computers.[1] An embedded computer is a programmable processor that is a component in a larger system that is not a general-purpose computer. Embedded computers help tame design complexity by separating (at least to some degree) hardware and software design concerns. A processor can be used as a pre-designed component—known as intellectual property (IP)—that operates at a known speed and power consumption. The software required to implement the desired functionality can be designed somewhat separately.

In exchange for separating hardware and software design, some elements traditionally found in hardware design must be transferred to software design. Software designers have traditionally concentrated on functionality while hardware designers have worried about critical delay paths, power consumption, and area. Embedded software designers must worry about real-time deadlines, power consumption, and program and data size. As a result, embedded SoC design disciplines require a blending of hardware and software skills.

This chapter considers the characteristics of SoCs built from embedded processors. The next section surveys the types of requirements that are generally demanded from embedded SoCs. Section 5.3 surveys the characteristics of components used to build embedded systems. Section 5.4 introduces the types of architectures used in embedded systems. Section 5.5 reviews design methodologies for embedded SoCs.

## 5.2 Requirements on Embedded SoCs

A digital system typically uses embedded processors to meet a combination of performance, complexity, and possibly design time goals. If the system's behavior is very regular and easy to specify as hardware, it may not be necessary to use embedded software. An embedded processor becomes more attractive when the behavior is too complex to be easily captured in hardwired logic.

Using embedded processors may reduce design time by allowing the design to be separated into distinct software and hardware units. In many cases, the CPU will be predesigned; even if the CPU and associated hardware is being designed for the project, many aspects of the hardware design can be performed separately from the software design. (Experience with embedded system designs does show, however, that the hardware and software designs are intertwined and that embedded software is prone to some of the same scheduling problems as mainframe software projects.)

But even if embedded processors seem attractive by reducing much of the design to "just programming," it must be remembered that embedded software design is much more challenging than typical applications programming for workstations or PCs. Embedded software must be designed to meet not just functional requirements—the software's input and output behavior—but also stringent nonfunctional requirements. Those nonfunctional requirements include:

- *Performance*—Although all programmers are interested in speed of execution, performance is measured much more precisely in the typical embedded system. Many embedded systems must meet real-time deadlines. The deadline is measured between two points in the software: if the program completely executes from the starting point to the end point by the deadline, the system malfunctions.
- *Energy/power*—Traditional programmers don't worry about power or energy consumption. However, energy and power are important to most embedded systems. Energy consumption is of course important in battery-operated systems, but the heat generated as a result of power consumption is increasingly important to wall-powered systems.
- *Size*—The amount of memory required by the embedded software determines the amount of memory required by the embedded system. Memory is often one of the major cost components of an embedded system.

Embedded software design resembles hardware design in its emphasis on nonfunctional requirements such as performance and power. The challenge in embedded SoC design is to take advantage of the best aspects of both hardware and software components to quickly build a cost-effective system.

## 5.3 Embedded SoC Components

### 5.3.1 CPUs

As shown in Fig. 5.1, a CPU is a programmable instruction set processor. Instructions are kept in a separate memory—a program counter (PC) that points to the current instruction. This definition does not consider reconfigurable logic to be a programmable computer, because it does not have a separate instruction memory and a PC. Reconfigurable logic can be used to implement sequential machines, and so a CPU could be built in reconfigurable logic. But the separation of CPU logic and memory is an important abstraction for program design.

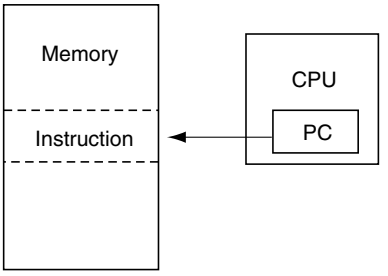



**FIGURE 5.1** A CPU and memory.

An embedded processor is judged by several characteristics:

- *Performance*—The overall speed of execution may be important in some systems, but in many cases we particularly care about the CPU's performance on critical sections of code.
- *Energy and power*—Processors provide different mechanisms to manage power consumption.
- *Area*—The area of the processor contributes to the total implementation cost of the SoC. The area of the memory required to store the program also contributes to implementation cost.

These characteristics are judged relative to the embedded software they are expected to run. A processor may exhibit very different performance or energy consumption on different applications.

RISC processors are commonly used in embedded computing. ARM[2] and MIPS[3] processors are examples of RISC processors that are widely used in embedded systems. A RISC CPU uses a pipeline to increase CPU performance. Many RISC instructions take the same amount of time to execute, simplifying performance analysis. However, many RISC architectures do have exceptions to this rule. An example is the multiple-register feature of the ARM processor: an instruction can load or store a set of registers, for which the instruction takes one cycle per instruction.

Most CPUs used in PCs and workstations today are superscalar processors. A superscalar processor builds on RISC techniques by adding logic that examines the instruction stream and determines, based on what CPU resources are needed, when several instructions can be executed in parallel. Superscalar scheduling logic adds quite a bit of area to the CPU in order to check all the possible conflicts between combinations of instructions; the size of a superscalar scheduler grows as $n^2$, where $n$ is the number of instructions that are under consideration for scheduling. Many embedded systems, and in particular SoCs, do not use superscalar processors and instead stick with RISC processors. Embedded system designers tend to use other techniques, such as instruction-set optimization caches, to improve performance. Because SoC designers are concerned with overall system performance, not just CPU performance, and because they have a better idea of the types of software run on their hardware, they can tackle performance problems in a variety of ways that may use the available silicon area more cost-effectively.

Some embedded processors are known as digital signal processors (DSPs). The term DSP was originally used to mean one of two things: either a CPU with a Harvard architecture that provided separate memories for programs and data; or a CPU with a multiply-accumulate unit to efficiently implement digital filtering operations. Today, the meaning of the term has blurred somewhat. For instance, version 9 of the ARM architecture is a Harvard architecture to better support digital signal processing. Modern usage applies the term DSP to almost any processor that can be used to efficiently implement signal processing algorithms.

The application-specific integrated processor (ASIP)[4] is one approach to improving the performance of RISC processors for embedded application. An ASIP's instruction set is designed to match the requirements of the application software it will run. On the one hand, special-purpose function units and instructions to control them may be added to speed up certain operations. On the other hand, function units, registers, and busses may be eliminated to reduce the CPU's cost if they do not provide enough benefit for the application at hand. The ASIP may be designed manually or automatically based on profiling information. One advantage of generating the ASIP automatically is that the same information can be used to generate the processor's programming environment: a compiler, assembler, and debugger are necessary to make the ASIP useful building blocks.

Another increasingly popular architecture for embedded computing is very long instruction word (VLIW). A VLIW machine can execute several instructions simultaneously but, unlike a superscalar processor, relies on the compiler to schedule parallel instructions at compilation time. A pure VLIW machine uses slots in the long, fixed-length instruction word to control the CPU's function units, with NOPs used to indicate slots that cannot be used for useful work by the compiler. Modern VLIW machines, such as the TI C6000[5] and the Motorola/Agere StarCore,[6] group single-operation instructions into execution packets; packets; the packet's length can vary depending on the number of instructions that the compiler was able to schedule for simultaneous operation. VLIW machines provide instruction-level parallelism with a much smaller CPU than is possible in a superscalar system; however,

the compiler must be able to extract parallelism at compilation time to be able to use the CPU's resources. Signal processing applications often have parallel operations that can be exploited at compilation time. For example, a parallel set of filter banks runs the same code on different data; the operations for each channel can be scheduled together in the VLIW instruction group.

### 5.3.2 Interconnect

Embedded SoCs may connect several CPUs, on-chip memories, and devices on a single chip. High-performance interconnect systems are required to meet the system's performance demands. The interconnection systems must also comply with standards so that existing components may be connected to them.

Busses are still the dominant interconnection scheme for embedded SoCs. Although richer interconnection schemes could be used on-chip, where they are not limited by pinout as in board-level systems, many existing architectures are still memory-limited and not interconnect-limited. However, future generations of embedded SoCs may need more sophisticated interconnection schemes.

A bus provides a protocol for communication between components. It also defines a memory space and the uses of various addresses in that memory space, for example, the address range assigned to a device connected to the bus. Busses for SoCs may be designed for high-performance or low-cost operation. A high-performance bus uses a combination of techniques—advanced circuits, additional bus lines, efficient protocols—to maximize transaction performance. One common protocol used for efficient transfers is the block transfer, in which a range of locations is transferred based on a single address, eliminating the need to transfer all the addresses on the bus. Some recent busses allow split transactions—the data request and data transfer are performed on separate bus cycles, allowing other bus operations to be performed while the original request is serviced. A low-cost bus design provides modest performance that may not be acceptable for instruction fetching or other time-critical operations. A low-cost bus is designed to require little hardware in the bus itself and to impose a small hardware and software overhead on the devices connecting to the bus. A system may contain more than one bus; a bridge can be used to connect one bus to another.

The ARM AMBA bus specification[7] is an example of a bus specification for SoCs. The AMBA spec actually includes two busses: the high-performance AMBA high-performance bus (AHB) and the low-cost AMBA peripherals bus (APB). The Virtual Sockets Interface committee has defined another standard for interconnecting components on SoCs.

### 5.3.3 Memory

One of the great advantages of SoC technology is that memory can be placed on the same chip as the system components that use the memory. On-chip memory both increases performance and reduces power consumption because on-chip connections present less reactive load than do pins and traces between chips; however, an SoC may still need to use separate chips for off-chip memory.

Although on-chip embedded memory has many advantages, it still is not as good as commodity memory. A commodity SRAM or DRAM's manufacturing process has been carefully tuned to the requirements of that component. In contrast, an on-chip memory's manufacturing needs must be balanced against the requirements of the logic circuits on the chip. The transistors, interconnections, and storage nodes of on-chip memories all have somewhat different needs than logic transistors.

Embedded DRAMs suffer the most because they need quite different manufacturing processes than do logic circuits. The processing steps required to build the storage capacitors for the DRAM cell are not good for small-geometry transistors. As a result, embedded DRAM technologies often compromise both the memory cells and the logic transistors, with neither being as good as they would be in separate, optimized processes. Although embedded DRAM has been the subject of research for many years, its limitations have kept its from becoming a widely used technology at the time of this writing.

SRAM circuits' characteristics are closer to those of logic circuits and so can be built on SoCs with less penalty. SRAM consumes more power and requires more chip area than does DRAM, but SRAM does not need refreshing, which noticeably simplifies the system architecture.

### 5.3.4 Software Components

Software elements are also components of embedded systems. Just as pre-designed hardware components are used to both reduce design time and to provide predictions of the characteristics of parts of the system, software components can also be used to speed up software implementation time and to provide useful data on the characteristics of systems.

CPU vendors often supply software libraries for their processors. These libraries generally supply code for two types of operations. First, they provide drivers for input and output operations. Second, they provide efficient implementations of commonly-used algorithms. For example, libraries for DSPs generally include code for digital filtering, fast Fourier transforms, and other common signal processing algorithms. Code libraries are important because compilers are still not as adept as expert human programmers at creating code that is both fast and small.

The real-time operating system (RTOS) is the second major category of software component. Many applications perform several different types of operations, often with their own performance requirements. As a result, the software is split into processes that run independently under the control of an RTOS. The RTOS schedules the processes to meet performance goals and efficiently utilize the CPU and other hardware resources. The RTOS may also provide utilities, such as interprocess communication, networking, or debugging. An RTOS's scheduling policy is necessarily very different from that used in workstations and mainframes, because the RTOS must meet real-time deadlines. A priority-driven scheduling algorithm such as rate-monotonic scheduling (RMS)[9] is often used by the RTOS to schedule activity in the system.

## 5.4 Embedded System Architectures

The hardware architecture of an embedded SoC is generally tuned to the requirements of the application. Different domains, such as automotive, image processing, and networking all have very different characteristics. In order to make best use of the available silicon area, the system architecture is chosen to match the computational and communication requirements of the application. As a result, a much wider range of hardware architectures is found for embedded systems as compared with traditional computer systems.

Figure 5.2 shows one common configuration, a bus-based uniprocessor architecture for an embedded system. This architecture has one CPU, which greatly simplifies the software architecture. In addition to I/O devices, the architecture may include several devices known as accelerators designed to speed up computations. (Though some authors refer to these units as co-processors, we prefer to reserve that term for units that are dispatched by the CPU's execution unit.) For example, a video operation's inner loops may be implemented in an application-specific IC (ASIC) so that the operation can be performed more quickly than would be possible on the CPU. An accelerator can achieve performance gains through several mechanisms: by implementing some functions in special hardware that takes fewer cycles than is required on the CPU, by reducing the time required for control operations that would require instructions on the CPU, and by using additional registers and custom data flow

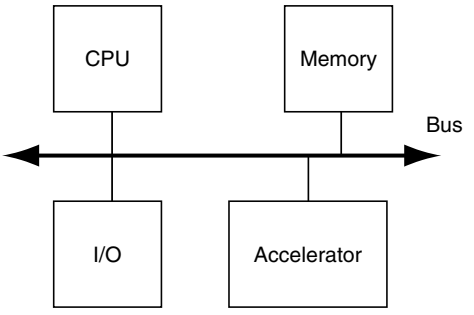



**FIGURE 5.2** A bus-based, single-CPU embedded system.

within the accelerator to more efficiently implement the available communication. The single-CPU/bus architecture is commonly used in applications that do not have extensive real-time characteristics and ones that need to run a wider variety of software. For example, many PDAs use this type of architecture. A single-CPU system simplifies software design and debugging since all the work is assumed to happen on one processing element. The single CPU system is also relatively inexpensive.

In general, however, a high-performance embedded system requires a heterogeneous multiprocessor—a multiprocessor that uses more than one type of processing element and/or a specialized communication topology. Scientific parallel processors generally use a regular architecture to simplify programming. Embedded systems use heterogeneous architectures for several reasons:

- *Cost*—A regular architecture may be much larger and more expensive than a heterogeneous architecture, which freed from the constraint of regularity, can remove resources from parts of the architecture where they are not needed and add them to parts where they are needed.
- *Real-time performance*—Scientific processors are desgined for overall performance but not to meet deadlines. Embedded systems must often put processing power near the I/O that requires realtime responsiveness; this is particularly true if the processing must be performed at a high rate. Even if a high-rate, real-time operation requires relatively little computation on each iteration, the high interrupt rate may make it difficult to perform other processing tasks on the same processing element.

Many embedded systems use heterogeneous multiprocessors. One example comes from telephony. A telephone must perform both control- and data-intensive operations: both the network protocol and the user interface require control-oriented code; the signal processing operations require data-oriented code. The Texas Instruments OMAP architecture, shown in Fig. 5.3, is designed for telephony: the RISC processor handles general-purpose and control-oriented code while the DSP handles signal processing. Shared memory allows processes on the two CPUs to communicate, as does a bridge. Each CPU has its own RTOS that coordinates processes on the CPU and also mediates communication with the other CPU.

The C-Port network processor,[11] whose hardware architecture is shown in Fig. 5.4, provides an example of a heterogeneous multiprocessor in a different domain. The multiprocessor is a high-speed bus. The RISC executive processor is C programmable and provides overall control, initialization, etc. Each of the 16 HDLC processors is also C programmable. Other interfaces for higher-speed networks are not general-purpose computers and can be programmed only with register settings.

Another category of heterogeneous parallel embedded systems is the networked embedded system. Automobiles are a prime example of this type of system: the typical high-end car includes over a hundred microprocessors ranging from 4-bit microcontrollers to high-performance 32-bit processors. Networks help to distribute high-rate processing to specialized processing elements, as in the HP DesignJet, but they are most useful when the processing elements must be physically distributed. When the processing elements are sufficiently far apart, busses designed for lumped microprocessor systems do not work well. The network is generally used for data transfer between the processing elements, with each processing element maintaining its own program memory as well as a local data memory. The processing elements communicate data and control information as required by the application. $I^2C$ and CAN are two widely-used networks for distributed systems.

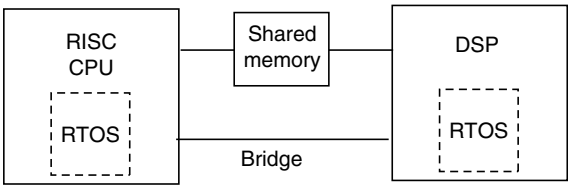



**FIGURE 5.3** The TI OMAP architecture.[10]

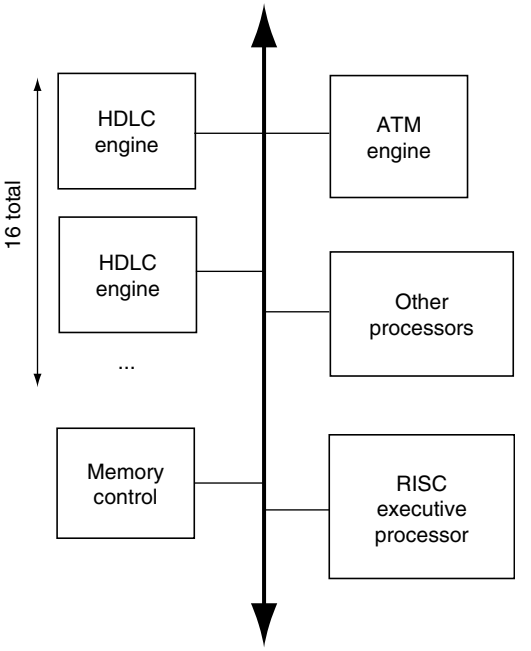



**FIGURE 5.4** Block diagram of the C-Port network processor.[11]

## 5.5 Embedded SoC Design Methodologies

### 5.5.1 Specifications

As described in Section 5.2, embedded computers are typically used to build systems with complex functionality. Therefore, capturing a functional description of the system is an important part of the design process. A variety of specification languages have been developed. Many of these languages were developed for software systems, but several languages have been developed over the past decade with embedded systems in mind.

Specification languages are generally designed to capture particular styles of design. Many languages have been created to describe control-oriented systems. An early example was Statecharts,[12] which introduced hierarchical states that provided a structured description of state machines. The SDL language[13] is widely used to specify protocols in telecommunications systems. The Esterel language[14] describes a reactive system as a network of communicating state machines.

Data-oriented languages find their greatest use in signal processing systems. Dataflow process networks[15] are one example of a specification language for signal processing. Object-oriented specification and design have become very popular in software design. Object-oriented techniques mix control and data orientation. Objects tend to reflect natural assemblages of data; the data values of an object define its state and the states of the objects define the state of the system. Messages providing communication and control. The real-time object-oriented Methodology (ROOM)[16] is an example of an object-oriented methodology created for embedded system design.

In practice, many systems are specified in the C programming language. Many practical systems combine control and data operations, making it difficult to use one language that is specialized for any type of description. Algorithm designers generally want to prototype their algorithms and verify them through experimentation; as a result, an executable program generally exists as the golden standard with which the implementation must conform. This is especially true when the product's capabilities are defined by standards committees, which typically generate one or more reference implementations,

usually in C. Once a working piece of code exists in C, there is little incentive to rewrite it in a different specification language; however, the C specification is generally a long way from an implementation. Algorithmic designs are usually written for uniprocessors and ignore many aspects of I/O, whereas embedded systems must perform real-time I/O and often distribute tasks among several processing elements. Algorithm designers often do not optimize their code for any particular platform, and their code is certainly not optimized for any particular embedded platform. As a result, a C language specification often requires substantial re-engineering before it can be used in an embedded system.

### 5.5.2 Design Flows

In order to better understand modern design methodologies for embedded SoCs, we can start with traditional software engineering methodologies. The waterfall model, one of the first models of software design, is shown in Fig. 5.5. The waterfall model is a top-down model with only local feedback. Other software design models, such as the spiral model, try to capture more bottom-up feedback from implementation to system design; however, software design methodologies are designed primarily to implement functionality and to create a maintainable design. Embedded SoCs must, as mentioned in Section 5.2, satisfy performance and power goals as well. As a result, embedded system design methodologies must be more complex.

The design of the architecture of an embedded SoC is particularly important because the architecture defines the capabilities that will limit both the hardware and software implementations. The architecture must of course be cost effective, but it must also provide the features necessary to do the job. Because the architecture is custom designed for the application, it is quite possible to miss architectural features that are necessary to efficiently implement the system. Retrofitting those features back into the architecture may be difficult or even impossible if the hardware and software design efforts do not keep in sync.

Important decisions about the hardware architecture include:

- How many processing elements are needed?
- What processing elements should be programmable and which ones should be hardwired?
- How much communication bandwidth is needed in the system and where is it needed?
- How much memory is needed and where should it go in the system?
- What types of components will be used for processors, communication, and memory?

The design of the software architecture is just as important and goes hand-in-hand with the hardware architecture design. Important decisions about the software architecture include:

- How should the functionality be split into processes?
- How are input and output performed?
- How should processes be allocated to the various processing elements in the hardware architecture?
- When should processes be scheduled?

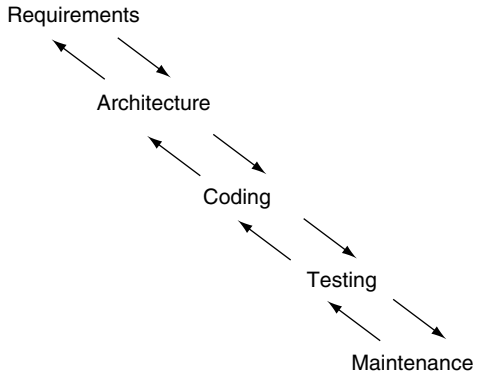



**FIGURE 5.5** The waterfall model of software development.

In practice, information required to make these decisions comes from several sources. One important source is previous designs. Though technology and requirements both change over time, similar designs can provide valuable lessons on how to (and not to) design the next system. Another important source is implementation. Some implementation information can come from pre-designed hardware or software components, which is

one reason why intellectual-property-based design is so important. Implementation can also come from early design efforts.

A variety of CAD algorithms have been developed to explore the embedded system design space and to help automate system architectures. Vulcan[21] and Cosyma[22] were early hardware/software partitioning systems that implemented a design using a CPU and one or more custom ASICs. Other algorithms target more general architectures.[23,24]

Once the system architecture has been defined, the hardware and software must be implemented. Hardware implementation challenges include:

- Finding efficient protocols to connect together existing hardware blocks
- Memory design
- Clock rate optimization
- Power optimization

Software implementation challenges include:

- Meeting performance deadlines
- Minimizing power consumption
- Minimizing memory requirements

The design must be verified throughout the design process. Once the design progresses to hardware and software implementation, simulation becomes challenging because the various components operate at very different levels of abstraction. Hardware units are modeled at the clock-cycle level. Software components must often be run at the instruction level or in some cases at even higher levels of abstraction. A hardware/software co-simulator[19] is designed to coordinate simulations that run at different time scales. The co-simulator coordinates multiple simulators—hardware simulators, instruction-level simulators, behavioral software processes—and keeps track of the time in each simulation. The co-simulator ensures that communications between the simulators happen at the right time for each simulator.

Design verification must include performance, power, and size as well as functionality. Although these sorts of checks are common in hardware design, they are relatively new to software design. Performance and power verification of software may require cache simulation. Some recent work has developed higher-level power models for CPUs.

### 5.5.3 Platform-Based Design

One response to the conflicting demands of SoC design has been the development of platform-based design methodologies. On the one hand, SoCs are becoming very complex. On the other hand, they must be designed very quickly to meet the electronics industry's short product lifecycles.

Platform-based design tries to tackle this problem by dividing the design process into two phases. In the first phase, a platform is designed. The platform defines the hardware and software architectures for the system. The degree to which the architecture can be changed depends on the needs of the marketplace. In some cases, the system may be customizable only by reprogramming. In other cases, it may be possible to add or delete hardware components to provide specialized I/O, additional processing capabilities, etc. In the second phase, the platform is specialized into a product. Because much of the initial design work was done in the first phase, the product can be developed relatively quickly based on the platform.

Platform-based design is particularly well suited to products derived from standards. On the one hand, all products must meet the minimum requirements of the standard. On the other hand, standards committees generally leave room for different implementations to distinguish themselves: added features, lower power, etc. Designers will generally want to modify their design to add features that differentiate their product in the marketplace.

Platform-based design also allows designers to incorporate design experience into products. Each product derived from the platform will teach something: how to better design part of the system, unexpected needs of customers, etc. The platform can be updated as new products are developed from it so that successive designs will be easier.

Platforms are usually designed within a technology generation. A new VLSI technology generally changes enough design decisions that platforms must be rethought for each new generation of technology. Therefore, the platform itself must be designed quickly and each product based on the platform must be completed quickly in order to gain effective use of the platform design effort in the 18-month lifecycle of a manufacturing technology.

### 5.5.4 Software Performance Analysis and Optimization

Although methods for hardware performance analysis and optimization are well-known, software techniques for optimizing performance have been developed only recently to meet the demands of embedded design methodologies.

The performance of an embedded system is influenced by several factors at different levels of abstraction. The first is the performance of the CPU pipeline itself. RISC design techniques tend to provide uniform execution times for instructions, but software performance is not always simple to predict. Register forwarding, which is used to enhance pipeline performance, also makes execution time less predictable. Branch prediction causes similar problems.

Superscalar processors, because they schedule instructions at execution time based upon execution data, provide much less predictable performance than do either RISC or VLIW processors. This is one reason why superscalar processors are not frequently used in real-time embedded systems.

The memory system is often an even greater source of uncertainty in embedded systems. CPUs use caches to improve average memory response time, but the effect of the cache on a particular piece of software requires complex analysis. In pathological cases, the cache can add uncertainty to execution times without actually improving the performance of critical software components. Cache simulation is often used to analyze the behavior of a program in a cache. Analysis must take into account both instructions and data. Unlike in workstation CPUs, in which the cache configuration is chosen by the CPU architect based on benchmarks, the designer of an embedded SoC can choose the configurations of caches to match the characteristics of the embedded software. Embedded system designers can choose between hardware and software optimizations to meet performance goals.

Analyzing the performance of a program requires determining both the execution path and the execution time of instructions along that path.[18] Both are challenging problems.[20] The execution path of a program clearly depends on input data values. To ensure that the program meets a deadline, the worst-case execution path must be determined. The execution time of instructions along the path depend on several factors: data values, interactions between instructions, and cache state.

### 5.5.5 Energy/Power Analysis and Optimization

Many embedded systems must also meet energy and power goals as well as performance goals. The specification may impose several types of power requirements: peak power consumption, average power consumption, energy consumption for a given operation.

To a first-order, high-performance design is low-power design. Efficient implementations that run faster also tend to reduce power consumption, but trade-offs between performance and power in embedded system design. For example, the power consumption of a cache depends on both its size and the memory system activity.[25] If the cache is too small, too many references require expensive main memory accesses. If the cache is too large, it burns too much static power. Many applications exhibit a sweet spot at which the cache is large enough to provide most of the available performance benefit while not burning too much static power. Techniques have been developed to estimate hardware/software power consumption.[26]

System-level approaches can also help reduce power consumption.[27] Components can be selectively turned off to save energy; however, because turning a component on again may consume both time and energy, the decision to turn it off must be made carefully. Statistical methods based on Markov models can be used to create effective system-level power management methodologies.

## 5.6 Summary

Embedded computers promise to solve a critical design bottleneck for SoCs. Because we can design CPUs relatively independently of the programs they run and reuse those CPUs design across many chips, embedded computers help to close the designer productivity gap. Embedded processors, on the other hand, require that many design techniques traditionally reserved for hardware—deadline-driven performance, power minimization, size—must now be applied to software as well. Design methodologies for embedded SoCs must carefully design system architectures that will allow hardware and software components to work together to meet performance, power, and cost goals while implementing complex functionality.

## References

1. Wayne Wolf, *Computers as Components: Principles of Embedded Computer System Design*, San Francisco: Morgan Kaufman, 2000.
2. http://www.arm.com.
3. http://www.mips.com.
4. G. Goossens, J. van Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P.G. Paulin, "Embedded software in real-time signal processing systems: design technologies," *Proceedings of the IEEE*, 85(3), March 1997, pp. 436–453.
5. http://www.ti.com.
6. http://www.lucent.com/micro/starcore/motover.htm.
7. ARM Limited, *AMBA(TM) Specification (Rev 2.0)*, ARM Limited, 1999.
8. http://www.vsi.com.
9. C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, 20(1), 1973, pp. 46–61.
10. http://www.ti.com/sc/docs/apps/wireless/omap/overview.htm.
11. http://www.cportcorp.com/products/digital.htm.
12. D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, 8, 1987, pp. 231–274.
13. Anders Rockstrom and Roberto Saracco, "SDL—CCITT specification and description language," *IEEE Transactions on Communication*, 30(6), June 1982, pp. 1310–1318.
14. Albert Benveniste and Gerard Berry, "The synchronous approach to reactive real-time systems," *Proceedings of the IEEE*, 79(9), September 1991, pp. 1270–1282.
15. E.A. Lee and T.M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, 83(5), May 1995, pp. 773–801.
16. Bran Selic, Garth Gullekson, and Paul T. Ward, *Real-Time Object-Oriented Modeling*, New York: John Wiley and Sons, 1994.
17. Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd, *Surviving the SOC Revolution: A Guide to Platform-Based Design*, Kluwer Academic Publishers, 1999.
18. Chang Yun Park and Alan C. Shaw, "Experiments with a program timing tool based on source-level timing scheme," *IEEE Computer*, 24(5), May 1991, pp. 48–57.
19. David Becker, Raj K. Singh, and Stephen G. Tell, "An engineering environment for hardware/software co-simulation," in *Proceedings, 29th Design Automation Conference*, IEEE Computer Society Press, 1992, pp. 129–134.

20. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe, "Performance estimation of embedded software with instruction cache modeling," in *Proceedings, ICCAD-95*, IEEE Computer Society Press, 1995, pp. 380–387.
21. Rajesh K. Gupta and Giovanni De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design & Test*, 10(3), September 1993, pp. 29–41.
22. R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design & Test*, 10(4), December 1993, pp. 64–75.
23. Wayne Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Transactions on VLSI Systems*, 5(2), June 1997, pp. 218–229.
24. Asawaree Kalavade and Edward A. Lee, "The extended partitioning problem: Hardware/software mapping, scheduling, and implementation-bin selection," *Design Automation for Embedded Systems*, 2(2), March 1997, pp. 125–163.
25. Yanbing Li and Joerg Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proceedings, 35th Design Automation Conference*, ACM Press, 1998, pp. 188–194.
26. W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano, "Power estimation of embedded systems: a hardware/software codesign approach," *IEEE Transactions on VLSI Systems*, 6(2), June 1998, pp. 266–275.
27. L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Transactions on VLSI Systems*, 8(3), June 2000, pp. 299–316.

# 6

# Embedded Processor Applications




Jonathan W. Valvano
*University of Texas at Austin*


## 6.1 Introduction

This chapter overviews the field of embedded processors and their applications. Basic concepts will first be introduced, followed by the examples of embedded systems. Each topic includes a problem statement, defines relevant terminology, discusses alternative hardware and software, and concludes with a specific solution. A systems-level approach to embedded processor applications is achieved by presenting a few case studies that illustrate the spectrum of applications that employ microcomputers.

As shown in Figure 6.1, the term embedded microcomputer system refers to an electronic device that contains one or more microcomputers inside. In this context, the word *embedded* means "hidden inside so we cannot see it." A *computer* is an electronic device with a processor, memory, and input/output (I/O) ports. Embedded systems perform very specific and predefined operations. The *processor* includes registers (which are high-speed memory), an arithmetic logic unit or ALU (to execute math functions), a bus interface unit or BIU (which communicates with memory and I/O), and a control unit or CU (for making decisions). *Memory* is high-speed storage containing software and data. Software consists of a sequence of commands, which in simple systems are executed in order. Complex systems implement multithreading, allowing many software sequences to be concurrently active. In an embedded system, we use read only memory (ROM) for storing the software and fixed constant data, and we use random access memory (RAM) for storing temporary information. The information in the ROM is nonvolatile,

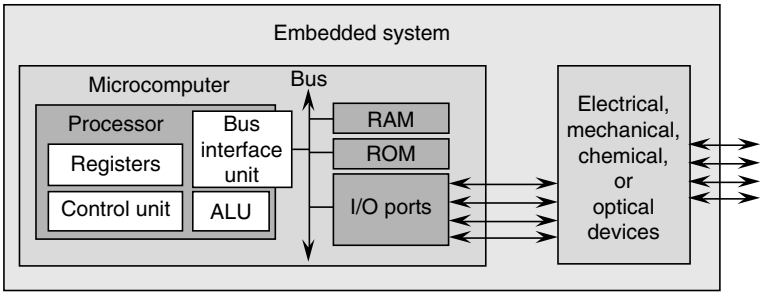



**FIGURE 6.1** An embedded system includes a microcomputer with electrical, mechanical, chemical, and optical devices. A microcomputer includes a processor with memory, input, and output devices.

meaning the contents are not lost when power is removed. I/O ports allow information to enter via the input ports and exit via the output ports. The I/O devices (e.g., parallel ports, serial ports, timer, DAC, ADC, and network ports) are a crucial part of an embedded system because they provide necessary functionality. Consequently, high-performance embedded processors, such as Freescale's 9S12X, contain a separate dedicated coprocessor just to manage I/O. The software together with the I/O ports and associated interface circuits gives an embedded computer system its distinctive characteristics.

In a developed country, we interact with embedded computers hundreds of times each day—at home, in car, at play, and at work. Table 6.1 illustrates the breadth of applications that use an embedded microcomputer [1].

**TABLE 6.1** Examples of Embedded Systems

| Category | Examples | What the Microcomputer Does? |
|---|---|---|
| Consumer | Washing machines | Controls the water and spin cycles |
| | TV remotes | Inputs from keys and sends IR |
| | Clocks and watches | Maintains the time and alarms |
| | Games and toys | Entertains the child |
| | Audio | Plays sounds and cancels noise |
| | Video | Plays and records movies |
| | Global positioning | Locates and directs position |
| Communication | Answering machines | Saves and organizes messages |
| | Phones and pagers | Communicates |
| | ATM machines | Maintains security and banking |
| Automotive | Automatic breaking | Stops on slippery surfaces |
| | Air bags | Improves safety |
| | Locks | Maintains security |
| | Electronic ignition | Controls sparks and fuel injectors |
| | Windows and seats | Remembers preferred settings |
| | Instrumentation | Collects and provides information |
| Military | Smart weapons | Recognizes friend vs. foe |
| | Missile guidance | Directs ordnance at the target |
| | Robotic vehicles | Gathers information |
| Industrial | Thermostats | Controls temperature |
| | Traffic control | Optimizes traffic flow |
| | Robot systems | Performs complex tasks |
| | Bar codes | Inventory control |
| Medical | Defibrillators | Restarts heartbeat |
| | Apnea monitor | Alarms if the baby stops breathing |
| | Pressure monitor | Measures heart function |
| | Pacemakers | Helps the heart to beat regularly |
| | Prosthetic devices | Increases mobility |

## 6.2 Embedded Processors

### 6.2.1 Embedded Microcomputer

The term *microcomputer* means a small computer. Small in this context describes its size and not its computing power; so a microcomputer can refer to a very wide range of products from the very simple to the most powerful Pentium-based PC. We typically restrict the term *embedded system* to refer to a system that does not look and behave like a typical computer. Most embedded systems do not have a keyboard, a graphics display, or secondary storage (disk). Embedded systems can be developed in two ways. The first technique uses the microcomputers that are available as single chips. These devices are suitable for low-cost, low-performance systems. On one end of the spectrum is the Microchip PIC10F200, which is a complete microcomputer in a 6-pin package with 256 by 12-bit ROM, 16 bytes RAM, and four I/O pins costing about 50¢. On the other end of the performance spectrum is the Freescale 9S12X, which is also a single-chip microcomputer, but it has 119 I/O pins, 512 MB of ROM, 32 KB of RAM, and runs at speeds up to 40 MHz costing about $15. For systems requiring more memory or faster execution, we can develop an embedded system around the PC architecture. These systems are first developed on a standard PC, and then the software and hardware are migrated to a stand-alone embedded-PC platform, such as the PC/104 (www.pc104.com).

The goal of this first design example is to control the room temperature within a range of 68°F–73°F. A sensor, which gives the current temperature in °F, is connected to input Port A. A heater is connected to output Port B, such that the heater turns off if Port B is 0, and the heater turns on if Port B is 1. This example also illustrates how an embedded system performs input/output. When an instruction is executed, the microprocessor often must refer to memory to read or write information. From a programming standpoint, most I/O ports are implemented as memory locations. For example, on the Freescale 6808 (www.freescale.com), I/O ports A and B exist as locations 0 and 1. So, when the program reads from or writes to locations 0 and 1, it is performing I/O. Most microcomputers allow the programmer to configure the I/O ports as inputs or outputs. The 6808 ports A and B have direction registers at locations 4 (DDRA) and 5 (DDRB), respectively. The software writes 0's to the direction register to specify the pins as inputs, and 1's to specify them as outputs. When the 6808 software reads from location 0, it inputs (gets information) from Port A (lda 0). When the software writes to location 1, it outputs (sends information) to Port B (sta 1). The solution to the thermostat problem is presented in Figure 6.2. This Freescale 6808 assembly language program reads from a sensor that is connected to Port A, if the temperature is above 73°F, it turns off the heat (by writing 0 to Port B). If the temperature is below 68°F, it turns on the heat by writing 1 to Port B.

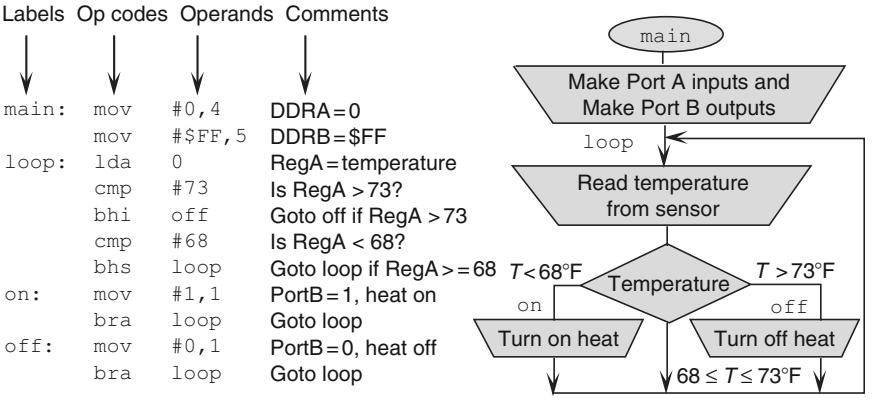



**FIGURE 6.2** A bang-bang temperature controller with hysteresis implemented using a Freescale 6808.

### 6.2.2 Choosing a Microcomputer

The computer engineer is often faced with the task of selecting a microcomputer for the project. Table 6.2 lists major manufacturers and their Web sites. Additional details can be found at http://www.microcontroller.com.

Often, only those devices for which the engineers have hardware and software experience are considered. Fortunately, this blind approach often yields an effective and efficient product, because many of the computers overlap in their cost and performance. In other words, if a microcomputer with which we are familiar can implement the desired functions for the project, then it is often efficient to bypass that perfect piece of hardware in favor of a faster development time. On the other hand, sometimes we wish to evaluate all potential candidates. Sometimes, it may be cost effective to hire or train the engineering personnel so that they are proficient in a wide spectrum of potential computer devices. Many factors must be considered when selecting an embedded microcomputer. The labor costs include training, development, and testing. The material costs include parts and supplies. The manufacturing costs depend on the number and complexity of the components. The maintenance costs involve revisions to fix bugs and perform upgrades. The ROM size must be big enough to hold instructions and fixed data for the software. The RAM size must be big enough to hold locals, parameters, and global variables. The EEPROM size must be big enough to hold nonvolatile fixed constants that are field configurable. The speed must be fast enough to execute the software in real time. The I/O bandwidth affects how fast the computer can input/output data. The data size (8-, 16-, or 32-bit) should match most of the data to be processed. Numerical operations like multiply, divide, signed, and floating point may be needed. Special functions like multiply and accumulate, fuzzy logic, and complex numbers are sometimes required. There must be enough parallel ports for all the I/O digital signals. The microcomputer needs the current number and type of network ports to interface with other computers or I/O devices. The timer functions can be used to generate signals, measure frequency, and measure period. Pulse-width modulation (PWM) is convenient for the output signals in many control applications. A DAC is used to convert digital numbers to analog outputs. An ADC is used to convert analog inputs to digital numbers. The package size and environmental issues affect many embedded systems. To meet manufacturing deadlines, the availability of a second source is

**TABLE 6.2** Web Sites of Companies That Make Microcontrollers

| Company | Example Products | Web Site |
|---|---|---|
| Analog devices | ADuC | http://www.analog.com |
| Atmel | ARM, AVR, 8051 | http://www.hitachi.com |
| Cypress | H8 | http://www.cypress.com |
| Dallas/Maxim | 8051 | http://www.maxim-ic.com |
| Freescale | HC08, 9S12, PowerPC, Arm, ColdFire | http://www.freescale.com |
| Hitachi/Renesas | H8 | http://www.renesas.com |
| Infineon | C16x, C500, TriCore | http://www.infineon.com |
| Intel | 8051, 8096, ARM | http://www.intel.com |
| Microchip | PIC | http://www.microchip.com |
| Mitsubishi | 740 7600 7700 M16C | http://www.mitsubishichips.com |
| National | CompactRISC | http://www.national.com |
| NEC | 78 K, V850 | http://www.nec.com |
| NetSilicon | ARM | http://www.netsilicon.com |
| Philips | 8051 | http://www.philips.com |
| Rabbit | 2000, 3000, 4000 | http://www.rabbitsemiconductor.com |
| Silicon Labs | 8051 | http://www.silabs.com |
| STMicroelectronics | ST6, ST7, ST9, ARM | http://www.st.com |
| Siemens | C500 C166 TriCore | http://www.siemens.com |
| Toshiba | TMP86 | http://www.toshiba.com |
| Texas Instruments | TMS320, MSP430 | http://www.ti.com |
| Zilog | Z8 | http://www.zilog.com |

advantageous. The availability of high-level language cross-compilers, simulators, emulators will facilitate software development. The power requirements will be important if the systems are battery operated.

When considering speed, it is best to compare time to execute a benchmark program similar to your specific application, rather than just comparing bus frequency. One of the difficulties is that the selection of microcomputer depends on the speed and size of the software, but the software cannot be written without the computer. Given this uncertainty, it is best to select a family of devices with a range of execution speeds and memory configurations. In this way, a prototype system is first simulated; then once the feasibility is confirmed, the specific version of the microcomputer can be selected, knowing the memory and speed requirements for the project.

## 6.3 Software Systems

### 6.3.1 Assembly Language

An *assembly language* program, like the one shown in Figure 6.2, has a one-to-one mapping with the machine code of the computer. In other words, one line of assembly code maps into a single machine instruction. The label field associates the absolute memory address with a symbolic label. The op code represents the machine instruction to be executed. The operand field identifies the data itself or the memory location for the data needed by the instruction. The comment field is added by the programmer to explain what, how, and why. The comments are not used by the computer during execution, but rather provide a means for one programmer to communicate with another, including oneself at a later time. This style of programming offers the best static efficiency (smallest program size) and best dynamic efficiency (fastest program execution). Another advantage of assembly language programming is the complete freedom to implement any arbitrary decision function or data structure. We are not limited to a finite list of predefined structures as is the case with high-level languages. For example, we can write assembly code with multiple entry points (places to begin the function).

### 6.3.2 High-Level Languages

Although assembly language enforces no restrictions on the programmer, most software developers argue that the limits placed on the programmer by a structured language, in fact, are a good idea. Building program and data structures by combining predefined components makes it easy to implement modular software that is easier to debug, verify correctness, and modify in the future. *Software maintenance* is the debug, verify, and modify cycle, and it represents a significant fraction of the effort required to develop products using embedded computers. Therefore, if the use of a high-level language sacrifices some speed and memory performance, but reduces the maintenance costs, most computer engineers will choose reliability and ease of modification over speed and memory efficiency. Cross-compilers for C, C++, Forth, and Basic are available for many single-chip microcomputers with C being the most popular.

One of the best approaches to this assembly versus high-level language choice is to implement the prototype in a high-level language, and see if the solution meets the product specifications. If it does, then leave the software in the high-level language because it will be easier to upgrade in the future. If the software is not quite fast enough (or small enough to fit into the available memory), then we might try a better compiler or a more powerful computer. Another approach is to profile the software execution, which involves collecting timing information on the percentage of time the computer takes executing each module. The profile allows us to identify a small number of modules that, if rewritten in assembly language, will have a big impact on the system performance.

### 6.3.3 Software Development

During the development phases of a project, we often would like the flexibility of accessing components inside the system. Therefore, the first step in product development is often *simulation.* Because of the high cost and long times required to create hardware prototypes, many preliminary feasibility designs

are now performed using hardware/software simulations. A simulator is a software application that models the behavior of the mechanical, electrical, and software components. If the external environment, hardware interfaces, and software program are simulated together, although the simulated time is slower than the actual time, the real-time hardware/software interactions can be studied. Potential solutions can be evaluated quickly. The qualities of a good simulator include accuracy, the ability to recreate a realistic external environment, and the flexible control over visibility [2].

*Debugging* embedded systems on the actual hardware system is very difficult for two reasons. First, the embedded system lacks the usual keyboard and display that assist us when we debug regular software. Second, the nature of embedded systems involves the complex and real-time interaction between the hardware and software. These real-time interactions make it impossible to test software with the usual single-stepping and print statements.

A *logic analyzer* is a multiple-channel digital oscilloscope. The logic analyzer can be used to record the digital signals at the microcomputer I/O ports. The advantages of the logic analyzer are very high-bandwidth recording (100 MHz–1 GHz), many channels (16–132 inputs), flexible triggering and clocking mechanisms, and personality modules that assist in interpreting the data.

Another approach is to use an in-circuit emulator (ICE). An ICE is a complex digital hardware device, which emulates (behaves in a similar manner) the I/O pins of the microcomputer in real time. The emulator is usually connected to a personal computer, so that emulated memory, I/O ports, and registers can be loaded and observed. To use an emulator, we first remove the microcomputer chip from the circuit and then attach the emulator pod into the socket where the microcomputer chip used to be. The only disadvantage of the ICE is its cost. To provide some of the benefits of this high-priced debugging equipment, some microcomputers have a background debug module (BDM). The BDM hardware exists on the microcomputer chip itself and communicates with the debugging personal computer via a dedicated 3-pin serial interface. Although not as flexible as an ICE, the BDM can provide the ability to observe software execution in real time, to set break points, to stop the computer, and to read and write registers, I/O ports, and memory.

### 6.3.4 Memory Allocation

Embedded systems group together in physical memory information that has similar logical properties. Because the embedded system does not load programs off disk when started, allocation is an extremely important issue for these systems. Typical software segments include global variables, local variables, fixed constants, and machine instructions. For single-chip implementations, different types of information are stored into the three types of memory. RAM is volatile and has random and fast access. EEPROM is nonvolatile and can be erased and reprogrammed. ROM is nonvolatile but can be programmed only once.

In an embedded application, structures that must be changed during execution are usually put in RAM. Examples include recorded data, parameters passed to subroutines, and global and local variables. We place fixed constants in EEPROM because the information remains when the power is removed, but can be reprogrammed at a later time. Examples of fixed constants include translation tables, security codes, calibration data, and configuration parameters. We place machine instructions, interrupt vectors, and the reset vector in ROM because this information is stored once and will not need to be reprogrammed in the future.

## 6.4 Interfacing

### 6.4.1 Digital Logic

Many logic families are available to design digital circuits. Each family provides the basic logic functions (AND OR NOT), but differs in the technology used to implement these functions. This results in a wide range of parameter specifications. Basic parameters of digital devices can be found in Refs. [1,3].

In general, it is desirable to design digital systems using all components from the same family. There are four basic considerations when using digital logic: speed, voltage levels, current levels, and capacitive loading.

One trend in the microcomputer-embedded systems field is the desire to implement higher and higher levels of functionality into smaller and smaller amounts of space using less and less power. Many examples of technology were developed according to these principles. Examples include portable computers, satellite communications, aviation devices, military hardware, and cellular phones. Simply using a microcomputer provides significant advantages in this faster–smaller race. The embedded system is not just a computer; so, there must also be mechanical and electrical devices external to the computer. To shrink the size and power required of these external electronics, they can be integrated into a custom integrated circuit (IC) called an application-specific integrated circuit (ASIC). An ASIC provides a high level of functionality squeezed into a small package. Advances in IC design allow more and more of these custom circuits (both analog and digital) to be manufactured in the same IC chip as the computer itself. In this way, single-chip solutions are possible.

### 6.4.2 Real-Time Systems

The microcomputer typically responds to external events with an appropriate software action. The time between the external event and the software action is defined as the interface latency. If an upper bound on the latency can be guaranteed, the system is characterized as real time or hard real time. If the system allows one software task to have priority over the others, then it is described as soft real time. Because most real-time systems utilize interrupts to handle critical events, we can calculate the upper bound on the latency as the sum of three components: (1) maximum time the software executes with interrupts disabled (e.g., other interrupt handlers, critical code); (2) the time for the processor to service the interrupt (saving registers on stack, fetching the interrupt vector); and (3) software delays in the interrupt handler before the appropriate software action is performed. Examples of events that sometimes require real-time processing include input, output, and alarms. When new input data are ready, the software must respond by reading the new input. When the output device is idle, the software must respond by giving more data to output. When an alarm condition occurs, the software must process the data until the time the alarm is processed.

Sometimes, the software must respond to internal events. Many real-time systems involve performing software tasks on a fixed and regular rate. For these systems, a periodic interrupt is employed to generate requests at fixed intervals. The microcomputer clock guarantees that the interrupt request is made exactly on time, but the software response (latency) may occur later. Examples of real-time systems that utilize periodic interrupts include data acquisition systems (DASs) where the ADC is sampled at a fixed rate, control systems where the software executes at the controller rate, waveform generation (e.g., music player) where the data are sent to the DAC at a fixed rate, and time of day clocks where the software maintains the date and time.

### 6.4.3 Keyboard Interfacing

Individual buttons and switches can be interfaced to a microcomputer input port simply by converting the on/off resistance to a digital logic signal with a pull-up resistor. When many keys are to be interfaced, it is efficient to combine them in a matrix configuration. $n^2$ Keys can be constructed as an $n$ by $n$ matrix. To interface the keyboard with $n^2$ keys, $2n$ I/O ports are needed; the rows to open collector (or open drain) microcomputer outputs and the columns to microcomputer inputs are connected. Open collector means the output will be low if the software writes a 0 to the output port, but will float (high impedance) if the software writes a 1. Pull-up resistors on the inputs will guarantee the column signals will be high if no key is touched in the selected row. The software scans the key matrix by driving one row at a time to 0, while the other rows are floating. If there is a key touched in the selected row, then the corresponding column signal will be 0. Many switches will bounce on/off for about 10 ms when touched or released. The software must read the switch position multiple times over a 20 ms

period to guarantee a reliable reading. One simple software method uses a periodic interrupt (with a rate slower than the bounce time) to scan the keyboard. In this way, the software will properly detect single-key touches. One disadvantage of the matrix-scanned keyboard is the fact that three keys simultaneously pressed sometimes looks like four keys are pressed.

### 6.4.4 Finite State Machine Controller

The objective of the second design example is to control traffic at an intersection of two one-way streets. This example also illustrates programmable logic and memory allocation. The finite state machine (FSM) has two inputs from sensors that identify the presence of cars as shown in Figure 6.3. There are also six outputs: red/yellow/green for the north road and red/yellow/green for the east road. In this FSM, each state has a 6-bit output value, a time to wait in that state, and four next states depending on if the input is 00 (no cars), 01 (car on the north road), 10 (car on the east road), or 11 (cars on both roads). The software will output the pattern for the current state, wait for the specified amount of time, input from the sensors, and jump to the next state depending on the input. The FSM table structure will be defined in EEPROM, and the program will be stored in ROM. The software for this system exhibits the three classic segments: RAM, EEPROM, and ROM. Since the variables (`Input`, `Pt`) have values that change during execution, they must be defined in RAM. We should be able to make minor modifications to the FSM (e.g., add/delete states, change I/O values) by changing the FSM data structure in EEPROM without modifying the controller (`main`) in ROM. A C-level program written for a 4 MHz Freescale 9S12 microcontroller is as follows:

```
const struct State{          // const means put in EEPROM
 unsigned char Out;          // 6-bit Output
 unsigned char Time;         // Time to wait in seconds
 unsigned char Next[4];};    // Next state if input=00,01,10,11
typedef const struct State StateType;
#define GoN     0   // Green on North, Red on East
#define WaitN   1   // Yellow on North, Red on East
#define GoE     2   // Red on North, Green on East
#define WaitE   3   // Red on North, Yellow on East
StateType fsm[4]={
  {0x21,  100, {GoN, GoN, WaitN, WaitN}},   // GoN        EEPROM
  {0x22,    8, {GoE, GoE, GoE, GoE}},       // WaitN      EEPROM
  {0x0C,  100, {GoE, WaitE, GoE, WaitE}},   // GoE        EEPROM
```

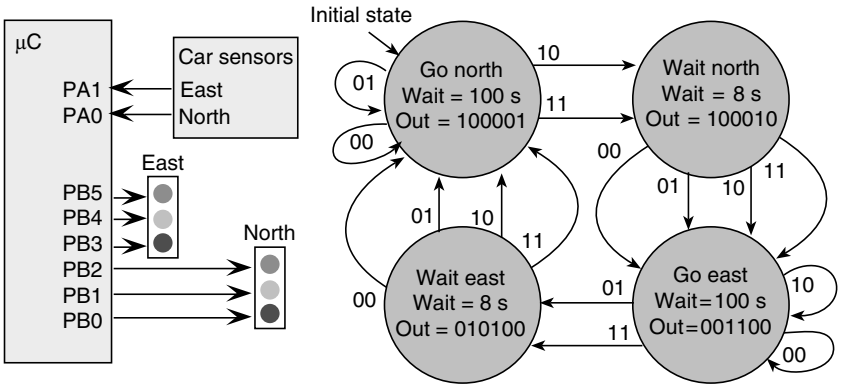



**FIGURE 6.3** A traffic controller implemented using a finite state machine (FSM).

```
  {0x14,   8,  {GoN, GoN, GoN, GoN}};          // WaitE    EEPROM
void Wait(unsigned short delay){               //          ROM
unsigned short startTime;                      //          RAM
  for(;delay>0;delay--){                       //          ROM
    startTime=TCNT;                            //          ROM
    while((TCNT-startTime)<= 31250){}}}        //          ROM
void main(void){              //                          ROM
 unsigned char Input;         //                          RAM
 unsigned int St;             //  Current State           RAM
 TSCR1=0x80;                  //  Enable TCNT             ROM
 TSCR2=0x07;                  //  divide by 128, 32us TCNT ROM
 St=GoN;                      //  Initial State           ROM
 DDRA=0, DDRB=0xFF;           //  A is input, B is output ROM
 while(1){                    //                          ROM
  PORTB=fsm[St].Out;          //  output for this state   ROM
  Wait(fsm[St].Time);         //  wait in this state      ROM
  Input=PORTA&0x03;           //  Input=00 01 10 or 11    ROM
  St=fsm[St].Next[Input];}}   //                          ROM
```

### 6.4.5 Current-Activated Output Devices

Many external devices used in embedded systems activate with a current, and deactivate when no current is supplied. The control element can be either a diode or a coil (resistor and inductor combination). The microcomputer controls the device by passing current or no current through the control element. Coil devices include electromagnetic relays, solenoids, DC motors, and stepper motors. Diode-based devices include LEDs, optosensors, optical isolation, and solid-state relays. Diode-based devices require a current-limiting resistor. The value of the resistor determines the voltage ($V_d$), current ($I_d$) operating point. The coil-based devices require a snubber diode to eliminate the large back EMF (>200 V) that develops when the current is turned off. The back EMF is generated when the large $dI/dt$ occurs across the inductance of the coil. The microcomputer output pins do not usually have a large enough $I_{OL}$ to drive these devices directly, so an open collector gate (such as the 7405, 7406, 75492, 75451, or NPN transistors) can be used to sink current to ground or use an open emitter gate (like the 75491 or PNP transistors) to source current from the power supply. The L293 has Darlington transistors both as current sinks (open collector) and current sources (open emitter) configured in an H-bridge, allowing the computer to control the amount and direction of the current. A device with an output current larger than the current required by the control element needs to be selected.

### 6.4.6 Stepper Motors

A bipolar stepper motor has only two coils and four wires, as shown in Figure 6.4. Using the full-step algorithm, current always passes through both coils. The computer controls a bipolar stepper by reversing the direction of the currents. If the computer generates the sequence (positive, positive) (negative, positive) (negative, negative) (positive, negative), the motor will spin. A unipolar stepper motor is controlled by passing current through four coils, exactly two at a time. There are five or six wires on a unipolar stepper motor. For both types of stepper motors, the software outputs the sequence 1010, 1001, 0101, 0110 to spin the motor. The software makes one change (e.g., change from 1001 to 0101) to affect one step. The software repeats the entire sequence over and over at regular time intervals between changes to make the motor spin at a constant rate. Some stepper motors will move on half steps by outputting the sequence 1010, 1000, 1001, 0001, 0101, 0100, 0110, 0010. Assuming the motor torque is large enough to overcome the mechanical resistance (load on the shaft), each output change causes the motor to step a predefined angle. One of the key parameters that determine whether the motor will slip (a computer change without the shaft moving) is the jerk, which is the derivative of the acceleration

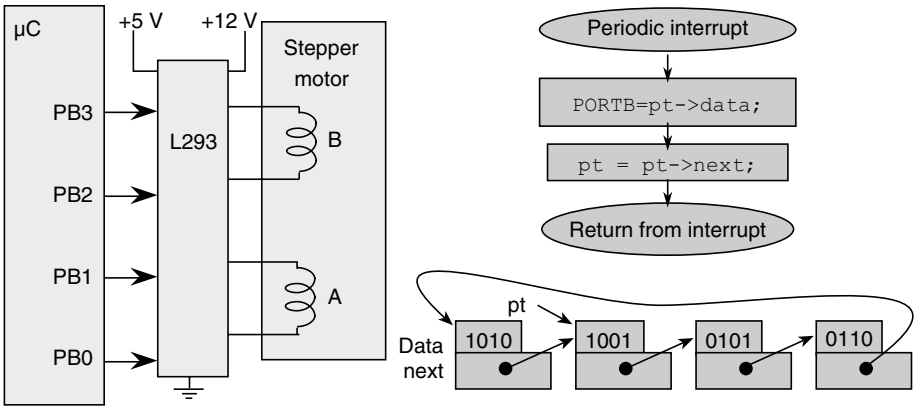



**FIGURE 6.4** A hardware interface of a bipolar stepper motor and the linked list data structure used by the software to spin the motor.

(i.e., third derivative of the shaft position). Software algorithms that minimize jerk are less likely to cause a motor slip. If the computer outputs the sequence in the opposite order, the motor spins in the other direction. A circular linked list data structure, as shown in Figure 6.4, is a convenient software implementation that guarantees the proper motor sequence is maintained.

## 6.5 Data Acquisition Systems

Before designing a DAS, the system goals must be clearly understood. The system can be classified as a quantitative DAS, if the specifications can be defined explicitly in terms of desired range, resolution, precision, and frequencies of interest. If the specifications are more loosely defined, we classify it as a qualitative DAS. Examples of qualitative DASs include systems that mimic the human senses where the specifications are defined using terms like "sounds good," "looks pretty," and "feels right." Other qualitative DASs involve the detection of events. In these types of systems, the specifications are expressed in terms of specificity and sensitivity. For example, some premature infants stop breathing during sleep. If we can detect this event and wake up the baby, the baby will start breathing again. An apnea monitor is attached to the baby as it sleeps to alert the parents to this life-threatening event. Other binary detection systems include the presence/absence of a burglar or the presence/absence of cancer. A true positive (TP) is defined when the condition exists (the baby stops breathing) and the system properly detects it (the alarm rings). A false positive (FP) is defined when the condition does not exist (the baby is breathing normally) but the system thinks it exists (the alarm rings). A false negative (FN) occurs when the condition exists (the baby stops breathing) but the system does not think it exists (the alarm is silent and baby dies). *Prevalence* is the probability the condition exists, sometimes called pretest probability, e.g., what percentage of babies will stop breathing. *Sensitivity* is the fraction of properly detected events (the baby stops breathing and the alarm rings) over the total number of events (the baby stops breathing). It is a measure of how well our system can detect an event. A sensitivity of 1 means the baby will not die. *Specificity* is the fraction of properly handled nonevents (the baby is breathing and the alarm is silent) over the total number of nonevents (the number of normal babies). A specificity of 1 means there will be no false alarms. The positive predictive value (PPV) of a system is the probability that the condition exists when restricted to those cases where the instrument says it exists. It is a measure of how much we believe the system is correct when it says it has detected an event. A PPV of 1 means that when the alarm rings, the baby is not breathing. The negative predictive value (NPV) of a system is the probability that the condition does not exists when restricted to those cases where the instrument says it does not exist. An NPV of 1 means if our instrument says the baby is breathing then that the baby is breathing.

$$\text{Prevalence} = (\text{TP} + \text{FN})/(\text{TP} + \text{TN} + \text{FP} + \text{FN}) \tag{6.1}$$

$$\text{Sensitivity} = \text{TP}/(\text{TP} + \text{FN}) \tag{6.2}$$

$$\text{Specificity} = \text{TN}/(\text{TN} + \text{FP}) \tag{6.3}$$

$$\text{PPV} = \text{TP}/(\text{TP} + \text{FP}) \tag{6.4}$$

$$\text{NPV} = \text{TN}/(\text{TN} + \text{FN}) \tag{6.5}$$

Many components are included in a DAS. The transducer converts the physical signal into an electrical signal. The amplifier converts the weak transducer electrical signal into the range of the ADC (e.g., 0–5 V). The analog filter removes unwanted frequency components within the signal. The analog filter is required to remove aliasing error caused by the ADC sampling. A periodic interrupt is used to control the sampling process. The interrupt service routine will sample the ADC and store the data in a first-in first-out queue. The data will be processed in the foreground by the main program. Examples of digital processing include digital filters, calibration calculations, event detection, and data display. Inherent in digital signal processing is the requirement that the ADC be sampled on a fixed time basis. Sampling at a known and fixed rate is particularly important when a digital filter is used.

The first decision to make is the ADC precision, which is the number of bits in the ADC. Whether we have a qualitative or quantitative DAS, we choose the number of bits in the ADC so as to achieve the desired system specification. For a quantitative DAS, this is a simple task because the relationship between the ADC precision and the system measurement precision is obvious. For a qualitative DAS, experimental trials are often employed to evaluate the relationship between ADC bits and system performance.

The next decision is the sampling rate, $f_s$. The Nyquist theorem states we can reliably represent a band-limited analog signal in digital form if we sample faster than twice the largest frequency that exists in the analog signal. For example, if an analog signal only has frequency components in the 0–100 Hz range, then if sample is taken at a rate above 200 Hz, the entire signal can be reconstructed from the digital samples. One of the reasons for using an analog filter is to guarantee that the signal at the ADC input is band limited. Violation of the Nyquist theorem results in aliasing. Aliasing is the distortion of the digital signal that occurs when frequency components above 0.5 $f_s$ exist at the ADC input. These high-frequency components are frequency shifted or folded into the 0–0.5 $f_s$ range.

## 6.6 Control Systems

### 6.6.1 Digital Control Equations

A control system is a collection of mechanical and electrical devices connected for the purpose of commanding, directing, or regulating a physical plant. The real state variables are the actual properties of the physical plant that are to be controlled. The goal of the sensor and DAS is to estimate the state variables. Any differences between the estimated state variables and the real state variables will translate directly into controller errors. A *closed-loop* control system uses the output of the state estimator in a feedback loop to drive the errors to 0. The control system compares these estimated state variables, $x(n)$, to the desired state variables, $x^*$ in order to decide appropriate action, $u(n)$. The terminology $(n)$ refers to the fact that these parameters are digital values sampled at finite-time intervals, where $n$ is the sample number. The actuator is a transducer, which converts the control system commands, $u(n)$, into driving forces, which are applied to the physical plant. The goal of the control system is to drive $x(n)$ to equal $x^*$. If the error $e$ is defined as the difference between the desired and estimated state variable

$$e(n) = x^* - x(n) \tag{6.6}$$

then the control system will attempt to drive $e(n)$ to 0. We usually evaluate the effectiveness of a control system by determining three properties: steady-state controller error, transient response, and stability. The steady-state controller error is the average value of $e(n)$. The transient response is how long does the system take to reach 99% of the final output after $x^*$ is changed. A system is stable if steady state (smooth constant output) is achieved. An unstable system may oscillate.

### 6.6.2 Pulse-Width Modulation

Many embedded systems must generate output pulses with specific pulse widths. The internal microcomputer clock is used to guarantee the timing accuracy of these outputs. Many microcomputers have built-in hardware that facilitates the generation of pulses. One classic example is the pulse-width-modulated motor controller. The motor is turned on and off at a fixed frequency (see the *Out* signal in Figure 6.5). The value of this frequency is chosen to be too fast for the motor to response to the individual on/off signals. Rather, the motor responses to the average. The computer controls the power to the motor by varying the pulse width or duty cycle of the wave. The IRF540 MOSFET can sink up to 28 A. To implement PWM, the computer (either with the built-in hardware or the software) uses a clock. The clock is a simple integer counter that is incremented at a regular rate. The *Out* signal is set high for time $t_h$ then set low for time $t_l$. Since the frequency of *Out* is to be fixed, $(t_h + t_l)$ remains constant, but the duty cycle $t_h/(t_h + t_l)$ is varied. The precision of this PWM system is defined to be the number of distinguishable duty cycles that can be generated. Let $H$ and $L$ be integer numbers representing the number of clock counts the *Out* signal is high and low, respectively. We can express the duty cycle as $H/(H+L)$. Theoretically, the precision should be $H+L$, but practically the value may be limited by the speed of the interface electronics.

### 6.6.3 Period Measurement

To sense the motor speed, a tachometer can be used. The AC amplitude and frequency of the tachometer output both depend on the shaft speed. It is usually more convenient to convert the AC signal into a digital signal (*In* shown in Figure 6.5) and measure the period. Again, many microcomputers have built-in hardware that facilitates the period measurement. To implement period measurement, the computer (again either with the built-in hardware or the software) uses a clock. Period measurement simply records the time (value of the clock) of two successive rising edges on the input and calculates the

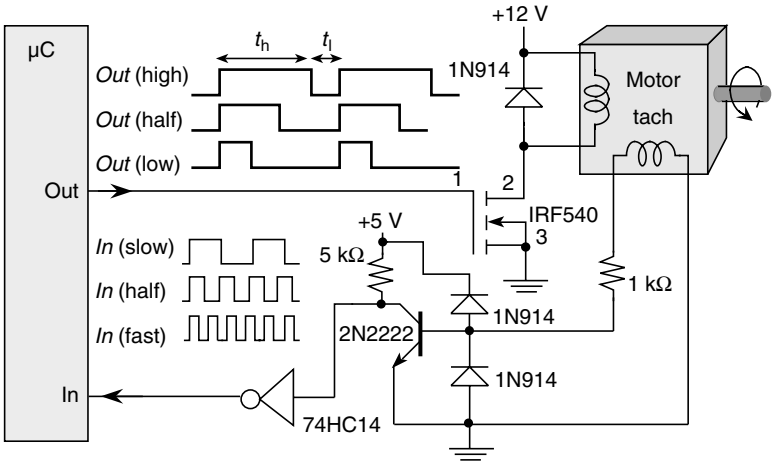



**FIGURE 6.5** An interface to a DC motor that uses PWM to control the delivered power and period measurement to determine the rotation speed.

time difference. The period measurement resolution is defined to be the smallest difference in period that can be reliably measured. Theoretically, the period measurement resolution should be the clock period, but practically the value may be limited by noise in the interface electronics. The software can calculate shaft speed, because the frequency is 1 over the period.

### 6.6.4 Control Algorithms

There are many common approaches to designing the software for the control system. The simplest approach to the closed-loop control system uses incremental control. In this motor control example, the actuator command, $u = H/(H + L)$, is the duty cycle of the pulse-width-modulated system. An incremental control algorithm simply adds or subtracts a constant from $u$ depending on the sign of the error. To add hysteresis to the incremental controller, we define two thresholds, $x_H$ and $x_L$, at values just above and below the desired speed, $x^*$. In other words, if $x < x_L$ (the motor is spinning too slow) then $u$ is incremented, and if $x > x_H$ (the motor is spinning too fast) then $u$ is decremented. It is important to choose the proper rate at which the incremental control software is executed. If it is executed too many times per second, then the actuator will saturate resulting in a bang-bang system like Figure 6.2. If it is not executed often enough, then the system will not respond quickly to changes in the physical plant or changes in $x^*$.

A second approach, called proportional integral derivative (PID), uses linear differential equations. To simplify the PID controller, we break the controller equation into separate proportion, integral, and derivative terms, where $p(n)$, $i(n)$, and $d(n)$ are the proportional, integral, and derivative components, respectively. In order to implement the control system with the microcomputer, it is imperative that the digital equations be executed on a regular and periodic rate (every $\Delta t$). The relationship between the real time, $t$ and the discrete time, $n$, is simply $t = n\Delta t$. If the sampling rate varies, then controller errors will occur. The proportional term makes the actuator output linearly related to the error. Using a proportional term creates a control system that applies more energy to the plant when the error is large.

$$p(n) = k_p e(n) \tag{6.7}$$

The integral term makes the actuator output related to the integral of the error. Using an integral term often will improve the steady-state error of the control system. If a small error accumulates for a long time, this term can get large. Some control systems put upper and lower bounds on this term, called anti-reset-windup, to prevent it from dominating the other terms. The implementation of the integral term requires the use of a discrete integral or sum. If $i(n)$ is the present control output and $i(n-1)$ is the previous calculation, the integral term is simply as follows:

$$i(n) = i(n-1) + k_i e(n)\Delta t, \quad \text{where} \quad i_{\min} \leq i(n) \leq i_{\max} \tag{6.8}$$

The derivative term makes the actuator output related to the derivative of the error. This term is usually combined with either the proportional or integral term to improve the transient of the control system. The proper value of $k_d$ will provide for a quick response to changes in either the set point or loads on the physical plant. An incorrect value may create an overdamped (very slow response) or an underdamped (unstable oscillations) response. There are a couple of ways to implement the discrete time derivative. A simple approach is

$$d(n) = k_d(x(n) - x(n-1))/\Delta t \tag{6.9}$$

In practice, this first-order equation is quite susceptible to noise. More sophisticated calculations can be found in Ref. [1]. The PID controller software is also implemented with a periodic interrupt every $\Delta t$. The interrupt handler first estimates the state variable, $x(n)$, and then calculates $e(n)$. The next actuator output is calculated by combining the three terms.

$$u(n) = p(n) + i(n) + d(n) \tag{6.10}$$

A third approach uses fuzzy logic to control the physical plant. Fuzzy logic can be much simpler than PID. It will require less memory and execute faster. When complete knowledge about the physical plant is known, then a good PID controller can be developed, i.e., the physical plant can be described with a linear system of differential equations, an optimal PID control system can be developed. Because the fuzzy logic control is more robust (still works even if the parameter constants are not optimal), then the fuzzy logic approach can be used when complete knowledge about the plant is not known or can change dynamically. Choosing the proper PID parameters requires knowledge about the plant. The fuzzy logic approach is more intuitive, following more closely to the way a human would control the system. If there is no set of differential equations that describe the physical plant, but there exists expert knowledge (human intuition) on how it works, then a good fuzzy logic system can be developed. It is easy to modify an existing fuzzy control system into a new problem. So if the framework exists, rapid prototyping is possible. Examples of fuzzy logic implementations can be found in Ref. [1].

## 6.7 Remote or Distributed Systems

Many embedded systems require the communication of command or data information to other modules at either a near or a remote location. We will limit our discussion with communication with devices within the same room as presented in Figure 6.6. A full-duplex channel allows data to transfer in both directions at the same time. In a half-duplex system, data can transfer in both directions but only in one direction at a time. Half duplex is popular because it is less expensive (two wires) and allows the addition of more devices on the channel without change to the existing nodes. If the distances are short, half duplex can be implemented with simple open collector TTL-level logic. Many microcomputers have open collector modes on their serial ports that allow a half-duplex network to be created without any external logic (although pull-up resistors are often used). Three factors will limit the implementation of this simple half-duplex network: (1) the number of nodes on the network, (2) the distance between nodes, and (3) the presence of corrupting noise. In these situations a half-duplex RS485 driver chip, such as the SP483 made by Sipex or Maxim, can be used. The master–slave system connects the master transmit output to all slave receive inputs. This provides for broadcast of commands from the master.

A very common approach to distributed embedded systems is called multidrop, as implemented in a controller area network (CAN). To transmit information to the other computers, the software activates the CAN driver and outputs the frame. The scheduling and error checking is handled by the low-level hardware [1].

Within the same room, IR light pulses can be used to send and receive information. This is the technology used in the TV remote control. In order to eliminate background EM radiation from triggering a false communication, the signals are encoded as a series of long and short pulses that resemble bar codes.

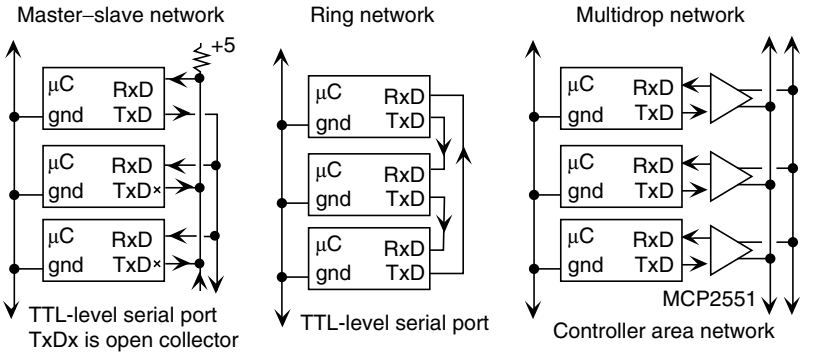



**FIGURE 6.6** Three simple configurations for distributed embedded systems.

A number of techniques are available for communicating across longer distances. Within the same building the Bluetooth protocol can be used. A second technique for longer distances is RF modulation. The information is modulated on the transmitted RF and demodulated at the receiver. Standard telephone modems and the Internet can also be used to establish long distance networks.

Two approaches are used to synchronize the multiple computers. In a master–slave system, one device is the master, which controls all the other slaves. The master defines the overall parameters that govern the functions of each slave and arbitrates requests for data and resources. This is the simplest approach but may require a high-bandwidth channel and a fast computer for the master. Collisions are unlikely in a master–slave system if the master can control access to the network.

The other approach is distributed communication. In this approach, each computer is given certain local responsibilities and certain local resources. Communication across the network is required when data collected in one node must be shared with other nodes. A distributed approach will be successful on large problems that can be divided into multiple tasks that can run almost independently. As the interdependence of the tasks increases so will the traffic on the network. Collision detection and recovery are required due to the asynchronous nature of the individual nodes.

## References

1. J.W. Valvano, *Embedded Microcomputer Systems: Real Time Interfacing*, 2nd ed., Thomson-Engineering Publishers, Toronto, Ontario, Canada, 2006.
2. J.W. Valvano, *Introduction to Embedded Microcomputer Systems: Motorola 6811 and 6812 Simulation*, Thomson-Engineering Publishers, Toronto, Ontario, Canada, 2002.
3. C.H. Roth, *Fundamentals of Logic Design*, Thomson-Engineering Publishers, Toronto, Ontario, Canada, 2004.

# 7

# An Overview of SoC Buses




Milica Mitić
Mile Stojčev
*University of Niš*

Zoran Stamenković
*IHP GmbH—Innovations for High Performance Microelectronics*


## 7.1 Introduction

The electronics industry has entered the era of multimillion-gate chips, and there is no turning back. This technology promises new levels of integration on a single chip, called the system-on-a-chip (SoC) design, but also presents significant challenges to the chip designers. Processing cores on a single chip may number well into the high tens within the next decade, given the current rate of advancements [1]. Interconnection networks in such an environment are, therefore, becoming more and more important [2]. Currently, on-chip interconnection networks are mostly implemented using buses. For SoC applications, design reuse becomes easier if standard internal connection buses are used for interconnecting components of the design. Design teams developing modules intended for future reuse can design interfaces for the standard bus around their particular modules. This allows future designers to slot the reuse module into their new design simply, which is also based around the same standard bus [3].

Shrinking process technologies and increasing design sizes have led to highly complex billion-transistor integrated circuits (ICs). As a consequence, manufacturers are integrating increasing numbers of components on a chip. A heterogeneous SoC might include one or more programmable components such as general-purpose processor cores, digital signal processor cores, or application-specific intellectual property (IP) cores, as well as an analog front end, on-chip memory, I/O devices, and other application-specific circuits. In other words, a SoC is an IC that implements most or all the functions of a complete electronic system [4].

On-chip bus organized communication architecture (CA) is among the top challenges in CMOS SoC technology due to rapidly increasing operation frequencies and growing chip size. In general, the performance of the SoC design heavily depends upon the efficiency of its bus structure. The balance of computation and communication in any application or task is, of course, known as a fundamental determinant of

delivered performance. Usually, IP cores, as constituents of SoCs, are designed with many different interfaces and communication protocols. Integrating such cores in a SoC often requires insertion of suboptimal glue logic. Standards of on-chip bus structures were developed to avoid this problem. Currently, there are a few publicly available bus architectures from leading manufacturers, such as CoreConnect from IBM [5], AMBA from ARM [6], SiliconBackplane from Sonics [7], and others. These bus architectures are usually tied to processor architecture, such as the PowerPC or the ARM processor. Manufacturers provide cores optimized to work with these bus architectures, thus requiring minimal extra interface logic.

This chapter gives an overview of the more popular on-chip standardized bus architectures such as AMBA, CoreConnect, Wishbone, STBus, and others, both from an industrial and research viewpoint. The crucial features, including bus topologies, arbitration methods, bus widths, and types of data transfers are considered.

The rest of this chapter is organized as follows: Section 7.2 presents background material on CAs, including a survey of typical topologies and communication protocols in use today. Section 7.3, as a central part of this chapter, gives an overview of several popular SoC CAs.

## 7.2 On-Chip Communication Architectures

### 7.2.1 Background

The design of on-chip CAs addresses the following three issues [8]:

*Definition of CA topology*: This defines the physical structure of the CA. Numerous topologies exist, ranging from single shared bus to more complex architectures such as bus hierarchies, token ring, crossbar, or custom networks.

*Selection and configuration of the communication protocols*: For each channel/bus in the CA, communication protocols specify the exact manner in which communication transaction occurs. These protocols include arbitration mechanisms (e.g., round-robin access, priority-based selection [5,6], time division multiplexed access [TDMA] [7]), which are implemented in centralized or distributed bus arbiters.

*Communication mapping*: This refers to the process of associating abstract system-level communications with physical communication paths in the CA topology [8].

### 7.2.2 Topologies

With respect to topology, on-chip communication architectures can be classified as follows:

*Shared bus*: The system bus is the simplest example of a shared communication architecture topology and is commonly found in many commercial SoCs [9]. Several masters and slaves can be connected to a shared bus. A block bus arbiter periodically examines accumulated requests from the multiple master interfaces and grants access to a master using arbitration mechanisms specified by the bus protocol. Increased load on a global bus lines limits the bus bandwidth. The advantages of shared-bus architecture include simple topology, extensibility, low area cost, efficient to implement, and easy to build. The disadvantages of shared-bus architecture are larger load per data bus line, longer delay for data transfer, larger energy consumption, and lower bandwidth. Fortunately, the above disadvantages, with the exception of the lower bandwidth, may be overcome by using a low-voltage swing signaling technique.

*Hierarchical bus*: This architecture consists of several shared buses interconnected by bridges to form a hierarchy. SoC components are placed at the appropriate level in the hierarchy according to the performance level they require. Low-performance SoC components are placed at lower performance buses, which are bridged to the higher performance buses so as not to burden the higher performance SoC components. Commercial examples of such architectures include the AMBA bus [6], CoreConnect [5], etc. Transactions across the bridge involve additional overhead, and during the transfer both buses remain inaccessible to other SoC components. Hierarchical buses offer large throughput improvements over the shared buses due to (1) the decreased load per bus and (2) the potential for transactions to proceed in parallel on different buses, and multiple ward communications can be preceded across the bridge in a pipelined manner [8].

*Ring*: In numerous applications, ring-based applications, such as network processors, and ATM switches are widely used [5,8]. In a ring, each node component (master–slave) communicates using a ring interface, and is usually implemented by a token-pass protocol.

### 7.2.3 On-Chip Communication Protocols

Communication protocols deal with different types of resource management algorithms used for determining access right to shared communication channels. From this point of view, in the rest of this section, we give a brief comment related to the main feature of the existing communication protocols.

*Static-priority*: This protocol employs an arbitration technique and is used in shared-bus communication architectures. A centralized arbiter examines accumulated requests from each master and grants access to the requesting master that is of the highest priority. Transactions may be of non-preemptive or preemptive type. AMBA, CoreConnect, etc., use this protocol [5,6].

*Time division multiple access*: The arbitration mechanism is based on a timing wheel with each slot statically reserved for a unique master. Special techniques are used to alleviate the problem of wasted slots. Sonics uses this protocol [7].

*Lottery*: A centralized lottery manager accumulates request for ownership of shared communication resources from one ore more masters, each of which has, statically or dynamically, assigned a number of lottery tickets [10].

*Token passing*: This protocol is used in ring-based architectures. A special data word, called token, circulates on the ring. An interface that receives a token is allowed to initiate a transaction. When the transaction completes, the interface releases the token and sends it to the neighboring interface.

*Code division multiple access* (CDMA): This protocol has been proposed for sharing on-chip communication channel. In a sharing medium, it provides better resilience to noise/interference and has an ability to support simultaneously transfer of data streams. But this protocol requires implementation of complex special direct sequence spread spectrum coding schemes, and energy/battery inefficient systems such as pseudorandom code generators, modulation and demodulation circuits at the component bus interfaces, and differential signaling [11].

### 7.2.4 Other Interconnect Issues

We now point to several interconnect issues that have direct impact on bus organization and its efficiency:

*Programming model*: This consists of a load and store operations. These operations are implemented as a sequence of primitive bus transactions. Modules issuing requests are called masters and those serving requests are called slaves [12].

*Split versus nonsplit buses*: If there is a single arbitration for a request–response pair, the bus is called nonsplit. In this case, the bus remains allocated to the master of the transaction until the response is delivered. Alternatively, in a split bus, the bus is released after the request to allow transactions from different masters to be initiated [13].

*Transaction ordering*: Usually, all transactions on a bus are ordered. However, on a split bus, a total ordering of transactions on a single master may cause performance degradation. This situation is typical when slaves respond at different speed. To solve this problem, recent extensions to bus protocols allow transactions to be performed on connections [14,15].

*Atomic chains of transactions*: This represents a sequence of transactions initiated by a single master that is executed on a single slave exclusively. During this activity, other masters are denied access to that slave until the end of the first transaction. This mechanism is standardly used to implement synchronization mechanisms between master modules (i.e., semaphores) [13].

*Media arbitration*: Bus master modules access the bus and the arbiter grants access. Arbitration is centralized as there is only one arbiter component. It is also global, since all requests, as well as the state of the bus, are visible to the arbiter. When a grant is given, the complete path from the source to the destination is exclusively reserved [12,13].

*Destination name and routing*: Command address and data are broadcasted on the bus. They reach every destination; only one of each activates, based on the broadcasted address, and executes the requested command [12,13].

*Latency*: This is caused by the following two factors: (1) the access time to the bus, which is the time until the bus is granted and (2) the latency introduced by the bus to transfer the data [12].

*Data format*: This is defined by separate wire groups for the transaction type, address, write data, read data, and return acknowledgments/errors [5,6,16,17].

### 7.2.5 Advantages and Disadvantages of On-Chip Buses

In the bus-based design approach, IP components communicate through one or more buses usually interconnected by bus bridges. Since the bus specification can be standardized, libraries of components whose interfaces directly match this specification can be developed. Even if components follow the bus standard, very simple bus interface adapters may still be needed. For components that do not directly match the specification, wrappers have to be built. Companies offer very rich component libraries and specialized development and simulation environments for designing systems around their buses. A somewhat different approach is a core-based design. In this case, IP components are compliant to a bus-independent and standardized interface and, thus, are directly connected to each other. Although the standard may support a wide range of functionalities, each component may have an interface containing only the functions that are relevant to it. These components may also be interconnected through a bus, in which case standard wrappers can adapt the component interface to the bus.

As a conclusion we can say that on-chip-bus-design and on-chip-core-based design methodologies are integration approaches that depend on standardized components or bus interfaces. They allow the integration of homogeneous IP components that follow these standards to be directly connected to each other, without requiring the development of complex wrappers. Let us note that on-chip buses rely on shared communication resources and on arbitration mechanism that is in charge of serializing bus access requests. This widely adopted solution unfortunately suffers from power and performance scalability limitations, and restricted sharing of resources between communicating entities. For bus networks, the bus is occupied by a single communication even if multiple communications could operate simultaneously on different portions on the bus. Therefore, a lot of effort has been devoted to the development of advanced bus topologies (e.g., partial or full crossbar, bridged buses) and protocols for better support of route-ability, flexibility, reliability, and reconfigure-ability. Therefore, a systematic way of designing networks with possibly arbitrary topology is gaining the importance [2].

In the long run, a more aggressive approach is needed. For particular needs, the SoC may be built around a sophisticated and dedicated network-on-chip that may deliver very high performance for connecting a large number of components. It seems that this design paradigm shifts toward a packetized on-chip communication based on micronetworks of interconnects or networks-on-chip [18]. More details concerning NoC design are given in Refs. [13,19,20].

## 7.3 System-On-Chip Buses

In the sequel, an overview of the more relevant SoC communication architectures will be given. Because of the space limitation, the discussion will be focused on describing the more distinctive features of each of them.

### 7.3.1 AMBA Bus

AMBA (advanced microcontroller bus architecture) [6,21] is a bus standard devised by ARM with aim to support efficient on-chip communications among ARM processor cores. Nowadays, AMBA is one of the leading on-chip busing systems used in high-performance SoC design. AMBA (see Figure 7.1) is hierarchically organized into two bus segments, system- and peripheral bus, mutually connected via bridge that

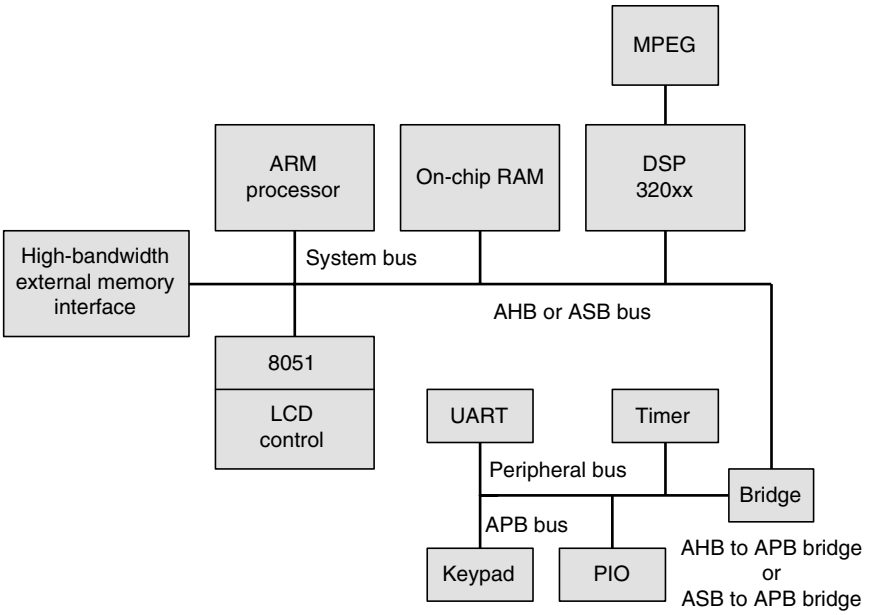



**FIGURE 7.1** AMBA-based system architecture.

buffers data and operations between them. Standard bus protocols for connecting on-chip components generalized for different SoC structures, independent of the processor type, are defined by AMBA specifications. AMBA does not define method of arbitration. Instead it allows the arbiter to be designed to suit the application needs, the best. The following are the three distinct buses specified within the AMBA bus:

1. *ASB (advanced system bus)*: First generation of AMBA system bus used for simple cost-effective designs that support burst transfer, pipelined transfer operation, and multiple bus masters.
2. *AHB (advanced high-performance bus)*: As a later generation of AMBA, this bus is intended for high-performance, high-clock synthesizable designs. It provides high-bandwidth communication channel between embedded processor (ARM, MIPS, AVR, DSP 320xx, 8051, etc.) and high-performance peripherals/hardware accelerators (ASICs MPEG, color LCD, etc.), on-chip SRAM, on-chip external memory interface, and APB bridge. AHB supports a multiple bus masters operation, peripheral and a burst transfer, split transactions, wide data bus configurations, and nontristate implementations. Constituents of AHB are AHB-master, -slave, -arbiter, and -decoder.
3. *APB (advanced peripheral bus)*: This bus is used to connect general-purpose low-speed, low-power peripheral devices. The bridge is peripheral bus master, whereas all bus devices (Timer, UART, PIA, etc.) are slaves. APB is static bus that provides a simple addressing with latched addresses and control signals for easy interfacing.

Recently, two new specifications for AMBA bus—multilayer AHB and AMBA AXI— are defined [6,22]. Multilayer AHB provides more flexible interconnect architecture (matrix that enables parallel access paths between multiple masters and slaves) with respect to AMBA AHB, and keeps the AHB protocol unchanged. AMBA AXI is based on the concept point-to-point connection. Good overview papers related to AMBA specifications are Refs. [6,22,23].

### 7.3.2 Avalon

Avalon bus (see Figure 7.2) is a bus architecture designed for connecting on-chip processors and peripherals together into a system-on-a-programmable-chip (SoPC). As an Altera's parameterized bus, Avalon is mainly used for FPGA SoC design based on Nios processor [24,25].

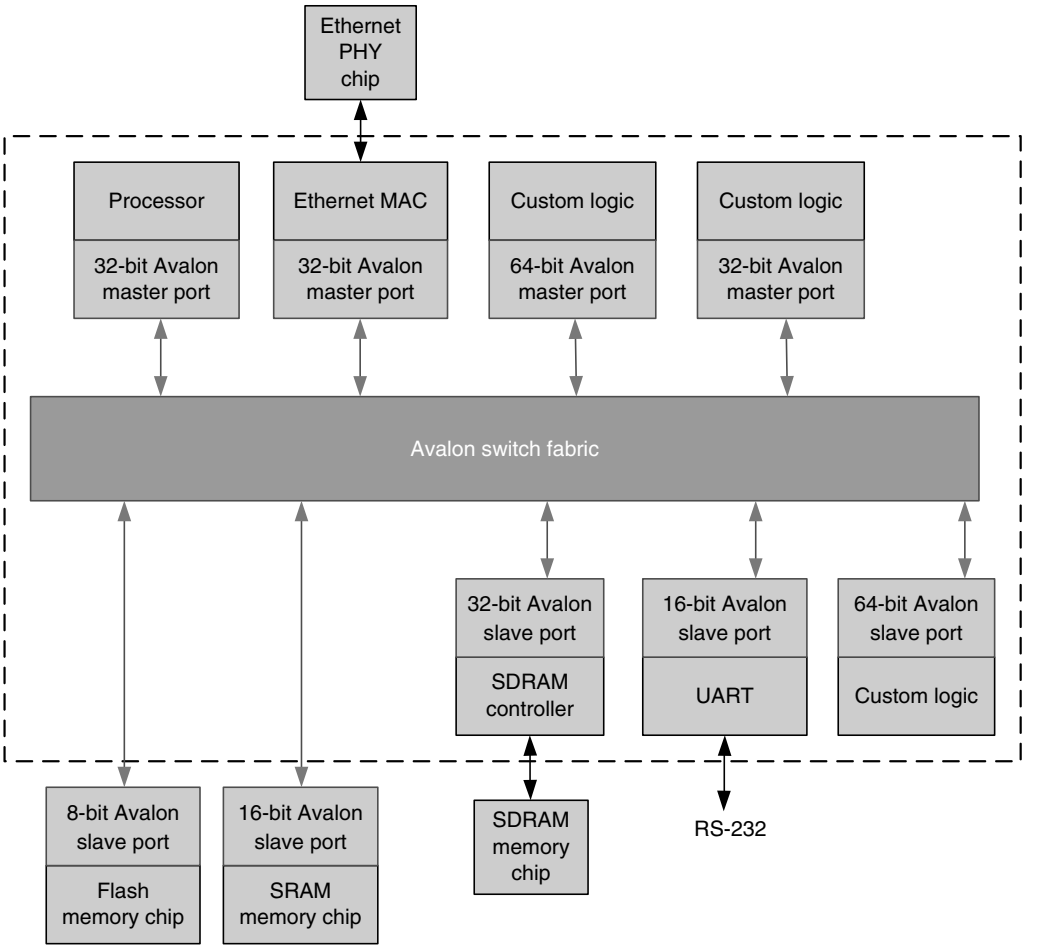



**FIGURE 7.2** Avalon bus-based system.

Avalon has a set of predefined signal types with which a user can connect IP blocks. Avalon is a synchronous interface and specifies the port connections between master and slave components and specifies the timing by which these components communicate. Basic Avalon bus transactions transfer one data item 8-, 16-, 32-, 64-, or 128-bits wide. Avalon uses separate address, data, and control lines.

This bus supports multiple bus masters. Masters and slaves interact with each other based on a technique called slave-side (distributed) arbitration.

The Avalon bus model (switch fabric) provides the following services to Avalon peripherals connected to the bus: data-path multiplexing, address decoding, wait-state generation, dynamic bus sizing, interrupt priority assignment, latent transfer capabilities, and a streaming read and write capabilities [24,25].

Altera's SoPC Builder, as a system development tool, automatically generates the switch fabric logic that supports each type of transfer supported by the Avalon interface.

### 7.3.3 CoreConnect

CoreConnect [5] is an IBM-developed on-chip bus. By reusing processor, subsystem, and peripheral cores, supplied from different sources, enables their integration into a single VLSI design. CoreConnect is a hierarchically organized architecture. It is comprised of three buses that provide an efficient interconnection of cores, library macros, and custom logic within a SoC (see Figure 7.3).

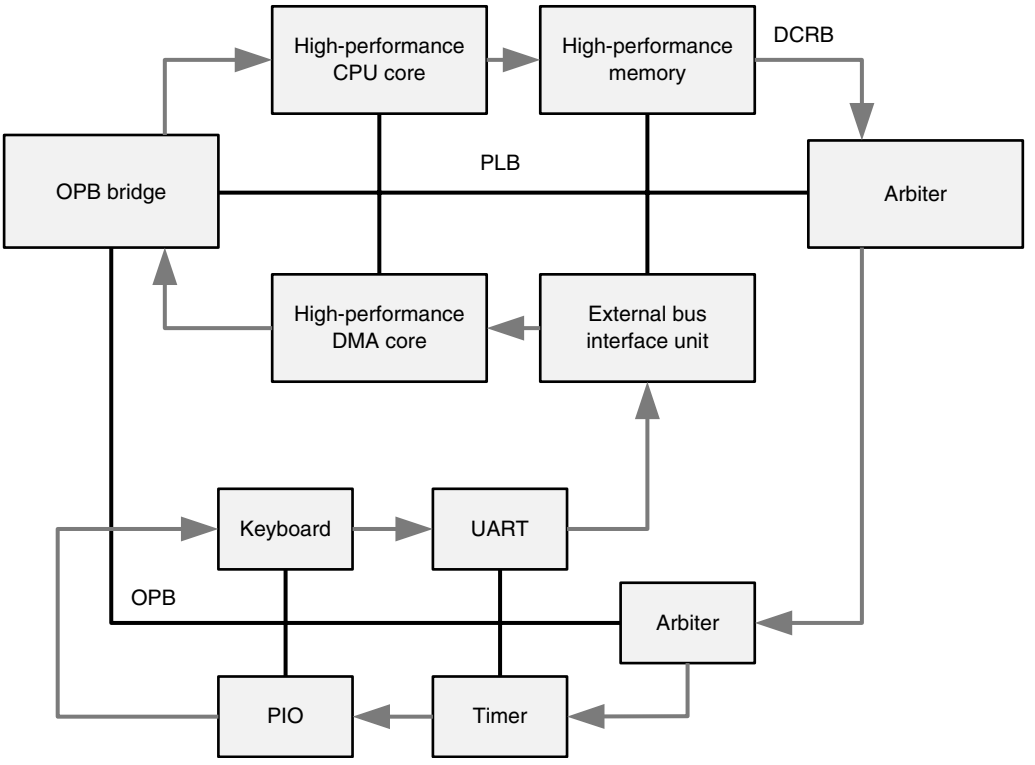



**FIGURE 7.3** CoreConnect bus-based system.

*Processor local bus* (PLB): This is the main system bus. It is synchronous, multi-master, central arbitrated bus that allows achieving high-performance and low-latency on-chip communication. Separate address and data buses support concurrent read and write transfers. PLB macro, such as glue logic, is used to interconnect various master and slave macros. Each PLB master is attached to the PLB through separate addresses, read-data and write-data buses, and other control signals. PLB slaves are attached to PLB through shared, but decoupled, address, and read-data and write-data buses. Up to 16 masters can be supported by the arbitration unit, whereas there are no restrictions in the number of slave devices [21].

*On-chip peripheral bus* (OPB): This bus is optimized to connect lower speed, low throughput peripherals, such as serial and parallel port, UART, etc. Crucial features of OPB are fully synchronous operation, dynamic bus sizing, separate address and data buses, multiple OPB bus masters, single cycle transfer of data between bus masters, single cycle transfer of data between OPB bus master and OPB slaves, etc. OPB is implemented as multi-master, arbitrated buses. Instead of tristate drivers, OPB uses distributed multiplexer. PLB masters gain access to the peripherals on the OPB bus through the OPB bridge macro. The OPB bridge acts as a slave device on the PLB and a master on the OPB.

*Device-control register bus* (DCRB): This is a single-master bus mainly used as an alternative relatively low speed data path to the system for (1) passing status and setting configuration information into the individual device-control registers between the processor core and other SoC constituents such as auxiliary processors, on-chip memory, system cores, peripheral cores, etc. and (2) design for testability purposes. DCRB is synchronous bus based on a ring topology implemented as distributed multiplexer across the chip. It consists of a 10-bit address bus and a 32-bit data bus. CoreConnect implements arbitration based on a static priority, with programmable priority fairness.

### 7.3.4 STBus

STBus is an on-chip bus protocol developed by STMicroelectronics [16]. It represents a set of protocol, interfaces, and architectural specifications intended to implement the communication network of digital systems. The STBus interfaces and protocols are closely related to the virtual component interface (VCI) industry standard. STBus implements both the protocols definition and the bus components. The following protocols are used [16,26]:

1. Type I (peripheral protocol): It is a simple synchronous handshake protocol with limited set of available command types, suitable for register access and slow peripherals. No pipelining is applied. Type I acts as a request-grant protocol. Only limited operation code and length are supported.
2. Type II (basic protocol): This protocol is more efficient than type I as it supports split transactions and adds pipelining features. The transaction set includes read/write operation with different sizes (up to 64 bytes) and also specific operations like read-modify-write and swap. Type II is equivalent to the request-grant-valid protocol. Transactions may also be grouped into chunks to ensure allocation of the slave and to ensure no interruption of the data stream. This protocol is typically suited for external memory controllers. A limitation of this protocol is that the traffic and symmetric transactions must be ordered (i.e., the number of the requesting cells equals to the number of the response ones).
3. Type III (advanced protocol): This is the most efficient protocol, as it adds support for split transactions, out-of-order executions, and asymmetric communications (i.e., the number of cells might differ between request and response). Type III is mainly used by CPUs, multichannel DMAs, and DDR controllers.

The STBus is modular and allows master and slaves of any protocol type and data size to communicate, through the use of appropriate type/size converters. A wide variety of arbitration policies is also available, such as bandwidth limitation, latency arbitration, least recently used (LRU), priority-based arbitration, etc.

The components interconnected by the STBus can be either initiators (initiates transactions on the bus by sending requests, such as CPUs or ASICs) or targets (responds to requests, such as memories, registers, or dedicated peripherals). Initiators can load or store data through the STBus backbone (see Figure 7.4). Some resources might be both initiators and peripheral/targets.

STBus-based system includes three kinds of components [26]:

Switch or node: This block arbitrates and routes the requests and responses. Different kinds of arbitrations are possible, including fixed priorities, variable priorities, dynamic priorities, latency based, bandwidth based, and LRU.

Converter or bridge domain: This converts the request from the protocol to another, for example from basic protocol to advanced protocol.

Size converter: It is used between two buses of same type of different widths; it includes buffering capacity.

STBus can instantiate different bus topologies as follows [21]:

Single shared bus: This is suitable for simple low-performance implementations; this bus characterizes minimal wiring area but limited scalability.

Full crossbar: This bus topology is intended for high-performance systems; wiring area is large.

Partial crossbar: This is used in medium-performance systems, represents a good compromise with respect to the previous two proposals.

### 7.3.5 Wishbone

Wishbone [27] bus architecture was developed by Silicore Corporation. In August 2002, OpenCores (organization that promotes open IP cores development) put it into the public domain. This means that Wishbone is not copyrighted and can be freely copied and distributed.

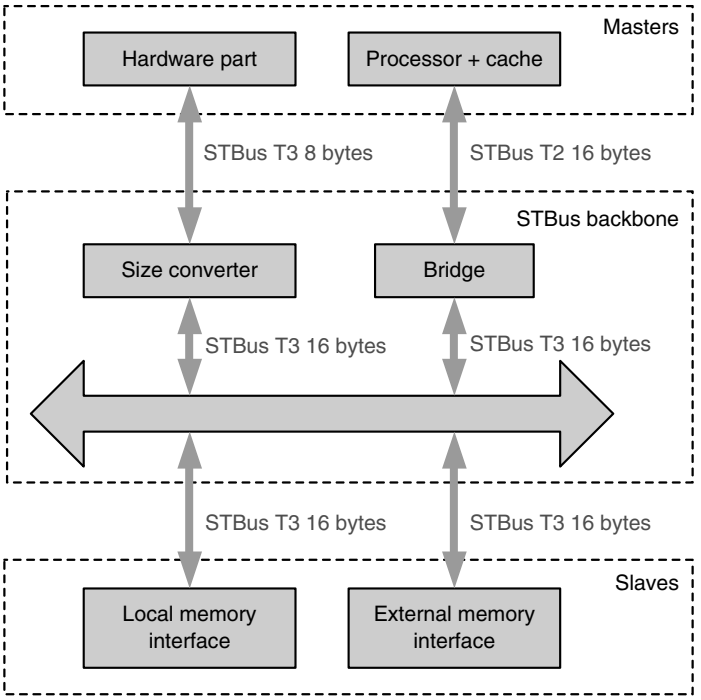



**FIGURE 7.4** STBus interconnect.

The Wishbone defines two types of interfaces, called master and slave. Master interfaces are IPs capable of initiating bus cycles, whereas slave interfaces are capable of accepting bus cycles [21]. The hardware implementations support various types of interconnection topologies (see Figure 7.5) such as

1. Point-to-point connection—used for direct connection of two participants that transfer data according to some handshake protocol.
2. Dataflow interconnection—used in linear systolic array architectures for implementation of DSP algorithms.
3. Shared bus—typical for MPSoCs organized around single system bus.
4. Crossbar switch interconnection—usually used in MPSoCs when more than one masters can simultaneously access several different slaves. The master requests a channel on the switch; once this is established, data is transferred in a point-to-point manner.

The Wishbone supports different types of bus transactions, such as read/write, implementing blocking/unblocking access. A read-modify-write transfer is also supported. Wishbone does not define hierarchical buses. In applications where two buses should exist, one slow and one fast, two separated Wishbone interfaces could be created. Designer can also choose the arbitration mechanism and implement it to fit the application needs best.

### 7.3.6 CoreFrame

The CoreFrame [28] architecture is low power high-performance on-chip interconnect architecture for integration of SoC blocks. From a high-level point of view, the CoreFrame architecture (see Figure 7.6) is viewed as a system of three buses (CPU bus, PalmBus, and MBus). The CPU bus is connected to PalmBus via PalmBus controller and to the MBus through a cache or bridge. The PalmBus and MBus are independent parallel buses, rather than a hierarchy of buses. Concurrent activities may be achieved on both buses maximizing available bandwidth resources. To avoid three-state buffering, CoreFrame does

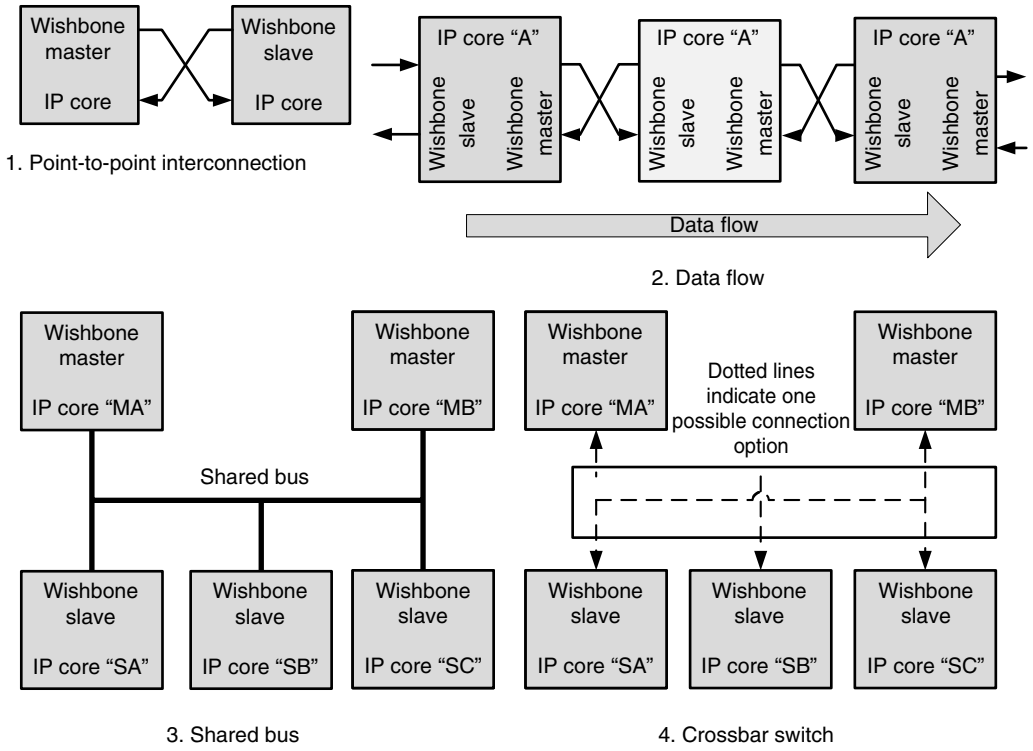



**FIGURE 7.5** Possible Wishbone interconnections.

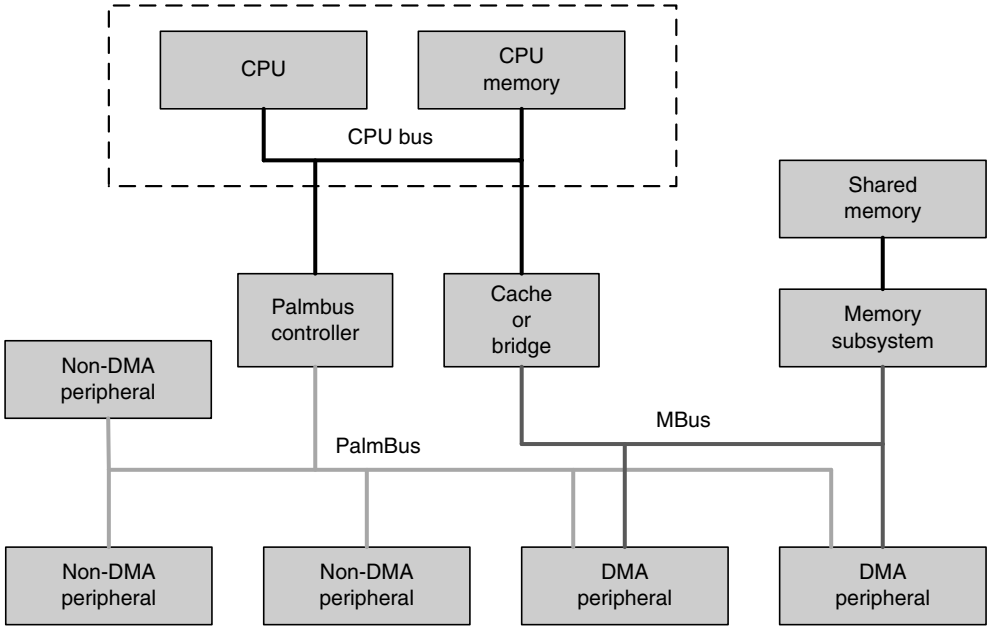



**FIGURE 7.6** CoreFrame architecture-based system structure.

not use shared signal lines. Instead it uses point-to-point signals and multiplexing. Communication between subsystems is carried out through shared memory variables.

A PalmBus represents a master–slave interface with a single master intended for communications between the CPU and peripheral blocks. It is not used to access memories. The PalmBus is designed for low-speed access from the CPU core and it provides the I/O backplane and allows the processor to configure and control peripheral blocks. Timings of the bus are synchronous with the CPU core. PalmBus is also designed with low-power consumption in mind [29].

The MBus is designed for high-speed accesses to shared memory from the CPU core and peripheral blocks. The MBus protocol is optimized for both ASIC-type implementations and data transfers to and from the data memory devices [30].

### 7.3.7 Manchester Asynchronous Bus for Low Energy

Manchester asynchronous bus for low energy (MARBLE) developed at the Manchester University is on-chip two channel micropipeline bus with centralized arbitration and address decoding that operates without global clock pulse. It is intended to provide interconnections of asynchronous macrocells within the VLSI ICs [31].

MARBLE is based on a split-transfer architecture allowing transfers between different initiators and targets to be interleaved without the needs for retries, thus giving low-energy operation and low latency. A MARBLE consists of two asynchronous multipoint channels. One of these channels carries the command from the initiator to the target returning either the accept status or defer status. The other multipoint channel carries a response from the target to the initiator (and the read data or write data in the appropriate direction). The two channels are used in a decoupled transfer scheme with loose coupling between channels to implement split transactions [32].

The interconnection provided by MARBLE is used in AMULET3H microprocessor (see Figure 7.7). It is intended to connect CPU core and DMA controller to RAM, ROM, and other peripherals [32]. In general, MARBLE demonstrates that all the features of a high-speed on-chip macrocell bus can be implemented efficiently in a fully asynchronous design style.

### 7.3.8 PI Bus

The PI (peripheral interconnect) bus was developed by several European semiconductor companies (Advanced RISC Machines, Philips Semiconductors, SGS-Thomson Microelectronics, Siemens, and TEMIC/MATRA MHS) within a framework of European project OMI (open microprocessor initiative framework). PI bus is an open standard published by OMI. For the purpose of SoC design, PI bus System Toolkit is developed. VHDL codes for master, slave, and control units are freely distributed. In addition, synthesis scripts for different ASIC and FPGA technologies, and examples of system solutions are available [9].

PI bus is a synchronous bus with unmultiplexed address and data signals that supports operation of multiple masters and bridges. It is an on-chip bus used in modular, highly integrated SoC designs. PI bus is designed for memory-mapped data transfers between its bus agents. Bus agents are on-chip modules equipped with PI bus interface and connected via PI bus signals. A PI bus agent acts as a PI bus master when it initiates data read/write operations because the bus ownership has been granted to the agent. A PI bus agent who is addressed at PI bus operation acts as a PI bus slave when it performs the requested data read/write operation. Typical masters are processor modules, coprocessors, or DMA modules, while typical slaves are on-chip memory and input–output interfaces to the external world (see Figure 7.8) [9].

The main features of PI bus are the following: (1) processor independent implementation and design; (2) demultiplexed operation; (3) clock synchronous; (4) peak transfer rate of 200 MHz (50 MHz bus clock); (5) address and data bus scalable (up to 32 bits); (6) 8-, 16-, 32-bit data access; (7) broad range of transfer types from single to multiple data transfers; and (8) multi-master capability. The PI bus does not provide (1) cache coherency support, (2) broadcast, (3) dynamic bus sizing, and (4) unaligned data access [9].

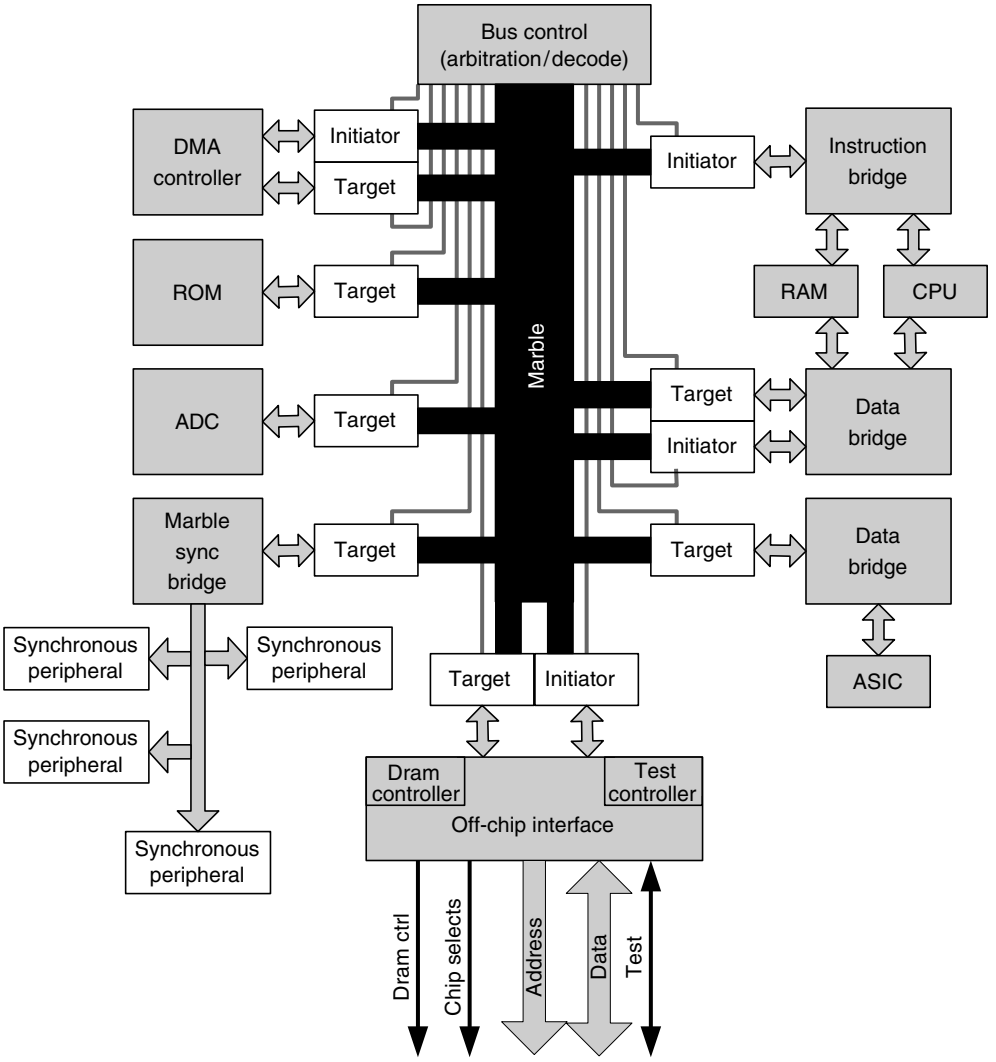



**FIGURE 7.7** AMULETH3H system.

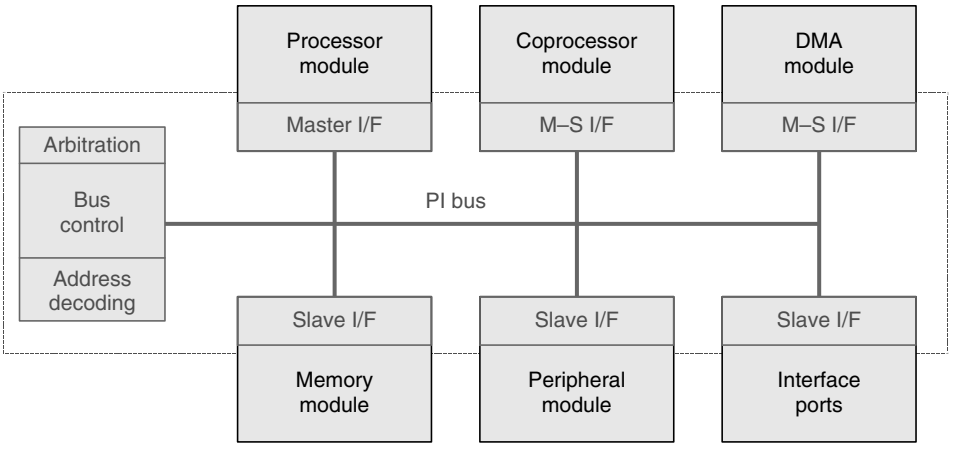



**FIGURE 7.8** Modules of a PI bus connected system.

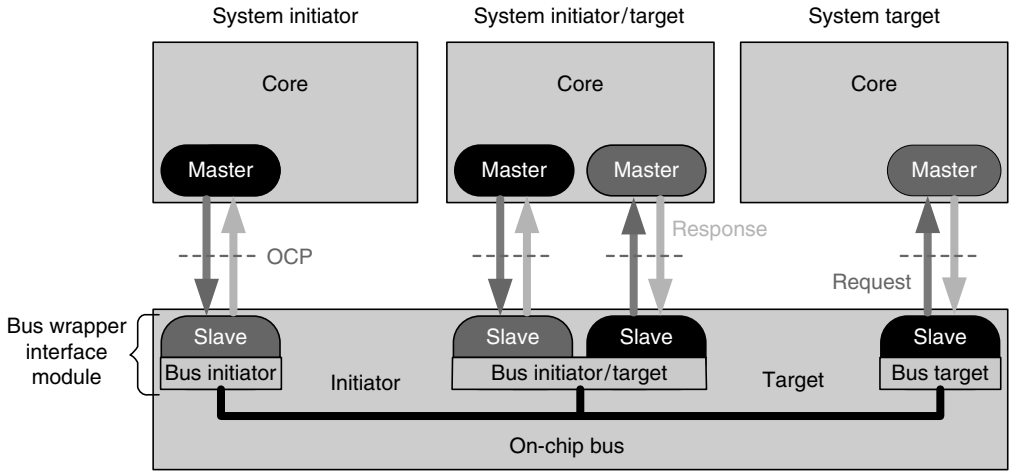



**FIGURE 7.9** Wrapped bus and OCP instances.

### 7.3.9 Open Core Protocol

Open core protocol (OCP) [14] is an interface standard that interconnects IP cores to on-chip bus. The OCP defines a comprehensive, bus-independent, high-performance, and configurable interface between IP cores and on-chip communication subsystems. A designer selects only those signals and features from the palette of OCP configurations needed to fulfill all of IP core's unique data, control, and test signaling requirements. Existing IP cores may be inexpensively adapted. Defining a core interface using the OCP provides a complete description for system integration. The following are the main features of OCP interface: (1) master–slave interface with unidirectional signals; (2) driven and sampled by the rising edge of the OCP clock; (3) fully synchronous, no multicycle timing paths; (4) all signals are strictly point-to-point (except clock and reset); (5) simple request/acknowledge protocol; (6) supports data transfer on every clock cycle; (7) allows master or slave to control transfer rate; (8) configurable data word width; (9) configurable address width; (10) pipelined or blocking reads; and (11) specific description formats for core characteristics, interfaces (signals, timing, and configuration), and performance [15].

Some of the standard on-chip buses, such as AMBA and SiliconBackplane μNetwork, use OCP. Communication requirements concerning IP core can be described using this protocol format. OCP interface is user settable, so the designer can define interface attribute, such as address and data bus width. Beside basic OCP version, there are four extensions: simple extension, complex extension, sideband extension, and debug and test interface extension. Basic OCP includes only data flow signals and is based on simple request and acknowledge protocol. However, the optional extensions support more functionality in control, verification, and testing. Simple extension and complex extension support burst transaction and pipelined write operations. In addition, sideband extension supports user-defined signals and asynchronous reset. Also, debug and test interface extension supports JTAG (Joint Test Action Group) and clock control. This is the reason why, when integrated in SoC, the OCP allows debugging and IP block test generating. Figure 7.9 presents SoC design based on the OCP.

### 7.3.10 Virtual Component Interface

The VCI [17] is an interface rather than a bus. Thus, the VCI specifies (1) a request–response protocol, (2) a protocol for the transfer of requests and responses, and (3) the contents and coding of these requests and responses. The VCI does not touch areas such as bus allocation schemes, competing for a bus, and so forth.

There are three complexity levels for the VCI: peripheral VCI (PVCI), basic VCI (BVCI), and advanced VCI (AVCI). The PVCI provides a simple, easily implementable interface for applications that do not

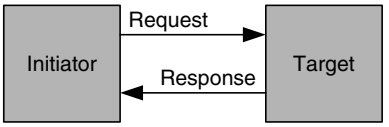



**FIGURE 7.10** VCI as a point-to-point connection.

need all the features of the BVCI. The BVCI defines an interface that is suitable for most applications. It has a powerful, but not overly complex protocol. The AVCI adds more sophisticated features, such as threads, to support high-performance applications. The PVCI is a subset of the BVCI, and the AVCI is a superset of the BVCI.

BVCI and AVCI make use of a "split protocol." That is, the timing of the request and the response are fully separate. The initiator can issue as many requests as needed, without waiting for the response. The protocol does not prescribe any connection between issuing of requests and arrival of the corresponding responses. The only thing specified is that the order of responses corresponds to the order of requests. In the AVCI, requests may be tagged with identifiers, which allow such requests and request threads to be interleaved and they response to arrive in a different order. Responses bear the same tags issued with the corresponding requests, such that the relation can be restored upon the reception of a response. As an interface, the VCI can be used as a point-to-point connection between two units called the initiator and the target, where the initiator issues a request and the target responds (see Figure 7.10).

The VCI can be used as the interface to a wrapper, which means a connection to a bus. This is how the VCI allows the VC to be connected to any bus. An initiator is connected to that bus by using a bus initiator wrapper. A target is connected to that bus by using a bus target wrapper. Once the wrappers for the bus have been designed, any IPs can be connected to that bus, as depicted in Figure 7.11.

### 7.3.11 SiliconBackplane μNetwork

Sonics μNetwork [7] consists of a set of architectures and SoC design tools. Defined architectures are SiliconBackplane for on-chip interconnection and multichip for off-chip interconnection. SiliconBackplane implements two-level arbitration, based on TDMA and round-robin.

SiliconBackplane μNetwork is a network on a chip that connects IP blocks in a SoC. μNetwork isolates the system of IP blocks from network by requiring all blocks to use single bus interface protocol, OCP. Every IP block communicates via "wrapper," which μNetwork calls an agent, using OCP. Agents communicate with each other through μNetwork. As system's requirements change, OCP and μNetwork support modification of many system's parameter in real time. System requirements relate to, for example, selection of arbitration scheme, definition of address space, etc. An agent is generated using the tool Fast Forward Development Environment, developed by Sonics. Basic building blocks of SiliconBackplane μNetwork are given in Figure 7.12.

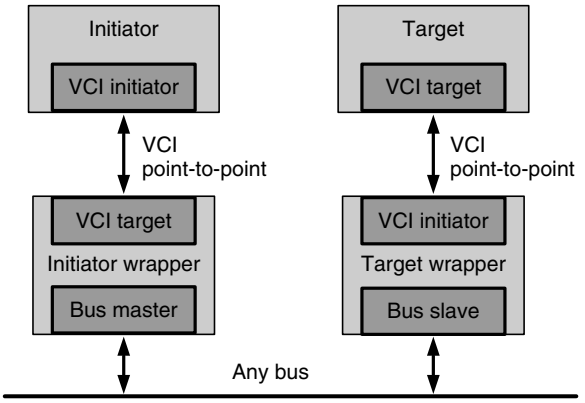



**FIGURE 7.11** Two VCI connections used to realize a bus connection.

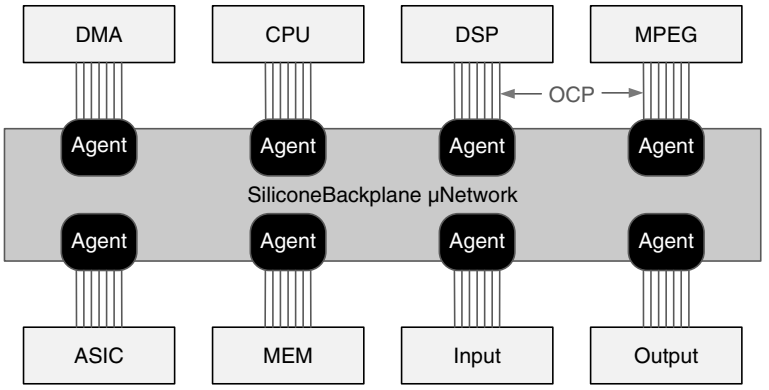



**FIGURE 7.12** SiliconBackplane μNetwork constituents.

## 7.4 Summary

Complex VLSI IC design has been revolutionized by the widespread adoption of the SoC paradigm. The benefits of the SoC approaches are numerous, including improvements in system performance, cost, size, power dissipation, and design turnaround time. Many SoC designs consist of one or more IPs, designed for a single or narrow set of applications with a highly characterizable communication. As the level of a chip integration continues to advances at a fast pace, the desire for efficient interconnects rapidly increases. Currently on-chip interconnection networks are mostly implemented using traditional interconnects like buses. The wide variety of buses used in SoC designs presents the major problem for reusable design. A number of companies and standard committees have attempted to standardize buses and interface with mixed results.

In this chapter we have given an overview of the most popular on-chip bus-based interconnection networks such as AMBA, Avalon, CoreConnect, STBus, Wishbone, etc. The main characteristics of the considered buses with respect to topology, arbitration method, bus width, and types of data transfers are discussed. In addition, we have pointed to some of the issues that SoC designers are facing in determining the bus architecture to use to provide flexible and high bandwidth between IP cores.

## References

1. Keating M. and Bricaud P., *Reuse Methodology Manual for System-on-a-Chip Designs*, 2/E/ Kluwer Academic Publishers, Boston, MA, 1999.
2. Ho W.H. and Pinkston T.M., A design methodology for efficient application-specific on-chip interconnects, *IEEE Trans. Parallel Distrib. Syst.*, 17(2): 174–190, February 2006.
3. Horspool N. and Gorman P., *The ASIC Handbook*, Prentice Hall, PTR, Upper Saddle River, NJ, 2001.
4. Benini L. and De Micheli G., Networks on chips: A new paradigm for component-based MPSoC design, chapter 3, pp. 49–80, in Jerraya A.A. and Wolf W. (Eds.), *Multiprocessor Systems-on-Chips*, Elsevier, Amsterdam, 2005.
5. CoreConnect Bus Architecture, IBM Microelectronics, available at http://www.ibm.com/chips/-products/coreconnect, January 2006.
6. ARM.AMBA Specifications v2.0, 1999, available at http://www.arm.com, January 2006.
7. Sonics μNetwork Technical overview, January 2002, available at http://www.sonicsinc.com, January 2006.
8. Lahiri K., Dey S., and Raghunathan A., Design of communication architectures for high-performance and energy-efficient systems-on-chip, chapter 7, pp. 187–222, in Jerraya A.A. and Wolf W. (Eds.), *Multiprocessor Systems-on-Chips*, Elsevier, Amsterdam, 2005.

9. Draft Standard OMI 324: PI-Bus, Rev. 0.3d, Open Microprocessor Systems Initiative, Copyright 1994 by Siemens AC, Munich, available at http://www.cordis.lu/esprit/src/omi-home.htm, August 2005.
10. Dally W.J. and Towel B., *Principles and Practices of Interconnection Networks*, Elsevier, Amsterdam, 2004.
11. Shandhag N.R., Reliable and efficient system-on-chip design, *IEEE Comput.*, 37(3): 42–50, March 2004.
12. Hennessy J. and Petterson D., *Computer Architecture: A Quantitative Approach*, Elsevier, Amsterdam, 2003.
13. Radulescu A. and Goossens K., Communication services for networks on chip, in Bhattacharyya S., Deprettere E., Teich J. (Eds.), *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, Marcel Dekker Inc., New York, 2004, pp. 193–213.
14. Overview of Open Core Protocol (OCP-2001-9-26), OCP International Partnership Association, Portland, OR 97221, USA, available at www.ocpip.org, January 2006.
15. Rowen C., *Engineering the Complex SoC: Facts, Flexible Design with Configurable Processors*, Prentice Hall, PTR, Upper Saddle River, NJ, 2004.
16. Strano G., Tiralongo S., and Pistrito C., *OCP/STBus Plug-In Methodology*, available at http://www.techoline.com/community/tech_group/com/tech_paper/37923, January 2006.
17. Virtual Component Interface Standard Version 2 (OCB 2 2.0), VCI Alliance, April 2001, available at www.vsi.org, March 2006.
18. Kogel T., Leupers R., and Meyr H., *Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processors Platforms*, Springer, Dordrecht, 2006.
19. Bertozzi D. and Benini L., Xpipes: A network-on-chip architecture for gigascale systems-on-chip, *IEEE Circ. Syst.*, 4(2): 18–31, Second Quarter 2004.
20. Jantasch A. and Tenhunen H., *Networks on Chip*, Kluwer Academic Publishers, Boston, MA, February 2003.
21. Ayala J., Lopez-Vellejo M., Bertozzi D., and Benini L., State-of-the-art SoC communication architectures, in Zurawski R. (Ed.), *Embedded Systems Handbook*, CRC Press, Boca Raton, FL, 2006, pp. 20.1–20.22.
22. ARM.AMBA Multi-Layer AHB Overview, 2001, available at http://www.arm.com, January 2006.
23. ARM.AMBA AXI Protocol Specifications, 2003, available at http://www.arm.com, January 2006.
24. Altera, Avalon Interface Specification, April 2005, available at www.altera.com, March 2006.
25. Altera, Avalon Bus Specification: Reference Manual, July 2003, available at www.altera.com, March 2006.
26. Pelissier G., Hersemeule R., Cambon G., Torres L., and Robert M., *Bus Analysis and Performance Evaluation on a SoC Platform at the System Level Design*, available at http:/www.ra.informatik.uni.stutgart.de/~pricopin/noc03/paper_44.pdf, January 2006.
27. WISHBONE System-on-Chip (SoC) *Interconnection Architecture for Portable IP Cores*, Revision: B.3, September, 2002, available at http://www.opencores.org/projects.cgi/web/wishbone/wishbone, March 2006.
28. Overview of the CoreFrame architecture, January 2002, available at http://www.palmchip.com, January 2006.
29. Palmchip, Overview of CoreFrame Architecture, White Paper, available at www.palmchip.com, February 2006.
30. Cordon B., *A Bus Architecture for System-on-Chip Designs, Palmchip Corporation*, available at www.palmchip.com, January 2006.
31. Bainbridge W., *Asynchronous system-on-chip interconnect*, Ph.D. Thesis, Department of Computer Science, University of Manchester, England, March 2000, pp. 119–140, available at www.cs.man chester.ac.uk/apt/projects/interconnect/, February 2006.
32. Bainbridge W. and Furber S., *Asynchronous macrocell interconnect using marble*, Technical Report, available at www.cs.manchester.ac.uk/apt/projects/interconnect/, February 2006.

# III

# Signal Processing

# 8

# Digital Signal Processing




Fred J. Taylor
*University of Florida*


## 8.1 Introduction

Signals are traditionally classified as being analog (continuous-time), discrete-time (sample-data), or digital. A continuous-time signal has infinite precision in both the time- and amplitude-domain. Discrete-time signals have infinite amplitude precision, but are discretely resolved in time (sampled). Digital signals are of finite precision in both the time (sampled) and amplitude (quantized). Digital signals are either synthesized by a digital system (e.g., computer) or by digitizing an analog signal using an analog-to-digital converter (ADC). A digital-to-analog converter (DAC) converts a digital signal into an analog signal. Signal processing refers to the science of analyzing, synthesizing, and manipulating audio, acoustic, speech, video, image, geophysical, radar, radio signals, plus a host of other waveforms using mathematics or technology. Signals may be an array of one-, two-, or *M*-dimensional samples, of finite or infinite duration. Digital signal processing (DSP) refers to the processing of digital or digitized signals exclusively with digital technologies and techniques. DSP systems and elements can be linear or nonlinear, and reside in the time (e.g., filter) or transform domain (e.g., frequency). DSP processing agents range from specialized mathematical and statistical abstractions, to software or hardware. In practice, DSP systems are often designed to meet very restrictive real-time speed, precision, dynamic range requirements, and operate in multisignal, multisystem environments. The design and study of a DSP solution, therefore, requires a concurrent knowledge of signal processing theory, application, and technology.

DSP is currently a major market force, consisting of semiconductor, hardware, software, applications, support, and training sectors. The origins of DSP are open to debate, but a seminal moment surely occurred when Claude Shannon developed an understanding of sample-data signal processing in the

middle of the twentieth century. Shannon's *sampling theorem* states that if an analog signal, having its highest frequency bounded below $B$ Hz, is sampled at a rate equal to or in excess of $f_s \geq 2B$ Hz, then the original signal can be perfectly reconstructed from its sample values. The critical parameter $f_N = f_s/2$ is called the Nyquist frequency and represents a strict upper bound on the frequency content of the sampled signal. Most DSP solutions are over-sampled, operating at a sample frequency far in excess of the minimally required value. If a signal is under-sampled at a rate below the minimum rate of $2B$ Hz, aliasing errors will occur. An aliased signal is a baseband signal whose sample values impersonate those of the sampled higher frequency signal. Another early enabler of the DSP revolution was the Cooley–Tukey FFT, the developers of fast Fourier transform (FFT) algorithm. The FFT made many signal-processing tasks practical for the first time using, in many instances, only software. Another defining DSP moment occurred when the first DSP microprocessors (DSP μp) made a marketplace appearance in the late 1970s. These devices immediately provided an affordable and tangible means of developing embedded solutions with a minimum risk and effort. Regardless of its origins, today DSP has become a pervasive technology, appearing in a myriad of applications, and supported with a rich and deep technological infrastructure. DSP is now a discipline unto itself, with its own professional societies, academic programs, trained practitioners, and industrial infrastructure.

## 8.2 Digital Signals and Systems

Digital systems can process digital signals in either the time- or frequency-domain. Systems are often characterized in the time-domain by their response to an impulse (i.e., $\delta[k]$), which is logically called the impulse response and is denoted by $h[k] = \{h[0], h[1], h[2], \ldots\}$. The sequence of sample values $h[k]$ is called a time series, which can be mathematically represented using the venerable $z$-transform. The $z$-transform of an arbitrary time series $x[k] = \{x[0], x[1], x[2], \ldots\}$ is formally given by $X(z) = \sum_{k=0}^{\infty} x[k]z^{-k}$ [7,8,9]. The $z$ operator is defined in terms of the delay theorem of Laplace transforms, namely $z = e^{sT_s}$, where $T_s$ is the sample period. The $z$-transforms of common signals are reported in standard table of $z$-transforms, exemplified by Table 8.1. The common signals shown in Table 8.1 can be manipulated and combined, using the property list shown in Table 8.2, to synthesize higher-order and more complex signals. In addition to the properties listed in Table 8.2, the initial value theorem $x[0] = \lim_{z \to \infty} X(z)$ and, with reservations, the final value theorem $x[\infty] = \lim_{z \to 1} (z-1)X(z)$ provide a convenient means of evaluating two end points of a time series. The mapping of a $z$-transformed signal $X(z)$ back into the time-domain is generally performed in a piecemeal manner. Specifically, the inverse $z$-transform of $X(z)$ is normally expressed as a partial fraction or Heaviside expansion, having the form $X(z) = \sum_{i=1}^{M} A_i X_i[z]$, where $X_i(z)$ is an element of Table 8.1 and corresponding to a discrete-time signal $x_i[k]$. The coefficient $A_i$ is called a Heaviside coefficient associated with the term $X_i(z)$. The inverse $z$-transform of $X(z)$ in partial fraction form is given by $x[k] = \sum_{i=1}^{M} A_i x_i[k]$.

The output of a linear system to an input $x[k]$ is the system's impulse response $h[k]$. The output to the system to an arbitrary input $x[k]$ is defined by the discrete-time linear convolution sum:

**TABLE 8.1** $z$-Transforms of Primitive Time Functions

| Discrete-Time Signal $x[k]$ | $z$-Transform $X(z)$ |
|---|---|
| $\delta[k]$ (impulse) | 1 |
| $u[k]$ (unit step) | $z/(z-1)$ |
| $a^k u[k]$ (exponential) | $z/(z-a)$ |
| $\sin[bkT_s]u[kT_s]$ (sine wave) | $\sin(bT_s)\ z/(z^2 - 2z\cos(bT_s) + 1)$ |
| $\cos[bkT_s]u[kT_s]$ (cosine wave) | $(z - \cos(bT_s))\ z/(z^2 - 2z\cos(bT_s) + 1)$ |
| $a^k \sin(bkT_s)u[kT_s]$ (damped sine) | $a\sin(bT_s)\ z/(z^2 - 2az\cos(bT_s) + a^2)$ |
| $a^k \cos(bkT_s)u[kT_s]$ (damped cosine) | $(z - a\cos(bT_s))\ z/(z^2 - 2az\cos(bT_s) + a^2)$ |

**TABLE 8.2** Properties of $z$-Transforms

| Property | Time Series | $z$-Transform |
|---|---|---|
| Linearity | $x_1[k] + x_2[k]$ | $X_1(z) + X_2(z)$ |
| Real scaling | $a\,x[k]$ | $a\,X(z)$ |
| Complex scaling | $w^k\,x[k]$ | $X(z/w)$ |
| Time reversal | $x[-k]$ | $X(1/z)$ |
| Modulation | $e^{-ak}\,x[k]$ | $X(e^a\,z)$ |
| Shift delay | $x[k-1]$ | $z^{-1}X(z) - zx[0]$ |

$$y[k] = h[k]x[k] = \sum_{m=0}^{\infty} h[m]x[k-m]$$

Computing the convolution sum is rare. Instead, a linear system is generally analyzed using simulation, emulation, or the $z$-transform. The convolution theorem states that the linear convolution $y[k] = h[k] * x[k]$ of a $z$-transformable impulse response $h[k]$ (i.e., $H(z) = \sum_{k=0}^{\infty} h[k]z^{-k}$) and input $x[k]$ (i.e., $X(z) = \sum_{k=0}^{\infty} x[k]z^{-k}$), is given by the inverse $z$-transform of the product $Y(z) = H(z)X(z)$. This method is only viable in instances where the $z$-transform of $h[k]$ and $x[k]$ are readily computed or tabled, and the inverse $z$-transform of $Y[z]$ can be efficiently computed. While $H(z)$ is generally known, most real signals are arbitrary and possibly noise contaminated, making the mathematical availability of $X(z)$ questionable. Nevertheless, the importance of this equation has resulted in the elements being given specific titles and meaning. The $z$-transform of the impulse response $h[k]$, namely $H(z)$, is called the system's transfer function and has the general form

$$H(z) = Y(z)/X(z) = \sum_{m=0}^{M} b_m z^{-m} \Bigg/ \sum_{m=0}^{M} a_m z^{-m} = N(z)/D(z)$$

The filter's poles ($p_m$) and zeros ($z_m$) are the roots of $D(z) = 0$ and $N(z) = 0$, respectively. The system's steady-state frequency response can be determined by evaluating the transfer function $H(z)$ along the trajectory $z = e^{j\varpi}$, where $\varpi \in [-\pi, \pi]$ and represents a normalized baseband frequency range $[-f_s/2, f_s/2]$ ($\pm$Nyquist frequency). Specifically, the steady-state frequency response of a linear system, in magnitude-phase form, is $H(e^{j\varpi}) = |H(e^{j\varpi})|\phi(e^{j\varpi})$.

## 8.3 Digital Filters

Transfer functions, when implemented in the time-domain, result in digital filters. The attributes of a digital filter can be specified in the time-domain, frequency-domain, or both. Digital filters can be grouped into three broad classes called finite impulse response (FIR), infinite impulse response (IIR), and multirate filters.

### 8.3.1 Finite Impulse Response Filters

An FIR filter possesses an impulse response that persists only for a finite number of sample values [5,6]. The impulse response of an $N$th-order FIR is given by $h[k] = \{h_0, \ldots, h_{N-1}\}$, and in the $z$-transform domain by $H(z) = \sum_{i=0}^{N-1} h_i z^{-i}$. One of the attributes of an FIR is its simplicity, consisting of a string of multiply-accumulations (MACs), and shift registers. The steady-state frequency response of an FIR is given by $H(e^{j\varpi}) = |H(e^{j\varpi})|\phi(e^{j\varpi})$. A system is said to possess a linear phase response if $\phi(e^{j\varpi}) = \alpha\varpi + \beta$ (i.e., linear in frequency). Linear phase filters are important in a number of applications, including: (1) synchronizing phase-modulated data streams, (2) anti-aliasing filters placed in front of signal phase-sensitive analysis subsystems (e.g., FFT), and (3) using phase-sensitive applications (e.g., image processing). Linear phase filtering is guaranteed whenever the coefficients of an $N$th-order FIR are symmetrically distributed about the filter's midpoint $L = (N-1)/2$ (i.e., $h_i = \pm h_{N-i}$, $i = 0, \ldots, L$). The resulting phase response satisfies the linear phase equation, which is given by $\phi(e^{j\varpi}) = -L\varpi + \{0, \pm\pi/2\}$. Another important phase response measure is called the group delay and is given by $\tau_g = -d\phi(e^{j\varpi})/d\varpi$. For a linear phase FIR, $\tau_g = L$, which indicates that the filter propagation delay is always $L$ clock cycles regardless of the input signal frequency.

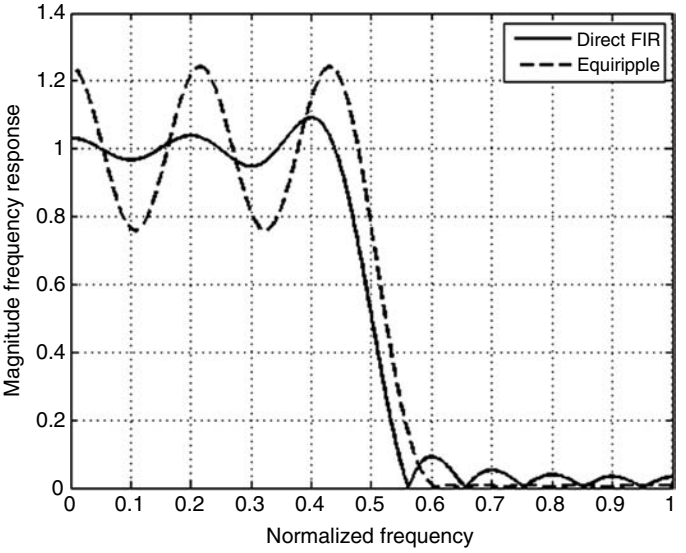



**FIGURE 8.1** Comparison of direct and equiripple FIR designs.

FIR design methods are well known. The simplest design technique is called the direct or window method. The design process begins with a specification of the desired filter frequency response $H(e^{j\varpi})$. An $M$-harmonic ($M \gg 1$) inverse Fourier transform (IFFT) of $H(e^{j\varpi})$ is computed, which results in an $M$-sample time series $h'[k]$ that produces a close approximation to the desired FIR frequency response. The long $M$-sample time series is then symmetrically reduced to a desired $N$-sample impulse response $h[k]$, defined by the $N$ central values of $h'[k]$, $N < M$. The major weakness of the direct design paradigm is that the approximation errors in the frequency-domain can be locally large at selected points as shown in Figure 8.1. Another commonly used design criteria is based on a minimax error criterion. The minimax criterion requires that the maximum value of the approximation error be minimized. A minimax FIR is characterized by the frequency-domain errors having an equiripple (equal ripple) envelope. Thus, this class of FIR is logically referred to as an equiripple filter and has a typical magnitude frequency response shown in Figure 8.1.

Windows are tools that are sometimes used to improve the shape of an FIR's frequency-domain envelope. An $N$-sample-data window is applied to an $N$th-order FIR on a sample-by-sample basis according to the rule $h_w[k] = h[k]w[k]$, where $h[k]$ is an FIR's impulse response, $w[k]$ is a window function, and $h_w[k]$ is the windowed FIR impulse response. In the frequency-domain, the effect of a window is defined by the convolution operation $H_w(n) = H(n)*W(n)$, which results in a tendency to smooth the envelope of the parent FIR's frequency response. The attributes of a window are defined by the width of the center (main) lobe and sideband suppression in the frequency-domain (see Table 8.3). Common window functions are rectangular, Hann, Hamming, Blackman, Kaiser, etc. The effect of a window on the direct FIR frequency response shown in Figure 8.1 is also displayed in Figure 8.2.

**TABLE 8.3** Effects of Data Windows

| Window | Transition Width $f_s/N$ | Highest Sidelobe in dB |
|---|---|---|
| Rectangular | 0.9 | −13 |
| Hann | 2.07 | −31 |
| Hamming | 2.46 | −41 |
| Blackman | 3.13 | −58 |
| Kaiser ($\beta = 2.0$) | 1.21 | −19 |

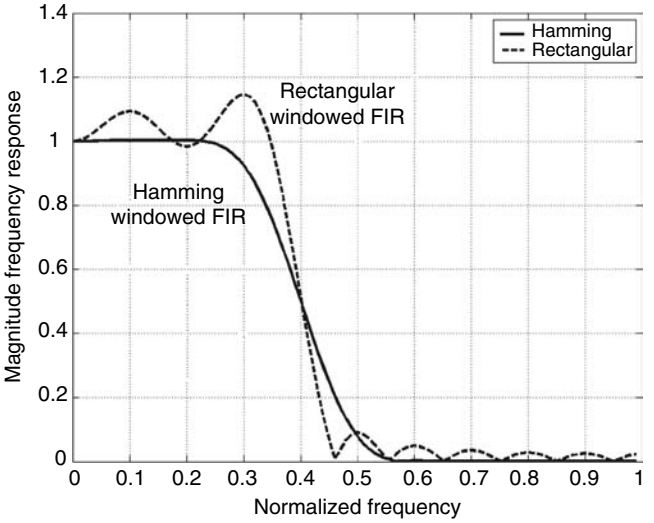



**FIGURE 8.2** Effects of windowing an FIR. Note that the windowed spectrum is smoothed and has an increased transition bandwidth.

### 8.3.2 Infinite Impulse Response Filters

Filters containing feedback are called infinite impulse response (IIR) filters [5,6]. With feedback, an IIR's impulse response can be infinitely long. The presence of feedback allows an IIR to achieve very high frequency selectivity and near resonance behavior. An $N$th-order constant coefficient IIR filter can be modeled by the transfer function:

$$H(z) = N(z)/D(z) = \sum_{i=0}^{M} b_i z^{-i} \Big/ \sum_{i=0}^{N} a_i z^{-i} = K(z^{N-M}) \prod_{i=0}^{M-1} (z - z_i) \Big/ \prod_{i=0}^{N-1} (z - p_i)$$

where the filter's zeros are $z_i$ and the filter's poles are $p_i$. The frequency response of an $N$th-order IIR is given by

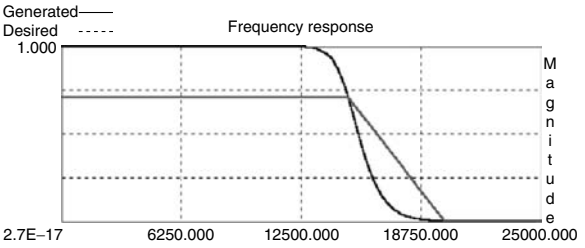
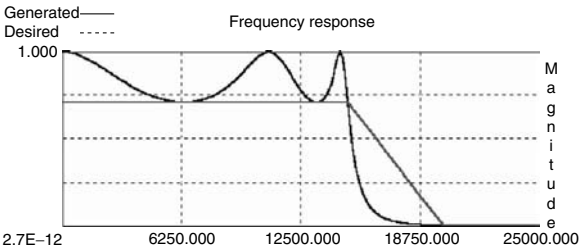
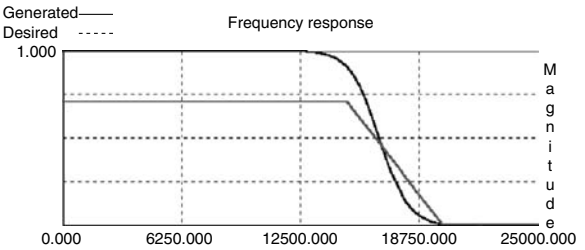
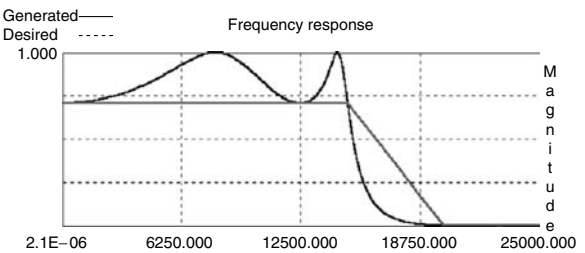
$$H(e^{j\varpi}) = \sum_{i=0}^{M} b_i e^{-ji\varpi} \Big/ \sum_{i=0}^{N} a_i \mathrm{e}^{-ji\varpi} = K\left(\mathrm{e}^{-j(N-M)} z^{N-M}\right) \prod_{i=0}^{M-1} \left(\mathrm{e}^{j\varpi} - z_i\right) \Big/ \prod_{i=0}^{N-1} \left(\mathrm{e}^{j\varpi} - p_i\right)$$

evaluated over the normalized frequency range $\varpi \in [-\pi, \pi)$, which defines the baseband frequency range bounded by $\pm f_s/2$ (Nyquist frequency).

The traditional IIR design strategy is based on converting classic analog filter models into their digital filter equivalents. Throughout the first half of the twentieth century, analog radio filter engineers created classic Bessel, Butterworth, Chebyshev, and Elliptic (Cauer) filter instantiations whose magnitude frequency response emulates that of an ideal filter. To standardize the analog filter design procedures, a set of normalized $-1$ dB or $-3$ dB low-pass filter models, having a 1.0 rad/s passband were created. These models were reduced to tables, charts, and graphs and are called analog prototype filters. The prototype low-pass filters can be the frequency scaled to define an analog low-pass, high-pass, band-pass, and band-stop filters $H(s)$, having desired frequency-domain attributes. The classic analog filter $H(s)$ is then converted into a digital filter model $H(z)$ to define a classic digital filter (see Table 8.4). The basic domain conversion techniques (i.e., $H(s) \rightarrow H(z)$) are (1) the impulse-invariant and (2) bilinear $z$-transform methods.

The impulse invariance filter design method results in a digital filter having an impulse response $h[k]$ that agrees with that of the parent analog filter's impulse response $h_a(t)$, up to a scale factor at every

**TABLE 8.4** Comparisons of *N*th-Order Classic IIR Lowpass Filters Having $f_s = 50$ kHz, a −3 dB 15 kHz Passband, 5 kHz Transition Band, and −50 dB Stopband

| Type | Order | Passband | Stopband | Magnitude Frequency Response |
|---|---|---|---|---|
| Butterworth | High ($N = 8$) | Smooth | Smooth |  |
| Chebyshev I | Medium ($N = 5$) | Ripple | Smooth |  |
| Chebyshev II | Medium ($N = 5$) | Smooth | Ripple |  |
| Elliptic | Low ($N = 4$) | Ripple | Ripple |  |



sample instant. An impulse-invariant design can be of significant value in applications, such as automatic control, where design objectives are defined in the time-domain (e.g., risetime, overshoot, settling time). If the parent analog filter's impulse response $h_a(t)$, or transfer function $H_a(s)$ is known, then the impulse-invariant digital filter is defined by

$$h_a(t) \Leftrightarrow H_a(s) = \sum_{i=1}^{N} a_i/(s + p_i) \overset{Z}{\leftrightarrow} (1/T_s) \sum_{i=1}^{N} a_i/(1 + e^{-p_i T_s} z^{-1}) = (1/T_s)H(z) \Leftrightarrow (1/T_s)h[k]$$

and in the frequency-domain by

$$H(e^{j\varpi}) = (1/T_s) \sum_{k=-\infty}^{\infty} H_a(j((\varpi/T_s) - (2\pi k/T_s)))$$

This equation exhibits a weakness of the impulse-invariant design method. For any physically meaningful sampling rate $f_s = 1/T_s$, aliasing errors can occur whenever the analog filter passes signal components at frequencies greater than the Nyquist frequency. Typically, analog filters have a gain that is finite for all frequencies. The high-frequency filter energy can be aliased back into the baseband and can distort (sometime significantly) the frequency response of an impulse-invariant filter. As a result, the impulse-invariant method is generally only used to design frequency-selective filters that are decidedly low pass.

When meeting frequency-domain specifications is the design objective, the bilinear $z$-transform method is normally used. The bilinear $z$-transform maps a classic analog filter $H_a(s)$ into a digital filter $H(z)$ without introducing aliasing errors. The bilinear $z$-transform establishes a relationship between the $s$- and $z$-domain, given by $s = (2/T_s)(z+1)/(z-1)$. The bilinear $z$-transform also defines an algebraic connection between the analog and digital frequency axis as $\Omega = (2/T_s)\tan(\varpi/2)$, where $\Omega$ is the analog frequency, $|\Omega| < \infty$, and $\varpi$ is the normalized digital frequency range $-\pi \leq \varpi < \pi$. The mapping from analog frequencies $\Omega$ to digital frequencies $\varpi$ is called warping, and prewarping in the opposite direction. The bilinear $z$-transform design paradigm is a multistep process consisting of the following steps:

1. Define the digital filter's frequency-domain attributes (gains at critical frequencies).
2. Prewarp the critical digital frequencies $\varpi$ into analog frequencies $\Omega$.
3. Design a prewarped classic analog filter $H_a(s)$ that meets specified pass-band and stop-band gain requirements.
4. Apply the bilinear z-transform to convert $H_a(s)$ into a digital filter $H(z)$. In the process, the prewarped analog filter frequencies $\Omega$ will be warped back to their original locations $\varpi$.

### 8.3.3 Multirate Systems

DSP systems that contain multiple sample rates are called multirate systems [7]. A signal $x[k]$, sampled at a rate $f_{in}$, is said to be decimated by $M$ if it is exported at a rate $f_{out} = f_{in}/M$, where $M > 1$. Mathematically, the decimated signal $x_d[k]$ can be expressed as $x_d[k] = x[Mk]$, indicating that only every $M$th sample of the fast sampled time series $x[k]$ is retained in the decimated signal $x_d[k]$. Decimation can also be modeled in the $z$-transform domain as $X_d(z) = X(z^M)$ and $X_d(e^{j\varpi}) = X_d(e^{jM\varpi})$ in the frequency-domain as suggested in Figure 8.3. To ensure that a signal $x[k]$ can be reconstructed from its decimated samples of $x_d[k]$, Shannon's sampling theorem must be obeyed. Specifically, if the minimum sampling frequency is bounded by $f_s = 2B$ Hz, the maximum decimation rate must be bounded by $M \leq f_s/2B$.

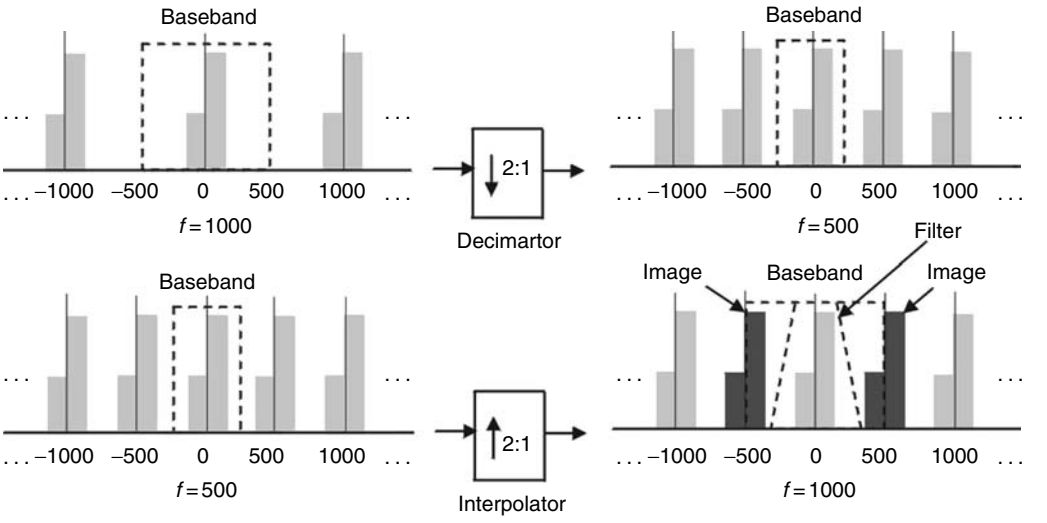



**FIGURE 8.3** Multirate system elements showing decimation (top) and interpolation (bottom).

Decimation is routinely found in audio signal and video data transmission and signal compression applications, and interfacing equipment with dissimilar fixed sample rates. By reducing the system's sample rate by a factor $M$, arithmetic bandwidth requirements can often be reduced by a similar amount.

Interpolation is the antithesis of decimation. While decimation is used to reduce the sampling rate, interpolation is used to increase the sample rate. A signal $x[k]$, sampled at a rate $f_{\text{in}}$, is said to be interpolated by $N$ if $x_i[k] = x[k]$ whenever $k \equiv 0$ modulo $(N)$, and zero elsewhere. The interpolated signal $x_i[k]$ is a time series consisting of $N-1$ zeros separated by actual sample values and clocked at a rate $f_{\text{out}} = Nf_{\text{in}}$, $N > 1$. In the $z$-transform domain, $X_i(z) = X(z^N)$, and $X_i(e^{j\varpi}) = X(e^{jN\varpi})$ in the frequency-domain, as shown in Figure 8.3. It can be noted that the interpolated spectrum contains multiple copies of the baseband spectrum $X(e^{j\varpi})$, where the unwanted copies can be removed using a low-pass filter.

## 8.3.4 Special Filter Cases

Besides baseline FIR, IIR, and multirate filters (which are based on FIR or IIR elements), other classes of digital filters are found in common use [12]. One of the most important of these is the adaptive filter. An adaptive filter modifies the filter coefficients during run-time in order to respond to measurable changes in the signal and system environment. The adaptation rules and procedures, generally based on a squared error criteria, range from simple to sophisticated, establishing trade-offs between implementation simplicity and accuracy. Adaptive filters that contain nonlinear elements are called neural networks. Some filter classes are defined in terms of special features of their defining mathematical framework. Wavelets, for example, are defined by basis functions that satisfy a formal set of scaling and dilatation rules. They often appear as a multirate solution consisting of collections of subfilters defined by wavelet basis functions that have been selected to match signal-specific signal attributes or features.

## 8.3.5 Digital Filter Architecture

The physical implementation of a particular FIR or IIR filter is called filter architecture [11]. Architectures specify how a digital filter is assembled using a collection of DSP primitive objects, such as shift registers, multipliers, and adders. The choice of architecture has a direct influence on the performance, cost, power consumption, and precision of the design outcome. Common FIR architectures are the direct, transpose, and lattice implementations. Common IIR architectures include (1) direct I and II, (2) normal (optimized second-order section), (3) cascade ($H(z) = \prod H_i(z)$, $H_i(z)$ a first- or second-order direct II or normal subsystem), (4) parallel ($H(z) = \sum H_i(z)$, $H_i(z)$ a first- or second-order direct II or normal subsystem), (5) ladder–lattice, and (6) wave. Architectures are often instantiated in terms of a state variable model. The state variable model for a single-input, single-output, $N$th-order IIR having an arbitrary architecture is given in terms of a state equation $\mathbf{x}[k+1] = \mathrm{A}\mathbf{x}[k] + \mathbf{b}u[k]$, and output equation $y[k] = \mathbf{c}^{\mathrm{T}}\mathbf{x}[k] + \mathbf{d}u[k]$, where $\mathbf{x}[k]$ is an $n$-vector, $y[k]$ and $u[k]$ are scalars, $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{b}$ and $\mathbf{c}$ are $n$-vectors, and $d$ is a scalar. The $i$th state of the digital filter, $\mathbf{x}_i[k]$ resides in the system's $i$th shift register. The coefficient $\mathbf{A}_{ij}$ denotes the filter gain existing between the $i$th and $j$th shift register, $\mathbf{b}_i$ represents the gain between input and $i$th shift register, $\mathbf{c}_i$ the gain between $i$th shift register and output, and $d$ is the direct path gain between input and output. The state variable model is interpreted in Figure 8.4, where the $n$ states of the system are shown stored in $n$ shift registers.

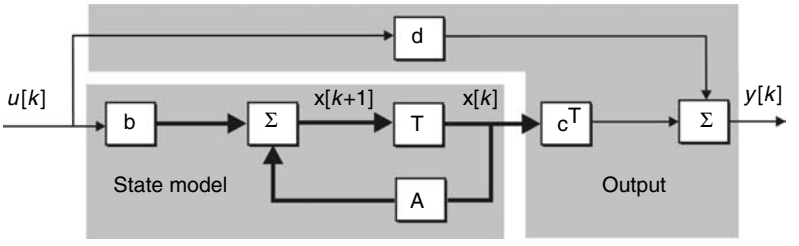



**FIGURE 8.4** State variable system model where $\mathbf{x}[k]$ is the state vector, $u[k]$ the input, and $y[k]$ the output.

The filter complexity and run-time dynamic range requirements of a system in state variable form can be mathematically computed or predicted. The dynamic range requirements are generally expressed in terms of the $l_p$ norm of the states, namely $||\mathbf{x}_i[k]||_p$, for $p = 1, 2, \ldots, \infty$, where $||\mathbf{x}_i[k]||_p = (\sum|\mathbf{x}_i[k]|^p)^{1/p}$. These norms are used to scale a filter in order to protect it against run-time register overflow, a serious error condition. Errors of less severity, which can nevertheless adversely influence system performance, are coefficient and arithmetic roundoff errors. Another minor error source is called limit cycling, a phenomenon that relates to the least significant bits (LSBs) of the output being "toggling" (producing a dynamically changing output) while the input is zero. Of the common architectural choices, cascade is the most popular and generally provides a good balance between performance, complexity, and precision. Direct II filters are known to be of low complexity, but often suffer from low precision. Parallel filters exhibit certain fault-tolerance attributes, ladder–lattice have good coefficient roundoff error immunity, but are comparatively complex.

## 8.4 Fourier and Spectral Analysis

The frequency-domain analysis and representation techniques can provide invaluable information about a signal and a system's environment. The mapping between the time- and frequency-domain is traditionally defined by a Fourier transform [1]. The historic difficulty in computing a Fourier transform of an arbitrary signal radically changed in 1965 when Cooley and Tukey introduced the now celebrated FFT algorithm. For over four decades, the FFT has been used to efficiently map a time series to and from the frequency-domain using a general-purpose digital computer. The FFT is a special manifestation of a more general class of transform called the discrete Fourier transform (DFT). The DFT defines a mapping of an $N$-sample time series $x_N[k]$ (possibly complex) into $N$ harmonics (complex) $X[n]$, where $X[n]$ is called the $N$th harmonic. The DFT of the $N$-sample time series $x_N[k]$ is given by the analysis equation:

$$X(n) = \sum_{k=0}^{N-1} x_N[k]\, W_N^{nk}; \quad n \in [0,\ N-1]$$

where $W_N = e^{-j2\pi/N}$. The DFT is also known to be periodic with period $N$ (i.e., $X[n] = X[n \pm kN]$). The inverse transform is given by synthesis equation:

$$x_N[k] = (1/N) \sum_{k=0}^{N-1} X[n]\, W_N^{-nk}; \quad k \in [0,\ N-1]$$

The DFT is parameterized in a manner shown in Table 8.5, and computes an $N$-harmonic spectrum using $N^2$ complex-multiply accumulates. The FFT algorithm significantly reduced the computational complexity of performing a DFT to $N \log_2(N)$. In general, a long FFT can be constructed from a

**TABLE 8.5** DFT Parameters

| DFT Parameter | Notation or Units |
|---|---|
| Sample size | $N$ samples |
| Sample period | $T_s$ seconds |
| Record length | $T = NT_s$ seconds |
| Number of harmonics | $N$ harmonics |
| Number of positive (negative) harmonics | $N/2$ harmonics |
| Frequency spacing between harmonics | $\Delta f = 1/T = 1/NT_s = f_s/N$ Hz |
| DFT frequency (one-sided baseband range) | $f \in (0, f_s/2)$ Hz |
| DFT frequency (two-sided baseband range) | $f \in (-f_s/2, f_s/2)$ Hz |
| Frequency of the $k$th harmonic | $f_k = kf_s/N$ Hz |

collection of small DFTs. Using the Cooley–Tukey FFT ordering algorithm, a DFT of length $N = \prod N_i$ can be created. Using the Good–Thomas ordering algorithm, a length $N = \prod N_i$ ($N_i$ and $N_j$ relatively prime) transforms result. For example, using a $N_1 = 15$, $N_2 = 16$, and $N_3 = 17$-point DFTs, an $N = 4080$-point Cooley–Tukey or Good–Thomas DFT can be computed.

In addition to the classic FFT, there are other spectral analysis techniques found in common use. One is called the chirp-z DFT that implements a DFT using linear convolution. The convolution filter has an impulse response that is equivalent to a linearly swept FM signal. Other DFT forms include filter banks and number theoretic transforms (NTT) that can compete with the FFT only in narrowly defined applications. Although not technically qualifying as a DFT, the discrete cosine transform (DCT) has significance in image compression applications, and like the FFT, has been reduced to both software and hardware instantiations.

The DFT and its derivatives are important signal analysis tools. They are sometimes used for off-line signal processing, whereas in other applications they must operate at real-time speeds using dedicated hardware, firmware, or software. An $N$-sample time series is often windowed (e.g., Hann) before being transformed to improve the interpretability of the resulting DFT. DFTs can also be used to convolve two time series if the DFTs are suitably modified. The DFT assumes that the signals being transformed are periodic, with period $N$. As a result, the convolution theorem for DFT is expressed as the periodic outcome $y[k] = x[k] \otimes h[k] = \text{IDFT}(\text{DFT}(x[k]) \times \text{DFT}(h[k]))$, where $\otimes$ denotes circular (periodic) convolution, and IDFT denotes an inverse DFT. A circular convolution can be functionally converted to behave like a linear convolution by adding a string of $N$ zeros to the time series $x[k]$ and $h[k]$ before performing the DFTs. This process, called zero padding, allows an efficient FFT replace an inefficient linear convolution sum. This advantage is exploited in high-order application, such as convolving two large, two-dimensional images.

One of the principal uses of a DFT or FFT is in performing spectral analysis [12]. Spectral analysis pertains to the study of signals and systems based on their frequency-domain signatures and attributes. The frequency-domain image of a signal or system is often interpreted in terms of a power spectrum that is a display of the power in a process on a per-harmonic basis. Spectral analysis methods generally fall into two classes, called parametric and nonparametric. Nonparametric spectral analysis methods (e.g., periodogram) are based on the DFT of one or more noise contaminated time series records. The individual spectra can be averaged or combined in various ways to create a more interpretable frequency-domain image of the signal or system under study. Parametric methods attempt to build a mathematical model of a process that approximates the measured power spectrum of a signal or system. The moving-average (MA) method is a parametric spectral estimation method that constructs an FIR (all zero) model of a signal process or system. Another parametric method is called auto-regressive (AR) and produces an all-pole IIR model of a signal or system, combining the two results in the parametric auto-regressive moving-average (ARMA) method. Other parametric methods are based on an eigenvalue analysis and result in a highly frequency-selective signal or system model.

## 8.5 DSP System Implementation

The implementation of a digital filter is an iterative process requiring design trade-off choices be made in the statement of filter specifications, filter type, architecture, and technology to achieve a design that meets performance, precision, and complexity (cost/power) requirements. Numerous software packages are commercially available to automatically design a baseline FIR or IIR filter of DFTs. Fewer software packages automatically support architectural or design optimization activities that can quantify run-time errors and register overflow saturation events. Furthermore, the majority of hardware-enabled DSP filters and transforms are implemented in fixed point, a point often ignored by existing design software tools [4]. The range of an unsigned $N$-bit fixed-point number $X$ is given by $R = X_{\max} - X_{\min}$, and has a resolution given by $Q = R/2^N$, where $Q$ is called the quantization step-size and is the weighted value of the LSB. The quantization error is defined to be the difference between a number's real and fixed-point

representation, specifically $e = X - X_Q$ (rounded). Statistically, the error is uniformly distributed over $[-Q/2,\ Q/2]$, with mean and variance is given as $E(e) = 0$ and $\sigma^2 = Q^2/12$, respectively. Of all the known fixed-point numbering systems, two's complement (2C) is by far the most popular and important. A 2C attribute, that makes it particularly attractive for DSP applications, is called the modulo $(2^N)$ wraparound property. This property states that if a string of valid 2C numbers $\{X_i\}$ are added to form $S = \sum X_i$, and if $S$ is a valid 2C number, then the final outcome will be correct regardless of possible overflows of intermediate sums.

For cases where higher dynamic ranges are needed, floating-point solutions are employed. The floating-point representation of a real number $X$ is given by $X \sim (-1)^S\, Mr^{\mathrm{E}}$, where $M$ is the mantissa, $r$ is the radix, E is the signed exponent, and $S$ is the sign bit. The mantissa is usually normalized to a value $1/r \leq M < 1$, and the format is defined by published standards (e.g., IEEE). A variation on the floating-point theme is called block floating point, a format used by a number of DSP chips, especially FFTs. A block floating-point representation of an array of numbers $\{x[k]\}$ is defined in terms of a maximum exponent E, where $|x[k]|_{\max} = r^{\mathrm{E}}$. A block floating-point representation of the number $x[k]$ is given by $x[k] = \pm M[k] r^{\mathrm{E}}$, where E is the fixed maximum exponent and $M[k]$ is a fractional mantissa ($M[k] \leq 1$). Since the scale factor $r^{\mathrm{E}}$ is known a priori, it need not be explicitly carried in number system representation.

The primary DSP arithmetic operation is the signed MAC. Fixed-point multipliers cover a wide range of speed, precision, and complexity trade-offs [2]. Compact low-complexity MACs can be designed using ripple adders. When adder area and power dissipation are not an issue, carry-lookahead adders can be used to accelerate wide wordlength adders and, therefore improve MAC speed. Carry-save adders (modified full adders) can also be an important element in implementing fast multipliers. Another fast multiplier architecture is based on Booth's algorithm and interprets strings of consecutive "ones" as multiplicative NO–OP operations. Fast multipliers can also be constructed using arrays of small wordlength multipliers. These architectures are referred to as cellular array multipliers, or simply array multipliers.

General-purpose programmable DSP μps make use of multipliers that map $XY \rightarrow P$, where $X$ and $Y$ are variables. Most DSP applications are SAXPY ($S = AX + Y$) intensive, which refers to multiplying a variable $X$ by a constant $A$ (e.g., filter coefficients), followed by an accumulation. Implementing SAXPY algorithms technically does not require general multiplication but rather an operation called scaling. Several techniques have been developed to exploit scaling in the implementation of DSP algorithms. They are particularly useful in implementing fixed-coefficient DSP algorithms with application-specific integrated circuits (ASIC), application-specific standard parts (ASSP), and field-programmable gate-array (FPGA) devices [10]. One scaling technique is called the reduced adder graph (RAG) method. RAG arithmetic is based on the theory of the ternary-valued ($\{0, \pm 1\}$) canonical sign-digit (CSD) numbers. For example, the 4-bit binary unsigned representation of the number 15 is $15_{10} \leftrightarrow 1111_2$, whereas the RAG representation is given by $15_{10} = 16_{10} - 1 \leftrightarrow 10001_{\mathrm{RAG}}$, which can be implemented using one adder and a shift register. The cost of an RAG multiplier is measured in terms of the number of adders needed to complete a design. Another scaling method is called distribute arithmetic (DA) and is applicable only to the implementation of constant DSP coefficient algorithms. As a point of reference, an $N$th-order FIR digital filter, having known coefficients $h_r$, $r \in [0, N)$, requires $N$-MAC operations be performed per cycle. The data is assumed to be coded as an $M$-bit 2C word, where

$$x[k] = -x[k:0] + x[k:1]2^{-1} + \cdots + x[k:N-1]2^{-(N-1)}$$

where $x[k:i]$ is the $i$th bit of sample $x[k]$. The output $y[k]$ is given by

$$y[k] = -\sum_{r=0}^{N-1} h_r x[k-r:0] + \sum_{i=1}^{M-1} 2^{-i} \sum_{r=0}^{N-1} h_r x[k-r:i] = \Theta[x[k:0]] + \sum_{i=1}^{M-1} 2^{-i} \Theta[x[k:i]]$$

here the mappings $\Theta[x[s:i]]$ are implemented using $2^N$-word semiconductor memory lookup table. The lookup table $\Theta$ maps an array of binary valued digits $x[k:i] = \{x[k:i], x[k-1:i], \ldots, x[k-M-1:i]\}$, taken from the $i$th common-bit location from $x[k]$, $k \in [0, \ldots, N-1]$, under the rule

$$\Theta[k:i] = \sum_{r=0}^{N-1} h_r x[k-r:i]; \quad x[s:i] \in [0,1]$$

Weighting the lookup value $\Theta[x[s:i]]$ by a factor $2^{-i}$ is implemented using a shift register. The result is generally a high-speed compact design.

## 8.6 DSP Technology

The semiconductor revolution, which began in the twentieth century, began to shape the field of DSP beginning in the late 1970s. Since then, DSP has been both a facilitating technology (replacing existing solutions) as well as an enabling technology (creating new solutions). The hallmark of the DSP technology revolution was the general-purpose DSP μp. The first generation DSP chips included on-chip ADC and DAC and large capable multiplier. The second-generation DSP μps overcame many of the first-generation device memory and precision limitations, and also removed the noisy on-chip ADCs and DACs. Since then, third-generation floating-point and fourth-generation multiprocessors have been added to the list of general-purpose DSP products [3]. Along with the DSP μp technology explosion, came the attendant improvements in software for both uni- and multiprocessor systems. High- and low-level software environments have been created to rapidly develop and test DSP solutions. Since DSP problems tend to be algorithmic, stressing real-time bandwidth, optimized solutions continue to be dominated by assembly language code solutions. Throughout these generational changes, DSP μps have maintained their dependence on capable MACs, tightly coupled memory, and a modified Harvard architecture. These trends continue to differentiate DSP μps from general-purpose microprocessors. Microprocessors emphasize (1) multiple data types, (2) multilevel cache memories, (3) paged virtual memory management in hardware, (4) support for hardware context management including supervisor and user modes, (5) large general-purpose register files, (6) orthogonal instruction sets, and (7) simple or complex memory addressing, depending upon whether the processor is RISC or CISC. DSP μps, however, typically have (1) only one or two data types supported by the processor hardware; (2) limited data cache memory; (3) no memory management hardware; (4) no support for hardware context management; (5) exposed pipelines; (6) predictable instruction execution timing; (7) limited register files with special-purpose registers; (8) nonorthogonal instruction sets; (9) enhanced memory addressing modes; (10) onboard fast RAM, ROM, and DMA; and (11) nonsequential access to data addressing modes (e.g., bit-reversed addressing). Techniques have also been developed to exploit opportunities for instruction-level parallelism, super-pipelining, and superscalar architectures. These innovations have led to very long instruction word (VLIW) architectures. Because of the upward spiral of software development costs, a significant amount of the academic and commercial activities have been directed to automatic compiler-based optimization of high-level language code.

In parallel with the explosion of general-purpose DSP μp products, there has been a growing presence of DSP-centric ASICs, ASSPs, and FPGAs. Although DSP μps enabled the DSP revolution, DSP technology innovations have become increasingly driven by intellectual property (IP) supplied by semiconductor houses, fabless semiconductor technology suppliers, and third-party IP providers. Their use and justification is based on performance, power dissipation, cost, and time-to-market considerations. At the beginning of the new millennium, the market value of ASICs and ASSPs exceeded that of general-purpose DSP μps. The trend toward ASICs and ASSPs, over DSP μps, is motivated by the need to achieve rapid system-on-a-chip (SOC) designs by integrating predefined DSP IP cores together using high-end electronic design automation (EDA) software. FPGAs are becoming an increasingly important DSP technology but continue to remain primarily prototype tools and useful in some low-volume applications.

## 8.7 Applications

DSP has become a well-known acronym, often appearing explicitly in the marketing vernacular of commercial electronic products. The sphere of influence and relevance of DSP continues to expand, often enabling solutions that could only be speculated a decade earlier. Modern DSP applications areas include general-purpose DSP (filtering, signal detection/classification, spectral analysis, adaptive filtering); instrumentation (waveform analysis, transient analysis); information/communication systems (speech, audio, voice over Internet, facsimile, modems, cellular telephones, wireless LANs); control systems (servos, disks, printers, automotive, guidance, vibration, power systems, robots); entertainment (sound, video, music); defense (radar, sonar, object recognition, ordinance) plus other areas such as biomedical, transportation, entertainment, and geophysical signal processing.

### References

1. Blahut, R., 1985, *Fast Algorithms for Digital Signal Processing*, Addison-Wesley, Reading, MA.
2. Brown, S. and Varnesic, Z., 2000, *Fundamental of Digital Logic with VHDL Design*, McGraw-Hill, New York.
3. Cavicchi, T., 2000, *Digital Signal Processing*, John Wiley & Sons, New York.
4. Koren, I., 1993, *Computer Arithmetic Algorithms*, Prentice-Hall, Englewood Cliffs, NJ.
5. Mitra, S., 2001, *Digital Signal Processing*, 2nd ed., McGraw-Hill, New York.
6. Mitra, S., 2006, *Digital Signal Processing*, 3rd ed., McGraw-Hill, New York.
7. Oppenheim, A. (Ed.) 1978, *Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.
8. Oppenheim, A.V. and Schafer, R.S., 1975, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.
9. Oppenheim, A. and Schafer, R., 1998, *Discrete-Time Signal Processing*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ.
10. Taylor, F., 1983, *Digital Filter Design Handbook*, Marcel Dekker, New York.
11. Taylor, F. and Mellott, J., 1998, *Hands-On Digital Signal Processing*, McGraw-Hill, New York.
12. Zelniker, G. and Taylor, F., 1994, *Advanced Digital Signal Processing*: *Theory Applicants*, Marcel Dekker, New York.

# 9

# DSP Applications




Daniel Martin
*Infineon*


## 9.1 Introduction

The story goes like this. In 1982, when Texas Instruments' (TI) engineers came up with their first general-purpose chip for DSP applications, they did not know how to call it. Terms like analog microprocessor or signal microprocessor sounded cumbersome for the user. Therefore, an engineer said, "why don't we confuse the chip and its application? In other words why don't we use the term DSP (digital signal processing) to describe our chip?" Hence, the DSP (digital signal processor) was born. Unfortunately, this still brings confusion 20 years later.

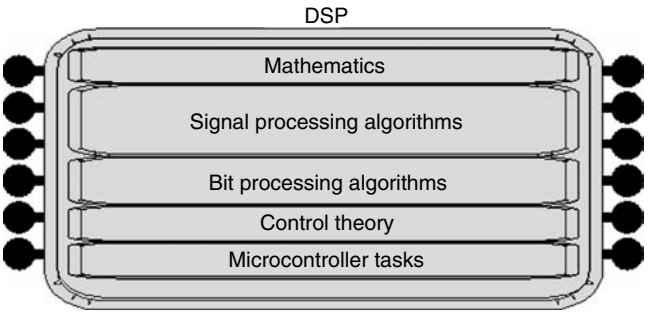



**FIGURE 9.1** A general purpose DSP can do more than DSP.

### 9.1.1 DSP (Digital Signal Processor) or DSP (Digital Signal Processing)?

What do we mean by DSP applications?

Applying the science of digital signal processing to the real world?

An application that uses a digital signal processor?

Although the two areas largely overlap, they are not identical. For instance, a typical digital signal processing application such as V90 modem is performed by a general purpose DSP but also by a custom chip or a Pentium.

### 9.1.2 A DSP Can Do More Than DSP Applications

A general purpose DSP can be efficient at many other tasks than pure processing of signals (Fig. 9.1). The reason is that a DSP is low cost and very efficient at processing in general. It is also good at processing math, bits, events, state-machines, etc. In addition, a DSP has a very deterministic behavior. Hence, it can precisely control hardware and multiple external events. It is the main reason that hard disk drives use a DSP as their main CPU. Disk drives and motor control represent one of the biggest applications for a DSP. They are classified under DSP applications, in reality they are more control-like; however, the "spread" of a general purpose DSPs into non-DSP applications is much less interesting than the discovery of new DSP applications.

In the following paragraph, we will concentrate on describing applications, which recently opened new markets thanks to some DSP techniques.

### 9.1.3 The Importance of DSP Applications

Over the last 20 years, the different market segments have made a different use of DSP applications (Fig. 9.2). The next 10 years will also bring its changes. For instance, the consumer market is likely to occupy more and more space in the life of DSP engineers.

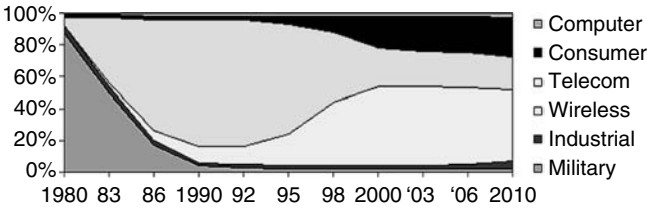



**FIGURE 9.2** Relative importance of market segments for DSP applications.

### 9.1.4 Classifying DSP Applications

It is usually to classify DSP applications following the market segments. For the purpose of this chapter, we conveniently classify the seven main areas of DSP applications without necessarily following the market segments of Fig. 9.1:

1. Military
2. Telecommunication terminals
3. Consumer products
4. Telecommunication infrastructure (including networking)
5. Computers, peripherals, and office automation
6. Automotive, industrial
7. Others, such as biometrics, biomedical, etc.

## 9.2 Military Applications

The first DSP applications were born in the 70s and were mainly military (radar, sonar). Today, the same applications exist with a much higher performance target. In addition, many military applications (vocoder) are taking advantage of "civilian" work. An interesting development for the military is the detection and disposal of mines [1]. It must be noted that, since the military community was the first DSP customer, strong links were created between DSP manufacturers and these pioneers. Despite its small size, the military community continues today to have a strong influence on the evolution of DSP architecture (floating-point, multiprocessing).

## 9.3 Telecommunication Terminals

By 1995, DSP has left its original circles (military, universities) and became a household name. The most popular DSP applications were telecommunications terminals such as cellular telephone, PC (modem), fax, and digital-answering machine (Table 9.1). Today, the quantity of telecom terminals that is produced per year reaches 450 M units for cell phones alone.

### 9.3.1 Phones and Answering Machines

The plain old residential telephone has very little DSP inside it (maybe calling ID). This is the exception among voice communication devices. For instance, second generation cordless phones (DECT) use digital techniques. Also all "packet telephones" such as Internet (IP) phones, LAN phones, are using voice compression, echo cancellation, and modem techniques to receive/transmit voice. An extreme telephone application can cater for up to 12 voice conference channels. This requires 12 decompression channels and mixing. Voice compression is not new, since they allowed the development of cheap "solid-state" DAM (digital answering machines). All put together, a combo device including a multi-channel cordless + a DAM + a connection to IP is a very demanding DSP application.

**TABLE 9.1** List of Telecom Terminals Using DSP Techniques

| Communication Function | List of Terminals |
|---|---|
| Voice (telephony) | Feature phone, DAM, cordless phone, Internet phone, business phone, LAN phone, DECT phone, combo products (LAN/POTS) |
| PC modem | (Voiceband) modem, DSL modem, cable modem |
| Fax | Color fax, fax/phone, fax/printer, fax/printer/scanner/copier |
| Web access | Web station, Web phone, Web pad |
| Videophony | Videophone |
| Cellular phone | Standard, voice + data cellular phone, smart phone, pager |

### 9.3.2 PC as a Terminal (Modem)

The PC is the second most successful telecom terminal of all times. For that it requires a modem (most advanced voiceband modem is V90). The modem was the application that created a mass market for DSP devices (1982–1992). Today, all DSP devices are trying to implement broadband modems. Roughly five classes of broadband modems are available, which all use massive amount of DSP power:

- DSL, which is made of three classes in order of difficulty (SDSL, ADSL, VDSL)
- Cable modem, which is classified as a set-top box peripheral
- Broadband wireless modem (LMDS, MMDS), which is also called the "wireless DSL"
- Broadband satellite modems
- Gigabit Ethernet (and above), which positions itself as the cheapest technology

### 9.3.3 Fax

A fax can be seen as medium range modem, plus a scanner writing bits into a graphics compression engine, and a decompression engine driving a printer. The three functions are all DSP-based. There is no reason why fax manufacturer will not develop DSL fax. Modern networks will allow a DSL fax to speak to a LAN fax. In fact, modern networks will allow "anything over everything" such as fax-over-IP and voice-over-DSL. You can bet that DSP will be in the middle of all that.

### 9.3.4 Web Access Terminals

Not to be confused with an IP phone (which is limited to voice communication), a Web access device is targeted at Web browsing and e-mail. Today (2001), all these types of devices and the so-called "Internet appliances" are struggling to find a mass-market acceptance. Despite this, three classes exist: Web station, Web phone, and Web pad.

#### 9.3.4.1 Web Station

It is a $99–299 consumer device in the form factor of a small laptop. It allows web browsing, send/receive e-mail, and (maybe) JPEG decode. Web browsing requires a modem (more likely V90 than DSL), which means DSP. Since V90 is less than 30 DSP MIPS and today's DSPs give anything from 100 to 1500 DSP MIPS, the use of a full-blown DSP might not be required. On the other hand, the unused performance can be put to good use: multimedia decode.

#### 9.3.4.2 Web Phone

This is the same as the Web station with the addition of telephony. Note that in the IP world, phoning requires more DSP MIPS than Web browsing.

#### 9.3.4.3 Web Pad

This is a cordless web station with a form factor identical to the pentop of 1992–1994. The DSP functions are fifty/fifty shared between the base and the tablet. The big advantage of a Web pad is to be network independent or modem independent. The big disadvantage is the price of the display.

### 9.3.5 Videophone

Videophone shares with speech recognition the honor of being the most promising 1971 DSP application. Thirty years later, many progresses have been made. The next 10 years will surely bring their annual series of breakthroughs.

### 9.3.6 Cell Phones

The modem put DSP on the radar screen in the 80s. By comparison, cell phones put DSP in the stratosphere in the 90s. By the end 1999, the cell phone was the star of the electronics world with more

**TABLE 9.2** A List of Possible Wireless Devices

| | Cellular | Proximity (Bluetooth) | Home RF (Residential) | DECT (Cordless) | Wireless LAN | Broadband | Satellite |
|---|---|---|---|---|---|---|---|
| Phone | | | | | | | |
| Modem | | | | | | | |
| Fax | | | | | | | |
| Web access | | | | | | | |
| Videophone | | | | | | | |
| Digital camera | | | | | | | |
| Palm-top | | | | | | | |
| DVD player | | | | | | | |
| DVR | | | | | | | |
| Set-top | | | | | | | |
| Digital TV | | | | | | | |
| Games | | | | | | | |
| MP3 player | | | | | | | |
| Home theater | | | | | | | |
| DAB | | | | | | | |
| MP3 Juke-box | | | | | | | |
| E-book | | | | | | | |
| PC | | | | | | | |
| Printer | | | | | | | |
| Car | | | | | | | |

than 300 million handsets a year, some containing multiple DSPs. Multiple DSPs are needed because a cell phone DSP function is traditionally divided into two parts: the speech coding and the channel coding.

*Speech coding* is a traditional speech codec (compression/decompression) algorithm varying from 5 to 12 kbit/s depending on standards, economic forces, and target quality. Above that, several speech-quality enhancement features are added. This includes echo cancellation and noise suppression. A promising trend is the use of wideband codec. All together, the sum of all speech functions put in a modern cell phone can require up to 100 DSP MIPS.

*Channel coding* is working on bits in transmission and (supposedly) in reception. As such, it does not qualify as a pure DSP application; however, in the first place, the reception is mainly done on samples and secondly equalization and other heavy DSP techniques (Viterbi algorithm) are classified under channel coding. Finally, the channel coding problems represent DSP research at its best today.

### 9.3.7 Wireless Terminals

The cell phone is the first of many types of wireless terminals that will come up over the next decade. In fact, wireless terminals are in a class of their own. Their rapid evolution differentiates them strongly from their wired cousins. Wireless is the technology with the most development potential over the next 10 years. It is easy to explain this statement by taking any existing equipment (from telephones to automobile) and turn it into a wireless device (Table 9.2). It is left to the reader to complete the table based on his or her own wishes.

## 9.4 Consumer Products

Section 9.3 proves that wireless will revolutionize many types of equipment. This is especially true for consumer devices. For instance, Bluetooth and GPS (both based on DSP) will be standard features on most consumer products described in the following subsections. In addition, consumer products have been traditionally nonconnected devices (camera, CD player) or passive devices (television). This is changing, in the form of access to the Web. This itself gives a big push to DSP applications.

## 9.4.1 Digital Cameras (and Digital Pictures)

One of the most promising consumer DSP applications is the field of digital pictures. Its most common incarnation is the digital camera. This very large field can be segmented in many ways, following these characteristics:

- Fixed pictures versus moving pictures (example: digital camera versus digital camcorder)
- Picture production (camera) versus picture consumption (digital frame)
- Portable versus semi-fixed/fixed equipment (example: digital camcorder versus webcam)
- Equipment versus module (example: digital camera versus add-on to a palm-type device)

### 9.4.1.1 Digital Camera

A digital camera is made of several functions: an image sensor, a processing part, and a storage element. The processing includes three main algorithms, front-end processing, image compression (DCT is mainly used here), and coding (Huffman coding).

In theory, a digital camera requires 10 DSP MIPS. Nevertheless, higher resolution, advanced algorithms (pixel by pixel) and sophisticated features such as the paparazzi effect turned the digital camera into a big DSP MIPS consumer. The paparazzi effect is when a series of pictures are taken at high speed (for example, 10 pictures in 1 s). In effect, we are not far from the performance of a video camera.

### 9.4.1.2 Digital Video Camera (Camcorder)

Big brother to the still camera, the video camera follows the same principle. It approximates the behavior of a digital camera except it has a better resolution and a continuous automatic stream of pictures. Another key difference is that it is a slave to the television set. Hence, decompression of pictures is as important as compression.

### 9.4.1.3 Web Camera

Not all video cameras need the sophistication of a camcorder. Common examples are surveillance cameras (slow speed, black and white) and Web cameras. The Webcam's block diagram is very similar to a digital (still) camera except the storage function has been replaced by a modem. Because the speed of the network is the bottleneck, there is no need to take more than one or two pictures every 5 s. Note also that a Web camera does not need any decompression algorithms.

### 9.4.1.4 PC Camera

The PC is a $10 digital video camera put on top of PC and used for video telephony or college room broadcasting. Its consists of a very low sensor quality and a sub-dollar micro-controller. The PC has taken the role of a DSP.

### 9.4.1.5 Modules and Toy Cameras

In the same spirit, any host can take the DSP role. For instance, there is the case of digital camera modules (host independent), add-on to a PDA (palm OS is the host), and toy cameras (PC is host).

### 9.4.1.6 Digital Picture Frame

Not the most fascinating killer application of all times (sending baby pictures to grandparents), the digital picture frame is exactly the opposite of the Web camera. The image first goes through a modem function, then through decompression, and ends up its life on a picture frame display; however; contrary to a Web camera there are large problems due to the human interface and the way we (the grandparents) interact with this kind of device.

## 9.4.2 PDAs (Handheld Devices, Palmtops)

PDA is not (yet) a big DSP platform. Still serious inroads are made. Two common ones are the use of a PDA as a common platform for digital camera and MP3 player. Also Web access (necessitating a

modem) and wireless access (obviously necessitating a wireless link) are the two good classical DSP applications, which are being pushed into these devices.

### 9.4.3 DVD Player (and Digital Storage Devices)

Storage devices such hard disk and CD-ROM players are basic sub-elements of the PC. The CD audio is also a well-known element of our life. The equipment, which really puts DSP into the consumer storage field, is the DVD player.

#### 9.4.3.1 DVD Player

A digital versatile disk (DVD) uses MPEG2 compression to store its video and audio tracks. A DVD player requires in the order of 200 DSP MIPS to decode the signal. Still, compared to some recent consumer platforms the function seems relatively straightforward, but this is only one-third of the DSP functions. The other two functions are the control of the disk (servo) and the reading and decoding of the stored data bits (channel coding).

#### 9.4.3.2 Universal Player/Recorder

Moreover, the DVD player is fast becoming a recorder device. MPEG2 coding algorithm necessitates many more DSP MIPS than decoding. Finally, the number of standards (in other words the number of DSP algorithms), which are currently supported by a DVD player, is mind-boggling. Effectively, the DVD player is the de facto universal home player/recorder. It can do nearly everything from recording MP3 audio to reading karaoke Chinese videodisk.

#### 9.4.3.3 DVR (Digital Video Recorder)

Further pushing this recording trend is the emergence of the DVR. Here there is no disk to read or to record. Or, more specifically, there is no BOUGHT disk; however, this is still a storage device (hard disk) on which television program can be stored (recorded) and read in nearly real-time. The DSP algorithm is the same as DVD (omnipresent MPEG2) but with the added complexity of simultaneous coding/decoding. In fact, there are two coding channels and one decoding channel requiring more than 700 DSP MIPS of DSP power.

### 9.4.4 Digital Set-Top Box (and Digital Television Peripheral Devices)

The DVR function just described can also be integrated in a set-top device. We will call set-top devices any consumer devices, which sits at home between the TV operator(s) and the television set (hence the name set top). A Web TV fits neatly into this definition.

#### 9.4.4.1 Digital Set Tops

Two types of digital set-top boxes are currently used, the wired and the wireless. The wired is the well-known connection to a cable, the wireless is the satellite type. Both require a massive amount of DSP in the demodulation/error correction schemes, followed by the good old MPEG2 decode. It must be noted that the DSP functions have a relatively minor role to play in the whole software. Set-top boxes are considered more of an open platform similar to a PC, than a closed device such as a DVD player. This comment was to introduce the current evolution of set-top boxes from one-way device to two-way devices (up-link is added to down-link).

#### 9.4.4.2 Two-Way Set Tops—Cable Modems and Web TVs

But what about the amount of DSP functions? Intuitively both devices would require twice the number of DSP workload since they now receive and transmit information. This is not so. The up-link is only for data, consequently the need for compression is null and the modem speed relatively low. To summarize, DSP did not drive the recent evolution of set-top boxes; however, this might change if they evolve into multimedia home gateways.

### 9.4.5 HDTV (and Digital Television)

If there is a domain in which DSP is bringing a lot, this is high definition TV (HDTV). This is not due to the high definition but to the use of digital techniques. Contrary to the current digital television, the digital functions (read MPEG2) are not put into a peripheral device but in the TV set. Even if there is still a lot of uncertainties in this market, there is no doubt about its massive use of DSP power.

### 9.4.6 GAMES (and Toys)

Although it is not the obvious place where DSP can found, games and toys have more and more needs of DSP because of the need for communication.

#### 9.4.6.1 Games Consoles (3D)

Games have massive amount of CPU and hardware power devoted to the manipulation of 3D graphics. It is interesting to know the three reasons why this cannot be classified as DSP. The first one is that 3D graphics is executed in floating point (whereas DSP is 95% fixed point). Reason number two is that graphics is a synthesized object whereas DSP manipulates real signals. Finally, DSP is software whereas graphics is a pipelined hardware. It is obvious that a lot of DSP applications can be found, which corresponds to the three above criteria. What about a 33-stage hardware floating point multi-channel polyphase audio synthesizer. Also, there are a lot of graphics algorithms, which are *not* floating point for instance. The bottom line is that the world of gaming, the world of video communications, and the world of image processing are now very close:

- Mixing of synthesized and real images found in modern games
- The adoption of MPEG4 as a telecom standard (MPEG4 principles rely on objects commonly found in PC graphics)

#### 9.4.6.2 Game Consoles as an Universal Platform

The same story as for set-top box or PDA applies here. Web browsing, modem, DVD player, MP3 player are all good examples of DSP applications. All are finding their way into game consoles.

#### 9.4.6.3 Toys

The first consumer device based on DSP was the Texas Instruments' speak and spell learning aid (1981). In fact, it was a toy disguised as a learning aid. Another TI DSP milestone was the famous "Julie doll" (1987). For the future, a lot of toys will be based on sophisticated electronics, adaptive behavior, and connected to the PC (possibly Bluetooth). All these functions have strong DSP contents.

### 9.4.7 MP3 Player (and Listening Musical Platforms)

Traditionally, the music industry was relying on very crude DSP in the consumer product (CD player). The explosion of MP3 portable devices is opening the doors to sophisticated DSP in mass-market audio devices.

#### 9.4.7.1 MP3 Player

When drawing a block diagram of a MP3 player, one can use the block diagram of a portable digital picture frame and replace the display by the connection to the speaker. This represents the simplicity of a MP3 player. The DSP MIPS number is low and the DSP functions pretty basic. Nevertheless, as for DVD, the difficulty is in the number of audio format to support (each one means a different DSP algorithm) and the security features. Note that a large number of MP3 players are built with a single DSP (no micro-controller host), which means than its 80% of its program is used for NON DSP work.

#### 9.4.7.2 Hi-Fi

A large number of high-fidelity equipment rely on DSP techniques. The most common is the Dolby standard, which can be found in cinema and home (5.1 channel) theater. The most exotic could be the digital speaker. The most difficult and resource intensive DSP application is the so-called 3D sound

**TABLE 9.3** Consumer Equipment—How Big Is the Market and How Much DSP Is Required?

| Consumer Equipment | Units Sold per Year | DSP MIPS |
|---|---|---|
| Digital camera | 10 M | 10 → 1000 |
| Palm top | 10 M | 50 |
| DVD player | 60 M | 300 |
| DVR | <1 M | 700 |
| Set top | 50 M | 300 |
| Digital TV | <1 M | 1000++ |
| Games | 100 M | 100,000+ (graphics) |
| MP3 player | 10 M | 20 |
| Home theater | <1 M | 1000+ |
| DAB | <1 M | 80 |
| MP3 juke box | <1 M | 20 |
| E-book | <1 M | 5 |

(PC games). Everyone knows about the difficulty of generating good 3D graphics. But one can appreciate the difficulty of generating 3D sounds when doing the comparison. An image is still displayed in a 2D world, whereas sound is really produced for a 3D world (by analogy: image will catch up with sound when it will be displayed as a hologram). In effect, we are speaking of thousands of DSP MIPS.

#### 9.4.7.3 Musical Instruments

In the professional musical world such as synthesizers, DSP first appeared as a label of quality. Music is a field where DSP can introduces massive improvements. For instance, adaptive techniques could make a good old country fiddle sounds like a Stradivarius. It does not sound like a very good idea, though.

### 9.4.8 Home Networking and Multimedia

MP3 player is the top of iceberg. The iceberg is the "connected" home. The infrastructure of this connected home is partially described later (refer to home gateway heading). Here, the new "gizmos" that this infrastructure allows are briefly described:

*MPEG4 player*: This is the same as MP3, except it also allows viewing video clips.

*Internet radio*: listening to radio on the Internet.

*MP3 juke box*: listening to MP3 clips; they had been previously stored on a hard disk drive (from the Internet Web sites). Similarly, we can add MPEG4 juke box.

*Home Storage or Multimedia Storage Box*: hard disk drive containing multimedia files.

*E-book*: Electronic book. Presently downloaded from the Web; in the future, this will done in two passes: first to the home storage and then to the e-book.

Table 9.3 summarizes the DSP requirements of some consumer devices. Knowing that most of them are starting their commercial life, one is impressed by the amount of work remaining for DSP engineers.

## 9.5 The Telecom Infrastructure

The telecom infrastructure could be divided into three spheres of influence (wired, wireless, networking). The convergence of all networks renders this distinction illusory. A very interesting trend is that infrastructure equipment such as servers, gateways, switches, radio relays are now finding their way into the home.

### 9.5.1 CTI (Computer Telephony Integration)

Before the net, CTI was the biggest infrastructure user of DSP. CTI means voice server, voice mail, and the infamous IVR (interactive voice response) machines "please hold on, etc." All these infrastructure

equipment are built using standard software modules and boards. For years CTI was the lifeline of many DSP board manufacturers.

### 9.5.2 Modem Banks

This is the first of the multi-channel DSP applications. Modem banks did not appear because of PC modems. They appeared because of the Internet (web servers) and remote workers (remote access servers). The number of required DSP MIPS is extremely high. For instance, a typical bank of 120 V90 modems requires 3600 DSP MIPS ($120 \times \sim 30$).

### 9.5.3 DSL Modem Banks

This is nothing compared to DSL, where (for instance) 30,000 DSP MIPS ($120 \times \sim 250$) is required for a 120 channels S-HDSL modem bank.

### 9.5.4 Broadband Line Card (Voice-over-Broadband)

The most likely example of broadband line card is a DSL line card, which fills the same function as modem banks plus the typical voiceband functions (echo cancellation, voice compression, DTMF detection, fax relay) found in gateways.

### 9.5.5 Gateway (Voice-over-Broadband)

Under this heading are included all recent buzzword equipment such as voice-over-DSL, voice-over-IP, voice-over-Packet, etc. A gateway can be on the periphery of the network (access), in the center (core) or on customer premises (private). Its main use is to translate from a circuit network to a packet network and back. A state-of-the-art gateway SOC (System-on-Chip) targets 200 channels, which translates into 4000–10,000 DSP MIPS depending on the voice compression quality. The voice quality has been the subject of a lot of debate (and hard-learnt lessons) in the IP community over the last four years.

### 9.5.6 Cellular Wireless Base Station

In wireless, voice quality is not a problem (relatively speaking) since compression algorithms are standardized. They require as low as 3 DSP MIPS (GSM full rate) to 30 DSP MIPS (third generation such as AMR) per channel. Cellular wireless base stations are half gateway (access to network) and half radio relay (air interface). It is this air interfaces, which presently (2001) presents a lot of challenges to the DSP world. Many MIPS-hungry techniques have been introduced (CDMA, turbo coder) and many more will be coming (smart antenna, multiple reception, software radio, etc.) over the next 20 years. In essence this is 25,000 DSP MIPS per channel. In other words, each channel requires a 25 GHz general purpose DSP. One can see the interest of application specific DSP and custom instruction set in this market.

### 9.5.7 Home Gateways and "Personal Systems"

A residential cordless base station is now the most common example of a "personal system." A personal system is a device having both characteristics of telecom terminals and telecom infrastructure. It is a terminal because:

1. It is sold in retail stores.
2. It is targeted at a small entity (single person, family, SOHO).

It is a telecom infrastructure equipment because:

1. It has no human interface.
2. It very often acts as a point-to-multipoint access device.

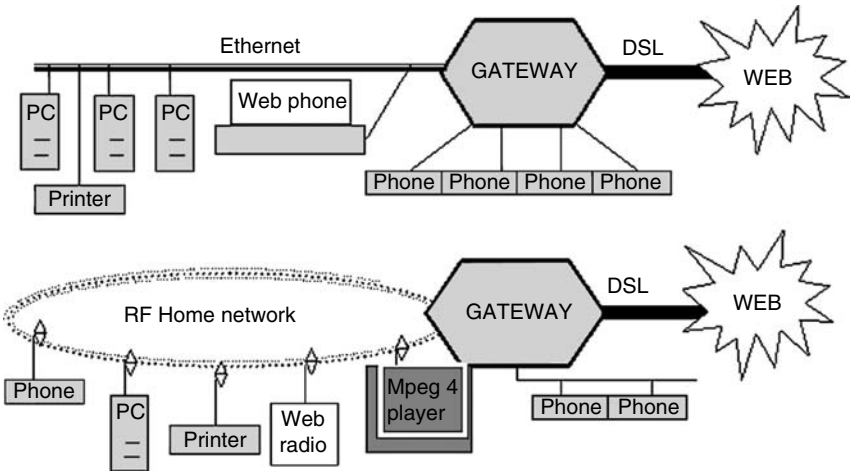



**FIGURE 9.3** Two examples of residential gateways—SOHO and HOME.

#### 9.5.7.1 What Is Residential Gateway?

This is the latest trend in "personal system." Two examples are given in Fig. 9.3. The first one is a DSL (external) to Ethernet/twisted pair (internal) gateway. It could be a typical SOHO scenario where phones are organized in star topology. Note that the LAN supports additional phones.

The second one is a HOME gateway, which differs by being driven more by entertainment than by work. It also uses an external DSL link but the internal communications are mainly wireless. The wireless network is used to avoid rewiring the house. The phones are organized in bus topology, which also corresponds to a typical home.

If the architecture of the home network evolves into thin-client terminals, home gateways are going to be a gold mine for DSP applications. DSP is required on all access points, DSP is required for voice compression, DSP can be required to do MP3 decompression (Internet radio), MPEG4 decompression (Internet clips), etc. The limit is one's wallet.

## 9.6 Computer, Peripherals, and Office Automation

### 9.6.1 PC as a Home Gateway

Obviously, the home gateway market is not leaving the PC industry passive, especially after 10 years of multimedia hype. The more likely scenario will NOT see any major integration of "telecom personal systems" into PC; however, the home gateway is having several major impacts on PC: integration of wireless functions, still more performance for multimedia (such as the typical MPEG4 clip already mentioned).

### 9.6.2 Printers

The benefit of a laser printer and color printer depends largely on the speed and quality of image processing. This is typical DSP task.

### 9.6.3 Hard Disk Drive

In the last years, disk drives know-how has changed from complex control theory to sophisticated DSP coding techniques. The algorithms used have more to do with wireless telecommunications than servo control. In addition, the emergence of network attached storage requires communications, which in turn means DSP.

## 9.7 Automotive, Industrial

Although it is often forgotten when discussing advanced digital developments, the automotive industry could be the surprise of the decade for the DSP industry.

### 9.7.1 Engine Control

Due to history and real-time constraints, the automotive industry uses the principle of table interpolation for engine control; however, the availability of faster CPUs and the development of sophisticated algorithms could change that in favor of more "classical" DSP techniques. In fact, automotive could become the first embedded mass-market where floating-point DSP is implemented.

### 9.7.2 Navigation Platform

GPS/navigation: In automotive, GPS is part of the dashboard platform. How big will this market be?

### 9.7.3 Industrial

A large application found in the industrial market segment is motor control. Quite a very different control from car engine control, the two applications are strongly related since they are the domain of microcontrollers. Identically to engine contol above, motor control is fast becoming a big DSP application.

## 9.8 Others

To finish, small many promising applications using DSP as their bases for new or more advanced features include:

*White appliances*: Refrigerators, washing machines, or any equipment requiring closed control will eventually be heavy DSP users.

*Biomedical*: A good example is the processing of image in medical equipment such as scanner.

*Audio aids*: This is a much larger application than previously thought. Gene FRANTZ [2] made a parallel between visual aids (glasses) and audio aids. Let us imagine the size of the market if everybody was wearing a hearing aid to cancel noise and unwanted conversation.

*Biometrics*: All recognition methods (fingerprint, retina, voice, etc.) rely on DSP.

## 9.9 Conclusions: General Trends

The time when a single application was driving DSP is finished. The next DSP application goal is now several thousands of DSP MIPS, and many applications are driving it:

- Smart and multiple antennae techniques in wireless base stations
- Third generation cellular wireless phones, smart-phones, and terminals
- Broadband access devices (VDSL modem, wireless broadband, gigabit Ethernet)
- Multi-channel application of the telecommunication infrastructure (typically: voice-over-broadband gateway).
- Multimedia home gateways, integrated access device (IAD), wireless Home/LAN access devices
- Streaming media devices (could be a MPEG4 player connected by Bluetooth to a home gateway)
- HDTV, high resolution cameras, 3D audio

Finally, even if no "broadband" applications existed, people would use DSP for cost reasons. When a very good sensor is needed, an imperfect sensor is a worthless commodity. By using DSP techniques

(interpolation, adaptive behavior, etc.) a worthless commodity can be turned into a production device. The author is eager to see the day where a 30-inch 2000 × 4000 color LCD matrix with 80% defects will be turned into a $100 HDTV screen. Only DSP can achieve that.

## References

Except for Will Strauss's unique (and expensive) monumental work, it is difficult to give a complete reference covering all those applications. Personal experience and key people from various conferences [3] were an invaluable tool. The author's recommendation is to go to the specialized electronics Web sites [4–12] and to type the keyword (e.g., e-book) in the search engine. Next, go to the Web sites of the principal DSP manufacturers [14–22] and look for products, white papers, and applications. The reader is encouraged to look into companies not so often associated with DSP, or smaller companies where the most innovative designs are found [23–45].

Finally, the author recommends the two INFINEON sites corresponding to the DSP chips on which he working or has recently worked: TriCore [46] and Carmel [47].

1. Strauss, Will (Aug., 1998) DSP strategies 2002—A study of markets driven by Digital Signal Processing Technology, Forward Concept. Website: www.forwardconcepts.com.
2. Frantz, Gene (October 31, 2000) Techonline—Online Symphosium for Electronic Engineers—Digital logic Design—SOC: A system perspective. www.techonline.com.
3. ICSPAT 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000.
4. EE TIMES: www.eetimes.com.
5. ICD magazine: icd.pennnet.com.
6. Newsletters from INSTAT: www.instat.com.
7. CHIP Center—electronics experts: www.chipcenter.com/eexpert.
8. EDTN network: www.edtn.com.
9. EDN magazine: www.ednmag.com/.
10. ELECTONIC DESIGN magazine: www.elecdesign.com.
11. ELECTRONIC NEWS online: http://www.electronicnews.com.
12. VISION MAGAZINE: www.ce.org/vision_magazine/.
13. ANALOG DEVICES: www.analog.com/industry/Industry_Solutions.html.
14. INTEL: http://developer.intel.com/platforms/.
15. IBM: www.chips.ibm.com.
16. INFINEON: www.infineon.com.
17. LUCENT/AGERE: www.lucent.com/micro.
18. MOTOROLA: http://e-www.motorola.com/solutions/index.html.
19. PHILIPS: www.semiconductors.philips.com.
20. STARCORE: www.starcore-dsp.com.
21. TEXAS: www.ti.com/sc/docs/innovate/index.htm.
22. TEXAS dsp village: http://dspvillage.ti.com/.
23. ALTERA: www.altera.com/html/products/products.html.
24. AMCC: www.amcc.com.
25. ATMEL: www.atmel.com.
26. ARM: www.arm.com.
27. BOPS: www.bopsnet.com.
28. C-CUBE: www.ccube.com/.
29. CIRRUS: www.cirrus.com.
30. CONEXANT: www.conexant.com/home.asp.
31. EQUATOR: www.equator.com.
32. IDT: www.idt.com/.
33. JACOBS PINEDA: www.jacobspineda.com.

34. LSI logic: www.lsilogic.com.
35. METALINK: www.metalink.co.il.
36. MITEL Semiconductor: www.semicon.mitel.com.
37. MORPHICS Technology: www.morphics.com.
38. NS: www.national.com/.
39. OAK TECHNOLOGY: www.oaktech.com.
40. PMC-SIERRA: www.pmc-sierra.com.
41. QUICKLOGIC: www.quicklogic.com.
42. SHARP: www.sharpmeg.com.
43. TERALOGIC: www.teralogic-inc.com.
44. VIRATA: www.virata.com.
45. XILINK: www.xilink.com.
46. INFINEON Universal Processor—TriCore: www.infineon.com/us/micro/tricore/.
47. INFINEON Carmel DSP: www.carmeldsp.com.

# 10

# Digital Filter Design




Worayot Lertniphonphun
James H. McClellan
*Georgia Institute of Technology*


## 10.1 Introduction

For computer and information technology (IT) applications, signal processing is an important tool. Nowadays, it is much more efficient and accurate to work with sampled (or digitized) signals rather than with analog (or electrical) signals. Once a signal has been sampled, it can be treated as a sequence of numbers that is a function of a discrete-time variable. When the sampling rate is greater than the Nyquist rate, the digital signal will completely represent the analog signal, because the analog signal can be reconstructed from the digital signal. Digital signal processing (DSP) implements various kinds of mathematical operations, so that physical electrical devices are replaced by computer software or hardware. Unlike analog systems, DSP can handle very sophisticated jobs with as much accuracy as needed. The theory of DSP can be found in three excellent references [1–3].

One very basic DSP operation is digital filtering. It is common to use many filters inside a larger DSP application. Digital filters have widely been used in the following applications:

- *Audio*: spectral shaping
- *Speech*: filter banks
- *Image*: de-blurring, edge-enhancement/detection
- *Communications*: bandpass filters
- *Radar*: matched filters

## 10.2 Digital Filters

The theory of digital filters can be found in references [4–6]. A digital filter is defined as a linear, time invariant operator on a discrete-time input signal, $x[n]$, that generates an output signal, $y[n]$. The filtering operation can always be written as a convolution

$$a[n]*y[n] = b[n]*x[n] \text{ (convolution)}$$

where $b[n]$ and $a[n]$ are the filter coefficients associated with the digital filter.

### 10.2.1 Implementation

In order to implement the filter as a causal operation, the number of filter coefficients must be finite and the coefficients should be nonzero for only positive indices. Then the output signal can be computed via the difference equation:

$$y[n] = \sum_{k=0}^{M} b[k]x[n-k] - \sum_{k=1}^{N} a[k]y[n-k]$$

where $N$ and $M$ are the number of poles and the number of zeros, respectively, and $N+M$ is the total order of the filter. If any one of the feedback coefficients $a[k]$ is nonzero for $k>0$, then the filter is called a recursive or infinite impulse response (IIR) filter. Otherwise, the filter is called a nonrecursive or finite impulse response (FIR) filter.

### 10.2.2 Frequency Response

The filtering process for linear, time-invariant (LTI) systems can be characterized by the frequency response

$$H(\omega) = \sum_{n} h[n]e^{-j\omega n}$$

which shows how the filter processes sinusoidal inputs. The discrete-time Fourier transform (DTFT) decomposes a general input signal as a superposition of harmonic signals,

$$x[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(\omega)e^{j\omega n}d\omega,$$

where the complex amplitudes of those harmonic signals are computed by the DTFT sum:

$$X(\omega) = \sum_{n} x[n]e^{-j\omega n}.$$

Then the behavior of the system can be described as a multiplication in the frequency domain:

$$Y(\omega) = H(\omega)X(\omega)$$

where $Y(\omega) = H(\omega)X(\omega)$ is the DTFT of the output. In terms of the filter coefficients we get

$$Y(\omega) = \frac{B(\omega)}{A(\omega)}X(\omega)$$

where $A(\omega)$, $B(\omega)$, $X(\omega)$, and $Y(\omega)$ are the DTFT of $a[n]$, $b[n]$, $x[n]$, and $y[n]$, respectively. The difference between FIR and IIR filters can be summarized as follows:

**FIR filter:** An FIR filter has $A(\omega) = 1$, so its frequency response is formed as a linear combination of complex exponential functions that is equivalent to a polynomial. Hence, the design problem can be formulated on a linear vector space and very efficient mathematical optimization methods are available for approximating the desired frequency response. The design methods are simple, and often guarantee convergence to an optimal solution. Finally, since FIR filters do not have feedback they do not suffer stability and sensitivity problems.

**IIR filter:** In contrast to the FIR case, IIR filters are rational functions, so the design problem is inherently nonlinear. No elegant mathematical method can guarantee convergence to the global optimum. In addition to the difficulty of numerical design, IIR filters might exhibit instabilities where a finite input can generate infinite output and high sensitivity, and where roundoff noise can be amplified; however, IIR filter design, has more design freedom, so IIR filters can have the same performance as FIR filters but with many fewer filter coefficients.

### 10.2.3 FFT Implementation

It is possible to implement a digital filter in the frequency domain with the fast Fourier transform (FFT) algorithm [7]. The implementation requires one FFT of the input signal, one multiplication of vectors, and one inverse FFT. The length of the FFT determines a block length so the signal must be segmented into sections for both the input and output. The frequency domain implementation actually uses circular convolution, so some care is needed to get the correct outputs. The FFT-based method of convolution is used in special circumstances because it is only practical for real-time systems when the FIR filter length is rather long—the major drawback is that it requires a large amount of buffer memory for the block processing.

### 10.2.4 Adaptive and Time-Varying Filters

Another important class of FIR filters is the class of adaptive filters [8], which find widespread application in areas such as equalizers for communication channels. The filter coefficients in an adaptive filter are continually changing as the input changes, so the filter design problem is quite different for these filters. The methods discussed in this chapter will not handle these cases where the coefficients are time varying.

## 10.3 Digital Filter Design Problem

### 10.3.1 Design Specification

A digital filter is usually designed so that its output has a desired frequency content, i.e., the frequency response is frequency selective. The filter coefficients are then optimized so that the frequency response $H(\omega)$ will best approximate an ideal frequency response $I(\omega)$. The ideal response varies for different applications.

**Frequency selective filter:** The ideal frequency response is either one or zero.

$$(\omega) = \begin{cases} 1, & \omega \text{ in the pass band} \\ 0, & \omega \text{ in the stop bands} \\ \text{don't care}, & \omega \text{ in the transition bands} \end{cases}$$

The frequency selective filter is designed so that the actual frequency response $H(\omega)$ is close to 1 in the passband and nearly 0 in the stopband. An example of a frequency selective filter is shown in Fig. 10.1.

**Equalizer:** Equalizers are applied to existing systems in order to remove distortion, or to improve the overall filter characteristic. Therefore, if the desired response of the system is $D(\omega)$, the ideal frequency response of the equalizer depends on the distortion filter $H_D(\omega)$, such that

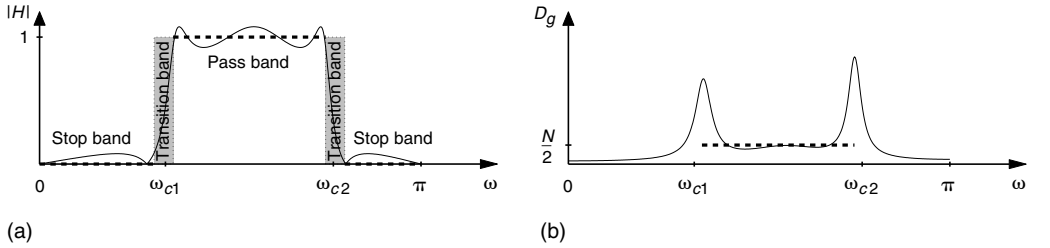



**FIGURE 10.1** Frequency selective (bandpass) filter: (a) shows the ideal magnitude response (thick dashed line) and an example of an elliptic (with 6 poles, 6 zeros) bandpass filter (thin solid line). The ideal filter has two cutoff frequencies, $\omega_{c1}$ and $\omega_{c2}$, that separate the two stop bands from the pass band; (b) shows the ideal group delay response (thick dashed line) and the group delay of the elliptic bandpass filter. Note that elliptic filters usually have severe phase distortion (i.e., a highly nonlinear group delay) in the passband.

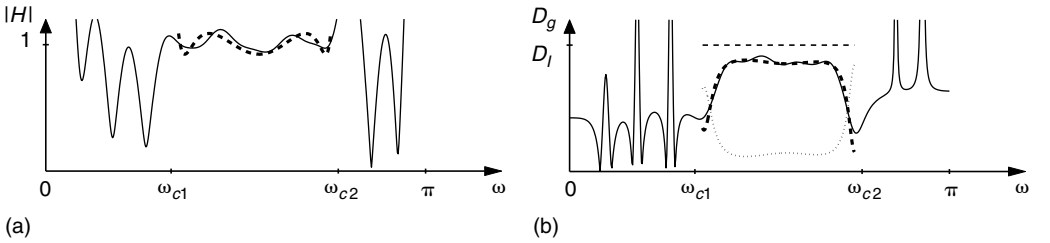



**FIGURE 10.2** The phase equalizer is designed to equalize the passband of the elliptic filter in Fig. 10.1 so that the group delay is flat in the pass band: (a) shows the ideal equalizer response (thick dashed line) and a FIR (order 25) equalizer; (b) shows the group delay, $D_g\{\bullet\}$, where $D_g\{I_{\text{Equal.}}(\omega)\} = D_g\{I(\omega)\} - D_g\{H_{\text{Elliptic}}(\omega)\}$ is the ideal group delay (thick dashed line).

$$I_{\text{Eq}}(\omega) = \frac{D(\omega)}{H_D(\omega)}$$

One example is shown in Fig. 10.2 where the equalizer is used to reduce the phase distortion of the filter in Fig. 10.1. The phase equalized filter is shown in Fig. 10.3.

**Filter bank:** A filter bank is a set of filters that sum to 1, the identity system:

$$I_k(\omega), \text{ for } k = 1, \ldots, P \quad \text{such that} \sum_{k=1}^{p} I_k(\omega) = 1$$

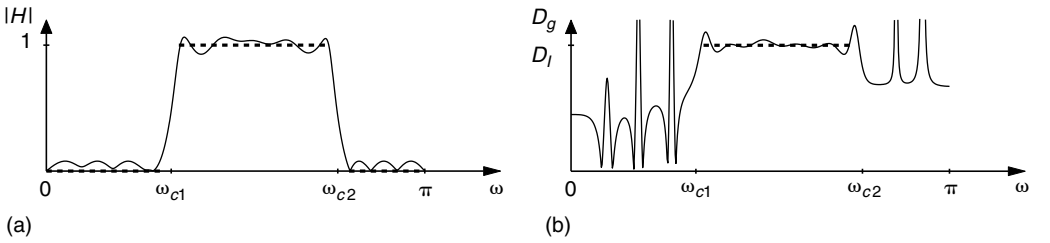



**FIGURE 10.3** Equalized filter. The figures show the magnitude and the group delay of the elliptic filter (with 6 poles, 6 zeros) after being equalized by the FIR (order 25) equalizer. The ideal filter for the example is a flat group delay frequency selective filter.

With actual filters, the sum might be approximately one. This property lets us decompose signals with a filter bank and then reconstruct perfectly. Filter banks are now widely used as analysis, storage, and compression tools for DSP.

**Differentiator:** The derivative operation is a filter whose ideal frequency response is

$$I_{\text{Diff.}}(\omega) = j\omega$$

Operators such as the first difference make poor filters because they do not work well for high-frequency signals. Filter design, however, can create high-order numerical differentiators that have excellent wide-band characteristics by approximating the ideal frequency response, $I_{\text{Diff.}}(\omega) = j\omega$.

## 10.3.2 Error Measurement

In order to have a filter whose frequency response is very close to a given ideal response, a norm for error measurement must be introduced. Then the filter design problem becomes a mathematical optimization problem. Many possible error norms can be used. For example, the most popular norms are:

- Maximal magnitude error, $\max\|I| - |H\|$, for a frequency-selective filter
- Maximal phase error, $\max |\angle I - \angle H|$, for an allpass filter
- Weighted complex error, $\|W(I - H)\|$, for general filters where the function $W(\omega)$ is a positive weight function

The design problem is usually carried out by minimizing one of these norms, but it is also possible to add constraints on the error magnitude, on the pole locations, the transition band overshoot, the smoothness of the error, or the magnitude of the filter coefficients. These various criteria lead to many different filter design methods that offer trade-offs with respect to efficiency and flexibility.

## 10.3.3 Filter Characteristics

Although many filter design papers and procedures have been published, only a few approaches have found widespread use in the 35-year history of DSP.

### 10.3.3.1 Optimal Magnitude Response

IIR filters with optimal magnitude error are generally easy to design partially because they usually require low order; however, these IIR filters usually have severe phase distortion that, in turn, limits the filter's application to cases such as audio where phase does not seem to be important.

### 10.3.3.2 Allpass Filters

The phase distortion of an optimal magnitude IIR filter is sometimes compensated by using an allpass equalizer, where the numerator and denominator of $H(\omega)$ have the same order, $M = N$, and the filter coefficients satisfy $a[k] = b^*[N - k]$. The allpass equalizer, however, is usually not an efficient way to implement filtering, because the equalizer usually has very high order compared to the original filter. This not only causes the filtering to become inefficient, but also causes a long delay in the output signal. Allpass filters can also be used for frequency selective filter design if a pair of allpass filters are connected in parallel. Details of this clever allpass design method can be found in [9–13].

### 10.3.3.3 Filters Designed by Optimizing a General Weighted Norm

In the most general case, filter design can be treated as the process of approximating a complex-valued function $H(\omega)$, where the filter coefficients are the approximating parameters. This treatment gives the filter design problem more degrees of freedom in choosing the ideal response because both magnitude and phase can be approximated. Figure 10.4 shows an example using the general weighted norm. The filter has a much better response than the filter in Fig. 10.3 with the same order as summarized in Table 10.1.

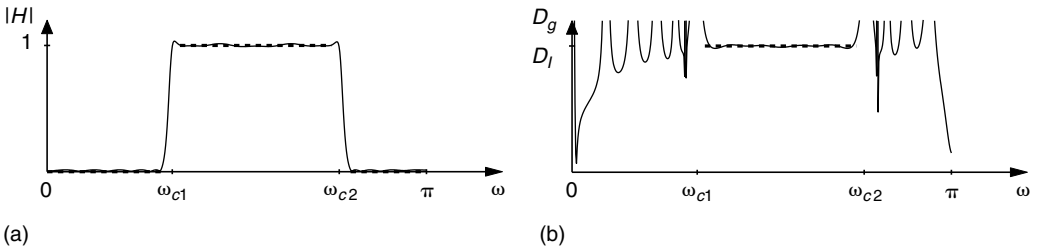



**FIGURE 10.4** Filter with optimal general weighted norm. The figures show the magnitude and the group delay of an IIR frequency selective filter (31 zeros, 6 poles) with flat delay passband. The ideal filter is the same as in Fig. 10.3, and the filter order is the same as the equalized elliptic filter.

**TABLE 10.1** Filter Design Comparison

| Filter | # Zeros | # Poles | Mag. Error | GD. Error | RMS GD. Error |
|---|---|---|---|---|---|
| Elliptic (Fig. 10.1) | 6 | 6 | 0.082 | 13.90 | 3.57 |
| Equalized elliptic (Fig. 10.3) | 31 | 6 | 0.079 | 3.15 | 0.76 |
| IIR (Fig. 10.4) | 31 | 6 | 0.015 | 2.40 | 0.32 |

*Note*: The table shows three features: maximal magnitude error, maximal group delay error, and RMSs of the group delay error, of the optimal response of the filter in Figs. 10.3 and 10.4 designed under different approaches.

Several optimization techniques are available to solve these general problems. In addition, the error can be controlled by the selection of an error constraint, an error weight, or a design norm; however, the optimization of general norms is often a difficult problem, especially in the complex domain. Most recent research has studied these general norm problems in order to improve the design when the goal is a simultaneous approximation of the magnitude and phase.

## 10.3.4 Filter Design as a Norm Problem

Filter design is usually done by minimizing either the worst-case error (Chebyshev norm), or the root mean squares (RMS) (least-squares norm) of the weighted error. Important norms from classical mathematics are listed below:

- Chebyshev norm: $\|E\|_\infty = \max_\omega |E(\omega)|$
- Least-squares norm: $\|E\|_2 = \left\{\int_{-\pi}^{\pi} |E(\omega)|^2 d\omega\right\}^{1/2}$
- $p$-norm: $\|E\|_p = \left\{\int_{-\pi}^{\pi} |E(\omega)|^p d\omega\right\}^{1/p}$ for $p \in [1,\infty]$
- Combined norm: $\|E\|_\alpha = \left\{\alpha\|E\|_\infty^2 + (1-\alpha)\|E\|_2^2\right\}$ for $\alpha \in [0,1]$

where $E = W(I - H)$ is the weighted complex error. When optimizing the Chebyshev norm, the resulting optimal filters have the smallest maximal error, while filters with minimal least-squares norm have the smallest RMS error. Preference for one norm over the other will generally depend on the application. In many cases, where both norms need to be small, filters should be designed under either the $p$-norm or the combined norm. Along with the norm, the numerical optimization can be done under design constraints, e.g., the most obvious one is a constraint on the magnitude of the error

$$\min \|E\| \text{ such that } |E(\omega)| < \varepsilon(\omega)$$

where $\varepsilon(\omega)$ is the error constraint.

Figure 10.5 shows the error of the filter with the same specification designed under four different norms. The RMS and maximal errors are summarized in Table 10.2.

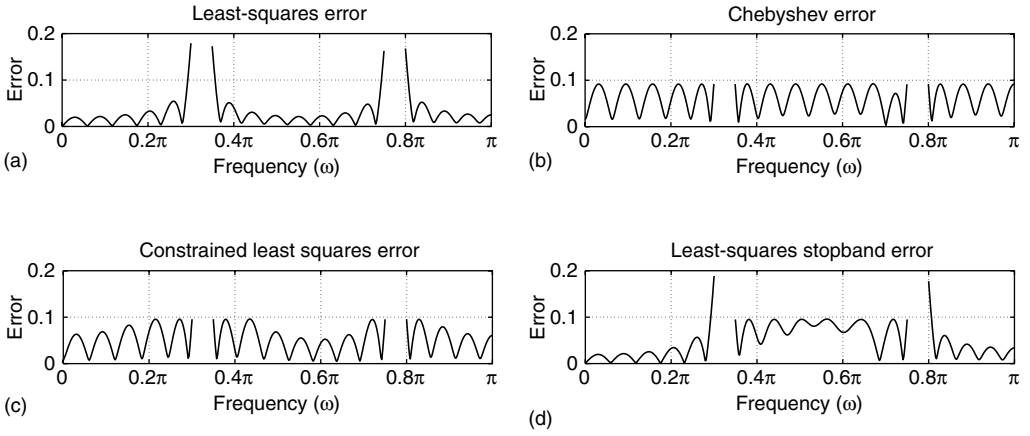



**FIGURE 10.5** Different error norms. The four filters were designed to approximate the same bandpass filter of order 25 with four different norms. The filter in (c) was designed by minimizing the least-squares norm under the constraint that the maximal error be smaller than 0.0959. Note that the filter in (c) can also be designed by minimizing the unconstrained combined norm problem with the norm weighting $\alpha = 0.4$. The filter can also be designed so that both the distortion in the pass band and the power of the stopband error are small. The filter in (d) was designed by optimizing the combination of the Chebyshev error norm of the passband plus the least-squares norm of the stopband.

**TABLE 10.2** Error Measurements for Fig. 10.5

| Filter | RMS Error | Maximal Error in Passband | RMS Error in Stopband |
|---|---|---|---|
| (a) Least-squares | 0.0361 | 0.1793 | 0.0348 |
| (b) Chebyshev | 0.0657 | 0.0923 | 0.0660 |
| (c) Constrained least-squares | 0.0562 | 0.0959 | 0.0571 |
| (d) Least-squares stopband | 0.0584 | 0.0958 | 0.0374 |

*Note*: These are the passband and stopband errors in bandpass filters designed by a different norm problem.

The norm optimization problem differs quite a bit for the FIR and IIR cases:

**FIR filter:** The problem is formed on a linear vector space and has been well studied. The optimal solution is unique by convexity. Many available design methods are not only elegant, but are also computationally efficient and have guaranteed convergence.

**IIR filter:** Although the IIR filter design problem does not have the same nice properties as the FIR filter design problem, optimizing the norm is relatively easy. One iterative approach to IIR filter design relies on a sub-procedure similar to the method for FIR filter design.

## 10.4 Conventional Design Methods

Although many filter design papers have been published in the 35 years of DSP, only a handful of filter design methods are widely used. Some of the older conventional methods can design filters with excellent magnitude response using a very simple procedure, but the variety of possible filter specifications and error norms are usually limited. More recent methods offer general design capabilities for both magnitude and phase approximation, but are based on numerical optimization.

### 10.4.1 IIR Filters from Analog Filters

Originally digital filters were derived from analog filters [14] because analog filter design techniques had been studied for a long time and the design usually involved algebraic formulas that were simple to carry out. The two main design methods are impulse-invariance and the bilinear transformation.

#### 10.4.1.1 Impulse Invariance

The design is carried out by starting with an already designed analog filter that is bandlimited. Let $h_a(t)$ denote the impulse response of the analog filter. Then the impulse response of the digital filter is obtained by sampling, i.e., by setting $h[n] = T_d h_a(nT_d)$; however, no analog filter is truly bandlimited, so the actual frequency response involves some aliasing:

$$H(\omega) = \sum_{k=-\infty}^{\infty} H_a\left(j\frac{\omega}{T_d} + j\frac{2\pi}{T_d}k\right)$$

where $H_a(s)$ is the Laplace transform system function of the analog filter. The aliasing effect usually causes only a slight perturbation of the digital filter with respect to the analog filter. The system function of the analog filter can be expressed in partial fraction form

$$H_a(s) = \sum_{k=1}^{N} \frac{A_k}{s - s_k}$$

After sampling the digital filter has a frequency response that is also a rational form:

$$\text{DTFT: } H(\omega) = \sum_{k=1}^{N} \frac{T_d A_k}{1 - e^{T_d S_k} e^{-j\omega}} = \frac{\sum_{k=0}^{N-1} b[n] e^{-j\omega n}}{1 + \sum_{k=1}^{N} a[n] e^{-j\omega n}}$$

where $b[n]$ and $a[n]$ are the coefficients of the designed filter. Impulse invariance is equivalent to a linear mapping of the analog frequency range $[-\pi/T_d, \pi/T_d]$ into the digital frequency range $[-\pi, \pi]$.

#### 10.4.1.2 Bilinear Transform

On the other hand, the bilinear transformation performs a nonlinear mapping of the whole analog frequency range $[-\infty, \infty]$ into the finite digital frequency range $[-\pi, \pi]$. The mapping of the $s$-plane to the $z$-plane is done by the bilinear transform:

$$s = \frac{2}{T_d}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right)$$

The resulting correspondence between the analog and digital frequency domains is a tangent function:

$$\omega_a = \frac{2}{T_d} \tan\left(\frac{\omega}{2}\right)$$

Despite the nonlinear nature of the mapping, it is relatively easy to turn the digital design specification into an analog design specification. The resulting filter is IIR and the filter coefficients can be computed with an algebraic form. The bilinear transform method is usually applied to four classical analog filter frequency selective filters: Butterworth, Chebyshev-I, Chebyshev-II, and elliptic filters. All these are well known for their frequency-selective behavior as lowpass, bandpass, or highpass filters. When using the bilinear mapping, elliptic IIR filters turn out to have the best magnitude response for given filter order, but elliptic filters have severe phase distortion, which can be a significant problem in advanced DSP applications such as telecommunications.

### 10.4.2 Windowing

IIR filter designs have poor phase response, so interest in FIR filters has always been strong. If the coefficients of an FIR filter are real and symmetric $b[k] = b^*[M - k]$ then the filter will have perfectly

linear phase. The first attempt to design FIR filters in the 1960s was to truncate the inverse DTFT of the ideal frequency response (which is the impulse response $h[n]$ of the ideal filter), so that the filter is symmetric and linear-phase. This requires the ideal filter to have linear-phase with slope $-\frac{1}{2}M$, where $M$ is the FIR filter order. This method of filter design turns out to give the optimal least-squares filter. However, the least-squares filter is not an acceptable filter, especially when the application calls for a frequency selective filter. The reason is that the least-squares approximation exhibits an overshoot called the *Gibbs' phenomenon*, which means that the magnitude of the error is large at the cutoff frequency regardless of the filter order. To reduce the magnitude error near the cutoff frequency, the strict truncation (done by applying a rectangular window) can be replaced by other windowing. Windowing for filter design involves the multiplication of a finite-length window shape times the ideal impulse response. For example, the ideal lowpass filter with delay $\mu = \frac{1}{2}M$ has an impulse response that is infinitely long:

$$h[n] = \frac{\sin(\omega_c(n-\mu))}{\pi(n-\mu)}, \quad -\infty < n < \infty$$

so the windowed filter coefficients are $b[n] = w[n]h[n]$ for $n = 0, 1, 2, \ldots, M$.

Different windows generate filter responses that allow a trade-off between the sharpness of transition region and the error magnitude. Popular windows are: Bartlett, Hamming, vonHann (or Hanning), and Kaiser, but for filter design the only important one is the Kaiser window, which is based on the modified Bessel function. The Kaiser window is defined as

$$w[n] = \frac{I_0\left(\beta\sqrt{1-(n-\mu)^2/\mu^2}\right)}{I_0(\beta)}, \quad n = 0, 1, 2, \ldots, M$$

where $I_0(x)$ is the modified Bessel function, and the parameter $\beta$ is chosen to control the ripple height in the stopband with the relationship:

$$\beta = \begin{cases} 0, & \delta\text{dB} < 21 \\ 0.5842(\delta_{\text{dB}} - 21)^{0.4} + 0.07886(\delta_{\text{dB}} - 21), & 21 < \delta\text{dB} < 50 \\ 0.1102(\delta_{\text{dB}} - 8.7), & \delta\text{dB} > 50 \end{cases}$$

where $\delta_{\text{dB}} = -20\ \log_{10}\ (\delta_{\text{stopband}})$ is the ripple height in dB. The design of the Kaiser window is illustrated in Fig. 10.6. Examples of digital filters designed via windowing are shown in Fig. 10.7.

#### 10.4.2.1 Frequency Sampling

Another common, but naive, approach to FIR design is the method of frequency sampling. In this case, the ideal frequency response is sampled over the range $-\pi < \omega \leq \pi$ at $M+1$ points and then the inverse FFT is computed to get the order-$M$ impulse response, which then contains the coefficients of the FIR filter. It is possible to let a few of the frequency samples be free parameters for a linear program that will optimize the resultant $H(\omega)$. This, in turn, improves the filter characteristics by making the error smaller near the cutoff frequency.

### 10.4.3 Weighted Least-Squares

Although frequency sampling filters and windowing designs have pretty good responses, neither one is an optimal filter. In the general optimization approach, the transition band of the frequency response should be treated as a "don't care" region. For common frequency selective filters, the optimal filter will have a smooth behavior in the transition band even though no optimization is done in that "don't care" region. The FIR filter can be designed by minimizing any norm with a guaranteed unique solution.

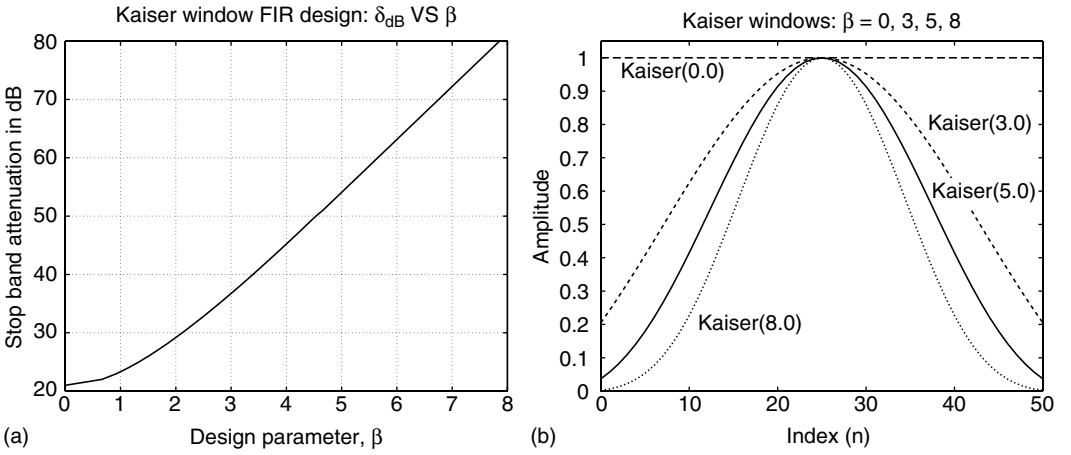



**FIGURE 10.6** The Kaiser window: (a) shows the relationship between $\beta$ and the ripple height in the stop band; (b) shows examples of length-51 Kaiser windows (i.e., filter order $=$ 50) with different parameters $\beta$. Note that, with $=$ 0, the window is the rectangular window and, with $\beta = 5$, the window is very similar to the Hamming window.

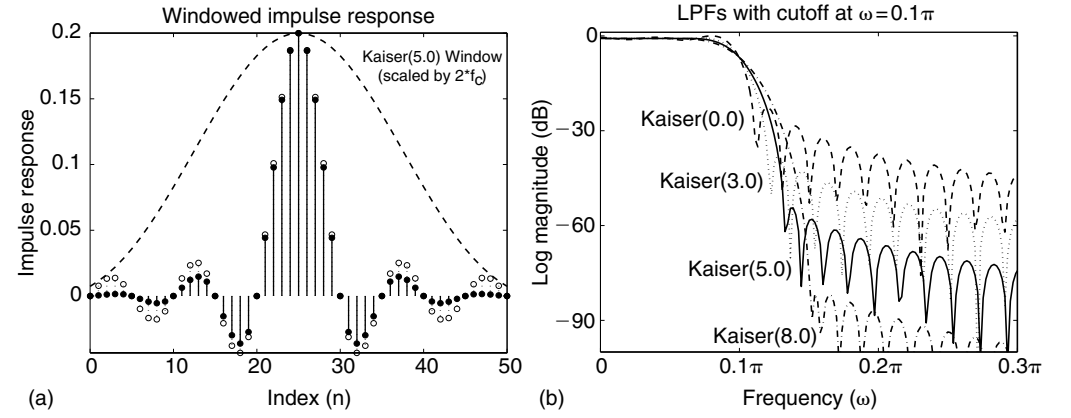



**FIGURE 10.7** Digital filter design via Kaiser windowing: (a) shows the impulse response of an ideal lowpass filter (circles with dotted lines) and the filter designed by windowing (filled circles with solid lines). The windowed filter is the product of the ideal impulse response and the Kaiser window with $\beta = 5$ (dashed line); (b) shows the log magnitude of four filters designed using the Kaiser window with different parameters $\beta$.

The design can be generalized further by using a weighting function on the error. For example, the weight can be used in clever ways to control the error. Here is the weight definition for an inverse filter (or equalizer).

$$I_{\mathrm{Eq}}(\omega) = \frac{D(\omega)}{H_{\mathrm{Sys}}(\omega)}, \quad W_{\mathrm{Eq}}(\omega) = \left|H_{\mathrm{Sys}}(\omega)\right| W(\omega)$$

The weighted design problem usually involves optimizing the norm of the error over the entire frequency domain, but that is done numerically by working on a dense frequency grid.

The easiest optimization problem is the least-squares norm minimization because the partial derivatives (which are the elements of the gradient) of the least-squares norm with respect to the filter coefficients are all linear combinations of the filter coefficients. This property implies that the optimal

filter can be found by solving the set of linear equations obtained by setting all those partial derivatives to zero. The solution for the weighted least-squares FIR filter is

$$\frac{\partial}{\partial b[n]} \int_{\omega} |W(I-H)|^2 d\omega = 0, \quad \text{for} \quad n = 0, 1, \ldots, M$$

$$\int_{\omega} |W|^2 \left( Ie^{j\omega n} - \sum_k b[k] e^{j\omega(n-k)} \right) d\omega = 0, \quad \text{for} \quad n = 0, 1, \ldots, M$$

For IIR filters, the problem is not nearly so easy because the denominator of the frequency response function makes the problem nonlinear. The solution can still be carried out by computing the partial derivatives and setting them equal to zero:

$$\int_{\omega} |W|^2 \left( \frac{e^{j\omega n}}{A^*} \left( I - \frac{B}{A} \right) \right) d\omega = 0, \quad \text{for } n = 0, 1, \ldots, M$$

$$\int_{\omega} |W|^2 \left( \frac{e^{j\omega n}}{A^*} \frac{B^*}{A^*} \left( I - \frac{B}{A} \right) \right) d\omega = 0, \quad \text{for } n = 0, 1, \ldots, N$$

The solution may not exist, however, but even if it does, it is often not unique. Furthermore, it is likely that only a locally optimal solution of the nonlinear equations can be found. Another approach is to use an iteration to find a close-to-optimal solution using the Steiglitz–McBride method [15]:

$$\min \left| W \left( I - \frac{B}{A} \right) \right| \rightarrow \min \left| \frac{W}{|A|} (IA - B) \right|$$

The solution can be realized by iteratively updating the rational functions $W/|A|$ and $B/A$.

### 10.4.4 Remez Exchange

Least-squares filters are not desirable in many applications because they exhibit large worst-case error near the transition band. On the other hand, the worst-case error can be minimized by reformulating the design problem as a Chebyshev (or min-max) problem.

$$\min_{b[n],a[n]} \max_{\omega} |W(I-H)|$$

This min-max problem is usually difficult to solve unless the problem can be transformed into a real problem. To do this, the ideal filter needs to be a linear-phase filter with a group delay of $\frac{1}{2}M$. Then the problem becomes an approximation of a real function by a sum of sinusoidal functions. For the special case of an even-order FIR filter with symmetric coefficients, the real problem becomes:

$$\min_{c_k} \max_{\omega} \left| W(\omega) \left( I(\omega) e^{j\omega M/2} - \sum_{k=0}^{M/2} c_k \cos \omega k \right) \right|$$

where $a = [1]$ and $b = \left[\frac{1}{2}c_{N/2}, \ \ldots, \ \frac{1}{2}c_1, \ c_0, \ \frac{1}{2}c_1, \ \ldots, \ \frac{1}{2}c_{N/2}\right]$. This min-max problem can be solved by the Remez algorithm [16–19]. The algorithm exploits the famous Alternation Theorem, which gives the necessary and sufficient condition for an optimal real Chebyshev solution as one that has at least $M+2$ alternating extremal points (i.e., points where the error is maximal). The operation of the algorithm involves an exchange that iteratively updates the extremal set and solves for the alternating error on that set. It turns out that the Remez Exchange algorithm is very efficient and always converges, so it has become a classical method for FIR filter design as the Parks–McClellan algorithm.

### 10.4.5 Linear Programming

Filter design by optimizing the norm of the weighted error can be further improved by applying constraints. However, only the magnitude constrained problem seems to be easy to solve

$$\min \|W(I-H)\| \quad \text{or} \quad \min \|E\|$$
$$\text{subject to} \quad |E(\omega)| < \varepsilon(\omega)$$

This constrained magnitude problem can be solved by Mathematical programming, which takes different forms depending on the norm. For the least-squares norm, the solution can be found by using quadratic programming.

Mathematical programming is also a tool for the nonlinear-phase Chebyshev problem [20], which can be rewritten as a constrained problem:

$$\min \delta$$
$$\text{subject to}$$
$$\mathcal{R}\{E(\omega)e^{j\theta}\} < \delta, \quad \text{for all } \omega \text{ and } \theta$$

This problem is a semi-infinite linear minimization (SILM). Linear programming can then be applied to the problem by sampling the parameters $\omega$ and $\theta$. The algorithm is not efficient for high-order filters because dense parameter sampling is needed to design filters with high precision. In order to improve efficiency, the SILM can be rewritten in a *dual form* [21–24].

## 10.5 Recent Design Methods

Because conventional design methods are available for only special types of digital filters, e.g., linear-phase, researchers have proposed various new methods that use complex approximation in filter design. Among those, only a few are discussed because they are elegant and useful in various applications.

### 10.5.1 Complex Remez Algorithm

The complex Chebyshev design problem is one of the most important approaches for designing digital filters. Unfortunately, it might need a general algorithm such as SILM, which requires a large number of frequency samples (with resulting high computation and high memory) when high precision is desired. For high order filters, linear programming is very inefficient for Chebyshev filter design. Instead, modifications of the Remez Exchange algorithm would be more desirable. Therefore, the complex Remez algorithm (CRemez) [25,26] was proposed using an exchange method search that is similar to the Remez Exchange; however, the original CRemez is most efficient only for the special case where the extremal error alternates. In general, nonlinear-phase filters are not guaranteed to have this strict alternating property, so the exchange method does not converge to the optimum. In order to get the optimum filter in the case of nonalternating extremal errors, a second stage is needed for CRemez. This second stage has to be a general optimization method that ends up being as inefficient as the SILM method. As a result, some filters are designed very quickly by CRemez, but others take a long time when the general optimization step must be invoked.

### 10.5.2 Constrained Least-Squares

Adams [27] suggested that Chebyshev digital filters do not always have the best overall characteristics. He found that by allowing the worst-case (Chebyshev) error to increase slightly, the least-squares error can be reduced significantly. To design this sort of filter, a constrained least-squares problem was

introduced. The problem has been solved by [28–31] with an algorithm that is quite efficient for designing FIR filters.

## 10.5.3 Generalized Remez Algorithm

The constrained least-squares methods have two design drawbacks: (1) error constraints are required to set up the problem, and (2) the existing methods only handle the FIR case. The first drawback is not severe, but it reduces the design efficiency because prior information such as a prior filter design procedure is needed to estimate the constraints; however, both drawbacks can be eliminated by using a different norm (called the combined norm) and by minimizing via the iterative reweighted least-squares (IRLS) technique.

### 10.5.3.1 IRLS Technique

Lawson [32] proposed that the Chebyshev problem be turned into a weighted least-squares (WLS) problem. In fact, any general norm problem can also be turned into a WLS problem

$$\text{IRLS: } \|E\| \rightarrow \|VE\|_2$$

The trick is to find the correct weight, $V$, which is an unknown that must be found by running an iterative update [32–34]. For example, the following iterations converge to the appropriate weight for the Chebyshev norm and the $p$-norm, respectively.

- Chebyshev update: $V^{(k+1)} = V^{(k)}\sqrt{|E|}$
- $p$-norm update: $V^{(k+1)} = \left\{V^{(k)}\sqrt{|E|}\right\}^{\frac{p-2}{p-1}}$

The convergence of the weight is dependent on the number of points in the frequency grid, so IRLS alone usually converges slowly.

## 10.5.4 Combined Norm

It can be shown that the solution of the combined norm problem is equivalent to a constrained least-squares problem. The solution has multiple extremals of the error similar to the Chebyshev solution. To solve the combined norm problem, the IRLS technique can be used after the problem is turned into a weighted least-squares problem:

$$\|E\|_\alpha \rightarrow \left\{\alpha\|VE\|_2^2 + (1-\alpha)\|E\|_2^2\right\}^{1/2}$$

By iterating on the weight, $V$, the solution generally converges quickly unless $\alpha$ is large. The optimization procedure can be improved by exploiting the multiple extremal error property of the optimal combined norm solution.

## 10.5.5 Generalized Remez Algorithm

The optimal filter design problem can be generalized further by considering the problem of minimizing the combined norm together with magnitude constraints. Using Lagrange multipliers, the problem can be turned into an unconstrained least-squares problem:

$$\min\left\{\alpha\|VE\|_2^2 + (1-\alpha)\|E\|_2^2 + \|\Lambda E\|_2^2\right\}^{1/2}$$

where $\Lambda$ is the Lagrange multiplier. This problem can be solved by the basic IRLS technique, but often converges slowly. On the other hand, it can be shown that the weight, $V$, and the multiplier, $\Lambda$, are nonzero for only a finite number of points and those points are extremal points and points where

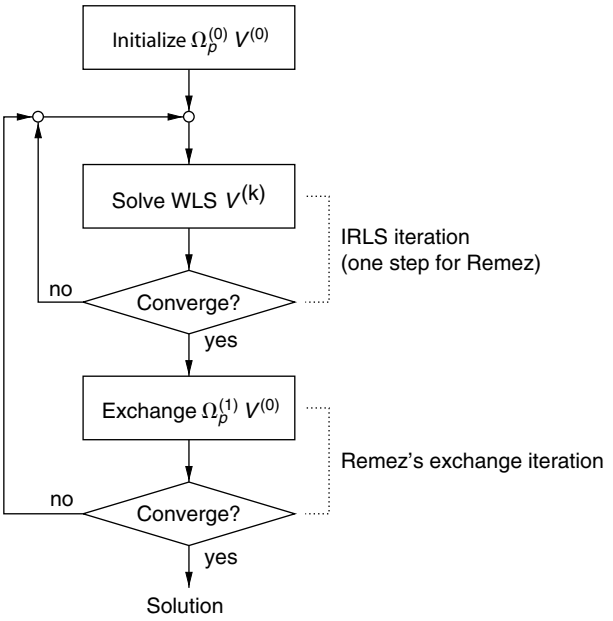



**FIGURE 10.8** Block diagram of the generalized Remez algorithm.

constraints are reached. So, the multiple exchange in the Remez algorithm can be used to find those points. After the points are found, the IRLS technique is applied to compute the filter coefficients. The solution will converge much more quickly than the classical IRLS because it deals with less frequency points. The authors call this new algorithm the "generalized Remez algorithm" or GRemez because its structure is equivalent to the Remez algorithm for linear phase FIR filter design [35]. Note that the GRemez algorithm is similar to a multiple exchange algorithm for the Chebyshev problem by Tseng [36]. The GRemez algorithm is summarized as a block diagram in Fig. 10.8.

Not only can the GRemez be used for the general constrained norm problem, but IIR filters can also be designed by the GRemez with the least-squares techniques presented in Section 10.4.3 to find either a close-to-optimal solution or a local optimal solution.

## 10.6 Summary

### 10.6.1 General Comment

This chapter has given a brief overview of various digital filter design methods. The theoretical ideas of frequency response and impulse response were reviewed in order to introduce the important design methods. In addition, the mathematical idea of optimizing with respect to a norm was discussed because many newer methods utilize numerical optimization to design general classes of filters. Readers are encouraged to find more details in the references that include many excellent DSP and digital filtering books [1–6].

Because digital filters must be designed for many diverse applications, a large number of design approaches are available; however, most filter applications can be addressed with an optimization algorithm when the general filter design problem is formed under the weighted norm of the complex error. This general problem can be difficult to solve in some cases. Fortunately, it is quite simple to design the most desirable filters, i.e., least-squares and Chebyshev FIR filters with linear-phase. Furthermore, filters with more general characteristics can be designed by methods presented in Section 10.5. We introduce the generalized Remez algorithm in the Section 10.4.3 in order to design both FIR and IIR filters under the weighted norm formulation.

Filter design methods presented in this paper are usually quite efficient, but some still require a fair amount of computation. For example, even though the least-squares method requires an amount of computation proportional to the cube of the filter order, $O(M^3)$, it is considered to be a relatively efficient design method using a norm minimization. Fortunately, the design time is hardly noticed on today's desktop computers, which have very fast processors.

Many other important issues in filter design were not treated here. These include filter order selection, filter pole location sensitivity, effect of implementing filter with fixed-point arithmetic, and multidimensional filter design [37–39]. Some details about these issues can be found in the references. In addition, the design of two-dimensional (2-D) filters has not been treated. Some of the optimization methods discussed here will also work for the 2-D case, but much of the theory of Chebyshev approximation no longer applies, so methods that exploit special features such as the Alternation Theorem will no longer be efficient.

### 10.6.2 Computer Tools

General filter design methods that are able to handle virtually all of the desired filter characteristics of common applications were discussed; however, most users would not want to be involved with details of programming the optimization algorithms. So, software applications have been built to help users skip the computer programming step and concentrate on entering the filter specifications for their application. Two types of interface for the design software are common: (1) allow users to set up the full design specification and (2) allow limited specifications.

The second type is generally implemented as a graphical user interface (GUI) that helps the user visualize all the steps of the design from creating the passband and stopband, setting the filter type (FIR or IIR), selecting the optimization tool, running the design program, and showing the designed filter responses. GUI software hides most of the design steps in the interest of simplicity, but it imposes a limitation on the amount of information that the user can see.

More advanced users probably need to design filters with more sophisticated specifications, more control on the error, or a wider variety of filter types. Therefore, they may need to enter the parameters manually in a command line to run the optimization function of the design algorithm. This normally requires some experience in the programming language and sometimes knowledge of the design program's source code.

One example of filter design programs can be found in the MATLAB© environment with its signal processing toolbox. Most of the design methods of filter design are available in the SP toolbox, and additional ones might be obtained by contracting the author who proposed the method. In MATLAB, the information for running the signal processing toolbox can be seen by typing "`help signal`." One attempt to make DSP simple to use is the MATLAB GUI program "`sptool`" that can upload and download signals, design and apply filters, and analyze the signal spectra. By pushing the button "`New Design`," the filter design GUI is called and the program gives the user an ability to design frequency selective filters by most of the conventional methods. Actually, the design GUI takes input parameters, changes them into the proper format, and then performs the design by calling design functions such as "`butter`," "`cheby1`," "`cheby2`," "`ellip`," "`firls`," and "`remez`." For more varieties of filter types, filters may be designed by calling "`cremez`," "`fircls`," etc., with appropriate parameters in the MATLAB command line. The authors have also developed the function "`gremez`" and "`gremez_gui`" as a MATLAB functions that can be downloaded from "`http://users.ece.gatech.edu/mcclella/gremez`."

## References

1. Oppenheim, A.V., Schafer, R.W., and Buck, J.A., *Discrete-time signal processing,* 2nd ed., Prentice-Hall, NJ, 1999.
2. McClellan, J.H., Schafer, R.W., and Yoder, M.A., *DSP first: a multimedia approach,* Prentice-Hall, NJ, 1998.

3. Proakis, J.G. and Manolakis D.G., *Digital signal processing: principles, algorithms, and applications,* Prentice-Hall, NJ, 1996.
4. Hamming, R.W., *Digital filters,* 3rd ed., Prentice-Hall, NJ, 1998.
5. Mersereau, R.M. and Smith, M.J.T., *Digital filtering: a computer laboratory textbook,* John Wiley & Sons, NY, 1994.
6. Parks, T.W. and Burrus, C.S., *Digital filter design,* John Wiley & Sons, NY, 1987.
7. Cooley, J.W. and Tukey J.W., An algorithm for the machine computation of complex Fourier series, *Math. Comput.,* 19, 297–301, April 1965.
8. Haykin, S.S., *Adaptive filter theory,* 3rd ed., Prentice-Hall, NJ, 1996.
9. Deczky, A.G., Recursive digital filter having equiripple group delay, *IEEE Trans. Circuits Syst.,* CAS-21, 131–134, Jan. 1974.
10. Deczky, A.G., Equiripple and minimax (Chebyshev) approximations for recursive digital filters, *IEEE Transactions on Acoust., Speech, Signal Processing,* ASSP-22, 98–111, April 1974.
11. Ikehara, M., Tanaka, H., and Kuroda, H., Design of IIR digital filters using allpass networks, *IEEE Trans. Circuits Syst. II,* 41, 231–235, March 1994.
12. Lang, M., Allpass filter design and applications, *IEEE Trans. Signal Processing,* 46, 2505–2513, Sep. 1998.
13. Zhang, X. and Iwakura, H., Design of IIR digital allpass filters based on eigenvalue problem, *IEEE Trans. Signal Processing,* 47, 554–559, Feb. 1999.
14. Weinberg, L., *Network analysis and synthesis,* R.E. Kreiger, Huntington, NY, 1975.
15. Steiglitz, K. and McBride, L.E., A technique for the identification of linear systems, *IEEE Trans. Automatic Control,* 10, 461–464, Oct. 1965.
16. Parks, T.W. and McClellan, J.H., Chebyshev approximation for nonrecursive digital filters with linear phase, *IEEE Trans. Circuit Theory,* CT-19, 189–194, March 1972.
17. McClellan, J.H. and Parks, T.W., A unified approach to the design of optimal FIR linear-phase digital filters, *IEEE Trans. Circuit Theory,* CT-20, 697–701, Nov. 1973.
18. McClellan, J.H., Parks, T.W., and Rabiner, L.R., A computer program for designing optimum FIR linear phase digital filters, *IEEE Trans. Audio Electroacoust.,* AU-21, 506–526, Dec. 1973.
19. Remez, E. Ya., General computational methods of Chebyshev approximation, *Atomic Energy Translation,* 4491, 1957.
20. Chen, X. and Parks, T.W., Design of FIR filters in the complex domain, *IEEE Trans. Acoust., Speech, Signal Processing*, ASSP-35, 144–153, Feb. 1987.
21. Alkhairy, A.S., Christian, K.G., and Lim, J.S., Design and characterization of optimal FIR filters with arbitrary phase, *IEEE Trans. Signal Processing,* 41, 559–572, Feb. 1993.
22. Komodromos, M.Z., Russell, S.F., and Tang, P.T.P., Design of FIR filters with complex desired frequency response using a generalized Remez algorithm, *IEEE Trans. Circuits Syst. II*, 42, 274–278, April 1995.
23. Burnside, D. and Parks, T.W., Optimal design of FIR filters with the complex Chebyshev error criteria, *IEEE Trans. Signal Processing,* 43, 605–616, March 1995.
24. Vuerinckx, R., *Design of Digital Chebyshev Filters in the Complex Domain,* Vrije Universiteit Brussel, Oct. 1997.
25. Karam, L.J., *Design of Complex Digital FIR Filters in the Chebyshev sense,* Georgia Institute of Technology, March 1995.
26. Karam, L.J. and McClellan, J.H., Chebyshev digital FIR filter design, *Signal Processing,* 76, 17–36, 1999.
27. Adams, J.W., FIR digital filters with least-squares stopbands subject to peak-gain constraints, *IEEE Trans. Circuits Syst.*, 39, 376–388, April 1991.
28. Sullivan, J.L. and Adams, J.W., Peak-constrained least-squares optimization, *IEEE Trans. Signal Processing,* 46, 306–321, Feb. 1998.
29. Sullivan, J.L. and Adams, J.W., PCLS IIR digital filters with simultaneous frequency response magnitude and group delay specification, *IEEE Trans. Signal Processing,* 46, 2853–2861, Nov. 1998.

30. Lang, M.C., An iterative reweighted least squares algorithm for constrained design of nonlinear phase FIR filters, *Proc. IEEE ISCAS,* 5, 367–370, 1998.
31. Lang, M.C., Multiple exchange algorithm for constrained design of FIR filters in the complex domain, *Proc. IEEE ICASSP,* 3, 1149–1152, 1999.
32. Lawson, C.L., *Contributions to the theory of linear least maximum approximations,* University of California, Los Angeles, 1961.
33. Lim, Y.C., Lee, J.H., Chen, C.K., and Yang, R.H., A weighted least squares algorithm for quasiequiripple FIR and IIR digital filter design, *IEEE Trans. on Signal Processing,* ASSP-40, 551–558, March 1992.
34. Burrus, C.S., Barreto, J.A., and Selesnick, I.W., Iterative reweighted least-square design of FIR filters, *IEEE Trans. Signal Processing,* 42, 2926–2936, Nov. 1994.
35. Lertniphonphun, W. and McClellan, J.H., Unified design algorithm for complex FIR and IIR filters, *Proc. IEEE ICASSP,* 2001.
36. Tseng, C.-Y., An efficient implementation of Lawson's algorithm with application to complex Chebyshev FIR filter design, *IEEE Trans. Circuits Sys. II,* 42, 245–260, April 1995.
37. Dudgeon, D.E. and Mersereau, R.M., *Multidimensional digital signal processing,* Prentice-Hall, NJ, 1984.
38. McClellan, J.H., The design of two-dimensional digital filters by transformations, *Proc. 7th Annual Princeton Conf. on Inform. Sci. and Syst.,* 247–251, 1973.
39. McClellan, J.H., *On the design of one-dimensional and two-dimensional FIR digital filters,* Rice University, Houston, TX, April 1973.

# 11

# Audio Signal Processing




Adam Dabrowski
Tomasz Marciniak
*Poznan University of Technology*


## 11.1 Introduction

Information and communication systems play bigger and bigger role in our modern society—the so-called information society. Sound (audio and speech) is one of the most important signals in these systems and the growing need for audio and speech processing (transmission, storing, etc.) generates new scientific problems (e.g., formulates new questions about data acquisition, compression, and coding), stimulates new technologies and techniques, as well as creates new areas of science and technology in informatics, computer engineering, communications, robotics, artificial intelligence, psychoacoustics, etc.

Applications of digital audio and digital speech processing systems are in audio production, storage, distribution, exchange, broadcasting, transmission, telephony, multimedia systems, computer games, Internet services, etc. Modern multimedia coding standards (e.g., moving picture expert group (MPEG) standards—MPEG-4, MPEG-7, and MPEG-21) [1–5] cover the whole range of audio signals starting

from high-fidelity audio, through the regular quality of audio and speech, down to relatively low-quality mobile access as well as synthetic speech and music.

To evaluate various audio coding systems, it is necessary to qualify the audio quality these systems offer. Generally, three main parameters are used to describe the quality of audio: bandwidth, fidelity, and spatial realism.

For high-fidelity (wideband) audio, a bandwidth of at least 20 kHz is needed. The acoustic signals with higher frequencies are not audible by human beings. Compact disc (CD)—the today's still most popular standard for digital audio representation—offers a bandwidth of 20–20,000 Hz. Traditional (analog) radio covers the bandwidth of up to 15 kHz for frequency modulation (FM) and only up to 4.5 kHz for amplitude modulation (AM). Wideband speech standard has a bandwidth of 50–7000 Hz, whereas the standard telephone speech is reduced to a bandwidth of merely 300–3400 Hz.

Fidelity is a (subjective) measure of perceptibility of impairment (noise) present in the reproduced audio. Audio fidelity is usually determined subjectively by means of an averaged judgment called the mean opinion score (MOS). It is typically based on a five-point grading scale: 5—impairment imperceptible, 4—perceptible but not annoying, 3—slightly annoying, 2—annoying, and 1—very annoying [6].

Spatial realism of an audio representation system describes the naturalness and quality of directional information about places of particular sound sources contained in the reproduced sound. The spatial realism depends first of all on the number of audio channels. Typical configurations are as follows: 1-channel audio (mono); 2-channel audio (stereo); and multichannel audio (surround sound), e.g., 4-channel (3 front and 1 rear), 5-channel (3 front and 2 rear), or 8-channel (6 front and 2 rear). An additional low-frequency enhancement (LFE) or subwoofer channel, supplementing the low-frequency content (in a bandwidth of approximately 15–150 Hz), can be added in any of these cases (e.g., a 5.1-channel format is a 5-channel configuration plus subwoofer). New multichannel standards allow for quite flexible extension of the number of channels, e.g., up to 13.1 channels for the Dolby Digital Plus standard [7]. Resulting high bitstream rates (even up to 6.144 Mbit/s) require high speed interfaces as, e.g., the new high-definition multimedia interface (HDMI) [8].

Thanks to modern electronic chips (first of all to digital signal processors, which are shortly described in Section 11.6) audio compression/decompression algorithms can be realized in real time. This is, for example, the case in well-known portable phones, audio players, and home theater systems [9,10]. Digital signal processors also give possibility for real-time noise reduction, echo cancellation, signal separation, spatial filtering, etc., which significantly improve operation quality of speech and audio processing systems, such as hearing aids, voice recognition, and speech recognition systems [11–16].

## 11.2 Elements of Technical Acoustics

For the purpose of this chapter, sound can be defined as a mechanical oscillation of an elastic medium that potentially can be heard. If acoustic vibrations are too high in frequency to be heard, they are referred to as ultrasonic oscillations. Consequently, if they are too low in frequency, they are called infrasonic oscillations. The sound starts in approximately 20 Hz and extends up to 20 kHz (thus it covers a bandwidth of approximately 10 octaves) [17–19].

A source of sound undergoes rapid changes of position (and size, or shape) that disturb positions of adjacent molecules of the surrounding medium (in most cases the atmosphere). Thus, these molecules start to oscillate about their equilibrium positions. These disturbances propagate elastically to neighboring particles and then gradually to larger and larger distances, thus constituting an acoustic wave traveling through the medium. The acoustic wave speed in air equals

$$c = c_0\sqrt{1 + \frac{\vartheta}{273}} \tag{11.1}$$

where $\vartheta$ is the room temperature in degree celsius and $c_0 = 331$ m/s is the sound speed at $\vartheta_0 = 0°C$. At room temperature ($\vartheta = 20°C$), the speed of sound is calculated to be 343 m/s.

A sound wave compresses and dilates the elastic medium it passes through, generating associated pressure fluctuations. The minimum fluctuation, to which the ear responds, is extremely small; e.g., at a frequency of 1000 Hz, the just noticeable effective air pressure amplitude, or in other words the root mean square (RMS) value, is approximately 20 μPa, i.e., less than $10^{-9}$ of the standard atmospheric pressure (equal to ~1000 hPa = $10^5$ Pa). The limit of danger followed by the threshold of pain corresponds to effective air pressure amplitude one million ($10^6$) times larger, but still less than one-thousandth of the atmospheric pressure [20].

Because of this wide range of acoustic pressure amplitudes, it has become conventional to specify the sound pressure level (SPL), $L_p$, in terms of a decimal logarithm with the (dimensionless) unit of the decibel (dB)

$$L_p = 20 \log_{10} \frac{p}{p_0} \tag{11.2}$$

where $p_0 = 20$ μPa.

Another quantity, which is often used, is the sound intensity level, $L_I$, defined in decibel as

$$L_I = 10 \log_{10} \frac{I}{I_0} \tag{11.3}$$

The reference in this case is the sound intensity $I_0 = 10^{-12}$ W/m$^2$. For a free progressive acoustic wave in air, SPL and sound intensity level are approximately equal.

## 11.3 Parametric Modeling of Audio Signals

A natural representation of an audio signal is its waveform $x(t)$ describing the sound pressure changes in time. Signal $x(t)$ occurs at a microphone output, excites the speaker, and generally, represents sound in analog audio processing systems. On the other hand, in digital systems, to reduce the required bitrate, the physical signal $x(t)$ can be replaced by a number of parameters, e.g., describing the way the speech or the audio signal originates (see Figure 11.1). The major problem, however, that immediately arises and has to be overcome, consists a fact that there exists no unique plausible model for production of all kinds of audio signals. On the contrary, for different types of audio, only different models (if any) can be proposed [2,3,21].

For example, for speech, efficient parametric description models can be developed by means of modeling of the human vocal tract [12]. Such parametric speech source models describe the speech production process by first modeling an excitation (noise for unvoiced speech and a periodic signal for voiced speech) and second, by representing the human voice tract by means of an appropriate infinite impulse response (IIR) filter. This is the so-called linear prediction coding (LPC) scheme [22]. The coder based on LPC concept referred to as the harmonic vector excitation coder (HVXC) is a part of the MPEG-4 audio standard. It has been developed to code narrowband speech at 8 ksamples/s sampling rate and bitrates of 2.0 or 4.0 kb/s [1].

Another signal example, for which an extremely efficient parametric description exists, is music. This description is the well-known musical score notation. Indeed, a kind of such a description has been applied in the musical instrument digital interface (MIDI). Although the musical score notation is extremely efficient and it does not probably exist any other representation for music that would be more efficient, the score as a means for audio coding has two major drawbacks. First, the whole information about the individual performer is lost. Second, an automatic transcription of audio into the musical score is very difficult. Thus, compromise solutions have to be searched for. Structured audio—a part of the MPEG-4 standard—uses such techniques. An audio signal is split into individual, meaningful source objects (natural audio objects, e.g., speech and music) and is treated as a composition of these [1,4,6]. This approach is also used and further developed in the newest standards MPEG-7 and MPEG-21 [2–4].

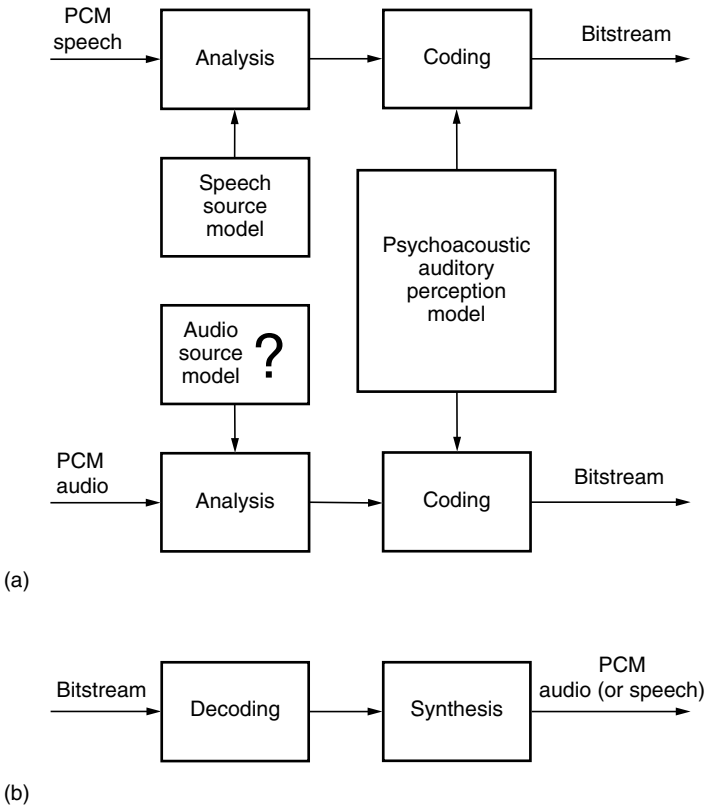



**FIGURE 11.1** Efficient audio (and speech) compression by exploiting the knowledge about the audio (and speech) production and perception: (a) coder scheme and (b) decoder scheme.

One of the most promising approaches to the parametric description of a wide class of audio signals consists in removing the redundancy contained in the original audio signal representation $x(t)$. This can be done by splitting the signal into a number of almost uncorrelated components. If these components are computed by means of some predefined transformation, the corresponding technique is referred to as the transform-based audio coding procedure. The simplest and the most popular method of this type is the time-to-frequency transformation by means of an appropriate analysis filter bank [23]. For example, in the MUSICAM standard [24] the whole audio band, which is in this case 24 kHz wide, is split into 32 uniform subbands of the width $24{,}000/32 = 750$ Hz each. Another widely used uniform filter bank is based on a modified discrete cosine transformation (MDCT) [25] (see Section 11.7).

Nonuniform, e.g., octave filter banks, can also be used. These optimized octave filter banks can be based on the so-called wave digital filters (WDFs) [26]. Efficient parametric description of audio signals, which is, in fact, a generalization of the octave filter bank approach and is, in other words, a simplified and formalized score-type representation, is the so-called discrete wavelet transformation (DWT) [27]. The DWT concept is briefly presented in Section 11.7.

Another quite efficient approach for the parametric description of audio is the so-called sinusoidal modeling often used for the analysis and synthesis of musical instrument sounds [21]. The audio signal $x(t)$ is modeled by a set of tones and noise

$$x(t) = \sum_k a_k(t) \sin\left(\int_{t=0}^{t} \omega_k(\tau) + \phi_\kappa\right) + \text{noise} \tag{11.4}$$

The tones have slowly varying parameters: amplitude $a_k(t)$ and frequency $\omega_k(t)$. Additionally, an appropriate noise model has to be used. A perceptually acceptable noise model can be obtained by adding some sinusoids with different frequencies and random phases. Alternative methods are based on the noise spectrum modeling.

An additional envelope model (with particular envelope attack and decay rates) can be added for some of the sinusoids to improve the sinusoidal model efficiency for highly nonstationary signals.

A dual approach to that described by Equation 11.4 is also possible. The signal $x(t)$ is first transformed into the frequency domain, e.g., by means of the discrete cosine transformation (DCT), then the sinusoidal modeling is realized in the frequency domain [21].

The MPEG-4 audio coding standard supports a related, advanced technique named HILN (harmonic and individual lines and noise) [28]. The HILN parametric audio encoder decomposes an audio signal into harmonic, individual sinusoidal, and noise components. Harmonic components are determined by means of their fundamental frequencies, amplitudes, and spectral envelopes. Sinusoidal components are represented by their amplitudes and frequencies. Noise is described by the amplitude and spectral envelope. Typically, frames of 32 ms duration are analyzed and information about the important harmonic content of a single frame is extracted by means of the short-time Fourier transformation. By matching amplitudes, phases, and frequencies of sinusoids across frames, the encoder groups them into harmonic or individual sinusoidal components. The longer the traces of components the encoder finds, the lower is the resulting bitrate. Finally, the encoder subtracts the superposition of all just-determined components from the original audio. The resulting residual signal is represented as the output of the matched linear filter excited with white noise. Such an encoder is capable of encoding a typical 8 kHz bandwidth audio signal with bitrate of 6–16 kb/s.

A related codec, developed by Philips and named SSC (sinusoidal codec), is based on splitting the audio signal into three objects: sinusoids, transients, and noise [29].

Other examples of parametric audio coding techniques are MPEG-4 PS (parametric stereo) and MPEG-4 SBR (spectral band replication) [30–34]. Both methods consist in omitting parts of original signal information and replacing it by a synthetic information such that the listener is satisfied with the final audio quality. An idea of SBR is explained in the next section.

## 11.4 Psychoacoustics and Auditory Perception

Understanding of psychoacoustics phenomena occurring during the auditory perception by humans is crucial for the design of efficient audio coding algorithms. An efficient audio coder (the so-called perceptual coder) should not only reduce redundant components in the audio representation, using an appropriate parametric audio model (see Section 11.3), but also remove irrelevant components from the source signal, i.e., those that are inaudible by humans.

Signal processing, which takes place in the human auditory system, can generally be divided into two stages: a preliminary phase realized in the acoustic auditory organs (ears) and the advanced phase done in auditory nervous system (in the brain). The auditory part of the inner ear, known as the cochlea because of its snail-like shape, performs a kind of the spectral analysis. The acoustic harmonic tones generate place selective oscillations distributed along the so-called basilar membrane, which extends down the cochlea. As a result, the frequency is mapped into a place on the basilar membrane and a frequency scale can be laid out at the basilar membrane with low frequencies near the apex and high frequencies near the base of the cochlea. According to the authors' results, the cochlear response is not a kind of a Fourier like transformation but, neglecting the nonlinearities, it is rather a kind of the nearly continuous wavelet transformation (CWT) [27]. Consequently, the cochlear response can be interpreted as if it were produced by a filter bank composed of highly overlapping bandpass filters with increasing passbands. These filters are referred to as the peripheral auditory filters.

Two widely accepted approaches are used for estimation of the passbands of the peripheral auditory filters. The older approach is based on the notion of critical bands $\Delta f_c$ [35–37]. The widths of the critical

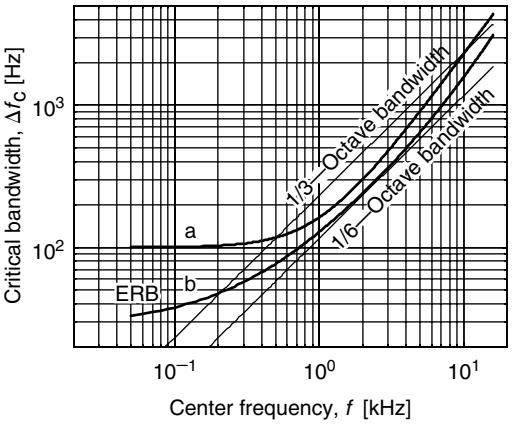



**FIGURE 11.2** Critical bandwidth and equivalent rectangular bandwidth (ERB) as functions of the passband center frequency: (a) critical bandwidth according to Equation 11.5 and (b) ERB according to Equation 11.6a.

bands vary from ~100 Hz for low frequencies (lower than 300 Hz) to about one-third of an octave for high frequencies (Figure 11.2a). The critical bandwidth as a function of its center frequency can be estimated in Hertz using the following expression:

$$\Delta f_c = 25 + 75(1 + 1.4f^2)^{0.69} \tag{11.5}$$

in which frequency $f$ is given in kilohertz [36].

The newer approach results from measurements of the frequency response shape of the peripheral auditory filters and uses a concept of equivalent rectangular bandwidth (ERB) [17]. ERB is a bandwidth of the equivalent ideal (rectangular) passband filter, which has the same center passband frequency as the respective peripheral auditory filter, transmits the same amount of power when excited with the same white noise, and has the passband gain equal to the maximum passband gain of the respective auditory filter. ERB as a function of frequency can be approximated in hertz as

$$\text{ERB} = 6.23f^2 + 93.3f + 28.52 \tag{11.6a}$$

where frequency $f$ is again given in kilohertz (see Figure 11.3b). Sometimes, a slightly simpler formula is used [17]:

$$\text{ERB} = 24.7(4.37f + 1) \tag{11.6b}$$

Chosen values of $\Delta f_c$ and ERB are listed in Table 11.1. Critical bandwidth is greater than ERB even three times for low frequencies but for higher frequencies, starting with ~500 Hz, it is only 1.3–1.5 times larger.

Approximately, 24 nonoverlapping critical bands cover the whole audible frequency range; but, it does not mean that there exist only 24 peripheral auditory filters. In fact, they occur as nearly continuously distributed filters along the frequency axis and any audible tone creates an individual peripheral auditory filter centered on it.

The ear canal acts as a resonator and increases the sound pressure at the tympanic membrane in the frequency range of 1.5–5 kHz, with a maximum at 3.5 kHz by about 10–15 dB. The sensitivity of the ear varies strongly with the frequency and reaches the maximum exactly in this band. In Figure 11.4, equal-loudness contours for pure tones are plotted. They are labeled in units of loudness level called phones. By definition, the loudness level in phones is numerically equal to SPL in decibels at the frequency

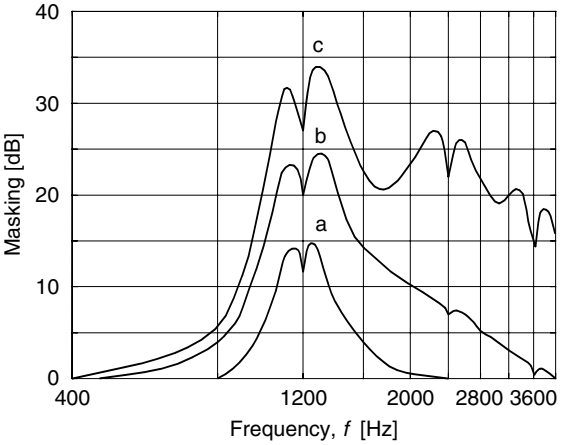



**FIGURE 11.3** Simultaneous tone-masking-tone effect relative to the threshold of audibility in quiet with a masker of frequency 1.2 kHz and of three different levels (SPL): (a) 40 dB, (b) 50 dB, and (c) 60 dB.

$f = 1000$ Hz. The lowest curve in Figure 11.4 represents the threshold of audibility (in quiet). This curve can be approximated [38] in decibels with expression given below:

$$L_{ptq} = 3.64f^{-0.8} - 6.5e^{-0.6(f-3.3)^2} + 10^{-3}f^4 \tag{11.7}$$

where frequency $f$ is as before given in kilohertz. The threshold of audibility computed with the above equation is shown in Figure 11.5.

A kind of positive feedback improves the sensitivity and selectivity of the basilar membrane oscillations. Its function can be compared with that of the so-called reaction used in early radio receivers to increase their amplification and to improve their frequency selectivity. The positive feedback effect decreases as sound intensity increases. Thus, the cochlea is less selective for intense sounds than for weak sounds. As a result, the peripheral auditory filters are nonlinear, thereby extending the overall dynamic range of the hearing system to the range of ~120 dB (see Figure 11.4).

Figure 11.6 shows the whole region of audibility extending from the threshold of audibility to the limit of danger (and further up to the threshold of pain). It also illustrates two important subregions: the region of speech and the region of music. The rest of the audibility area is a reserve of the human hearing system. Speech covers the frequency band of ~200 Hz to 5 kHz and the dynamic range ~50 dB. Music occupies larger area, i.e., the frequency band of 50 Hz to 10 kHz and the dynamic range of ~70 dB. For

**TABLE 11.1** Critical Bandwidth and Equivalent Rectangular Bandwidth (ERB) as Functions of the Respective Center Frequency

| Center Frequency $f_c$ (Hz) | Critical Bandwidth $\Delta f_c$ (Hz) | ERB (Hz) | $\Delta f_c$/ERB |
|---|---|---|---|
| 50 | 100 | 33 | 3.0 |
| 100 | 100 | 38 | 2.7 |
| 200 | 100 | 47 | 2.2 |
| 500 | 120 | 77 | 1.5 |
| 1000 | 160 | 128 | 1.3 |
| 2000 | 300 | 240 | 1.3 |
| 5000 | 900 | 651 | 1.4 |
| 10,000 | 2300 | 1585 | 1.5 |

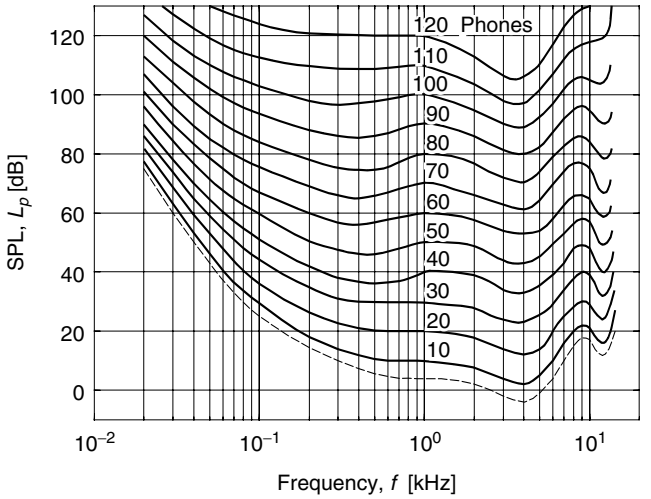



**FIGURE 11.4** Equal-loudness contours for pure tones.

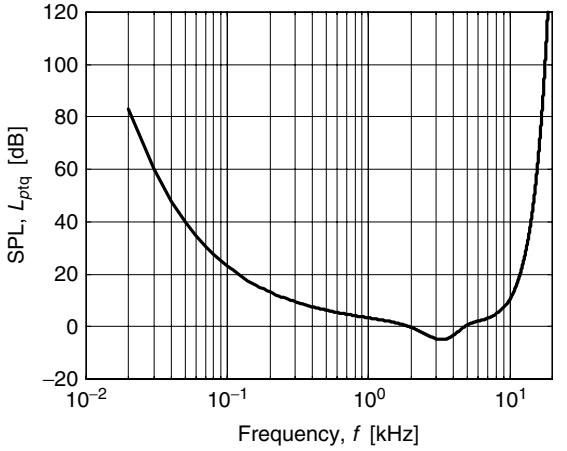



**FIGURE 11.5** Threshold of audibility in quiet approximated by Equation 11.8.

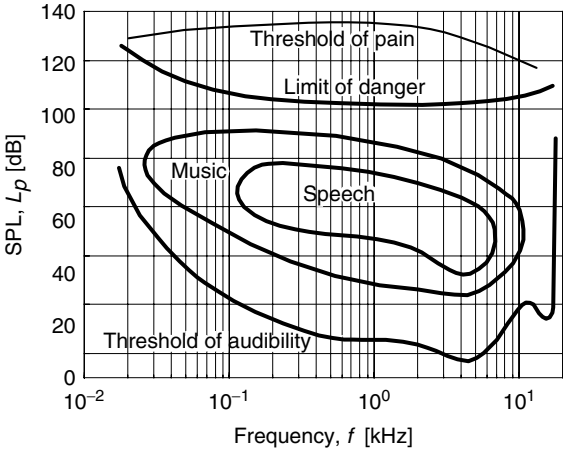



**FIGURE 11.6** Region of audibility.

the representation of high-quality audio it is, however, necessary to cover and reproduce practically the whole region of audibility, i.e., the frequency band of 20 Hz to 20 kHz and the dynamic range of at least 80–90 dB.

Because of diffraction produced by the head, the sound that reaches the ears depends on the sound source direction. The difference between the arrival time of sound at each of the two ears together with the difference in the intensity of the sound that reaches each ear is used by the auditory nervous system to determine the location of the sound source. This ability manifests mostly in the horizontal plane. In audio coding systems, it is represented by stereo (2-channel) or more exactly by surround (multichannel) sound.

The spectral components of a sound are coded for intensity and time in the auditory nervous system, but not always all components are audible. This interesting phenomenon called masking is extremely important for efficient digital coding of audio. Masking is a kind of interference with the audibility of a sound (called probe or maskee) caused by the presence of another sound (called masker), if both these sounds are close enough to each other in frequency and occur simultaneously or closely to each other in time. If a lower level probe is inaudible, because of a simultaneous existence of a higher level masker, this effect is referred to as the simultaneous masking. If an inaudible probe precedes the masker or follows the masker, this phenomenon is called temporal masking. Masking is typically described by the minimum shift of the probe intensity level above its threshold of audibility in quiet, necessary for the probe to be heard in the presence of the masker.

Four different cases for masking can be distinguished: tone-masking-tone, noise-masking-tone, tone-masking-noise, and noise-masking-noise. The latter two cases are particularly important for the design of effective perceptual audio coders, because masking can be exploited to make the quantization noise inaudible. The first two cases were, however, so far, much more intensively investigated. In Figure 11.3, a simultaneous tone-masking-tone effect relative to the threshold of audibility in quiet is illustrated. Masker is a pure harmonic tone of frequency 1.2 kHz and of three different SPLs: 40, 50, and 60 dB. The following effects can be observed. First, the higher the level of the masker, the greater is the masking. Second, masking is largest for probe frequencies slightly above or below the masker frequency. Third, masking decreases as probe frequency gets closer to that of the masker. This phenomenon is observed only for tone-masking-tone case and is caused by audible beats between the two tones, which make the presence of the probe more apparent. Fourth, masking is greater on frequencies above the masker frequency than on frequencies below it. Fifth, because of the nonlinearity of the human hearing system, the masking curve has similar shape for various masker harmonics. This phenomenon is also typical only for the tone-masking-tone case.

Analyzing curves in Figure 11.3 and taking the threshold of audibility in Figure 11.4 or 11.5 into account, the maximum probe-to-masker ratios (PMRs) can be determined. For example, for a 40 dB SPL masker, the maximum PMR is $15 + 3 - 40 = -22$ dB (the maximum level of the fully masked probe relative to the threshold of audibility for the curve in Figure 11.3a is almost 15 dB and the threshold of audibility for the tone of frequency 1.2 kHz is about 3 dB SPL). Similarly, for 50 and 60 dB maskers, the maximum PMRs are calculated to be $-22.5$ and $-23$ dB, respectively.

Masking curves for tone-masking-noise are similar but smoother, because no audible beats occur in this case. To reduce the influence of audible beats, also in the tone-masking-tone case, a tone-like narrowband noise instead of a pure tone should be used as masker. In practical audio signals, this situation is observed rather than appearance of audible beats. That is why both cases with the tone as masker can be reduced to only one: a tone-like masker. The maximum PMR can be approximated [39] in decibels by the following expression:

$$\mathrm{PMR}_t = -(14.5 + z) \tag{11.8}$$

where $z$ is numerically equal to the critical band index in Bark [18] defined as

$$z = 13\arctan(0.76f) + 3.5\arctan(f/7.5)^2 \tag{11.9}$$

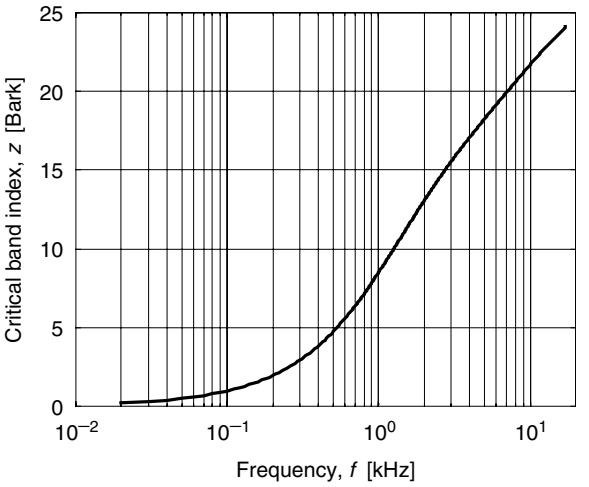



**FIGURE 11.7** Critical band index in Bark according to Equation 11.9.

where frequency $f$ is in kilohertz. The curve determined by Equation 11.9 is plotted in Figure 11.7.

When a wideband flat noise is used to mask a pure tone, masking is much stronger than that just considered. It should, however, be stressed that only a narrow frequency band (the critical band) of the noise centered at the tonal frequency causes masking of this tone. If the bandwidth of the previously wideband masking noise is made narrower than the respective critical bandwidth (noise with the constant power spectral density is considered) and if the previous probe tone level was just below the masking threshold, then the intensity of this tone has to be lowered before it can be masked again. On the other hand, if the noise bandwidth is wider than this critical bandwidth, no significant change in the masking effect can be observed. In this case, the maximum PMR, illustrated in Figure 11.8, can be determined in decibels [40] by the following expression:

$$\mathrm{PMR}_n = -2.0 - 20.5\arctan(f/4) - 0.75\arctan(f^2/2.56) \tag{11.10}$$

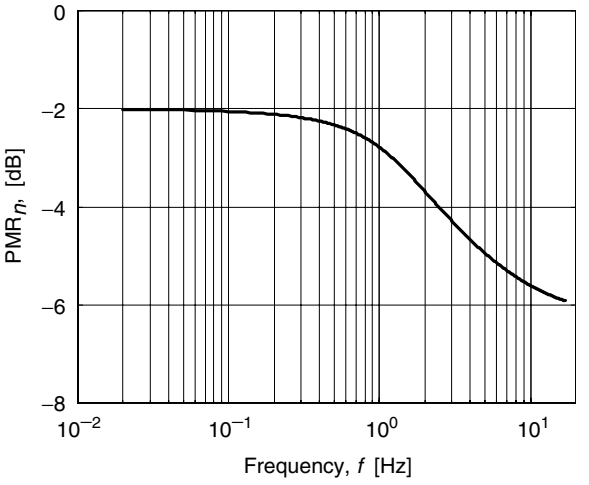



**FIGURE 11.8** Maximum PMR for noise masker.

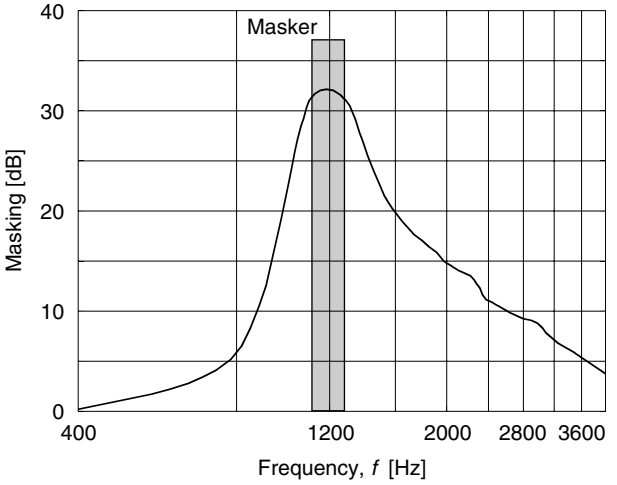



**FIGURE 11.9** Simultaneous noise-masking-tone effect relative to the threshold of audibility in quiet with a masker of center frequency of 1.2 kHz, critical bandwidth, and 40 dB SPL.

where frequency $f$ is again given in kilohertz. Pessimistically, a constant value $\mathrm{PMR}_n \approx -5.5$ dB can be used independently from frequency. Simultaneous noise-masking-tone effect relative to the threshold of audibility in quiet with a masker of center frequency $f_c = 1.2$ kHz, the critical bandwidth, and 40 dB SPL, is illustrated in Figure 11.9.

Although masking is typically measured as a shift of the threshold level of hearing above the threshold level of audibility in quiet, its mathematical model should be based on the additivity of signal powers (linear scale) rather than on the additivity of levels (logarithmic scale). In this context, a notion of psychoacoustic excitation is widely used [17]. Particular excitations are approximately additive in terms of power; however, it is also convenient to introduce the excitation level (i.e., excitation described in the logarithmic scale) because the masking threshold level $L_{\mathrm{tm}}$ can be modeled as the excitation level shifted by the PMR. In all masking cases, the simplest mathematical description for the masking threshold level $L_{\mathrm{tm}}$ is a triangular shape shown in Figure 11.10. In the abscissa axis, the critical band index $z$ in Bark is

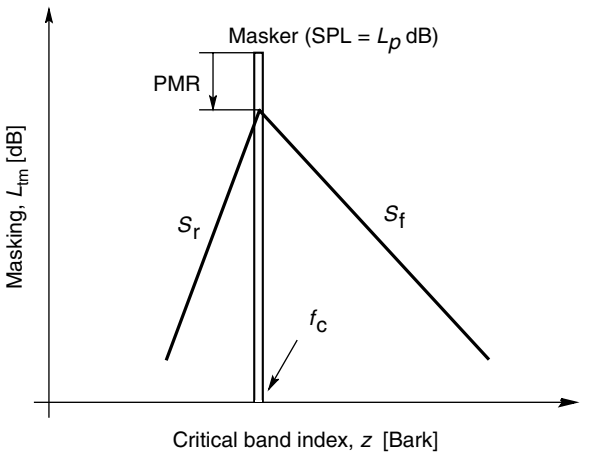



**FIGURE 11.10** Simplified model of the masking threshold level $L_{\mathrm{tm}} = L_{\mathrm{tm}}(z, z_c, L_p, \alpha)$.

used. The masking threshold peak can be determined in decibel by the maximum PMR according to the following equation [39,41]:

$$\mathrm{PMR} = \alpha \mathrm{PMR}_t + (1 - \alpha)\mathrm{PMR}_n \tag{11.11}$$

where $0 \leq \alpha \leq 1$ is the masker tonality index defined in such a way that $\alpha = 0$, if the masker is a white noise and $\alpha = 1$, if the masker is a tone. Parameter $\alpha$ can be determined using the so-called spectral flatness measure (SFM) of the masker, defined as a decimal logarithm of the geometric average to the arithmetic average ratio of the masker power spectral density distribution in the masker frequency band. $\mathrm{SFM}_{\max} = 0$ dB for a white noise masker, and $\mathrm{SFM}_{\min} \cong -60$ dB for a practically pure tone masker. Parameter $\alpha$ could be computed as

$$\alpha = \mathrm{SFM}/\mathrm{SFM}_{\min} \tag{11.12a}$$

but owing to possible computational inaccuracies the result computed using expression given by Equation 11.20a could be greater than 1. Therefore, a slightly more complicated expression should be used

$$\alpha = \min(\mathrm{SFM}/\mathrm{SFM}_{\min}, 1) \tag{11.12b}$$

The rising slope $S_\mathrm{r}$ (dB/Bark) of the masking threshold triangle is approximately constant and equals [38]

$$S_\mathrm{r} = (25/27) \tag{11.13a}$$

The falling slope $S_\mathrm{f}$ (dB/Bark) is smaller and depends on the masker SPL, $L_p$, in decibel and its center frequency, $f_c$, in kilohertz

$$S_\mathrm{f} = -24 + 0.2L_p - 0.23f_c^{-1} \tag{11.13b}$$

In most typical situations, the falling slope can be approximated as $S_\mathrm{f} \approx -10$ dB/Bark.

Consequently, the masking threshold level $L_\mathrm{tm}$ is a function of the critical band index $z$, the masker center position $z_c$, the masker SPL, $L_p$, and the masker tonality index $\alpha$, i.e., $L_\mathrm{tm} = L_\mathrm{tm}(z, z_c, L_p, \alpha)$. If many simultaneous maskers occur together, the overall masking effect as a function of frequency can be determined by the global threshold of hearing $L_{p\mathrm{tg}}$ (Figure 11.11). To determine this threshold, additivity of signal powers or respective psychoacoustic excitations should be taken into account. Thus, the following approximate expression for the global threshold of hearing in decibel can be used:

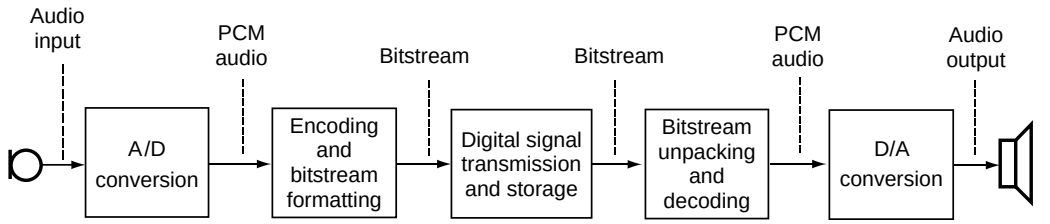



**FIGURE 11.11** General scheme of a digital audio processing system.

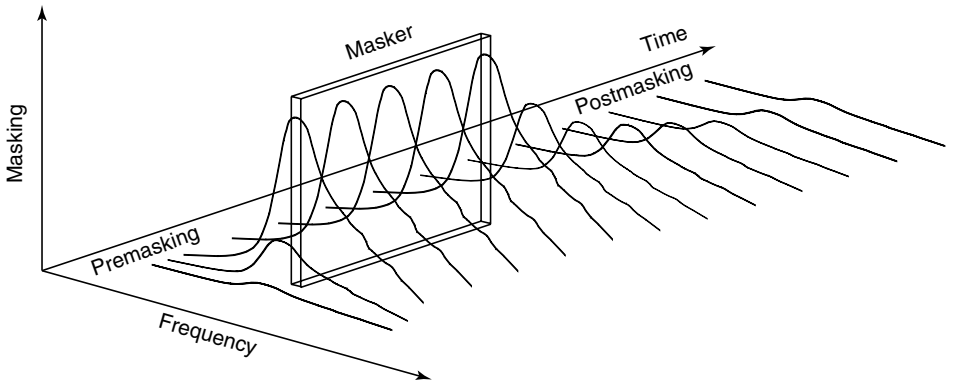



**FIGURE 11.12** Combined noise-masking-tone effect in frequency and time.

$$L_{p\mathrm{tg}} = 10\log_{10}\left(10^{L_{p\mathrm{tq}}(z)/10} + \sum_i 10^{L_{\mathrm{tm}}(z,z_{cj},L_{pj},\alpha_j)/10}\right) \tag{11.14}$$

where $L_{p\mathrm{tq}}(z)$ is the threshold of audibility in quiet, $L_{\mathrm{tm}}(z, z_{cj}, L_{pj}, \alpha_j)$ are particular masking threshold levels, and index $j$ indicates the $j$th masker.

Finally, the signal-to-mask ratio (SMR) can be computed in decibels as follows:

$$\mathrm{SMR}_j = L_{pj} - L_{p\mathrm{tg}}(z_j) \tag{11.15}$$

where $z_j$ corresponds to the smallest $L_{p\mathrm{tg}}$ within the critical band of the $j$th masker. This is usually the left-hand side edge of this critical band.

Masking also occurs when the signal either precedes or follows the masker (Figure 11.12). This is the already mentioned phenomenon of temporal masking (Figure 11.13). Note that both Figures 11.9 and 11.13 show respective cross sections of the 2D surface in Figure 11.12.

In backward masking (premasking), the signal precedes the masker, whereas in forward masking (postmasking) the signal follows the masker. The premasking effect appears in 10–20 ms before the masker, whereas the postmasking effect is by one order of magnitude longer, i.e., in the order of 50–200 ms after the masker ends. In order to take the postmasking into consideration, the signal power $P_j = 10^{L_{pj}/10}$ occurring in time $t$ should be seemingly increased according to the following equation:

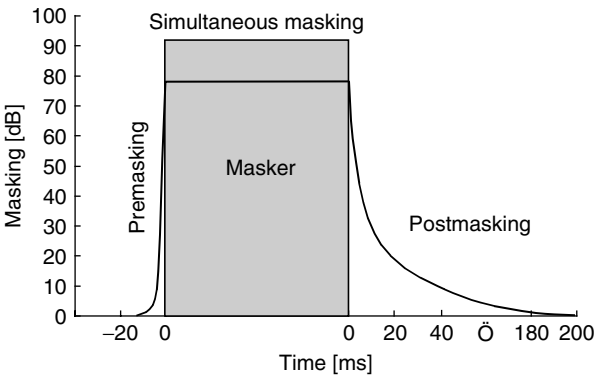



**FIGURE 11.13** Temporal masking effect.

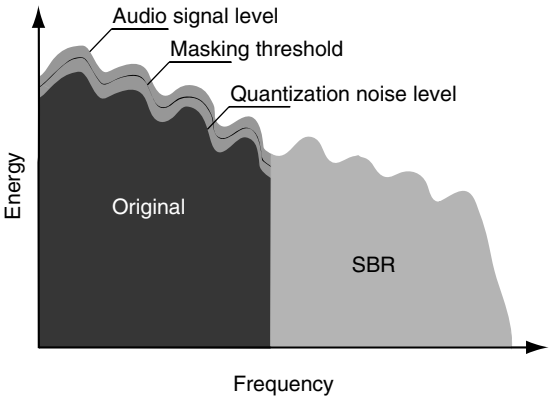



**FIGURE 11.14** Spectral band replication (SBR) principle.

$$p_{sj}(t) = P_j(t) + c_j(\Delta t)p_j(t - \Delta t) \tag{11.16}$$

in which the postmasking coefficient $c_j(\Delta t)$ is given by

$$c_j(\Delta t) = \exp(\Delta t/\tau_j) \tag{11.17}$$

but $\tau_j$ is a time constant depending on the $j$th critical band. As a result, the simultaneous masking threshold rises according to the increased level $L_{psj} = 10\log_{10} p_{sj}$.

All described masking effects are exploited for data compression in modern perceptual audio coders (e.g., in the MUSICAM procedure, MPEG standards, etc.).

A new audio representation and coding idea, recently approved for MPEG-4 as audio extension standard, is the so-called SBR [32–34]. This is a bandwidth extension technique related to the way the sounds (music and speech) originate and are perceived by humans. The SBR principle is based on reduction of the audio signal bandwidth to the low-frequency region (e.g., to the lower half) only and, consequently, on respective lowering (halving) of the sampling rate, then on retrieving the missing high-frequency content (the upper bandwidth half) by mapping the low-frequency portion together with appropriate energy shaping, and by adding some still missing information as tone and noise components. All this should be done using as small amount of additional information as possible just to guarantee high subjective quality of the final audio signal (Figure 11.14).

The SBR technology is very useful for low bitrate audio delivery, such as the Internet streaming for audio on demand.

## 11.5 Principles of Audio Coding

To digitally process audio, it is first necessary to sample and quantize the data, i.e., to convert the analog signal $x_c(t)$ into a digital form. This is realized in an analog-to-digital converter (ADC). The digital data can then be compressed and encoded in a digital audio coder (transmitter), transmitted through a communication channel, decoded in a receiver, and finally recovered in a digital-to-analog converter (DAC). A general scheme of a digital audio processing system is shown in Figure 11.11.

Sampling of a continuous-time signal

$$X = X_c(t), \quad -\infty < t < \infty \tag{11.18}$$

is a process of time discretization. It consists in representing the signal $x_c(t)$ with a series of samples

$$X_n = X(n) = X_c(t_n), \quad n = 0, \pm 1, \pm 2, \ldots \tag{11.19}$$

referred to as the discrete-time signal or sampled-data signal. Uniform time discretization with sampling period $T_s > 0$ and rate

$$F_S = 1/T_S \tag{11.20}$$

is defined by

$$X_n = X(n) = X_d(nT_s) = X_c(t_n),\ t_n = nT_s - \tau,\ n = 0, \pm 1, \pm 2, \ldots \tag{11.21}$$

where $\tau > 0$ is some (usually unavoidable) system delay.

According to the sampling theory, a low-band continuous-time signal $x_c(t - . \tau)$, i.e., the signal, whose spectrum extends from zero to some maximum frequency, can be reconstructed on the basis of the discrete-time signal $x(n)$, if the sampling rate $F_s$ is greater or at least equal to the Nyquist sampling rate, which is twice as high as the greatest frequency contained in the continuous-time signal spectrum, or in other words, if the whole signal spectrum lies below $F_s/2$, called the Nyquist frequency. In practice, sampling rate $F_s$ has to be somewhat greater than the Nyquist sampling rate [22]. Typical sampling rates for audio are 8 ksamples/s for telephony (the signal spectrum extends up to 4 kHz, and thus covers most of the frequencies contained in speech), 32 ksamples/s for medium quality digital audio (audible frequency band up to 16 kHz is covered), 44.1 ksamples/s for a CD standard (audio frequency band up to 22.05 kHz is represented), 48 and 96 ksamples/s for high-quality digital audio (the represented frequencies range up to 24 and 48 kHz, respectively).

An important generalization of the classic sampling theory applies to band signals [42]. A continuous-time signal, whose spectrum is limited to some frequency band

$$\Delta f = f_1 - f_2, \quad f_2 > f_1 \tag{11.22}$$

can be sampled with a sampling rate of at least $F_s = 2\Delta f$ only (i.e., critically sampled), if both spectrum border frequencies $f_1$ and $f_2$ in Equation 11.22 are consecutive multiples of the Nyquist frequency $F_s/2$, i.e., if

$$f_1 = kF_s \quad \text{and} \quad f_2 = (k+1)F_s/2 \tag{11.23}$$

where $k$ is an integer. Such a signal is referred to as the integer-band signal. Audio signals are not by themselves integer-band signals but they can be split with an analysis filter bank to some subband signals, which all are integer-band signals, and thus, can be critically sampled. This is indeed the case in many digital audio coders; e.g., in the MUSICAM standard the input audio signal, initially sampled with 48 ksamples/s, is split into 32 subbands with bandwidths of $24{,}000/32 = 750$ Hz each. Signals in each subband are sampled with $48/32 = 1.5$ ksamples/s sampling rate.

Another signal discretization process is quantization, i.e., the procedure of converting a signal with continuously distributed values into a signal with discrete values. Unlike sampling, which, under some conditions, can be considered lossless, i.e., the original signal can—at least theoretically—be perfectly recovered after sampling, quantization is an inherently lossy operation [22,42].

The error due to the quantization has a nature of noise and is referred to as the quantization noise. Although this noise is unavoidable and cannot be removed from the signal, it can be made inaudible by controlling its level and forcing it to lie under the threshold of audibility. Masking effects, discussed in Section 11.4, can be very effectively exploited with this end in view.

The quantization noise is usually analyzed under the following simplifying assumptions:

- The quantization steps are uniform
- The number of quantization levels is high

The first assumption is not fulfilled in many quantization techniques for audio signals. This is because the perception of noise does not depend on its absolute power but on the signal-to-noise ratio (SNR). Thus, it is reasonable to quantize audio signals nonuniformly, with quantization steps proportional to the signal values. If the steps are not uniform, then the quantization error will be a function of the input signal, and consequently, it will not be an additive noise any more. Fortunately, in most procedures for the quantization of audio signals, quantization steps are at least range by range uniform and the first assumption can be considered as approximately valid. The second assumption is usually satisfactorily fulfilled. Owing to this assumption, the quantization noise has a uniform probability density distribution and is not correlated with the signal [43].

Denote by $Q$ the quantization step and by $p(x)$ the probability distribution function of the quantization error. Then

$$\int_{-Q/2}^{Q/2} p(x)\mathrm{d}x = 1 \tag{11.24}$$

where

$$p(x) = \begin{cases} 1/Q & \text{for } x \in [-Q/2, Q/2] \\ 0 & \text{otherwise} \end{cases} \tag{11.25}$$

From Equations 11.24 and 11.25, the average quantization noise power $p_\mathrm{q}$ can be calculated as

$$P_\mathrm{q} = \int_{-\infty}^{\infty} x^2 p(x)\mathrm{d}x = \int_{-Q/2}^{Q/2} \frac{X^2}{Q}\mathrm{d}x = \frac{Q^2}{12} \tag{11.26}$$

The SNR in decibels is then

$$\mathrm{SNR} = 10\log_{10}\left(\frac{P_\mathrm{S}}{P_\mathrm{q}}\right) \tag{11.27}$$

where $P_\mathrm{s}$ is the time-averaged signal power. Assume that the ADC has a full scale of $m$ bits. Then the maximum input signal amplitude is

$$A = (2^m - 1)Q \tag{11.28}$$

and thus

$$P_\mathrm{S} \propto [(2^m - 1)Q]^2 \tag{11.29}$$

From Equations 11.26, 11.27, and 11.29 it follows that

$$\mathrm{SNR}_{(m)} = 20m\log_{10} 2 + \text{constant} \approx 6.02m + \text{constant} \tag{11.30}$$

Thus, each additional bit in the quantized signal resolution means ~6 dB improvement in the SNR (or equivalently in the dynamic range). The "constant" in Equation 11.30 is of secondary importance. Its value depends on the signal probability density distribution and the ADC range. For instance, for an ADC range equal to $(-4\sqrt{p_\mathrm{S}}, 4\sqrt{p_\mathrm{S}})$, the respective value is constant $\approx -7.3$ dB.

Representing a signal just as a stream of uniformly quantized samples is referred to as the pulse code modulation (PCM) and is usually considered as the original digital audio signal of the maximum

achievable quality. Typical resolutions in bits per sample (bps) are 16, 20, 24, 32, and even 48 bps. For instance, for a CD standard with two stereo channels, 44.1 ksamples/s sampling rate and 16-bit resolution, the resulting audio net bitrate is $2 \times 44,100 \times 16 = 1.41$ Mb/s. In reality, the CD standard has a large overhead bitrate due to 49-bit representation of every 16-bit sample. The resulting total bitrate is thus equal to $(49/16)\ (1.41) = 4.32$ Mb/s.

PCM representation is not an efficient method for high-quality audio. In order to reduce the required bitrate, various data compression and coding techniques can be used. Simple but not very efficient approaches preserve the signal waveform and are therefore referred to as lossless coding techniques (Section 11.8). Data compression facility of lossless audio coders is rather moderate. Average achievable bit per sample values are only slightly greater than 4.5 bps [6]. Sophisticated techniques, which are still subject of an intensive research, allow for a drastic reduction of this value—at least by one order of magnitude. These coding techniques are lossy in the sense that they corrupt the signal; however, this corruption can be controlled in such a way that it is inaudible. Such audio coders are called transparent (Section 11.9). In order to efficiently and transparently compress audio or speech, the knowledge about the speech and audio production (the parametric audio coding discussed in Section 11.3 instead of the classic waveform audio coding) as well as the knowledge concerning the human auditory perception (discussed in Section 11.4, resulting in the perceptual audio coding) should be exploited (Figure 11.1).

First, efficient transparent or nearly transparent audio codecs that spread around due to Internet applications were based on waveform perceptual coding schemes. Among them are MPEG-1 layer III standard (commonly known as MP3) and MPEG-4 AAC (advanced audio codec) standard (see Section 11.10). The first one provides compression of factor 10 and the latter around 15. Parametric audio codecs, such as SSC and SBR (see Section 11.3), beat these results by more than three times. It is believed that by optimization of parametric modeling of audio signals, further improvement of the compression factor will still be possible.

## 11.6 Digital Audio Signal Processing Systems

Fast development of very large-scale integration (VLSI) electronic chips (digital signal processors [DSPs], field programmable gate arrays [FPGAs], ADCs and DACs, audio codecs, sampling rate converters, etc.) gives possibility for effective digital processing of audio signals [10,44,45].

Digital audio processing equipment can generally be divided into three groups:

- High-performance audio processing systems (e.g., high-end consumer audio devices, professional mixers)
- Personal audio devices (e.g., car audio systems, musical instruments, multitrack recorders)
- Portable audio devices (e.g., MP3 and AAC players, toys, and handheld game players)

A simplified scheme of a typical digital audio processing system is shown in Figure 11.15. Its heart is usually a DSP. At present, several world-leading companies manufacture various DSPs for audio. It should be mentioned that they in particular offer:

- General-purpose (fully) programmable digital signal processors (PDSPs)
- Processor cores, which can be used to design the customer own chips
- Application-specific standard products (ASSPs), e.g., the specialized audio DSPs

Among the most popular manufacturers of PDSPs are Texas Instruments, Analog Devices, Cirrus Logic, Freescale, LSI Logic, Microchip, NEC, Renesas, and others.

Comparison of selected fixed-point and floating-point processors is presented in Table 11.2 [46–48]. It should be taken into account that particular types of processors undergo fast changes (alike in the common computer trade). Independent benchmark analysis of DSPs is performed by specialized companies, e.g., Berkley Design Technology Inc. (www.bdti.com).

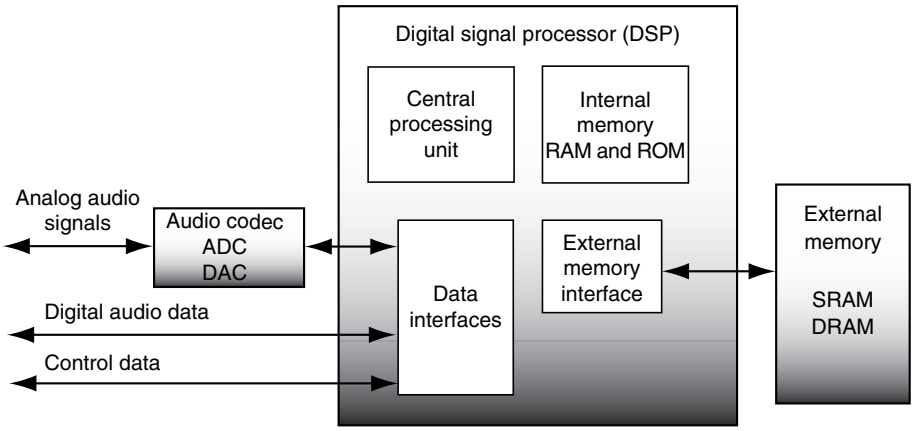



**FIGURE 11.15** A digital audio signal processing system.

**TABLE 11.2** Main Features of Selected Programmable DSPs

| Processor Family | Arithmetic Type | Data Formats (bits) | Clock Speed (MHz) | Peak Performance | On-Chip Memory (kB) |
|---|---|---|---|---|---|
| ADSP-BF53x Blackfin | Fixed point | 16 | 400–750 | 800–1500 MIPS | 52–308 |
| ADSP-213xx SHARC | Fixed point/ floating point | 32/40 | 200–400 | 1200–2400 MFLOPS | 320–1024 |
| ADSP-TS20x TigerSHARC | Fixed point/ floating point | 8/16/32/40 | 500–600 | 3000–3600 MFLOPS | 512–3072 |
| DSP563xx | Fixed point | 24 | 120–180 | 120–180 MIPS | 33–648 |
| TAS3108 | Fixed point | 28/32 | 135 | 675 MIPS | 48 |
| TMS320C54x | Fixed point | 16 | 80–160 | 80–160 MIPS | 20–336 |
| TMS320C55x | Fixed point | 16 | 160–300 | 320–600 MIPS | 64–352 |
| TMS320C62x | Fixed point | 16/32/40 | 167–300 | 1336–2400 MIPS | 72–896 |
| TMS320C64x | Fixed point | 8/16/32/40/64 | 400–1000 | 3200–8000 MIPS | 160–1056 |
| TMS320C67x | Floating point | 32 | 150–300 | 900–1800 MFLOPS | 72–672 |

A DSP can operate with fixed-point or with floating-point arithmetic. Although floating-point processors typically with 32-bit IEEE-754 standard (or even 64-bit double-precision format), e.g., the TMS320C67xx processors [48], offer very effective calculation performance, fixed-point processors are also often used, because they are characterized by a much lower supply power consumption (less by about one decimal order of magnitude than that of the floating-point processors), furthermore, because of a very low standby power (e.g., 0.12 mW for the TMS320C5509 processor), and last but not least, because fixed-point processors are much cheaper [48].

Using facilities of modern DSPs, not only sophisticated digital filters and filter banks but also whole audio multichannel compression/decompression systems can usually be realized in real time with a single processor. Real-time implementation of DSP algorithms is possible due to many important features of modern DSPs. Among the most important are the following:

- Hardware multiplier and long accumulator
- Harvard type and multibus architecture
- An on-chip memory with no additional wait-state cycles (including cache memory)

- Bit-reversed addressing used in fast Fourier transformation (FFT) algorithms
- Circular buffers—a key feature of many DSP routines (e.g., in the realization of FIR digital filters)
- Very long instruction word (VLIW) architecture
- Flexible data flows as single instruction multiple data (SIMD), multiple instruction single data (MISD), and multiple instruction multiple data (MIMD)
- Multiple execution units
- Integration of the complex instruction set computer (CISC) DSP core with a reduced instruction set computer (RISC) microcontroller core in a single chip

It can be observed that the increase of the computational efficiency of new DSPs is often achieved by means of multiple execution units, which are comprised of ALUs, multiplier, and adder/subtracter (for address generation), and data registers (important, e.g., for storing temporary data). As illustrative examples of architectural concepts, which are computationally very efficient, can, e.g., serve

- Texas Instruments VelociTI, which is an advanced VLIW architecture [48]
- Analog Devices Super Harvard Architecture (SHARC) with a SIMD facility [46]

Effective utilization of these highly parallel architectures needs an efficient C-compiler and an efficient assembly optimizer.

As is shown in Figure 11.15, the overall efficiency of a digital audio system depends on effective communication of the DSP with external peripheral units [46–48]. An appropriate digital data interface set can consist of the following:

- Parallel host port interface of 8/16/32-bit width
- Inter-integrated circuit ($I^2C$) interface
- Digital audio transmitter (e.g., S/PDIF, IEC958, or AES/EBU)
- Multichannel buffered serial port
- Asynchronous sampling rate converter

All the above interfaces together with the external memory interface, which is an intelligent controller for glueless connection with external SRAMs and DRAMs, operate independently from the CPU of the DSP and reduce the overall number of the system components.

An interesting feature of some DSPs is integration of the DSP core with the microcontroller core in a single chip. An example is the Texas Instruments OMAP59xx processor, which consists of TMS320C55x DSP and ARM9 microcontroller [48].

Besides the above discussed functional features of the DSPs, a very important issue for the designers is a convenient programming environment, which should help in implementation of basic audio processing algorithms. Examples of such DSP environments are Code Composer Studio [48] and VisualDSP++ [46]. DSP producers offer also specialized graphical user interface (GUI) environments, which streamline the design of audio systems (e.g., the VisualAudio Designer [46]).

DSPs are often offered as a complete solution together with an on-chip ROM containing, e.g., audio decoding algorithms such as Dolby, DTS, MPEG, or WMA (SHARCMelody [46]).

## 11.7 Audio Processing Basics

### 11.7.1 DFT, DCT, and Related Transformations

Discrete Fourier transformation (DFT) is a powerful tool for the analysis of discrete-time signals. A block of $N$ samples $x(n)$, $n = 0, 1, \ldots, N-1$, is considered and its harmonic components are extracted

under assumption that they also describe an infinitely long, block wise periodic extension of signal $x(n)$. DFT is defined as follows:

$$X(k) = \frac{1}{N}\sum_{n=0}^{N-1} x(n) W_n^{kn}, \quad k = 0, 1, \ldots, N-1 \tag{11.31a}$$

where $W_N = e^{j(2\pi/N)}$. The inverse DFT (IDFT) is then given by

$$x(n) = \sum_{k=0}^{N-1} X(k) W_N^{-nk}, \quad n = 0, 1, \ldots, N-1 \tag{11.31b}$$

Computation of DFT and IDFT is usually realized using the so-called FFT algorithms, which reduce the computational complexity of DFT from $\propto N^2$ to $\propto N \log N$ under assumption that $N = 2^K$, where $K$ is a natural number. Two main FFT types can be distinguished: decimation in time and decimation in frequency [22,27,49,50].

Assuming a typical DSP, realization of an FFT of the length $N = 512$ requires about 200 words of the program memory and $4\ N + 1050$ words of data memory. Using a fixed-point DSP, the number of necessary instruction cycles is about 33,000. Assuming a moderate instruction cycle period of 25 ns, sampling rate of 44.1 ksamples/s ($T_s = 22.676$ μs). Accumulation time of 512 input samples is $512 \times T_s = 11.61$ ms. FFT analysis takes $33{,}000 \times 25$ ns $= 0.825$ ms only. This example shows that even a multichannel "online" audio range FFT analysis is easily possible with common DSPs [51].

A block of $N$ samples $x(n)$ can be mirrored before it is periodically extended. This results in the so-called DCT. Because of the mirroring symmetry, DCT gives sharper spectrum than DFT. This is the main advantage of this transformation.

In perceptual audio coders, signals are often mapped into the frequency domain by means of the so-called MDCT [25]. This is a type of DCT with overlapped power complementary time windows (Figure 11.16). By this means, blocking and time-aliasing effects get cancelled. Denote by $x_l(n)$, $n = 0, 1, \ldots, N-1$, time-domain signal samples in the $l$th block of $N$ samples. MDCT is defined as

$$X_l(k) = \sum_{n=0}^{N-1} w(n) x_l(n) \cos\left[\frac{\pi}{2N}\left(2n + 1 + \frac{N}{2}\right)(2k+1)\right] \quad \text{for } k = 0, 1, \ldots, \frac{N}{2} - 1 \tag{11.32a}$$

where

$X_l(k)$ are samples in the frequency domain
$N$ is the number of input samples
$N/2$ is the frequency-domain blocklength
$w(k)$ is the time window function

Division of the input signal into MDCT block is quite flexible. A long block can be split into shorter blocks. Figure 11.16 presents possible transitions MDCT windows between long and short block modes. The length of the block depends on stationarity of the input signal.

The respective inverse discrete cosine transform (IMDCT) is defined as follows:

$$y_l(n) = w(n)\frac{N}{4}\sum_{k=0}^{N/2-1} X_l(k) \cos\left[\frac{\pi}{2N}\left(2n + 1 + \frac{N}{2}\right)(2k+1)\right] \quad \text{for } n = 0, 1, \ldots, N-1 \tag{11.32b}$$

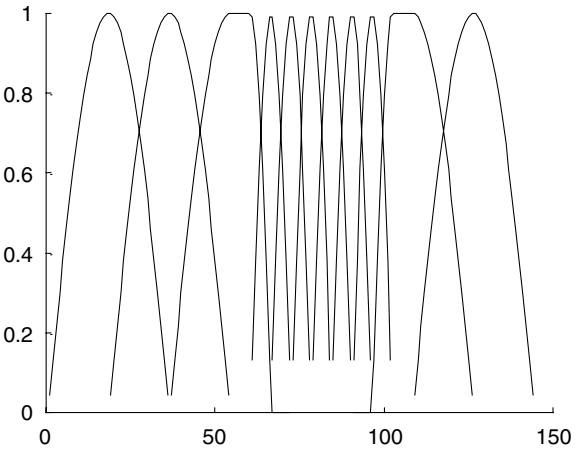



**FIGURE 11.16** Overlapped, power complementary MDCT windows.

The input signal is recovered with an overlap and add operation

$$\tilde{x}_l(n) = y_{l-1}\left(n + \frac{N}{2}\right) + y_l(n) \quad \text{for } n = 0, 1, \ldots, \frac{N}{2} - 1, \tag{11.32c}$$

which cancels the time-domain aliasing.

## 11.7.2 Discrete Wavelet Transformation

One of the newest mathematical tools, which is widely used in the area of audio signal processing, and which is, in fact, a generalization of an octave filter bank approach, is the so-called DWT [23,27,49,50,52]. To introduce the DWT concept, it should first be noticed that the signal $x(t)$ can often be expressed as a linear expansion

$$x(t) = \sum_m c_m \psi_m(t) \tag{11.33}$$

where

$m$ is an integer index
$c_m$ are the real-valued expansion coefficients (parameters describing the signal)
$\psi_m(t)$ is a set of real-valued functions of time $t$ called the expansion set

The expansion set is called *basis*, if the representation in Equation 11.33 is unique, i.e., if functions $\psi_m(t)$ are linearly independent. The most interesting case is the orthogonal or even orthonormal basis. For example, in a Fourier series, the orthogonal basis functions $\psi_m(t)$ are cos $k\omega_0 t$ and sin $k\omega_0 t$, where $\omega_0$ is related to the signal period $T$ according to equation $\omega_0 = 2\pi/T$.

For the DWT, a two-dimensional set of coefficients $a_{kl}$ is constructed such that

$$x(t) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} 2^{k/2} a_{kl} \psi(2^k t - l) \tag{11.34}$$

where the function $\psi(t)$, called *wavelet* (concentrated or short wave), generates the expansion set $\psi(2^k t - l)$, which is an orthogonal basis. The wavelet $\psi(t)$ is an oscillating and quickly decaying function, which has its energy sufficiently well localized in time and in frequency. Several different wavelet classes have already been proposed [27].

Introducing another basic function $\psi(t)$, called the scaling function, a multiresolution signal representation, starting from some resolution $k$, can be formulated:

$$x(t) = \sum_{l=-\infty}^{\infty} 2^{k/2} b_{kl} \varphi(2^k t - l) + \sum_{\kappa=k}^{\infty} \sum_{l=-\infty}^{\infty} 2^{\kappa/2} a_{\kappa/} \psi(2^{\kappa} t - l) \tag{11.35}$$

Two fundamental self-similarity equations have to be fulfilled:

$$\varphi(t) = \sum_{n} \sqrt{2} h_0(n) \varphi(2t - n) \tag{11.36a}$$

$$\psi(t) = \sum_{n} \sqrt{2} h_1(n) \varphi(2t - n) \tag{11.36b}$$

where $h_0(n)$ and $h_1(n)$ are impulse responses of two discrete-time complementary filters—a lowpass filter and a highpass filter, respectively. For a finite even length $N$, responses $h_0(n)$ and $h_1(n)$ are related to each other by the following equation:

$$h_1(n) = (-1)^n h_0(N - 1 - n), \quad n = 0, \ldots, N - 1 \tag{11.37}$$

If resolution $k$ is large enough, $\kappa = k$ can only be taken into account in Equation 11.35. In other words, we can assume that

$$x(t) = \sum_{l=-\infty}^{\infty} 2^{k/2} b_{kl} \varphi(2^k t - l) + \sum_{l=-\infty}^{\infty} 2^{k/2} a_{kl} \psi(2^k t - 1) \tag{11.38}$$

Thus, from Equations 11.36a and 11.36b, we conclude that $x(t)$ is a signal of resolution $k+1$ and can be modeled as

$$x(t) = \sum_{n=-\infty}^{\infty} 2^{(k+2)/2} b_{(k+1)n} \varphi(2^{k+1} t - n) \tag{11.39}$$

It should be stressed that scaling coefficients $b_{(k+1)n}$ in Equation 11.39 approximate signal samples, i.e., $x_n \approx b_{(k+1)n}$, because for high enough scale $k+1$, the scaling functions $\varphi(2^{k+1} t - n)$ act as delta functions. In this case, sampling period $T_s = 1/2^{k+1}$.

Assuming that functions $\varphi(2^k t - l)$ and $\psi(2^k t - l)$ in Equation 11.38 form an orthonormal basis, after some manipulations, we conclude that

$$b_{kl} = \sum_{n} h_0(n - 2l) b_{(k+1)n} \tag{11.40a}$$

$$a_{kl} = \sum_{n} h_1(n - 2l) b_{(k+1)n} \tag{11.40b}$$

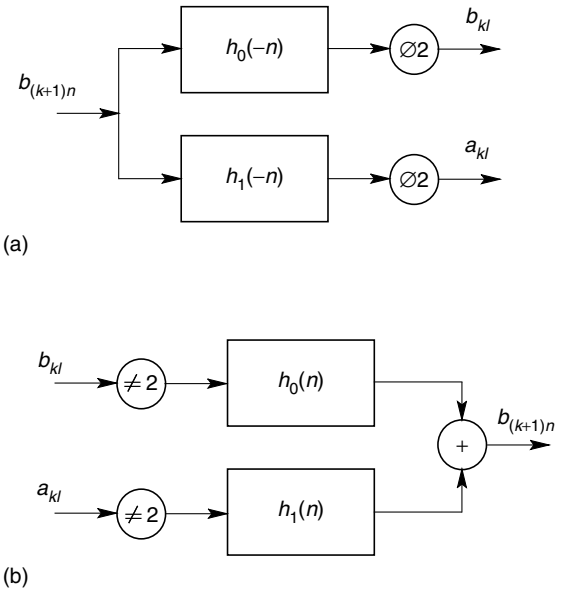



**FIGURE 11.17** Two-band filter bank: (a) analysis bank and (b) synthesis bank.

If the signal $x(t)$ is of finite duration, the sums in Equations 11.38 and 11.39 are finite. The sets $\{b_{(k+1)n}\}$ and $\{b_{kl}, a_{kl}\}$ form alternative parametric descriptions for the signal $x(t)$. Although both sets have the same number of parameters, description $\{b_{kl}, a_{kl}\}$ is somehow more efficient because parameters $a_{kl}$ are less important than $b_{kl}$ and consequently $\{b_{(k+1)n}\}$. From Equations 11.40a and 11.40b, we conclude that parameters $b_{kl}$ and $a_{kl}$ result from a lowpass and a highpass filtering of parameters $b_{(k+1)n}$, respectively, with a two-band splitting filter bank of impulse responses $h_0(-n)$ and $h_1(-n)$, respectively, followed by a downsampling with factor 2 (Figure 11.17a). This procedure can be continued many times to obtain even more efficient parametric representation. If only parameters $b_{kl}$ are split, which is the case in the classical DWT, a kind of an octave signal analysis filter bank results (Figures 11.18a and 11.19). If also

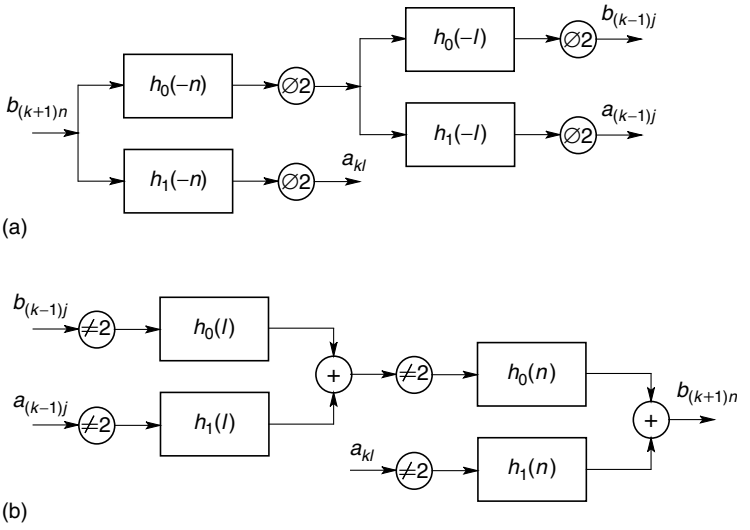



**FIGURE 11.18** Discrete wavelet transformation (DWT): (a) two-band analysis tree and (b) two-band synthesis tree.

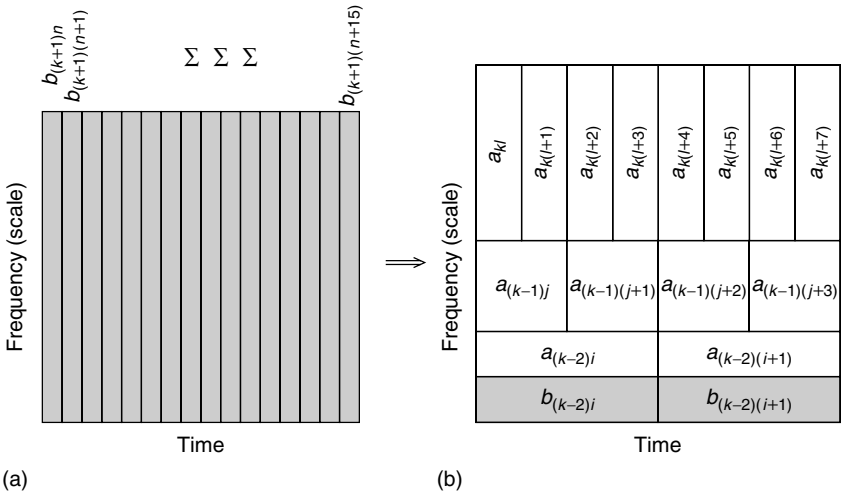



**FIGURE 11.19** Time–frequency (scale) signal representation patterns: (a) an initial $(k+1)$-resolution scale pattern corresponding to Equation 11.39 and (b) DWT pattern after three transformation steps (the first step is made according to Equation 11.38).

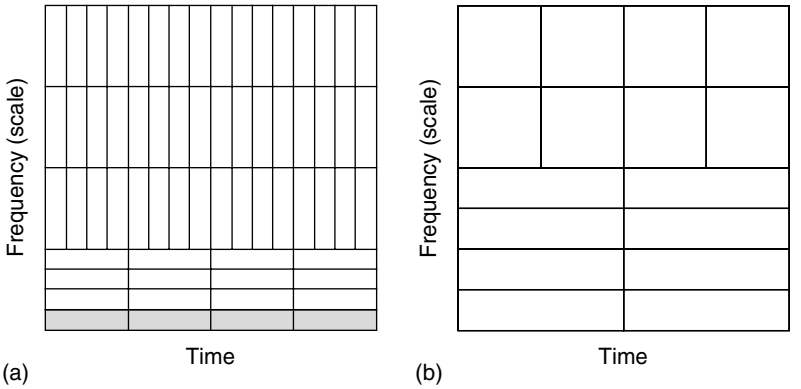



**FIGURE 11.20** Time–frequency (scale) signal representation patterns: (a) a pattern after two transformation steps with a four-band wavelet basis and (b) an example of a two-band wavelet packet transformation.

parameters $a_{kl}$ are split (wavelet packet) and a multiband splitting filter bank is used (multiband wavelet system) [27], very flexible analysis filter banks can be realized (Figure 11.20), e.g., those simulating along the frequency axis the distribution of a set of nonoverlapping peripheral auditory filters (see Section 11.4).

### 11.7.3 FIR Filters

Basic operation in digital audio signal processing is frequency selective filtering. It can be realized in frequency domain, e.g., using FFT and in time domain using finite impulse response (FIR) and IIR filters [22,49,50,53]. Mostly FIR filters are used because they are always stable and can be easily designed with a perfect linear phase characteristic.

**TABLE 11.3** Impulse Response of Ideal Filters

| Filter Type | Impulse Response | |
|---|---|---|
| Lowpass | $h_d(n) = \begin{cases} \frac{\sin(n\omega_c)}{n\pi} \\ \frac{\omega_c}{\pi} \end{cases}$ | $n = 0$<br>$n = 0$ |
| Highpass | $h_d(n) = \begin{cases} \frac{\sin(n\omega_c)}{n\pi} \\ 1 - \frac{\omega_c}{\pi} \end{cases}$ | $n = 0$<br>$n = 0$ |
| Passband | $h_d(n) = \begin{cases} \frac{\sin(n\omega_2) - \sin(n\omega_1)}{n\pi} \\ \frac{\omega_2 - \omega_1}{\pi} \end{cases}$ | $n = 0$<br>$n = 0$ |
| Stopband | $h_d(n) = \begin{cases} \frac{\sin(n\omega_1) - \sin(n\omega_2)}{n\pi} \\ 1 + \frac{\omega_2 - \omega_1}{\pi} \end{cases}$ | $n = 0$<br>$n = 0$ |

Assuming an ideal filter frequency response given by the following expression:

$$H_d(e^{j\omega}) = \sum_{n=-\infty}^{+\infty} h_d(n) e^{-j\omega n} \tag{11.41}$$

where

$$h_d(n) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} H_d(e^{j\omega}) e^{j\omega n} d\omega \tag{11.42}$$

The respective FIR filter impulse response is given by the following expression:

$$h_{\mathrm{FIR}}(n) = w_N(n) h_d(n) \tag{11.43}$$

where $w_N(n)$ is a specially selected time window (e.g., Hanning, Hamming, Blackman, or Kaiser window), used to reduce the so-called Gibbs phenomenon [22]. Depending on filter type, the ideal filter coefficients can be calculated using equations listed in Table 11.3.

Another, more advanced, method for the design of FIR filters, is an optimization procedure developed by Parks and McClellan (also known as the Remez algorithm). This method is implemented in the MATLAB environment with two functions: *remezord* to estimate the filter order and *remez* to compute the filter coefficients [52]. This optimization method should be used, if a relatively high stopband attenuation is required, e.g., with a 20-bit resolution for representation of signal samples, we usually need a stopband attenuation of approximately 120 dB. As a design example, Figure 11.21 presents the frequency response of a lowpass FIR filter designed with the Parks–McClellan method, with the normalized cutoff frequency of $\pi/64$. This filter can be used as a prototype filter for the design of analysis and synthesis filter banks for audio coders, according, e.g., to the MUSICAM and MPEG-1 standards.

Using modern DSPs, FIR filters can easily be implemented with the MACD instruction, which realizes multiplication, accumulation, and data move. When used with repeat next instruction (RPT), MACD

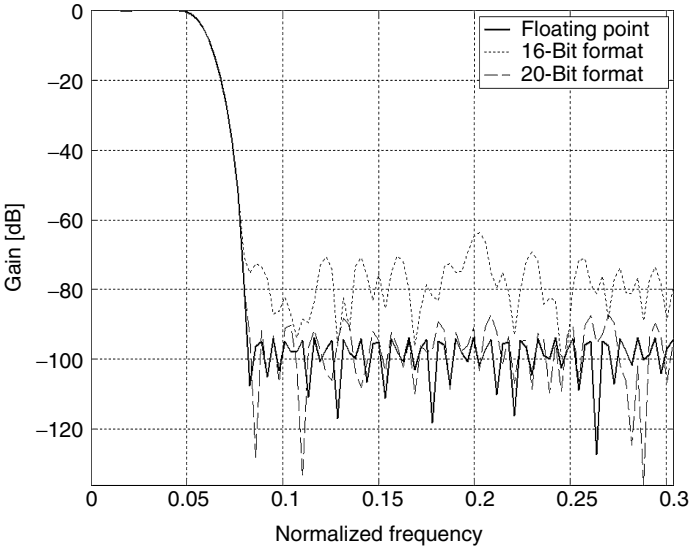



**FIGURE 11.21** Frequency response of an FIR lowpass filter designed with the Parks–McClellan method.

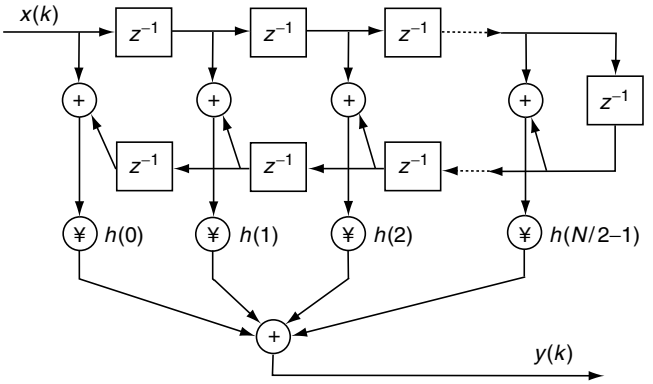



**FIGURE 11.22** Symmetrical FIR filter.

becomes a single-cycle instruction once the RPT pipeline is started. The theoretical maximum length $L$ of the FIR filter can be computed as

$$L = \frac{T_S}{T_c k} \tag{11.44}$$

where
$T_s$ is the sampling period
$T_c$ is the instruction cycle
$k$ is the number of converted channels

Assuming sampling rate of 48 ksamples/s, 25 ns instruction cycle of the DSP and six output channels, we can realize FIR filters with the maximum length of about 138.

Another method for the implementation of FIR filters in DSPs consists in the use of two further new features of modern DSPs, namely the circular addressing and the FIRS instruction. This possibility can be effectively used, if the filter has a symmetric impulse response $h(n)$ (see Figure 11.22), i.e., if the filter output signal is given by

$$y(n) = \sum_{k=0}^{N/2-1} h(n)\{x(n-k) + x[n-(N-1-k)]\} \tag{11.45}$$

The FIRS instruction can add two data values (stored in a circular buffer) in parallel with the multiplication of this result by a filter coefficient. Once the repeat pipeline is started, this instruction becomes also a single-cycle instruction. A computational complexity is in this case reduced by half and makes it possible to realize FIR filters with the double length as compared with the programming technique previously described.

### 11.7.4 IIR Filters

Although FIR filters have important advantages as linear phase, stability, robustness, easy design, and implementation, their IIR counterparts will have complexity (the transfer function degree) reduced by some decimal orders of magnitude [53]. Therefore, IIR filters are advantageous over and above FIR filters in particular applications. IIR filters are typically designed starting with an analog reference filter and then performing the bilinear transformation [22]. Transfer function of the analog reference filter is denoted by $H(s)$. Then the resulting IIR filter transfer function $H(z)$ is calculated as

$$H(z) = H(s)\big|_{s=\frac{2}{T_S}\frac{z-1}{z+1}} \tag{11.46}$$

where $T_s$ is the sampling period. The respective transformation of the analog frequency $\omega_a$ into the digital frequency $\omega_d$ is given by

$$\omega_d = \frac{2}{T_s}\arctan\left(\frac{T_S}{2}\omega_a\right) \tag{11.47}$$

It is reasonable to apply the bilinear transformation in Equation 11.46 directly to elements of the analog reference filter rather than to its transfer function, thus preserving the ladder or the lattice analog circuit structure. The resulting digital signal flow graphs realize the so-called WDFs [27,48,53], which are famous, because of robustness in stability, small sensitivity to coefficient changes, low complexity, and great dynamic range.

## 11.7.5 Filter Banks

A filter bank is a collection of digital filters with a multiple input and a multiple output [23,50]. The filter bank with one input and $M$ outputs is referred to as the analysis filter bank. On the other hand, the synthesis filter bank consists of $M$ inputs and one output (see Figure 11.23). Splitting of the input signal into decimated subbands via an analysis filter bank and then reconstructing the initial signal from subband signals with a respective synthesis filter bank is referred to as the subband coding (SBC) technique commonly used for nearly lossless data compression.

A filter bank in the main path of the MPEG-1 audio coder [54] consists of 32 subband filters with a normalized bandwidth of $\pi/(32T_s)$, where $T_s$ is the input audio signal sampling period. The impulse responses of particular filters in this filter bank are defined as

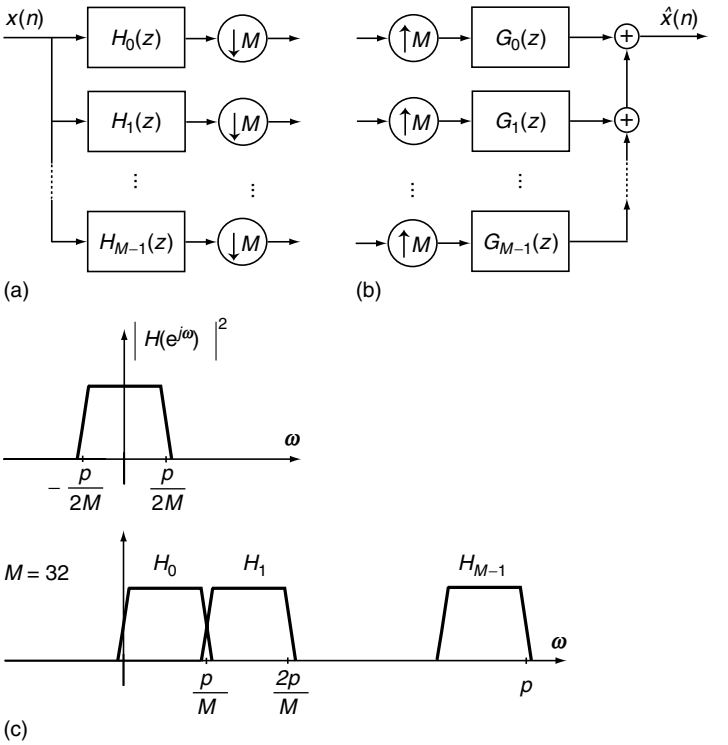



**FIGURE 11.23** Filter banks: (a) analysis filter bank, (b) synthesis filter bank, and (c) subband pattern.

$$H_i(n) = h(n)\cos\left[\frac{(2i+1)(n-16)\pi}{64}\right] \tag{11.48}$$

where $h(n)$ is an impulse response of the prototype lowpass filter. In the analysis filter bank, the output signal in $i$th subband is defined as a convolution:

$$S_i(m) = \sum_{n=0}^{511} x(m-n)H_i(n) \tag{11.49}$$

In the MPEG-1 encoder, an efficient polyphase filter bank realization is implemented, using the following steps:

1. Thirty-two new input samples $x(n)$ are shifted into a 512-point FIFO buffer.
2. Five hundred and twelve samples $x(n)$ are multiplied by the modified (the so-called analysis window) coefficients $C(n)$.

$$Z(n) = C(n)x(n) \tag{11.50a}$$

   where $C(n) = -h(n)$ if the integer part of $n/64$ is odd, otherwise $C(n) = h(n)$, $n = 0, 1, \ldots, 511$.

3. An intermediate result is calculated as follows:

$$Y(k) = \sum_{j=0}^{7} Z(k+64j) \quad \text{for } k = 0, 1, \ldots, 63 \tag{11.50b}$$

4. Thirty-two new output samples are computed

$$S_j = \sum_{k=0}^{63} M_i(k)Y(k) \quad \text{for } i = 0, 1, \ldots, 31 \tag{11.50c}$$

   where $M_i(k) = \cos\{[(2i+1)(k-16)\pi]/64\}$ are the modulation (or analysis) matrix coefficients.

## 11.7.6 Sampling Rate Conversion

Currently, digital audio signals are used with various sampling rates. Typical values are 8, 16, 22.05, 32, 44.1, 48, and even 96 ksamples/s. Thus, an online sampling rate conversion is a very important task in digital audio signal processing algorithms [42]. This task can nowadays be realized using digital signal processors [55] or specialized chips [10,46]. Generally, three different approaches are possible:

- Natural approach based on, first, interpolation with integer factor $M$, and then, decimation with integer factor $N$ (Figure 11.24)
- Time-domain approach based on direct interpolation (or decimation) in time, i.e., on the realization of a sequence of noninteger delays (Figure 11.25)
- Frequency-domain approach based on, first, a blockwise DFT; second, on respective spectrum modification in each block—a throw-in of zero spectral samples in the middle of the DFT spectrum for interpolation or cutting out of some of spectral samples in the middle of the DFT spectrum for decimation (see Figure 11.26); and third, the backward

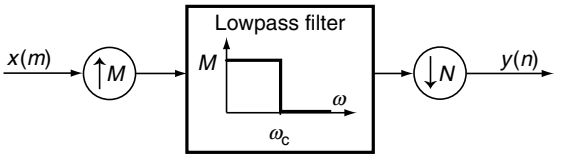



**FIGURE 11.24** Basic system for sampling rate conversion.

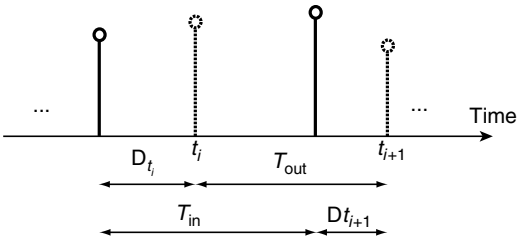



**FIGURE 11.25** Time relationships between input and output samples.

blockwise IDFT transformation (refer to the interpft function in the MATLAB environment [52])

A scheme of the simplest system for the synchronic sampling rate conversion is shown in Figure 11.24. The output samples are calculated using difference equations, which utilize the up- and downsampling and the filtering in between. Table 11.4 presents the respective up- and downsampling factors for the considered sampling rate conversions. These factors are equal to the least common multiple of a pair of sampling rates (the input and the output sampling rates), divided by the respective sampling rate. Because the up- and downsampling factors for the rate conversion to and from 44.1 ksamples/s are inadmissibly large, a slightly lower sampling rate, namely 44 ksamples/s (see Table 11.5) can usually be accepted. This would introduce a small, inaudible, error with a relative value of $\delta = 0.22676\%$.

The filtering operation between an interpolator and a decimator should be realized via a lowpass filter with gain $M$ and the normalized cutoff frequency $\omega_c = \min(\pi/M, \pi/N)$ [23]. The respective FIR filter can be designed, e.g., using the Parks–McClellan method. Depending on the converted rates and the desired signal resolution (16 or 20 bits, corresponding to the stopband attenuation of 96 or 120 dB, respectively) the required length $L$ of FIR filters varies between 154 and 198. Depending on the upsampling coefficient $M$, the number of effective filter taps, which have to be calculated, is reduced to $L/2M$.

Sampling rate alteration using the time-domain approach can be applied in asynchronic systems. An output signal sample can be determined using the following relationship:

$$\Delta t_{i+1} = (\Delta t_i + T_{\text{out}}) \bmod T_{\text{in}} \tag{11.51}$$

where

$T_{\text{in}}$ is the input sampling interval
$T_{\text{out}} = t_{i+1} - t_i$ is the output sampling interval
$t_{i+1}$ is the instant in which a new output sample should occur

The above relationships are illustrated in Figure 11.25. Input samples are indicated with solid lines and output samples with dotted lines.

One of the simplest time-domain sampling rate conversion methods is a high oversampling and then choosing appropriate output samples (those, which are the nearest to the required positions in time). A multistage approach of this type is illustrated in Figure 11.27 [56]. Interpolators with factors 64 and 128 are controlled by a time-analysis unit, which measures the ratio between the input and the output sampling rates. An advantage of this method (in comparison with the natural method) is the possibility for the use of the same filter coefficients for different sampling rate conversion ratios, and thus, a simplified realization of the interpolation filters.

Time-domain conversion can also be based on various numerical methods, e.g., on polynomial interpolation. Lagrange interpolation or spline interpolation can effectively be used [56,57]. In the case of an $N$th-order spline with a function defined in interval $[x_k, \ldots, x_{k+m}]$ as

$$M_k^N(x) = \sum_{i=k}^{k+m} a_i \phi_i(x) \tag{11.52}$$

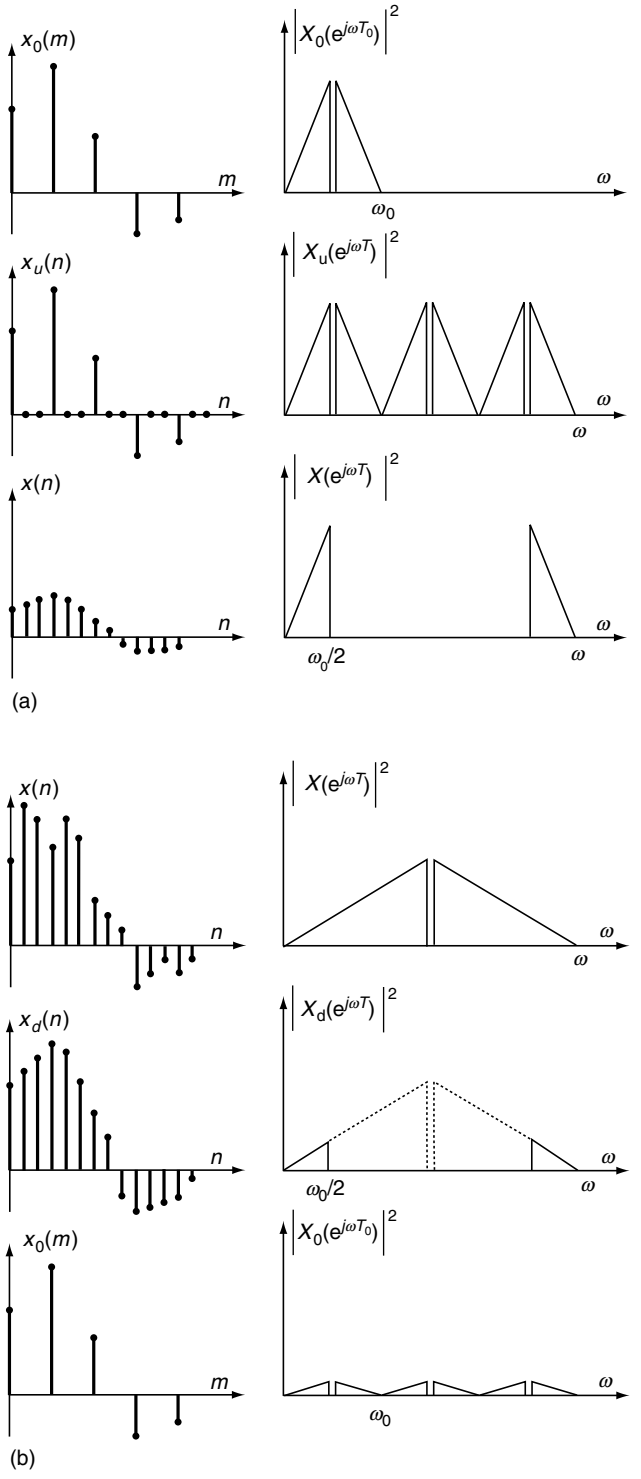



**FIGURE 11.26** Sampling rate conversion in frequency domain: (a) interpolation and (b) decimation. These first two approaches can be mixed, resulting in substantial reduction of the required intermediate sampling rate.

**TABLE 11.4** Sampling Rate Conversion Factors

| Sampling Rate Conversion [ksamples/s] | Upsampling Coefficient $M$ | Downsampling Coefficient $N$ | Least Common Multiple of a Pair of Sampling Rates |
|---|---|---|---|
| 16 → 48 | 3 | 1 | 48,000 |
| 32 → 48 | 3 | 2 | 96,000 |
| 16 → 44.1 | 441 | 160 | 7,056,000 |
| 32 → 44.1 | 441 | 320 | 14,112,000 |
| 44.1 → 48 | 160 | 147 | 7,056,000 |

**TABLE 11.5** Sampling Rate Conversion to/from 44 ksamples/s

| Sampling Rate Conversion [ksamples/s] | Upsampling Coefficient $M$ | Downsampling Coefficient $N$ |
|---|---|---|
| 16 → 44 | 11 | 4 |
| 32 → 44 | 11 | 8 |
| 44 → 48 | 12 | 11 |

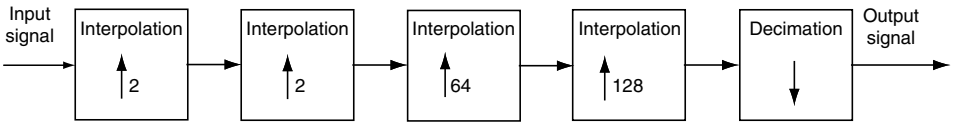



**FIGURE 11.27** Sampling rate conversion with multistage oversampling.

where

$$\phi_i(x) = (x - x_i)_+^N = \begin{cases} 0 & x < x_i \\ (x - x_i) & x \geq x_i \end{cases}$$

a sixth-order interpolation is used with a simple FIR filter to compensate $\text{sinc}^7$ distortion in the frequency domain caused by the spline interpolator [56].

## 11.8 Lossless Audio Coding

### 11.8.1 Pulse Code Modulation

The most typical digital waveform coding is the PCM, in which a stream of uniformly distributed digitally coded samples, which represent a given analog continuous-time signal is used. Basic PCM coder consists of an antialiasing filter, sampling device, and a quantizer. In practice, to improve the subjective audio quality, the quantizer should have a nonlinear (logarithmic) characteristic based on, e.g., a 13-segment A-law or a 15-segment μ-law used in telephone systems [49]. The normalized characteristics are given by

- A-law

$$f(x) = \begin{cases} \frac{Ax}{1+\ln A} & 0 \leq x \leq \frac{1}{A} \\ \frac{1+\ln Ax}{1+\ln A} & \frac{1}{A} \leq x \leq 1 \end{cases} \tag{11.53}$$

- μ-law

$$f(x) = \frac{\ln(1+\mu x)}{\ln(1+\mu)} \quad \text{for } 0 \le x \le 1 \tag{11.54}$$

For the compression from 16 to 8 bits, typical values of the coefficients are $A = 87.6$ and $\mu = 255$.

In most cases, PCM bitstream has highly redundant information. Thus, using a number of previous samples of the input signal, we can predict the next sample with a relatively small error. This feature is used in differential pulse code modulation (DPCM), in which a difference between input sample and its estimation is coded. The prediction is realized with appropriate FIR filter. In the case, in which the statistics of the input signal changes in time is unknown, the prediction should be made adaptive. An adaptive coding is realized in an adaptive difference pulse code modulation (ADPCM). The respective schemes, i.e., those of the ADPCM encoder and the ADPCM decoder are shown in Figure 11.28.

A special case of the DPCM approach is delta modulation (DM). The DM encoder is very simple to implement because it uses a 1-bit quantizer and a first-order predictor (see Figure 11.29). The encoder is so strongly simplified that high sampling rates are required. Among disadvantages of DM are possible slope overload and granularity noise. Both can, however, be easily reduced by adaptive versions of DM, i.e., ADM.

Continuous variable slope delta modulation (CVSDM) is an example of the ADM. CVSDM effectively reduces the DM slope overload [12]. An interesting advantage of this method is its resistance to transmission errors. Figure 11.30 presents the structure of the CVSDM encoder.

The output signal of the CVSDM encoder is given by

$$y(n) = \text{sgn}\{x(n)\text{-}\hat{x}(n\text{-}1)\} \tag{11.55}$$

where $x(n)$ is the input PCM sample and $(n)$ is the estimated sample.

The parameter $\alpha$ depends on the signal slope, i.e., $J$ bits in $K$ output bits of $y(n)$:

$$\alpha = \begin{cases} 1 & j \text{ bits are the same} \\ 0 & \text{else} \end{cases} \tag{11.56}$$

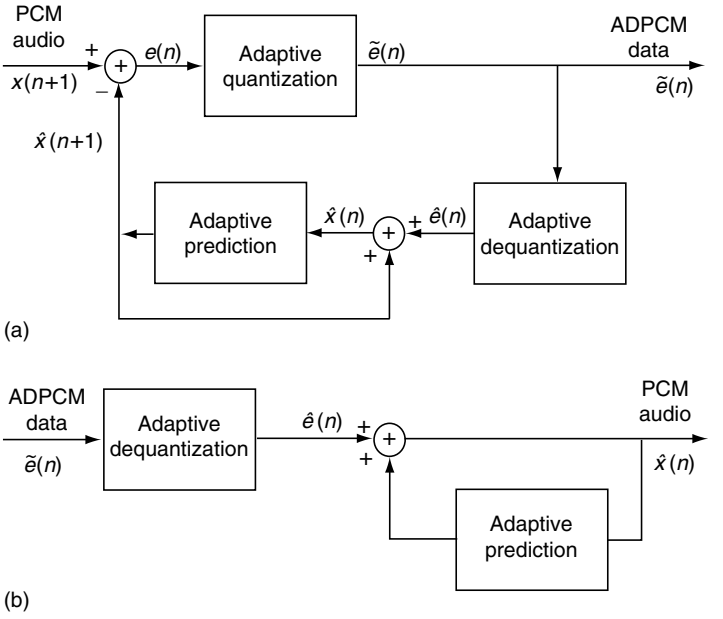



**FIGURE 11.28** ADPCM: (a) encoder and (b) decoder.

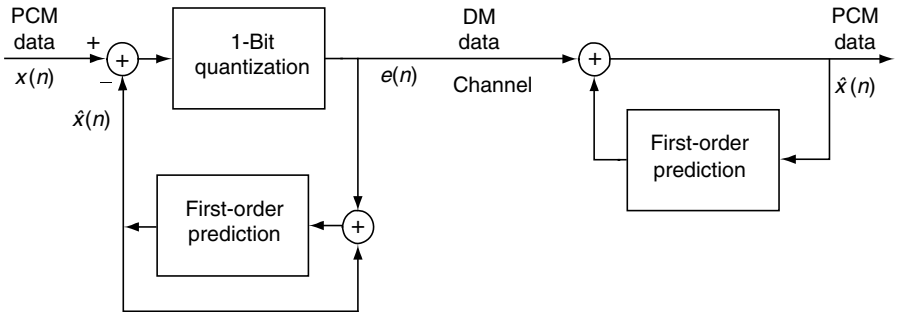



**FIGURE 11.29** Delta modulation system.

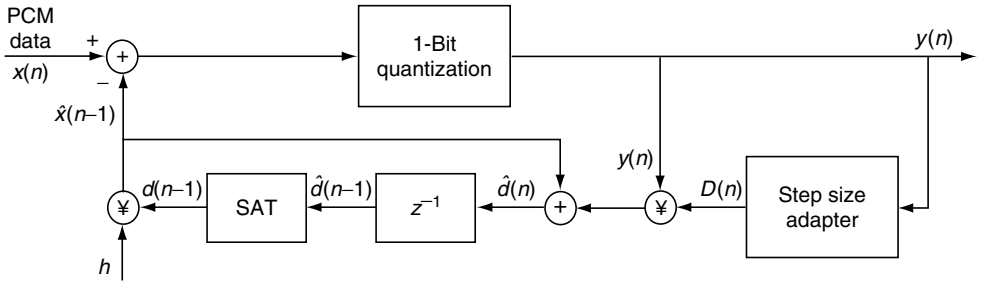



**FIGURE 11.30** CVSDM encoder.

The quantization step $\Delta(n)$ is increased or decreased using parameter $\alpha$:

$$\Delta(n) = \begin{cases} \min\{\Delta(n-1) + \Delta_{\min}, \Delta_{\max}\} & \text{for } \alpha = 1 \\ \max\{\beta\Delta(n-1), \Delta_{\min}\} & \text{for } \alpha = 0 \end{cases} \tag{11.57}$$

where

$\beta$ = step decreasing coefficient
$\Delta_{\min}$ = minimum step size
$\Delta_{\max}$ = maximum step size

The estimated value $\hat{x}(n-1)$ is given by

$$\hat{x}(n-1) = hd(n-1) \tag{11.58}$$

where

$$d(n-1) = \begin{cases} \min\{\hat{d}(n-1), d_{\max}\} & \text{for } \hat{d}(n-1) \geq 0 \\ \max\{\hat{d}(n-1), d_{\min}\} & \text{for } \hat{d}(n-1) < 0 \end{cases}$$

$h$ is the accumulator decay coefficient.

## 11.8.2 Entropy Coding Using Huffman Method

The entropy coding is a lossless bitstream reduction and can be used on its own or as a supplement to other methods, e.g., after the DPCM. This coding approach is based on the statistical redundancy, when the signal samples or sequences (blocks) have different probabilities. The entropy of a signal is defined as the following average:

$$H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{11.59}$$

where $-\log_2 p_i$ is an information of the $i$th code word and $p_i$ is the probability of its occurrence. The most popular method for the entropy coding is the Huffman coding method [58,59], in which the optimal code can be found using an iterative procedure based on the so-called Huffman tree. The Huffman coding is, e.g., used in MPEG-1 audio standard to reduce the amount of output data in layer III. The set of 32 Huffman tables is specially tuned for statistics of the MDCT coefficients (see Section 11.7) divided into some regions and subregions [60].

## 11.9 Transparent Audio Coding

A need for reduction of bitrate required for the transmission of high-quality audio signals draws a growing attention to lossy audio coding techniques. Lossy audio coding will be fully acceptable, if it is perceptually transparent, i.e., if the corruption of the audio signal waveform is inaudible. An efficient transparent audio coding algorithm (Figure 11.1) should

1. Remove redundancy contained in the original audio signal
2. Remove the perceptual irrelevancy

The first task requires an efficient parametric description of audio, e.g., a plausible mathematical model of its production, as the signal should be split into almost uncorrelated components. Although this is a relatively simple task for speech, and at least a conceivable one for music, for general audio signals this is a very complex problem. Therefore, instead of a real audio production model, a compromise solution can be used, namely a general signal analysis model, e.g., in the frequency domain or in the scale-of-resolution domain (see Section 11.7), reducing redundancy (correlation) of the signal components.

In case of waveform audio codecs, this is done by means of a proper analysis filter bank. At the receiver side, the signal components are recombined via the corresponding synthesis filter bank. This procedure should allow perfect or at least nearly perfect signal reconstruction under ideal conditions.

In case of parametric audio codecs, parameters of the signal components are not predefined and are to be searched for with the encoding algorithm.

The second task listed above should be realized by means of a precise psychoacoustic hearing model, which should take all masking effects into account. This subject has been discussed in Section 11.4. The hearing model provides information about the dynamic range, which is necessary for the proper representation of parameters (signal components) contained in the signal analysis model. Thus, it allows for efficient dynamic bit allocation to particular signal parameters or components to guarantee that the quantization noise is inaudible.

A general scheme of the typical transparent audio coder is shown in Figure 11.31. The input audio signal is first analyzed via an analysis filter bank. Taking different possible analysis filter banks into account, state-of-the-art waveform coders can be divided into two historically relevant categories: subband coders (SBCs) and transform coders (TCs). TCs operate usually with much greater frequency resolution than SBCs. In typical SBCs, uniform polyphase analysis filter banks are used (see Section 11.7). On the other hand, TCs typically employ the MDCT [25]. Other types of analysis filter banks,

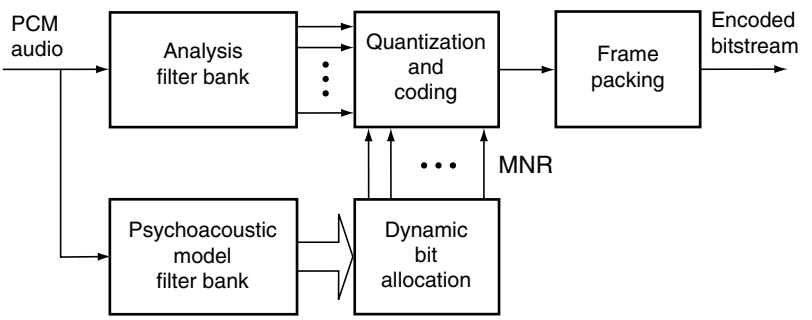



**FIGURE 11.31** General scheme of the transparent audio coder.

e.g., octave filter banks, can also be very efficiently used. They can be realized with the DWT discussed in Section 11.7, or with WDFs [27,48]. Another approach, which is implemented in MPEG-1 layer III audio coder, is a hybrid filter bank, which is combination of a coarse frequency resolution subband filter bank followed by a fine frequency resolution transformation (MDCT in this case).

Parallelly, the input signal is also analyzed with a psychoacoustic model filter bank. This filter bank should estimate a number of nonoverlapping peripheral auditory filters, which cover the whole audible frequency range. In these audio coders, which for parametric signal description exploit octave analysis filter banks, this can be just the same filter bank; however, typically, the psychoacoustic model filter bank is realized separately with the FFT.

Using the psychoacoustic auditory model, a global dynamic threshold of hearing is computed, e.g., by using the way shown in Equation 11.14 or more precisely in Equation 11.66. Then in each analysis filter subband, the respective $\mathrm{SMR}_i$ is computed with Equation 11.67. Finally, the mask-to-noise ratio (MNR) is computed as

$$\mathrm{MNR}_i(m) = \mathrm{SNR}(m) - \mathrm{SMR}_i \tag{11.60}$$

where, $\mathrm{SNR}(m)$, the SNR determined by Equation 11.30, resulting from an $m$-bit quantization is denoted. Within the $i$th critical band, the quantization noise will be inaudible as long as the $\mathrm{MNR}_i(m)$ is positive. This observation can be used for efficient dynamic bit allocation, which can be realized, e.g., with the following procedure:

- The bit allocation unit searches for the analysis filter subband with the lowest MNR and allocates code bits to this subband; then the $\mathrm{SNR}(m)$ value is updated for this subband and the actual MNR is computed with Equation 11.60.
- The process is repeated until no more code bits can be allocated.

An important problem, resulting from the transformation of the audio signal (via an analysis filter bank) into the frequency domain, is the appearance of preechoes, occurring in silent signal periods followed by sudden sound attacks (e.g., of a percussive character). This phenomenon is caused by quantization errors, which are irrelevant in loud and stationary signal parts but are immediately audible in silent signal parts. In TCs, the inverse transform in the receiver distributes the quantization errors over the whole block of samples cut with the respective time window. In SBCs, this effect occurs due to transients. A possible method for suppression of preechos is the adaptive window switching (see Figure 11.16) [9]. Windows of short lengths should be used in nonstationary parts of the signal, whereas in stationary parts of the signal, wide windows (improving the overall coding efficiency) should be used. Typically, the block size varies between $N = 64$ and $N = 1024$.

Further reduction of audio bitrate is still possible by resignation from the full perceptual transparency. In many cases, especially in multimedia or in mobile-access applications, a not annoying reduction of fidelity of some audio components of secondary importance is acceptable. The whole audio scene can be divided into a number of individual audio objects: a conversation, a background noise, a background music, sounds produced by particular sources, etc. These objects can be coded and transmitted separately. Furthermore, some of them may be added synthetically at the receiver. Such coding philosophy is used in the so-called structured audio format implemented in the MPEG-4 standard (see Section 11.10). By this means, a very flexible scalability of audio quality can be realized. This is very useful when audio has to be transmitted through channels of varying capacity or is to be received with decoders of various quality and complexity.

## 11.10 Audio Coding Standards

### 11.10.1 Lossless and Lossy Standards

Despite the popularity of lossy coders, there also exists a group of lossless standards. Selected lossless standards are listed in Table 11.6 [61]. Is should be noticed that average maximum compression factor in this case is only about 50% [62].

**TABLE 11.6** Audio Coders

| Lossless | Lossy |
|---|---|
| Apple Lossless | ADPCM |
| Audio Lossless Coding (MPEG-4 ALS) | AAC & AAC+ (MPEG-2, MPEG-4) |
| FLAC | ATRAC |
| Meridian Lossless Packing | Dolby Digital (AC3) |
| Monkey's Audio | MP3 (MPEG-1 Audio layer III) |
| Shorten | Ogg Vorbis |
| Windows Media Audio | Windows Media Audio |

## 11.10.2 MUSICAM and MPEG Standards

Among standards for digital coding of high-quality audio, the most important role play MPEG standards are designed for various communications and multimedia applications. They are elaborated as a result of efforts of the working group WG 11 within the International Organization for Standardization (ISO/IEC).

The first result was MPEG-1 standard IS 11172 designed (in its audio part) for a 2-channel audio, approximately with a CD quality [54]. This standard consists of three layers—I, II, and III, of increasing efficiency. For transparent transmission, they enable bitrates of 384, 192, and 128 kb/s, respectively. MPEG-1 supports sampling rates of 32, 44.1, and 48 ksamples/s. Layer II of MPEG-1 is based on the masking-pattern universal subband integrated coding and multiplexing (MUSICAM) standard designed for digital audio broadcasting (DAB) system [24,63]. Layer III of MPEG-1 (commonly called MP3) has become very popular in Internet due to very widely spread $*$.mp3 audio files [64].

The next step of the standardization was MPEG-2 AAC standard IS 13818 designed for high-definition television (HDTV) [64,65]. In comparison with MP3, it offers a multichannel (surround) sound for high spatial realism (up to 48 channels), provides low bitrate audio (below 64 kb/s), supports low sampling rates of 16, 22.05, and 24 ksamples/s, and uses higher efficiency and simpler filterbank (pure MDCT instead of hybrid). Higher coding efficiency of AAC in relation to MP3 by factor 1.5 is also result of better selected blocksizes: 1024 samples for stationary signals (MP3: 576 samples) and 128 samples for transient signals (MP3: 192 samples).

SSC coder (Figure 11.32) developed by PHILIPS is one of the first very efficient parametric audio coders (see Section 11.3). Its coding efficiency comparing to that of the MP3 is higher by factor 3.5 [29].

Expanded ACC system is MPEG-4 AAC-Plus, which has two versions:

- Version 1, which is a combination of AAC + SBR
- Version 2, which is a combination of AAC + SBR + PS (parametric stereo)

Version 2 has found important applications, e.g., in DRM (digital radio mondiale) broadcasting and Internet streaming (see Section 11.11).

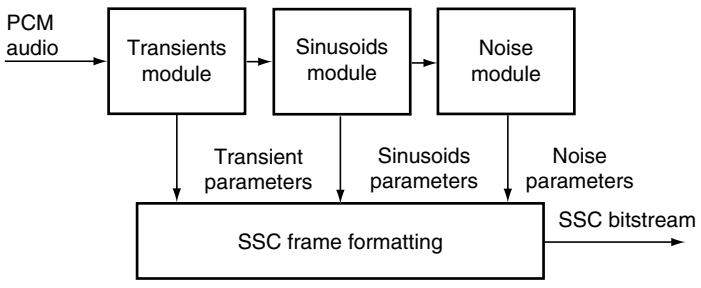



**FIGURE 11.32** Structure of SSC encoder.

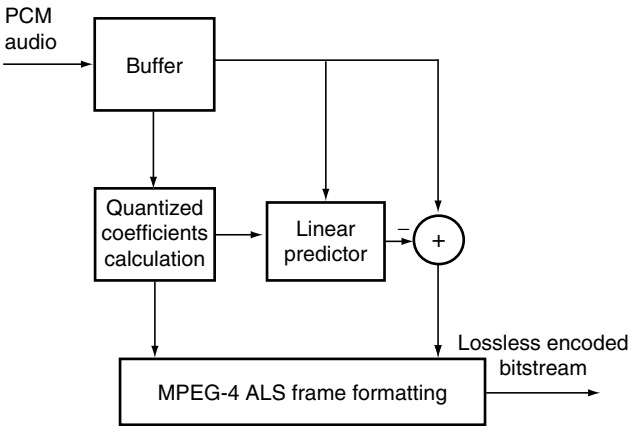



**FIGURE 11.33** Structure of MPEG-4 ALS encoder.

In the frame of MPEG-4 audio lossless standard (ALS) can also be found [62]. A simplified scheme of an ALS encoder is shown in Figure 11.33. The ALS coder works using forward-adaptive linear prediction with estimation of the optimal predictor coefficients and calculation of a suitable prediction order. Entropy coding is realized using Rice codes (simple mode) and block Gilbert–Moore codes (more complex but more efficient) [62]. An interesting feature of ALS coder is the possibility of floating-point data processing.

The third generation standard MPEG-4 has been designed for a broad area of various communications (especially mobile-access) and multimedia applications and is characterized by high flexibility, scalability, and universality [1]. It supports bitrates between 2 and 64 kb/s and offers additional services as text-to-speech (TTS) conversion, structured audio format, and interface between TTS and synthetic moving face models (talking heads), which are driven from speech.

A new standard MPEG-7, named the multimedia content description interface, is aimed to support as broad range of communications and multimedia applications as possible [2]. MPEG-7 audio merges five different technologies: the audio description framework (scalable series, low-level descriptors, and uniform silence segments), musical instrument timbre descriptors, sound recognition tools, spoken content descriptors, and melody descriptors. To describe the low-level audio features, regions of similarity and dissimilarity within the sound are searched for. This can be done using either samples taken at regular intervals or segments of samples. The relevant samples are then further manipulated to form a scalable series, which allows to progressively downsample the data contained in a series, according to the application, bandwidth, or storage requirements.

The scope of newest MPEG-21 standard is the integration of technologies enabling transparent and augmented use of multimedia resources across a wide range of networks and devices to support functions, such as content creation, content production, content distribution, content consumption and usage, content packaging, intellectual property management and protection, content identification and description, financial management, user privacy, terminals and network resource abstraction, content representation, and event reporting [3].

MUSICAM as well as MPEG-1 layers I and II coders have the same structure shown in Figure 11.34. The input audio signal is transmitted via a 32-band polyphase analysis filter bank (Figure 11.23a) with equally spaced passbands, according to Equations 11.50a through 11.50c. All subband filters with impulse responses $H_i(n)$, $i = 0, 1, \ldots, 31$, determined by Equation 11.48, are obtained by modulation of a single prototype lowpass filter with the impulse response $h(n)$, as is illustrated in Figure 11.23c. Their output signals are critically decimated. For a 48 ksamples/s sampling rate, each subband filter has a passband width of 750 Hz. Although these filters are highly overlapping, they can guarantee a perfect (or at least a nearly perfect) signal reconstruction (via the synthesis filter bank in Figure 11.23b) due to

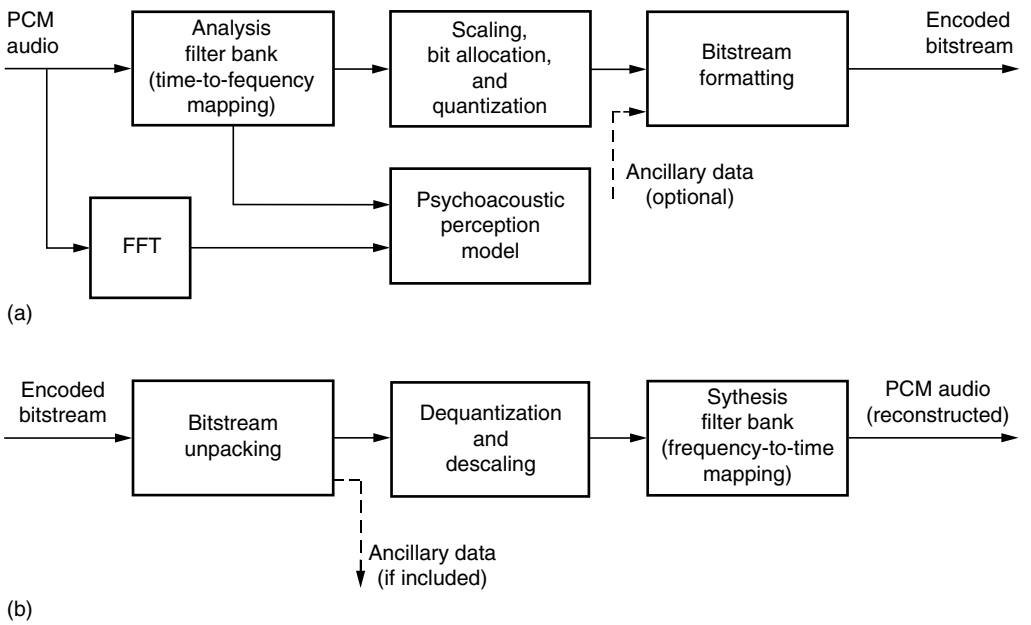



**FIGURE 11.34** Structure of MUSICAM and MPEG-1 layer I and II coders: (a) encoder and (b) decoder.

the power complementarity. For instance, at multiples of 750 Hz the respective filters, i.e., those with neighboring passbands exhibit a 3 dB attenuation.

Samples of subband signal components are quantized with a number of uniform midtread quantizers with 3, 5, 7, . . . , 65, 535 possible levels. Blocks of samples are formed (e.g., blocks of 12 samples in layer I) and divided by a scalefactor $s_{\text{sf}}$ selected in such a way that the sample with the largest magnitude is scaled to 1. By this means, a quite large overall dynamic range of approximately 126 dB is reached. Proper quantizers are selected with the dynamic bit allocation algorithm described in Section 11.9, controlled by a psychoacoustic audibility model, to meet the bitrate and the global threshold of hearing requirements. The whole procedure is described below.

The psychoacoustic model filter bank is based on a 512-point FFT for layer I and on a 1024-point FFT for layers II and III. First, samples of the spectral power density $P(k)$ and the respective level $L_P(k)$ in decibels are computed:

$$p(k) = |X(k)|^2 \tag{11.61a}$$

$$L_p(k) = 10 \log_{10} p(k) \tag{11.61b}$$

where $X(k)$ are DFT spectrum samples defined by Equation 11.31a. Next in each of $i = 0, 1, \ldots, 31$ subbands, the signal SPL is computed as follows:

$$L_{pi} = \max \left[ L_{P\max i}, 20 \log_{10} (32768 s_{\text{sf}\max i}) - 10 \right] \tag{11.62}$$

where $L_{P\text{max}i}$ is the maximum $L_P(k)$ value in $i$th subband.

Next, the relevant masker levels $L_{P\text{m}}(z_j)$ are searched for: tone masker levels $L_{P\text{tm}}(z_j)$, $j = 1, 2, \ldots, m_{\text{tm}}$ and noise masker levels $L_{P\text{nm}}(z_j)$, $j = 1, 2, \ldots, m_{\text{nm}}$. Then, the masking indices are computed (in dB):

$$a_{tm}(z_j) = -1.525 - 0.275 z_j - 4.5 \tag{11.63a}$$

$$a_{nm}(z_j) = -1.525 - 0.175 z_j - 0.5 \tag{11.63b}$$

where by $z_j$ the $j$th critical band index in Bark is denoted.

Individual masking threshold levels are computed (in dB) as

$$L_{ttm}(z, z_j, L_{pm}) = L_{Ptm}(z_j) + a_{tm}(z_j) + V(\Delta z, z_j) \tag{11.64a}$$

$$L_{tnm}(z, z_j, L_{pm}) = L_{Pnm}(z_j) + a_{nm}(z_j) + V(\Delta z, z_j) \tag{11.64b}$$

for tone maskers and for noise maskers, respectively. The so-called masking function $v(\Delta z, z_j)$ is defined by

$$V(\Delta z, z_j) = \begin{cases} 17(\Delta z + 1) - 0.4L_{pm}(z_j) + 6 & \text{for } \Delta z < -1 \\ (0.4L_{pm}(z_j) + 6)\Delta z & \text{for } -1 \le \Delta z < 0 \\ -17\Delta z & \text{for } 0 \le \Delta z < 1 \\ (\Delta z - 1)(-17 + 0.15L_{pm}(z_j)) + 17 & \text{for } 1 \le \Delta z \end{cases} \tag{11.65}$$

where $\Delta z = z - z_j$. The above equation gives significant values in range $-3 \le \Delta z \le 8$ only. Outside this region we can assume that $v(\Delta z, z_j) \to -\infty$.

Using Equation 11.14, the global threshold of hearing $L_{ptg}$ (Figure 11.35) can now be computed (in dB):

$$L_{ptg}(z) = 10\log_{10}\left(10^{L_{ptq}(z)/10} + \sum_{j=1}^{m_{tm}} 10^{L_{ttm}(z,z_j,L_{pm})/10} + \sum_{j=1}^{m_{nm}} 10^{L_{tnm}(z,z_j,L_{pm})/10}\right) \tag{11.66a}$$

where $L_{ptq}(z)$ is the threshold of audibility in quiet. Consequently,

$$L_{ptg\,min}(i) = \min\left(L_{ptg}(z_j)\right) \tag{11.66b}$$

for $i$th subband is computed over all critical bands $j$ contained in this subband. Now in each subband, the $\mathrm{SMR}_i$ can be computed (in dB):

$$\mathrm{SMR}_i = L_{pi} - L_{ptg\,min}(i) \tag{11.67}$$

Finally, the dynamic bit allocation algorithm based on maximization of the mask-to-noise ratio $\mathrm{MNR}_i(m)$ defined with Equation 11.60 assign bits to each block and each subband.

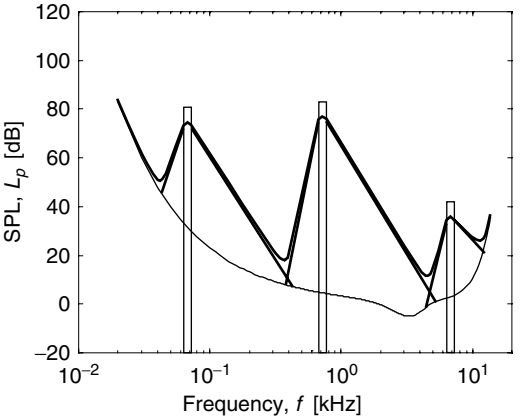



**FIGURE 11.35** Global hearing threshold $L_{ptg}$ for several simultaneous maskers as a function of frequency.

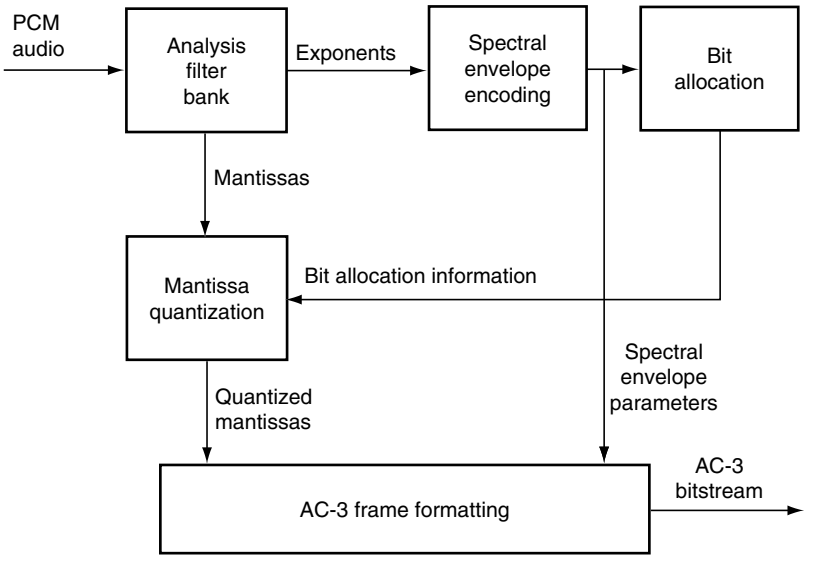



**FIGURE 11.36** AC-3 encoder.

## 11.10.3 Dolby AC-3 Standard

Dolby AC-3 standard gives the possibility for multichannel audio compression (from 1 to 5.1 channels) [66]. The bitstream ranges from 32 to 640 kb/s. Coding operation in AC-3 format is realized using time division aliasing cancelation (TDAC) filter bank. The sample block of length 512 is transformed into the frequency domain and each sample block is overlapped by 256 samples. Spectral components are represented in a floating-point format and the exponents and mantissas are coded separately. A set of exponents is coded into the spectral envelope. One AC-3 synchronization frame is composed for six audio blocks (1536 audio samples). Simplified AC-3 encoder is presented in Figure 11.36. The psychoacoustic model used in the AC-3 standard divides the audio band (0–24 kHz) into 50 subbands.

## 11.10.4 ATRAC Standard

An adaptive transform acoustic coder (ATRAC) developed by Sony has been designed for the MiniDisk system [67]. The bitstream of 16-bit audio signal with sampling rate 44.1 ksamples/s (705.6 kb/s) is reduced to 146 kb/s. The input audio signal (512 samples per channel) is decomposed into spectral coefficients, which are grouped into 52 block floating units (BFUs). The spectral coefficients are

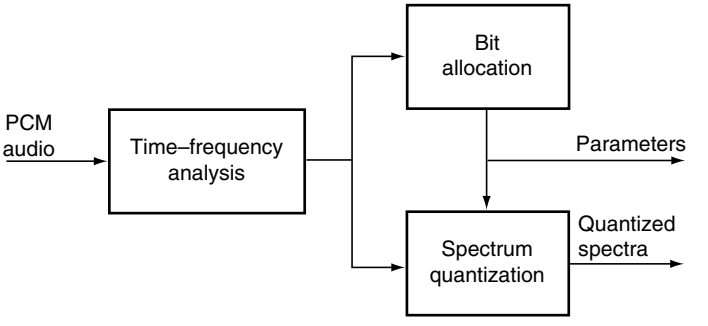



**FIGURE 11.37** ATRAC encoder.

normalized in each BFU, which are then quantized to the specified word length. Using QMF filters, the time–frequency analysis unit divides the input signal into three subbands: 0–5.5 kHz, 5.5–11 kHz, and 11–22 kHz. Each subband is transformed into the frequency domain using the MDCT. Figure 11.37 presents the general scheme of the ATRAC encoder.

New versions of ATRAC coder, i.e., ATRAC3 (the signal analysis in 4 bands) and ATRAC3plus (the signal analysis in 16 bands with longer windows) can compress audio data to, respectively, ∼10% and 5% of the CD data rate. The ATRAC family includes also ATRAC advanced lossless coder [67].

## 11.11 Digital Audio Transmission and Storage

### 11.11.1 Digital Audio Broadcasting

Digital audio broadcasting (DAB) system has been developed as a result of the EUREKA-147 Project—a worldwide consortium [63]. More than 30 countries are members of the World DAB forum (excluding the United States, where the system called HD [high-definition] radio is promoted). DAB uses two (terrestrial and satellite) frequency bands: band III (174–240 MHz) and L-band (1452–1492 MHz).

Audio coding is performed with MPEG-1 layer II standard (MP2) at 192 kb/s. Six radio program channels (each 192 kb/s) are organized in a DAB multiplex. Radio modulation is based on the coded orthogonal frequency division multiplexing (COFDM) and the coding is realized using the Viterbi algorithm [63].

### 11.11.2 Digital Radio Mondiale

Digital radio mondiale (DRM) is an open standard digital radio system, which is designed for revitalization of the old analog radio transmission, which uses the long and short frequency bands below 30 MHz (120 MHz limit has been voted) [68]. The DRM uses also the COFDM transmission and is fitted into the old AM band with the existing broadcasting band plan based on a 10 kHz bandwidth.

The DRM can use three different types of audio coding:

- HE-AAC (high efficiency advanced audio codec) for audio transmission
- CELP (code excited linear predicyion) for voice transmission only
- HVXC in the case of speech programs

It is possible to enhance the performance of DRM encoders by application of the SBR technology (see Sections 11.3 and 11.4).

### 11.11.3 Internet Transmission

Development of broadband Internet access (more than 256 kb/s) using a cable modem or a digital subscriber loop (DSL) facilities gives the possibility of Internet radio broadcasting. Nowadays hundreds of radio stations can already be found in the Internet. They mainly transmit music. Organizing your own radio station is very simple—there exists an open source and free-of-charge software (see www.shoutcast.com).

Two types of transmissions are used: live and on demand. The audio coding transmission is mainly realized with MP3 (stereo signals with typical bitrate of 128 kb/s) or AAC-Plus (stereo signals with 64 kb/s or 5.1 channels with 160 kb/s).

### 11.11.4 Digital Audio Storage

It can be observed that in the era of Internet, the digital audio files in lossless and lossy formats can be stored on hard disk drives (HDDs) of personal computers or computer servers. An HDD with capacity of, e.g., 200 GB can store 3640 h of digital music with 128 kb/s bitstream. Next popular storing medium is flash memories (used mainly in personal players) with bigger and bigger capacity (e.g., 4 MB).

The data storing costs are the lowest for optical disks [69]. Perhaps, therefore, the most popular format is still the compact disk digital audio (CD-DA) with stereo PCM signals with 16-bit resolution and sampling rate of 44.1 ksamples/s.

Multichannel coding (1–5.1 channels) with high resolution of 16, 20, or 24 bits can be found in DVD-audio (digital versatile disk audio) format that uses lossless compression: LPCM (linear pulse coded modulation) or MLP (meridian lossless packing) [70]. Sampling rate in this standard may be set at 44.1, 48, 88.2, 96, 176.4, or 192 ksamples/s (two highest sampling rates only for two channels).

Second multichannel (stereo or 5.1) optical standard is SACD (super audio compact disk), which uses 1-bit sigma-delta modulation called direct stream digital standard with sampling rate of 2.8224 MHz. SACD gives a possibility for hybrid recording, i.e., CD-DA together with SACD.

Audio data streams (up to 8) can also be recorded on DVD-video with maximal data rate of 6.144 Mb/s. In this format mainly lossy standards are used (Dolby Digital [7]).

## References

1. ISO/IEC JTC1/SC29, MPEG-4 audio, Doc. N2431, 1998.
2. ISO/IEC JTC1/SC29, MPEG-7, Martínez, J.M. (Ed.), Doc. WG11, N4031, 2001.
3. ISO/IEC JTC1/SC29, MPEG-21, Bormans, J. and Hill, K. (Eds.), Doc. WG11 N4041, 2001.
4. MPEG home page, http://www.chiariglione.org/mpeg/, 2006.
5. Noll, P., MPEG audio coding standards, *IEEE Signal Process. Mag.*, 1997.
6. Noll, P. and Liebchen, T., Digital audio: From lossless to transparent coding, Dabrowski, A. (Ed.), IEEE Poland Section, Chapter Circuits and Systems, *Proceedings of the IEEE Signal Processing Workshop*, Poznan, Poland, 1999, 53.
7. Dolby Digital Plus, Dolby Laboratories Inc., www.dolby.com, 2006.
8. *High-Definition Multimedia Interface* (*HDMI*) *Specification,* Version 1.2, 2005.
9. Audio and Multimedia Realtime Systems, Fraunhofer IIS, http://www.iis.fraunhofer.de/amm/download/index.html, 2006.
10. Audio Solutions Guide—Analog and Digital Amplifiers, Clock Distribution Circuits, Data Converters, Digital Signal Processors, Interface, Power Management, Texas Instruments, www.ti.com, 2006.
11. Dabrowski, A., Marciniak, T., and Pawlowski, P., Chosen digital signal procedures for hearing aids, *Arch. Control Sci.*, 15 (LI), 3, 2005, 291.
12. Gold, B. and Morgan, N., *Speech and Audio Signal Processing—Processing and Perception of Speech and Music*, John Wiley & Sons, New York, 2000.
13. Huang, Y. and Benesty, J., *Audio Signal Processing for Next Generation Multimedia Communication Systems*, Kluwer Academic Publishers, Boston, MA, 2004.
14. Marciniak, T., Dabrowski, A., and Cetnarowicz, D., Voice signal enhancement for human–machine interfaces, *Foundations of Control and Management Sciences*, Publishing House of Poznan University of Technology, Poznan, Poland, 02, 2004, 45.
15. Rochowniak, R., Marciniak, T., and Dabrowski, A., Recognition of noised speech using HMM-based approach, IEEE Poland Section, Chapter Circuits and Systems, *Proceedings of the IEEE Signal Processing Workshop*, Poznañ, Poland, 2005, 133.
16. Schroeter, J., Mehta, S.K., and Carter, G.C., Acoustic signal processing, *The Electrical Engineering Handbook*, Chapman & Hall/CRCnetBASE, Boca Raton, FL, 2000, 19.1.
17. Moore, B.C.J., *An Introduction to the Psychology of Hearing*, 4th ed., Academic Press, London, 1997.
18. Zwicker, E. and Fastl, H., *Psychoacoustics*, 2nd ed., Springer-Verlag, Berlin, 1999.
19. Zwicker, E. and Feldtkeller, R., *Das Ohr als Nachrichtenempfänger*, Hirzel-Verlag, Stuttgart, 1967.
20. Kinsler, L., Frey, A., Coppens, A., and Sanders, J., *Fundamentals of Acoustics*, John Wiley & Sons, New York, 2000.
21. Purnhagen, H., Advances in parametric audio coding, *Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, New York, 1999, W99–1.

22. Oppenheim, A., Schafer, R., and Buck, J., *Discrete-Time Signal Processing*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1998.
23. Vaidyanathan, P.P., *Multirate Systems and Filter Banks*, Prentice-Hall, Englewood Cliffs, NJ, 1993.
24. Dehery, Y.F., Stoll, G., and Kerkhof, L.V.D., MUSICAM source coding for digital sound, *Proceedings of the 17th International Television Symposium*, Montreux, Switzerland, 1991, 612.
25. Princen, J., Johnson, A., and Bradley, A., Subband transform coding using filter bank designs based on time domain aliasing cancellation, *Proceedings of ICASSP*, 1987, 2161.
26. Fettweis, A., Wave digital filters: Theory and practice, *Proc. IEEE*, 74(2), 1986, 270.
27. Burrus, C.S., Gopinath, R.A., and Guo, H., *Introduction to Wavelets and Wavelet Transforms*, Prentice-Hall, Upper Saddle River, NJ, 1998.
28. Purnhagen, H., Meine, N., HILN—the MPEG4 parametric audio coding tools, *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, Geneva, 3, 2000, 2001.
29. Muzzi, M.G., *Improvement of the Audio Quality of a Parametric Coder*, Philips Digital System Laboratories, Eindhoven, The Netherlands, 2003.
30. Chong, K.S., et al. Low power spectral band replication technology for the MPEG-4 audio standard, *Proceedings of ICICS-PCM*, Singapore, 2003, 1408.
31. Dietz, M. and Meltzer, S., *CT-aacPlus—a State-of-the-Art Audio Coding Scheme, EBU Technical Review*, Coding Technologies, Germany, 2002, 1/7.
32. Ekstrand, P., Bandwidth extension of audio signals by spectral band replication, *Proceedings of the 1st IEEE Workshop on Model Based Processing and Coding of Audio*, Leuven, Belgium, 2002.
33. ISO/IEC JTC1/SC29/WG11 N6130, Parametric coding of high quality audio, text 14496–3, approved 2003.
34. Kunz, O., *Spectral Band Replication Explained: White Paper*, Coding Technologies, http://www.codingtechnologies.com/products/sbr.htm, 2006.
35. Fletcher, H., Auditory patterns, *Rev. Mod. Phys.*, 12, 1940, 47.
36. Scharf, B., Critical bands, *Foundations of Modern Auditory Theory*, Vol. 1, Tobias, J.V. (Ed.), Academic Press, New York, 1970, 157.
37. Zwicker, E., Flottrop, G., and Stevens, S.S., Critical bandwidth in loudness summation, *J. Acoust. Soc. Am.*, 29, 1957, 548.
38. Terhardt, E., Calculating virtual pitch, *Hear. Res.*, 1, 1979, 155.
39. Hellman, R.P., Asymmetry in masking between noise and tone, *Percept. Psychophys.*, 11, 1972, 241.
40. Kapust, R., A human ear related objective measurement technique yields audible error and error margin, *Proceedings of the 11th International AES Conference on Test and Measurement*, Portland, 1992, 191.
41. Johnston, J.D., Transform coding of audio signals using perceptual noise criteria, *IEEE J. Sel. areas Commun.*, 6(2), 1988, 314.
42. Dabrowski, A., *Multirate and Multiphase Switched-Capacitor Circuits*, Chapman and Hall, London, 1997.
43. Jayant, N.S. and Noll, P., *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
44. Audio DSPs, Cirrus Logic, http://www.cirrus.com/en/products/pro/techs/T6.html, 2006.
45. DSP Device Overview, Altera, www.altera.com, 2006.
46. Audio/Video Products, Analog Devices, www.analog.com, 2006.
47. Digital Signal Processors and Controllers, Freescale Semiconductor, www.freescale.com, 2006.
48. DSP Selection Guide, Texas Instruments, www.ti.com, 2006.
49. Bellanger, M., *Digital Processing of Signals: Theory and Practice*, 3rd ed., John Wiley & Sons, New York, 2000.
50. Mitra, S.K., *Digital Signal Processing*, 3rd ed., McGraw-Hill, New York, 2006.
51. Marciniak, T. and Dabrowski, A., Aspects of audio processing, *Online Symposium for Electronics Engineers* (*OSEE*), www.techonline.com, 2000.
52. MATLAB: *Signal Processing Toolbox User's Guide Ver. 6*, The MathWorks Inc., Natick, MA, 2006.

53. Dabrowski, A., Figlak, P., Golebiewski, R., and Marciniak, T., *Signal Processing Using Signal Processors* (in Polish), PUT Press, Poznan, Poland, 2001.
54. ISO/IEC JTC1/SC29, MPEG-1, Information technology—coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s—IS 11172 (Part 3, Audio), 1992.
55. Beckmann, P. and Stilson, T., An efficient asynchronous sampling-rate conversion algorithm for multi-channel audio applications, *Proceedings of the 119th AES Convention*, New York, 2005.
56. Zölzer, U., *Digitale Audiosignalverarbeitung*, Teubner, Stuttgart, 1997.
57. Ramstadt, T.A., Digital methods for conversion between arbitrary sampling frequencies, *IEEE Trans. Acoust., Speech Signal Process.*, ASSP-32, 3, 1984, 577.
58. Huffman, D.A., A method for the construction of minimum redundancy codes, *Proc. IRE*, 40, 1952, 1098.
59. Salomon, D. *Data Compression: The Complete Reference*, 3rd ed., Springer-Verlag, New York, 2004.
60. Pan, D.Y., A tutorial on MPEG/audio compression, *IEEE Multimedia*, 2, 1995, 60.
61. List of codecs, en.wikipedia.org, 2006.
62. Liebchen, T., The MPEG-4 audio lossless coding (ALS) standard-technology and applications, *Proceedings of the AES 119th Convention*, New York, 2005.
63. Hoeg, W. and Lauterbach, T., *Digital Audio Broadcasting—Principles and Applications of Digital Radio*, 2nd ed., John Wiley & Sons, West Sussex, England, 2003.
64. Brandenburg, K., MP3 and AAC explained, *Proceedings of AES 17th International Conference on High Quality Audio Coding*, Erlangen, Germany, 1999.
65. ISO/IEC JTC1/SC29, MPEG-2, Information technology—generic coding of moving pictures and associated audio information—IS 13818 (Part 7, Audio), 1997.
66. Jayant, N., Digital audio communications, *Digital Signal Processing Handbook (Part IX)*, CRC Press LLC, Boca Raton, FL, 1999.
67. ATRAC Technology, Sony, http://www.sony.net/Products/ATRAC3/, 2006.
68. ETSI ES 201 980 V2.2.1, *Digital Radio Mondiale* (*DRM*) *System Specification*, 2005.
69. Delux Media Global Services, www.deluxmedia.com, 2006.
70. Stuart, J., Craven, P., Gerzon, M., Law, M., and Wilson, R., MLP lossless compression, *Proceedings of the AES 9th Regional Convention*, Tokyo, http://www.meridian-audio.com/lib_pap.htm, 1999.

# 12

# Digital Video Processing




Todd R. Reed
*University of Hawaii at Manoa*


## 12.1 Introduction

Rapid increases in performance and decreases in cost of computing platforms and digital image acquisition and display subsystems have made digital images ubiquitous. Continued improvements promise to make digital video as widely used, opening a broad range of new application areas. In this chapter, some of the key aspects of this evolving data type are examined.

### 12.1.1 Some Historical Perspective

The use of image sequences substantially predates modern video displays [1]. As might be expected, the primary initial motivation for using these sequences was the depiction of motion. One of the earlier approaches to motion picture display was invented by the mathematician William George Horner in 1834.

Originally called the Daedaleum (after Daedalus, who was supposed to have made figures of men that seemed to move), it was later called the zoetrope (life turning) or the wheel of life. The Daedaleum works by presenting a series of images, one at a time, through slits in a circular drum as the drum is rotated.

Although this device is very simple, it illustrates some important concepts. First and foremost, the impression of motion conveyed by a sequence of images is illusory. It is the result in part of a property of the human visual system (HVS) referred to as persistence of vision. An image is perceived to remain for a period after it has been removed from view. This illusion is the basis for all motion picture displays. When the drum in the device is rotated slowly, the images appear (as they are) as a disjoint sequence of still images. As the speed of rotation increases (the images are displayed at a higher rate), a point is reached at which motion is perceived, even though the images appear to "flicker". Further increasing the speed of rotation, a point is reached at which flicker is no longer perceived (the critical fusion frequency). Finally, the slits in the drum illustrate a critical aspect of this illusion. To perceive motion from a sequence of images, the stimuli that are represented by the individual images must be removed for a period between each presentation. If not, the sequence of images simply merges into a blur, and no motion is perceived.

These concepts (rooted in the nature of human visual motion perception) are fundamental, and are reflected in all motion picture acquisition and display systems.

### 12.1.2 Video

Unlike image sequences on film, video is represented as a 1-D signal, derived by scanning the camera sensor. The fact that the signal is derived by scanning imposes a particular signal structure, an example of which is shown in Figure 12.1 for a noninterlaced system.

In principle, there are many ways in which scanning could be done. The simplest in concept is noninterlaced, line-continuous scanning (which yields the video signal just discussed). This approach is also referred to as progressive scanning. Viewed in the 2-D plane (either at the camera or display), this approach appears as shown in Figure 12.2.

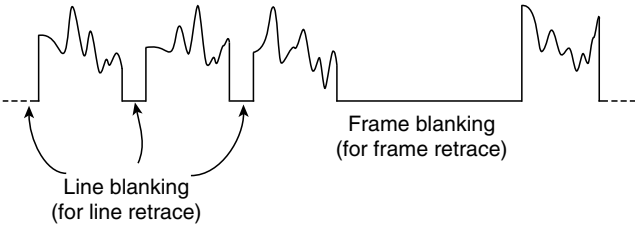



**FIGURE 12.1** A noninterlaced video signal.

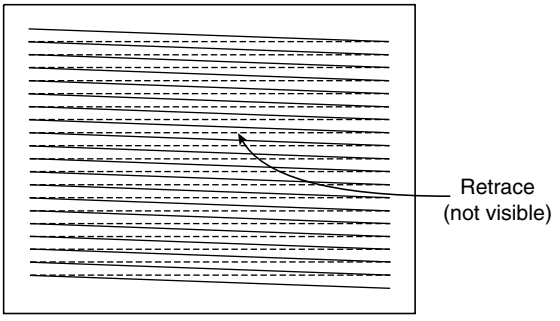



**FIGURE 12.2** A noninterlaced scanning raster.

The bandwidth of the resulting video signal is relatively high. Transmitting a frame of 485 lines*, with 4:3 aspect ratio (NTSC resolution), at 60 frames/s requires roughly twice the available channel bandwidth (6 MHz). 60 updates/s are needed to avoid wide area flicker, dictated by the temporal response of the HVS. One approach to reducing the signal bandwidth is to send half as many samples (lines). This cannot be accomplished by reducing the frame rate to 30 fps, because an unacceptable degree of flicker is introduced. Reducing the spatial resolution of each frame results in unacceptable blurring. Interlaced scanning is a compromise between the two approaches.

As used in NTSC television, each complete scan (a frame) contains 525 lines, and occurs every 1/30 s. The frame consists of two fields (even and odd), $262\frac{1}{2}$ lines each. These fields are interlaced to form the frame. Fields are scanned every 1/60 s (reducing flicker). Because two fields are interlaced to form one frame, this is called 2:1 interlace. Two interlaced fields (NTSC) are shown in Figure 12.3.

Image acquisition and display via scanning have several disadvantages. Nonideal aspects of the scanning system (e.g., nonzero spot size), and under some circumstances the act of scanning itself, leads to a reduction in vertical resolution below that predicted by the sampling theorem. The ratio of the actual to ideal resolution is called the Kell factor $k$, $0 \leq k \leq 1$. Typical values of $k$ are $.6 < k < .8$, with interlaced systems having lower $k$. Scanning also causes distortion when objects in the scene are in motion. For example, a vertical line in motion will result in a tilted scanned image (not because of the tilt of the scan line, but because points on the line at the bottom of the screen are reached later than points at the top). Finally, different points in space within the frame do not correspond to the same point in time. Viewed in the spatiotemporal volume, each frame is tilted, with the upper left corner of the frame corresponding to a significantly earlier time than the lower right corner. This can make the accurate analysis of the image sequence difficult.

Interlaced scanning has additional disadvantages. Interlaced display systems suffer from interline flicker (particularly in regions of the image with nearly horizontal structure). Interlacing results in reduced vertical resolution that increases aliasing. It also increases the complexity of subsequent processing or analysis (such as motion estimation). Interoperability with other systems, such as computer workstations (which use noninterlaced displays) is made difficult. Still images extracted from interlaced video (freeze fields) are generally of poor quality. Often, only freeze fields are provided. This last point can be seen by considering the case of an edge in horizontal motion (Figure 12.4). To merge two fields to get a still image of reasonable quality, or to get a good progressively scanned sequence from an interlaced one, is a nontrivial problem.

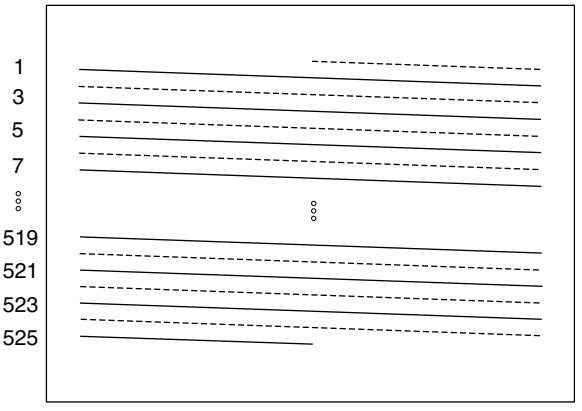



**FIGURE 12.3** An NTSC frame, formed by interlacing two fields (2:1 interlace).

*NTSC consists of 525 lines, but only ~485 lines are active.

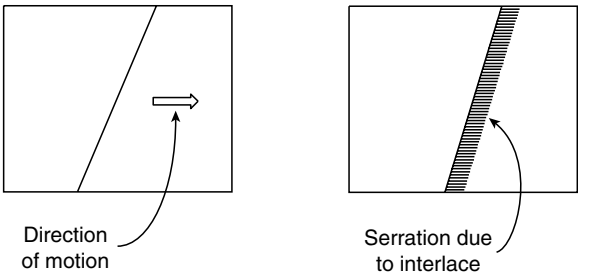



**FIGURE 12.4** Effect of interlace on an edge in horizontal motion.

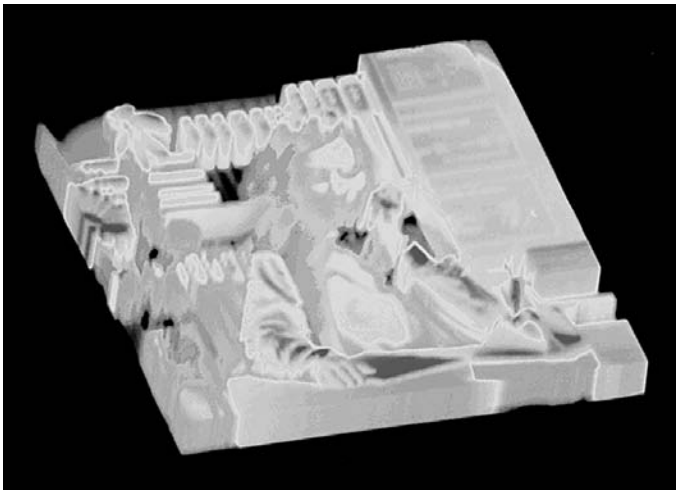


**FIGURE 12.5** An image sequence represented as a spatiotemporal volume, raytraced to exhibit its internal structure.

### 12.1.3 Image Sequences as Spatiotemporal Data

As discussed above, the scanning process makes the precise specifications of an image sequence difficult (since every spatial point exists at a different time). Interlace complicates matters further. In the remainder of this chapter, the simplifying assumption that each point in a frame corresponds to the same point in time is made. This is analogous to the digitization of motion picture film, or the sequence that results from a CCD camera with a shutter. It is a reasonable assumption in progressive or interlaced video systems when scene motion is slow compared to the frame rate. The series of frames are no longer tilted in the spatiotemporal domain, and can be "stacked" in a straightforward way to form a spatiotemporal volume (see Figure 12.5).

## 12.2 Some Fundamentals

Following are some notational conventions and basic principles used in the remainder of this chapter. A continuous sequence is denoted as $u(x, y, t)$, $v(x, y, t)$, etc., where $x$ and $y$ are the continuous spatial variables and $t$ is the continuous temporal variable. Similarly, a discrete sequence is denoted as $u(m, n, p)$, $v(m, n, p)$, etc., where $m$ and $n$ are the discrete (integer) spatial variables and $p$ is the discrete (integer) temporal variable.

### 12.2.1 A 3-D System

As in 1-D and 2-D, a 3-D discrete system can be defined as

$$y(m,n,p) = H[x(m,n,p)] \tag{12.1}$$

where $H$ is the system function. In general, this function need be neither linear nor shift invariant. If the system is both linear and shift invariant (LSI), it can be characterized in terms of its impulse response $h(m,n,p)$. The linear shift invariant system response can then be written as

$$\begin{aligned} y(m,n,p) &= \sum_{m'=-\infty}^{\infty} \sum_{n'=-\infty}^{\infty} \sum_{p'=-\infty}^{\infty} x(m',n',p')h(m-m',n-n',p-p') \\ &\equiv x(m,n,p)*h(m,n,p) \end{aligned} \tag{12.2}$$

where "$*$" denotes (discrete) convolution. Similarly, for the continuous case,

$$g(x,y,t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x',y',t')h(x-x',y-y',t-t')\mathrm{d}x'\ \mathrm{d}y'\ \mathrm{d}t' \tag{12.3}$$

### 12.2.2 3-D Fourier Transform

The 3-D continuous Fourier transform can be expressed as

$$F(\xi_x,\xi_y,\xi_t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y,t)\mathrm{e}^{-j2\pi(x\xi_x+y\xi_y+t\xi_t)}\mathrm{d}x\ \mathrm{d}y\ \mathrm{d}t \tag{12.4}$$

where $\xi_x$, $\xi_y$, and $\xi_t$ are the spatiotemporal-frequency variables and $f(x, y, t)$ is a continuous spatiotemporal signal. As in the 2-D case, the 3-D Fourier transform is separable:

$$F(\xi_x,\xi_y,\xi_t) = \int_{-\infty}^{\infty} \left[ \int_{-\infty}^{\infty} \left[ \int_{-\infty}^{\infty} f(x,y,t)\mathrm{e}^{-j2\pi x\xi_x}\mathrm{d}x \right] \mathrm{e}^{-j2\pi y\xi_y}\mathrm{d}y \right] \mathrm{e}^{-j2\pi t\xi_t}\mathrm{d}t \tag{12.5}$$

Also as in the 1-D and 2-D cases, if

$$g(x,y,t) = h(x,y,t)*f(x,y,t) \tag{12.6}$$

then

$$G(\xi_x,\xi_y,\xi_t) = H(\xi_x,\xi_y,\xi_t)F(\xi_x,\xi_y,\xi_t) \tag{12.7}$$

If $h(x, y, t)$ is the LSI system impulse response then $H(\xi_x, \xi_y, \xi_t)$ is the frequency response of the system.

The spatiotemporal discrete Fourier transform is defined as

$$v(h,k,l) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} \sum_{p=0}^{N-1} u(m,n,p)W_N^{hm}W_N^{kn}W_N^{lp} \tag{12.8}$$

where $0 \le h, k, l \le N-1$ and $W_N = \mathrm{e}^{\frac{-j2\pi}{N}}$.

The inverse transform is

$$u(m,n,p) = \frac{1}{N^3}\sum_{h=0}^{N-1}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} v(h,k,l)\,W_N^{-hm}W_N^{-kn}W_N^{-lp} \tag{12.9}$$

where $0 \le m, n, p \le N-1$.

### 12.2.3 Moving Images in the Frequency Domain

Following the discussion in Ref. [2], a moving monochrome image can be represented by an intensity distribution $f(x, y, t)$. The image is static if $f(x, y, t) = f(x, y, 0)$ for all $t$. The velocity of the image can be expressed via the image velocity vector

$$\vec{r} = (r_x,\ r_y) \tag{12.10}$$

If the (initially static) image translates at a constant velocity $\vec{r}$ then

$$f_r(x, y, t) = f(x - r_x t,\ y - r_y t,\ t) \tag{12.11}$$

Consider the case of a simple 2-D "image" $f(x, t)$. Let

$$\vec{a} = \begin{pmatrix} x \\ t \end{pmatrix} \text{ and } \vec{b} = \begin{pmatrix} \xi_x \\ \xi_t \end{pmatrix} \tag{12.12}$$

where $\xi_x$ and $\xi_t$ are the spatial and temporal frequency variables. Then the transform pair can be written as

$$f(\vec{a}) \overset{F}{\rightarrow} F(\vec{b}) \tag{12.13}$$

Now, translation can be represented as a coordinate transformation:

$$\vec{a}' = \begin{pmatrix} x - r_x t \\ t \end{pmatrix} = \mathbf{A}\vec{a} \tag{12.14}$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & -r_x \\ 0 & 1 \end{bmatrix} \tag{12.15}$$

and $r_x$ is the horizontal speed.

Using the expression for the Fourier transform after an affine coordinate transformation (any combination of scaling, rotation, and translation),

$$f(\vec{a}') \overset{F}{\rightarrow} F\left[(\mathbf{A}^{-1})^T\vec{b}\right] \tag{12.16}$$

where

$$(\mathbf{A}^{-1})^T = \begin{bmatrix} 1 & 0 \\ r_x & 1 \end{bmatrix} \tag{12.17}$$

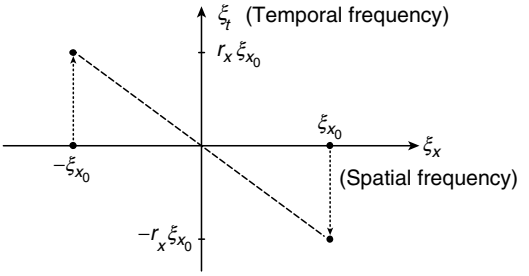



**FIGURE 12.6** A two component 1-D signal in translational motion.

so that

$$f(x - r_x t, t) \xrightarrow{F} F(\xi_x, \xi_t + r_x \xi_x) \tag{12.18}$$

**Example**

Consider a simple static image with only two components (Figure 12.6). As the image undergoes translation with horizontal speed $r_x$, all temporal frequencies are shifted by $-r_x\, \xi_x$. Spatial frequency components remain unchanged. That is, all frequency components of an image moving with velocity $r_x$ lie on a line through the origin, with slope $-r_x$.

Extending the analysis to the 3-D case ($f(x, y, t)$), let the velocity $\vec{r} = (r_x, r_y)$ Then

$$f(x - r_x t, y - r_y t, t) \xrightarrow{F} F\big(\xi_x, \xi_y, \xi_t + r_x \xi_x + r_y \xi_y\big) \tag{12.19}$$

Each temporal frequency is shifted by the dot product of the spatial frequency vector $\vec{s} = (\xi_x, \xi_y)$ and the image velocity vector $\vec{r} = (r_x, r_y)$. If the image was originally static, then

$$\xi_t = -\vec{r} \cdot \vec{s} = -(r_x \xi_x + r_y \xi_y) \tag{12.20}$$

Geometrically, the image motion changes the static image transform (which lies in the $(\xi_x, \xi_y)$ plane) into a spectrum in a plane with slope $-r_y$ in the $(\xi_y, \xi_t)$ plane and $-r_x$ in the $(\xi_x, \xi_t)$ plane. As in the 2-D case, the shifted points lie on a line through the origin. Note that this represents a relatively sparse occupation of the frequency domain (of interest for compression applications) (Figure 12.7). A 3-D

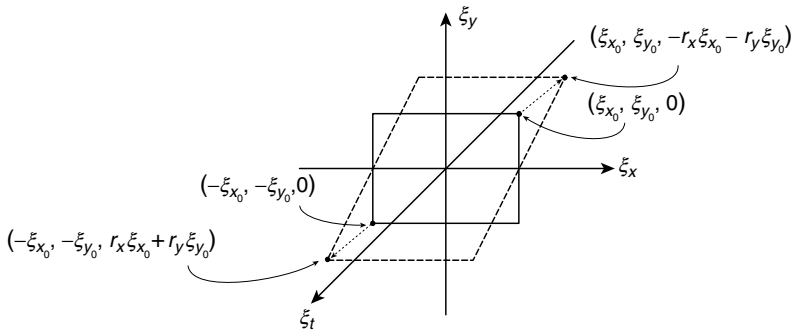



**FIGURE 12.7** A two component 2-D signal in translational motion.

volume of data has been compressed into a plane. This compactness is not observed in the spatiotemporal domain.

In summary, the spectrum of a stationary image lies in the $(\xi_x, \xi_y)$ plane. When the image undergoes translational motion, the spectrum occupies an oblique plane which passes through the origin. The orientation of the plane indicates the speed and direction of the motion. It is therefore possible to associate energy in particular regions of the frequency domain with particular image velocity components. By filtering specific regions in the frequency domain, these image velocity components can be detected. As will be seen shortly, other effects (such as the visual impact of temporal aliasing) can also be understood in the frequency domain.

### 12.2.4 Three-Dimensional Sampling

In its simplest form (regular sampling on a rectangular grid, the method used here), 3-D sampling is a straightforward extension of 2-D (or 1-D) sampling (Figure 12.8). Given a band-limited sequence

$$f(x, y, t) \xrightarrow{F} F(\xi_x, \xi_y, \xi_t) \tag{12.21}$$

with

$$F(\xi_x, \xi_y, \xi_t) = 0 \text{ whenever } |\xi_x| > \xi_{x_0}, |\xi_y| > \xi_{y_0}, \quad \text{or} \quad |\xi_t| > \xi_{t_0} \tag{12.22}$$

the continuous sequence can be reconstructed from a discrete set of samples whenever

$$\xi_{x_s} > 2\xi_{x_0},\ \xi_{y_s} > 2\xi_{y_0}, \quad \text{and} \quad \xi_{t_s} > 2\xi_{t_0} \tag{12.23}$$

where $\xi_{x_s}$, $\xi_{y_s}$, $\xi_{z_s}$ are the sampling frequencies. Equivalently, the sequence can be reconstructed if the intervals between samples are such that

$$\Delta x < \frac{1}{2\xi_{x_0}},\ \Delta y < \frac{1}{2\xi_{y_0}}, \quad \text{and} \quad \Delta t < \frac{1}{2\xi_{t_0}}. \tag{12.24}$$

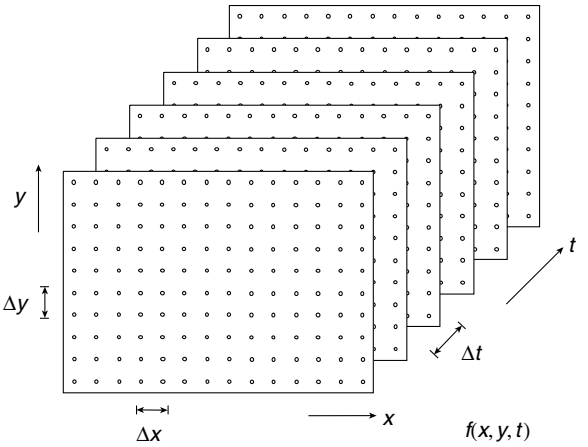



**FIGURE 12.8** A sampled spatiotemporal signal (image sequence).

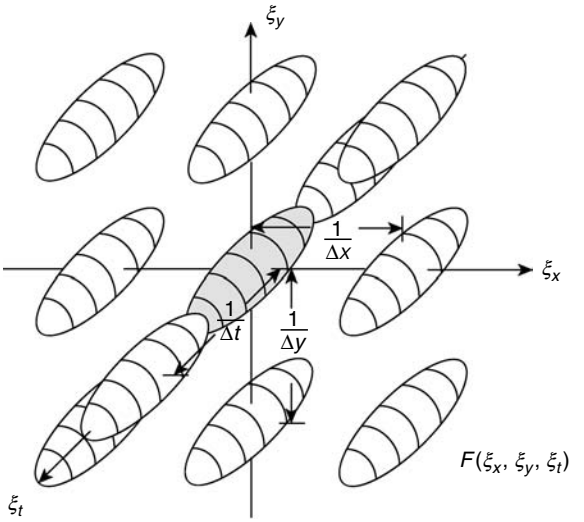



**FIGURE 12.9** An image sequence with insufficiently high sampling in the temporal dimension.

If any of the sampling frequencies fall below the specified rates, then the neighboring spectra (replications of the continuous spectrum, produced by the sampling process) overlap, and aliasing results. A case for which the temporal sampling frequency is too low is shown in Figure 12.9. The appearance of aliasing in the spatial domain, where it commonly manifests as a jagged approximation of smooth, high contrast edges, is relatively familiar and intuitive. The effect of sampling at too low a rate temporally is perhaps less so.

Consider the earlier simple example of a 1-D image with only two components, moving with velocity $r_x$. The continuous case, as derived previously, is shown in Figure 12.10. $\xi_{x_0}$ is the frequency of the static image. Suppose this image is sampled along the temporal dimension at a sampling frequency $\xi_{t_s}$ less than the Nyquist rate ($\xi_{t_N} = 2r_x\xi_{x_0}$), and the image is reconstructed via an ideal lowpass filter with temporal cutoff frequencies at plus and minus half the sampling frequency (Figure 12.11). What is the visual effect of the aliased components?

As seen previously, the velocity of motion is reflected in the slope of the line connecting the components. For the situation shown, a sinusoidal grid

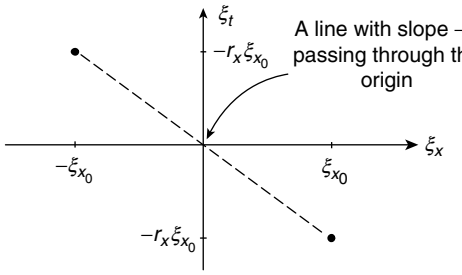



**FIGURE 12.10** A continuous two component 1-D signal in translational motion.

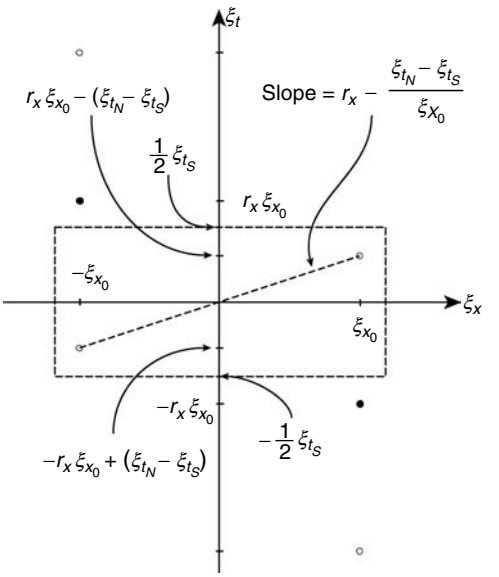



**FIGURE 12.11** A reconstruction of a sampled 1-D signal with temporal aliasing.

(of the same frequency as the original) moving in the opposite direction, with speed $r_x\xi_{x_0} - (\xi_{t_N} - \xi_{t_s})$ is observed. As the sampling frequency drops, the velocity decreases, eventually reaching zero. Continued reduction in $\xi_{t_s}$ results in motion in the same direction as the original image, increasing in velocity until (at $\xi_{t_s} = 0$) the velocities of the two components are identical.

In the simple example just considered, the image was spatially homogeneous, so that the effects of aliasing were seen throughout the image. In general, this is not the case. As in the 1-D and 2-D cases, the temporal aliasing effect is seen in regions of the sequence with sufficiently high temporal frequency components to alias. Circumstances leading to high temporal frequencies include high velocity (large values of $r_x$ in our simple example) and high spatial frequency components with some degree of motion (high $\xi_{x_0}$ in our example). Higher spatial frequency components require slower speeds to cause aliasing.

A well-known example of temporal aliasing is the so-called wagon wheel effect, in which the wheels of a vehicle appear to move in a direction opposite to that of the vehicle itself. The wheels have both high spatial frequency components (due to their spokes) and relatively high rotational velocity. Hence, aliasing occurs (the wheels appear to rotate in reverse). The vehicle itself, however, which is moving more slowly and is also generally composed of lower spatial frequency components, moves forward (does not exhibit aliasing effects).

## 12.3 Perception of Visual Motion

Visual perception can be discussed at a number of different levels: the anatomy or physical structure of the visual system, the physiology or basic function of the cells involved, and the psychophysical behavior of the system (the response of the system to various stimuli). Following is a brief discussion of visual motion perception. A more extensive treatment can be found in Ref. [3].

### 12.3.1 Anatomy and Physiology of Motion Perception

The retina (the hemispherical surface at the back of the eye) is the sensor surface of the visual system, consisting of two major types of sensor elements. The rods are long and thin structures, numbering approximately 120 million. They provide scotopic "low-light" vision and are highly sensitive to motion. The cones are shorter and thicker and substantially fewer in number (approximately six million per retina). They are less sensitive than the rods, providing photopic "high-light" and color vision. The cones are much less sensitive to motion.

The rods and cones are arranged in a roughly hexagonal array. However, they are not uniformly distributed over the retina. The cones are packed in the fovea (hence color vision is primarily foveal). The rods are primarily outside the fovea. As a result, motion sensitivity is higher outside the fovea, corresponding to the periphery of the visual field.

Visual information leaves each eye via the optic nerve. The nerves from each eye split at the optic chiasma, pass through the lateral geniculate nucleus, and continue to the visual cortex. Information is retinotopically mapped on the cortex (organized as in the original scene, but reversed). Note, however, that the mapping is not one-to-one (one retinal rod or cone to one cortical cell). As mentioned above, there are approximately 120 million rods and 6 million cones in each eye, but only 1 million fibers in the associated optic nerve. This 126:1, apparently visually lossless compression is one of the motivations for studying perceptually inspired image and video compression techniques, as discussed later in this chapter.

To achieve this compression, each cortical cell receives information from a set of rods or cones. This set makes up the receptive field for that cell. The response of a cortical cell to stimuli at different points in this field can be measured (e.g., via a moving spot of light), and plotted just as one might plot the impulse response of a 2-D filter.

Physiologically, nothing mentioned so far seems specifically adapted to the detection (or measurement) of motion. It might be reasonable to expect to find cells that respond selectively to, e.g., the direction of motion. There appear to be no such cells in the human retina (although other species do

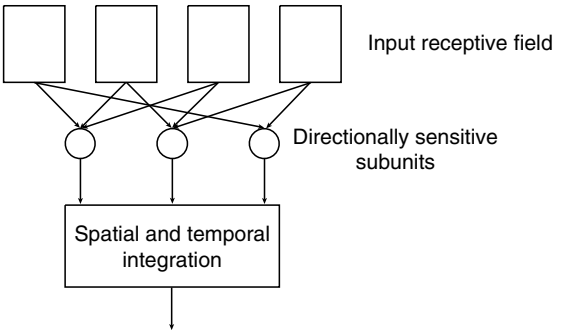



**FIGURE 12.12** A common organizational structure for modeling complex cell behavior.

have retinal cells that respond in this way). There are, however, cells in the mammalian striate cortex that exhibit this behavior (the complex cells).

How these cells come to act this way remains under study. However, most current theories fit a common organizational structure [4], shown in Figure 12.12. The input receptive fields are sensitive both to the spatial location and spatial frequency of the stimulus. The role, if any, of orientation is not widely agreed upon. The receptive field outputs are combined, most likely in a nonlinear fashion, in the directionally sensitive subunits to produce an output highly dependent either on the direction or velocity (or both direction and velocity) of the stimulus. The output of these subunits is then integrated both spatially and temporally.

Consider the hypothetical directionally sensitive mechanism in more detail for the case of rightward moving patterns (Figure 12.13). For example, suppose the receptive fields are symmetric, and C is a comparator that requires both inputs to be high to output a high value. If a pattern that stimulates receptive field 1 (RF1) moves a distance $\Delta x$ in time $\Delta t$ (so that it falls within receptive field 2 [RF2]), then the comparator will "fire."

Although it is simple, such a model establishes a basic link between moving patterns on the retina and the perception of motion. Additional insight can be obtained by considering the problem from a systems perspective.

## 12.3.2 Psychophysics of Motion Perception

### 12.3.2.1 Spatial Frequency Response

In the case of spatial vision, much can be understood by modeling the visual system as shown in Figure 12.14. The characteristics of the filter $H(\xi_x, \xi_y)$ have been estimated by determining the threshold visibility of sine wave gratings. The resulting measurements indicate visual sensitivity as a function of spatial frequency that is approximately lowpass in nature. The response peaks in the vicinity of 5 cycles/degree, and falls off rapidly beyond 10 cycles/degree.

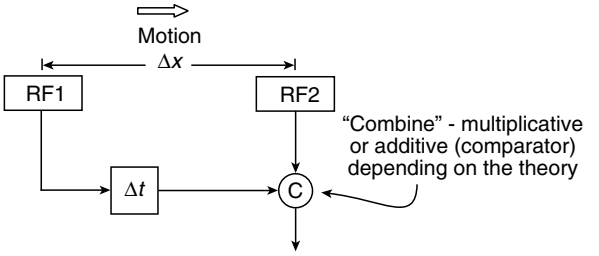



**FIGURE 12.13** A mechanism for the directionally sensitive detection of motion.

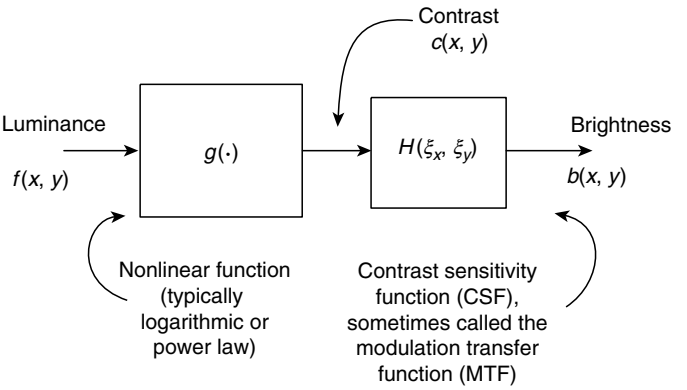



**FIGURE 12.14** A simple block diagram of the modeling spatial vision.

If it were separable (i.e., $H(\xi_x, \xi_y)$ could be determined by finding $H(\xi_x)$ and $H(\xi_y)$ independently), with $H(\xi_x) = H(\xi_y)$, or isotropic, the spatial response could be characterized via a single 1-D function. Although the assumption of separability is often useful, the spatial contrast sensitivity function (CSF) of the human visual system is not, in fact, separable. It has been shown that visual sensitivity is reduced at orientations other than vertical and horizontal. This may be due to the predominance of vertical and horizontal structures in the visual environment, leading to the development or evolution of the visual system to be particularly sensitive at (or conversely, less sensitive away from) these orientations. This is referred to as the "oblique effect."

#### 12.3.2.2 Temporal Frequency Response

The most straightforward approach to extending the above spatial vision model to include motion is to modify the CSF to include temporal frequency sensitivity, so that $H(\xi_x, \xi_y)$ becomes $H(\xi_x, \xi_y, \xi_t)$.

One way to estimate the temporal frequency response of the visual system is to measure the flicker response. Although the flicker response varies with intensity and with the spatial frequency of the stimulus, it is again generally lowpass, with a peak in response in the vicinity of 10 Hz. The attenuation of the response above 10 Hz increases rapidly, so that at 60 Hz (the field rate of NTSC television), the flicker response is very low.

It is natural, as in the 2-D case, to ask whether the spatiotemporal-frequency response $H(\xi_x, \xi_y, \xi_t)$ is separable with respect to the temporal frequency. There is evidence to believe that this is not the case. The flicker-response curves for high and low spatial frequency patterns do not appear consistent with a separable spatiotemporal response.

#### 12.3.2.3 Reconstruction Error

To a first approximation, the data discussed above indicate that the HVS behaves as a 3-D lowpass filter, with bandlimits (for bright displays) at 60 cycles/degree along the spatial frequency axes, and 70 Hz temporally. This approximation is useful in understanding errors that may occur in reconstructing a continuous spatiotemporal signal from a sampled one. Consider the case of an image undergoing simple translational motion. This spatiotemporal signal occupies an oblique plane in the frequency domain. With sampling, the spectrum is replicated (with periods determined by the sampling frequencies along the respective dimensions) to fill the infinite 3-D volume. The spectrum of a sufficiently sampled (aliasing-free) image sequence produced in this way is shown in Figure 12.15.

The 3-D lowpass reconstruction filter (the spatiotemporal CSF) can be approximated as an ideal lowpass filter, as shown in Figure 12.16. As long as the cube in Figure 12.16 completely encloses the spectrum centered at DC, without including neighboring spectra, there is no reconstruction error. This case included no aliasing. If aliasing is included (the sample rate during acquisition is too low) then the aliased components will be visible only if they fall within the passband of the CSF filter.

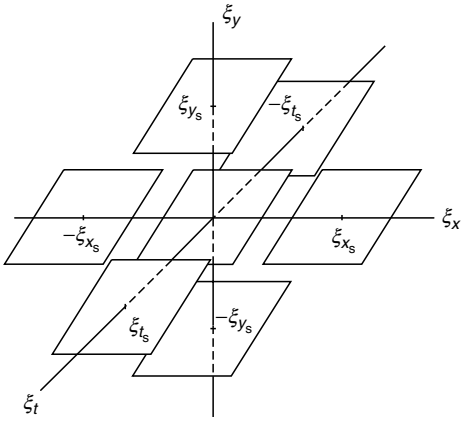



**FIGURE 12.15** Spectrum of a sampled image undergoing uniform translational motion.

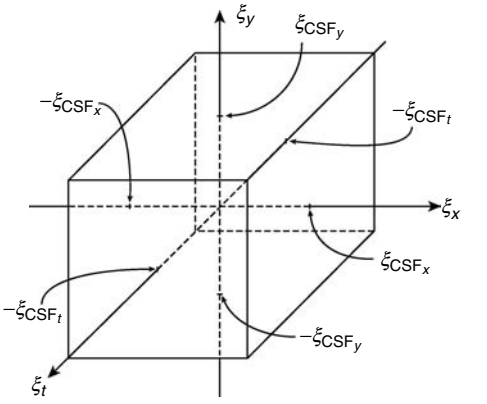



**FIGURE 12.16** An ideal 3-D lowpass reconstruction filter, with cutoff frequencies determined by the spatiotemporal contrast sensitivity function (CSF).

The above frequency domain analysis explains some important aspects of human visual motion perception. Other observations are not as easily explained in this way, however. As observed in Ref. [5], perceived motion is local (different motions can be seen in different areas of the visual field) and spatial-frequency specific (individual motion sensors respond differently (selectively) to different spatial frequencies). These two observations suggest an underlying representation that is local in both the spatiotemporal and spatiotemporal-frequency domains. Examples of such representations are discussed in the following section.

### 12.3.3 Effects of Eye Motion

The analysis of motion perception above assumed a "passive" view. That is, any change in the pattern of light on the retinal surface is due to motion in the scene being viewed. That this is not the case can be seen by considering the manner in which static images are viewed. They are not viewed as a whole, but in a series of "jumps" from position to position. These jumps are referred to as "saccades" (meaning jolt or jerk in French).

Even at the positions where the eye is at rest, it is not truly static. It undergoes very small motions (microsaccades) of 1–2 min of arc. In fact, the eye is essentially never at rest. It has been shown that if the eye is stabilized, vision fades away after about a second. The relevance of this to the current discussion is that although the eye is in constant motion, so that the intensity patterns on the retina are constantly changing, when viewing a static scene no motion is perceived. Similar behavior is observed when viewing dynamic scenes [6]. Obviously, however, in the case of dynamic scenes, motion is often perceived (even though the changes in intensity patterns on the retina are not necessarily greater than for static images).

Two hypotheses might explain these phenomena. The first is that the saccades are so fast that they are not sensed by the visual system. However, this does not account for the fact that motion is seen in dynamic scenes, but not static ones. The second is that the motion sensing system is turned off under some circumstances (the theory of corollary discharge). The basic idea is that the motor signals that control eye movement are also involved in the perception of motion, so that when intensity patterns on the retina change and there is a motor signal present, no motion is perceived. When intensity patterns change but there is no motor signal, or if there is no change in intensity patterns but there is a motor signal, motion is perceived. The later situation corresponds to the tracking of moving objects (smooth pursuit). The first hypothesis (the less plausible of the two) can be easily modeled with temporal linear filters. The second, more interesting behavior can be modeled with a simple comparator network.

## 12.4 Image Sequence Representation

### 12.4.1 What Does Representation Mean?

The term "representation" may require some explanation. Perhaps the best way to do so is to consider some examples of familiar representations. For simplicity, 2-D examples are used. Extension to 3-D is relatively straightforward.

#### 12.4.1.1 Pixel Representation

The pixel representation is so common and intuitive that it is usually considered to be "the image." More precisely, however, it is a linear sum of weighted impulses:

$$u(m,n) = \sum_{m'=0}^{N-1} \sum_{n'=0}^{N-1} u(m',n')\delta(m-m',n-n') \tag{12.25}$$

where $u(m, n)$ is the image, $u(m', n')$ are the coefficients of the representation (numerically equal to the pixel values in this case) and the $\delta(m-m', n-n')$ play the role of basis functions.

#### 12.4.1.2 DFT

The next most familiar representation (at least to engineers) is the DFT, in which the image is expressed in terms of complex exponentials:

$$u(m,n) = \frac{1}{N^2} \sum_{h=0}^{N-1} \sum_{k=0}^{N-1} v(h,k) W_N^{-hm} W_N^{-kn} \tag{12.26}$$

where $0 \leq m, n \leq N-1$ and

$$W_N = \mathrm{e}^{\frac{-j2\pi}{N}} \tag{12.27}$$

In this case, $v(h, k)$ are the coefficients of the representation and the 2-D complex exponentials $W_N^{-hm} W_N^{-kn}$ are the basis functions.

The choice of one representation over the other (pixel vs. Fourier) for a given application depends on the image characteristics that are of most interest. The pixel representation makes the spatial organization of intensities in the image explicit. Since this is the basis of the visual stimulus, it seems more natural. The Fourier representation makes the composition of the image in terms of complex exponentials (frequency components) explicit. The two representations emphasize their respective characteristics (spatial vs. frequency), to the exclusion of all others. If a mixture of characteristics is desired, different representations must be used.

### 12.4.2 Spatial/Spatial-Frequency Representations

A natural mixture is to combine frequency analysis with spatial location. An example of a 1-D representation of this type (a time/frequency representation) is a musical score. The need to know not only what the frequency content of a signal is but also where in the signal the frequency components exist is common to many signal, image, and image sequence processing tasks [7]. There are a variety of approaches [8,9] that could be taken to developing a representation to facilitate these tasks. The most intuitive approach is the finite-support Fourier transform.

#### 12.4.2.1 Finite-Support Fourier Transform

This approach to local frequency decomposition has been used for many years for the analysis of time-varying signals. In the 2-D continuous case,

$$F_{x,y}(\xi_x, \xi_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_{x,y}(x', y') e^{-j2\pi(\xi_x x' + \xi_y y')} \, dx' \, dy' \tag{12.28}$$

where

$$f_{x,y}(x', y') = f(x', y') h(x - x', y - y'), \tag{12.29}$$

$f(x', y')$ is the original image, and $h(x - x', y - y')$ is a window centered at $(x, y)$.

The properties of the transform depend a great deal on the properties of the window function. Under certain circumstances (i.e., for certain windows), the transform is invertible. The most obvious case is for nonoverlapping (e.g., rectangular) windows.

The windowed transform idea can, of course, be applied to other transforms, as well. An example that is of substantial practical interest is the discrete cosine transform, with a rectangular nonoverlapping window:

$$F(h, k) = \alpha(h)\alpha(k) \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} f(m, n) \cos\left(\frac{(2m+1)h\pi}{2N}\right) \cos\left(\frac{(2n+1)k\pi}{2N}\right) \tag{12.30}$$

where $h, k = 0, 1, \ldots, N-1$,

$$\alpha(h) = \begin{cases} \sqrt{\frac{1}{N}} & \text{for h = 0} \\ \sqrt{\frac{2}{N}} & \text{otherwise,} \end{cases} \tag{12.31}$$

$\alpha(k)$ is defined similarly, and the window dimensions are $N \times N$. This transform is the basis for the well-known JPEG and MPEG compression algorithms.

#### 12.4.2.2 Gabor Representation

This representation was first proposed for 1-D signal analysis by Dennis Gabor in 1946 [10]. In 2-D [11], an image can be represented as the weighted sum of functions of the form

$$g(x, y) = \hat{g}(x, y) e^{j2\pi\left[\xi_{x_0}(x - x_0) + \xi_{y_0}(y - y_0)\right]} \tag{12.32}$$

where

$$\hat{g}(x, y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\frac{1}{2}\left[\left(\frac{x - x_0}{\sigma_x}\right)^2 + \left(\frac{y - y_0}{\sigma_y}\right)^2\right]} \tag{12.33}$$

is a 2-D Gaussian function, $\sigma_x$ and $\sigma_y$ determine the extent of the Gaussian along the respective axes, $(x_0, y_0)$ is the center of the function in the spatial domain, and $(\xi_{x_0}, \xi_{y_0})$ is the center of support in the frequency domain. A representative example of a Gabor function is shown in Figure 12.17.

Denoting the distance between spatial centers as $D$ and the distance between their centers of support in the frequency domain as $W$, the basis is complete if $W \times D = 2\pi$. These functions have a number of interesting aspects. They achieve the lower limits of the Heisenberg uncertainty inequalities:

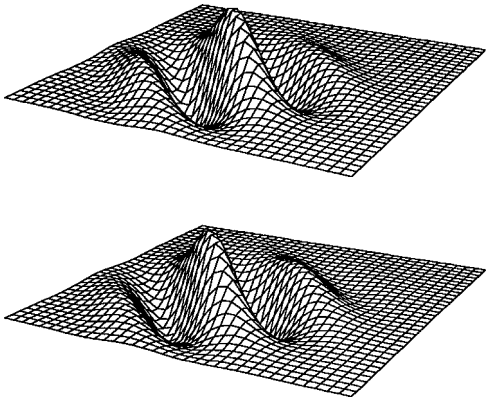


**FIGURE 12.17** Real (top) and imaginary (bottom) parts of a representative 2-D Gabor function.

$$\Delta x \cdot \Delta \xi_x \geq \frac{1}{4\pi}, \Delta y \cdot \Delta \xi_y \geq \frac{1}{4\pi} \tag{12.34}$$

where $\Delta x$, $\Delta y$, $\Delta \xi_x$, and $\Delta \xi_y$ are the effective widths of the functions in the spatial and spatial-frequency domains. By this measure, then, these functions are optimally local. Their real and imaginary parts also agree reasonably well with measured receptive field profiles. However, the basis is not orthogonal. Specifically, the Gabor transform is not equivalent to the finite-support Fourier transform with a Gaussian window. For a cross section of the state of the art in Gabor transform-based analysis, see Ref. [12].

### 12.4.2.3 Derivative of Gaussian Transform

In 1987, Young [13] proposed a receptive field model based on the Gaussian and its derivatives. These functions, like the Gabor functions, are spatially and spectrally local and consist of alternating regions of excitation and inhibition in a decaying envelope. Young showed that Gaussian derivative functions more accurately model the measured receptive field data than do the Gabor functions [14].

In Ref. [15], a spatial or spatial-frequency representation based on shifted versions of the Gaussian and its derivatives was introduced (the derivative of Gaussian transform [DGT]). As with the Gabor transform, although this transform is nonorthogonal, with a suitably chosen basis it is invertible. The DGT has the significant practical advantage over the Gabor transform that both the basis functions and coefficients of expansion are real valued.

The family of 2-D separable Gaussian derivatives centered at the origin can be defined as

$$\begin{aligned} g_{0,0}(x, y) &= g_0(x) g_0(y) \\ &= \mathrm{e}^{-(x^2+y^2)}/2\sigma^2 \end{aligned} \tag{12.35}$$

$$\begin{aligned} g_{m,n}(x, y) &= g_m(x) g_n(y) \\ &= \frac{\mathrm{d}^{(m)}}{\mathrm{d}x^{(m)}} g_0(x) \frac{\mathrm{d}^{(n)}}{\mathrm{d}y^{(n)}} g_0(y) \end{aligned} \tag{12.36}$$

This set can then be shifted to any desired location. The variance $\sigma$ defines the extent of the functions in the spatial domain. There is an inverse relationship between the spatial and spectral extents, and the value of this variable may be constant or may vary with context.

The 1-D Gaussian derivative function spectra are bimodal (except for that of the original Gaussian which is itself a Gaussian) with modes centered at $\pm\Omega_m$ rad/pixel:

$$\Omega_m = \frac{\sqrt{m}}{\sigma} \tag{12.37}$$

where $m$ is the derivative order. The order of derivative necessary to center a mode at a particular frequency is therefore

$$m = (\Omega_m \sigma)^2 \tag{12.38}$$

#### 12.4.2.4 Wigner Distribution

The examples above indicate that a local frequency representation need not have an orthogonal basis. In fact, it need not even be linear. The Wigner distribution (WD) was introduced by Eugene Wigner in 1932 [16] for use in quantum mechanics (in 1-D). In 2-D, the WD can be written as

$$W_f(x, y, \xi_x, \xi_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f\left(x + \frac{\alpha}{2}, y + \frac{\beta}{2}\right) f^*\left(x - \frac{\alpha}{2}, y - \frac{\beta}{2}\right) \mathrm{e}^{-j2\pi(\alpha\xi_x + \beta\xi_y)} \mathrm{d}\alpha \, \mathrm{d}\beta \tag{12.39}$$

where the "*" (asterisk) denotes complex conjugation. The WD is real valued, so does not have an explicit phase component (as seen in, e.g., the Fourier transform). A number of discrete approximations to this distribution (sometimes referred to as pseudo-WDs) have also been formulated.

### 12.4.3 Spatial/Scale Representations (Wavelets)

Scale is a concept that has proven very powerful in many applications, and may under some circumstances be considered as fundamental as frequency. Given a set of (1-D) functions

$$W_{jk}(x) = W(2^j x - k) \tag{12.40}$$

where the indices $j$ and $k$ correspond to dilation (change in scale) and translation, respectively, a signal decomposition

$$f(x) = \sum_j \sum_k b_{jk} W_{jk}(x) \tag{12.41}$$

emphasizes the scale (or resolution) characteristics of the signal (specified by $j$) at specific points along $x$ (specified by $k$), yielding a multiresolution description of the signal.

A class of functions $W_{jk}(x)$ that have proven extremely useful are referred to as wavelets. A detailed discussion of wavelets is beyond the scope of this chapter (refer to Refs. [17–19] for excellent treatments of this topic). However, an important aspect of any representation (including wavelets) is the resolution of the representation, and how it can be measured.

### 12.4.4 Resolution

In dealing with joint representations, resolution is a very important issue. It arises in a number of ways. In discussing the Gabor representation, it was noted that the functions minimized the uncertainty inequalities, e.g.,

$$\Delta x \cdot \Delta \xi_x \geq \frac{1}{4\pi} \tag{12.42}$$

Note that it is the product that is minimized. Arbitrarily high resolution cannot be achieved in both domains simultaneously, but can be traded between the two domains at will. The proper balance depends on the application. It should be noted that the effective width measures $\Delta x$, $\Delta \xi_x$, etc., (normalized second moment measures) are not the only way to define resolution. For example, the degree of energy concentration could be used (leading to a different optimal set of functions, the prolate spheroidal functions). The appropriateness of the various measures again depends on the application. Their biological (psychophysical) relevance remains to be determined.

All the above points are relevant for both spatial/spatial-frequency and spatial/scale representations (wavelets). Wavelets, however, present some special considerations. Suppose one wishes to compare the

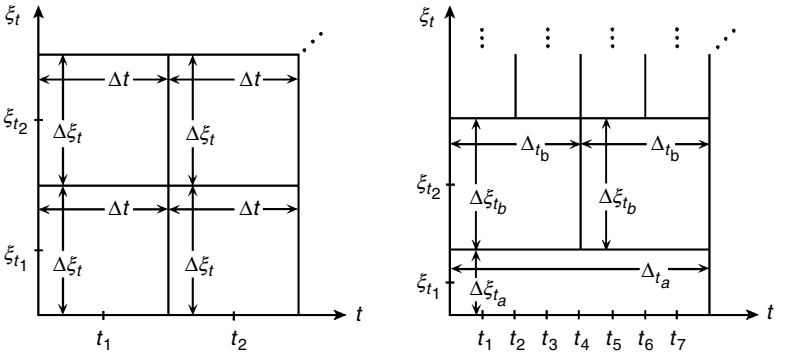



**FIGURE 12.18** Resolution of a time/frequency representation and a wavelet representation in the time–frequency plane.

resolutions of time/frequency and wavelet decompositions? Specifically, what is the resolution of a multiresolution method? This question can be illustrated by considering the 1-D case, and examining the behavior of the two methods in the time–frequency plane (Figure 12.18).

In the time/frequency representation, the dimensions $\Delta t$ and $\Delta\xi_t$ remain the same throughout the time–frequency plane. In wavelet representations, the dimensions vary but their product remains constant. The resolution characteristics of wavelets may lead one to believe that the uncertainty of a wavelet decomposition may fall below the bound in Equation 12.42. This is not the case. The trade-off between $\Delta t$ and $\Delta\xi_t$ simply varies. The fundamental limit remains.

A final point relates more specifically to the representation of image sequences. The HVS has a specific (band-limited) spatiotemporal-frequency response. Beyond indicating the maximum perceivable frequencies (setting an upper bound on resolution), it seems feasible to exploit this point further, to achieve a more efficient representation. Recalling the relationship between motion and temporal frequency, a surface with high spatial frequency components, moving quickly, has high temporal frequency components. When it is static, it does not. The characteristics of the spatiotemporal CSF may lead us to the conclusions that static regions of an image require little temporal resolution, but high spatial resolution, and that regions in an image undergoing significant motion require less spatial resolution (due to the lowered sensitivity of the CSF), but require high temporal resolution (for smooth motion rendition).

The first conclusion is essentially correct (although not trivial to exploit). The second conclusion, however, neglects eye tracking. If the eye is tracking a moving object, the spatiotemporal-frequency characteristics experienced by the viewer are very similar to those in the static case. That is, visual sensitivity to spatial structure is not reduced significantly.

## 12.5 Computation of Motion

There are many approaches to the computation of motion (or, more precisely, the estimation of motion based on image data). Before examining some of these approaches in more detail, however, it is worthwhile to review the relationship between the motion in a scene and the changes observed in an image of the scene.

### 12.5.1 Motion Field

The motion field [20] is determined by establishing a correspondence between the motion of points in the scene (the real world) and the motion of points in the image plane. This correspondence is found

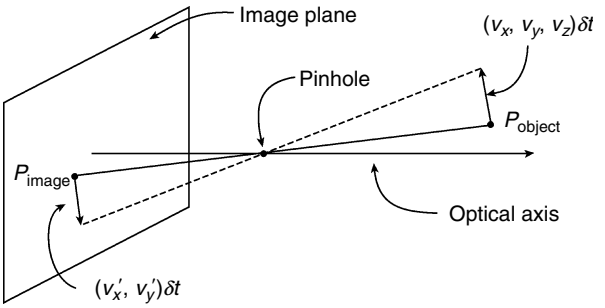



**FIGURE 12.19** Motion field based on a simple pinhole camera model.

geometrically, and is independent of the brightness patterns in the scene (e.g., the presence or absence of surface textures, changes in luminance, etc.).

Consider the situation in Figure 12.19. At a particular instant of time, a point $P_{\text{image}}$ in the image corresponds to some point $P_{\text{object}}$ on the surface of an object. The two points are related via the perspective projection equation. Now, suppose the object point $P_{\text{object}}$ has velocity $(v_x, v_y, v_z)$ relative to the camera. The result is a velocity $(v_x', v_y')$ for the point $P_{\text{image}}$ in the image plane. The relationship between the velocities can be found by differentiating the perspective projection equation with respect to time. In this way, a velocity vector can be assigned to each image point, yielding the motion field.

## 12.5.2 Optical Flow

Usually, the intensity patterns in the image move as the objects to which they correspond move. Optical flow is the motion of these intensity patterns. Ideally, optical flow and the motion field correspond. However, this need not always be the case. For a perfectly uniform sphere rotating in front of an imaging system, there is shading over the surface of the sphere (because of the shape of the sphere), but it does not change with time. The optical flow is zero everywhere, while the motion field is not. For a fixed sphere illuminated by a moving light source, the shading changes with time, although the sphere is not in motion. The optical flow is nonzero, while the motion field is zero.

Furthermore, optical flow is not uniquely determined by local information in the changing image. Consider, for example, a region with uniform brightness that does not vary with time. The most likely optical flow value is zero, but (as long as there are corresponding points of equal brightness in both images) there are many correct flow vectors. What we would like is the motion field, but what we have access to is optical flow. Fortunately, the optical flow is usually not too different from the motion field.

## 12.5.3 Calculation of Optical Flow

There are a wide variety of approaches to the calculation of optical flow. The first, below, is a conceptually simple yet very widely used method. This approach is particularly popular for video compression, and is essentially that used in MPEG-1, 2, and 4.

### 12.5.3.1 Optical Flow by Block Matching

The calculation of optical flow by block matching is the most commonly used motion estimation technique. The basic approach is as follows. Given two successive images from a sequence, the first image is partitioned into nonoverlapping blocks (e.g., $8 \times 8$ pixels in size, Figure 12.20 [left]). To find the motion vector for each block, the similarity (e.g., via mean squared error) between the block and the intensities in the neighborhood of that block in the next frame (Figure 12.20 [right]) is calculated. The location that shows the best match is considered the location to which the block has moved. The motion vector for the block is the vector connecting the center of the block in frame $n$ to the location of the best match in frame $n+1$.

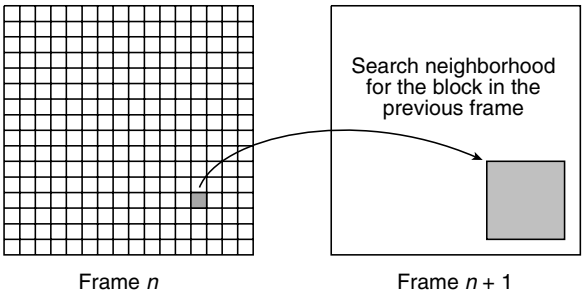



**FIGURE 12.20** Motion estimation by block matching.

The approach is simple. There are, however, a number of things to consider. The size of the search neighborhood must be established, which in turn determines the maximum velocity that can be estimated. The search strategy must be decided, including the need to evaluate every potential match location and the precision with which the match locations must be determined (e.g., is each pixel a potential location? Is subpixel accuracy required?). The amount of computation time/power available is a critical factor in these decisions. Even in its simplest form, block matching is computationally intensive. If motion estimates must be computed at frame rate (in 1/30 of a second or less) this will have a strong effect on the algorithm design. A detailed discussion of these and related issues can be found in Ref. [21].

### 12.5.3.2 Optical Flow via Intensity Gradients

The calculation of optical flow via intensity gradients, as proposed by Horn and Shunck [22], is a classical approach to motion estimation.

Let $f(x, y, t)$ be the intensity at time $t$ for the image point $(x, y)$, and let $r_x(x, y)$ and $r_y(x, y)$ be the $x$ and $y$ components of the optical flow at that point. Then for a small time interval $\delta t$,

$$f\left(x + \underbrace{r_x \delta t}_{\delta x}, y + \underbrace{r_y \delta t}_{\delta y}, t + \delta t\right) = f(x, y, t). \tag{12.43}$$

This single equation is not sufficient to determine $r_x$ and $r_y$. It can, however, provide a constraint on the solution. Assuming that intensity varies smoothly with $x$, $y$, and $t$, the left-hand side of Equation 12.43 can be expanded using Taylor's series:

$$f(x, y, t) + \delta x \frac{\partial f}{\partial x} + \delta y \frac{\partial f}{\partial y} + \delta t \frac{\partial f}{\partial t} + \text{higher-order terms} = f(x, y, t) \tag{12.44}$$

Ignoring the higher-order terms, canceling $f(x, y, t)$, dividing by $\delta t$, and letting $\delta t \to 0$,

$$\frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} + \frac{\partial f}{\partial t} = 0 \tag{12.45}$$

or

$$f_x r_x + f_y r_y + f_t = 0 \tag{12.46}$$

where $f_x$, $f_y$, and $f_t$ are estimated from the image sequence.

This equation is called the optical flow constraint equation, since it constrains $r_x$ and $r_y$ of the optical flow. The values of $(r_x, r_y)$ that satisfy the constraint equation lie on a straight line in the $(r_x, r_y)$ plane.

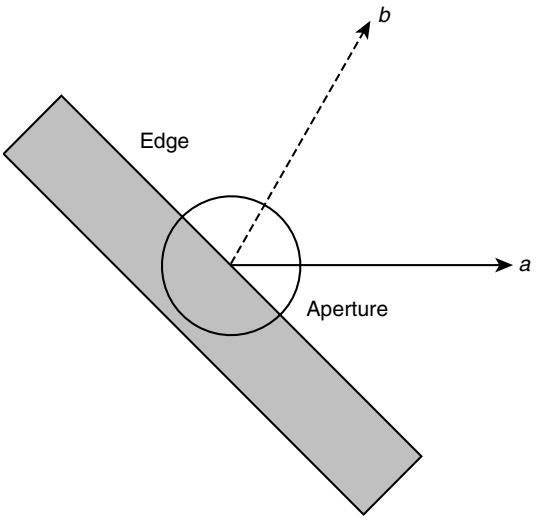



**FIGURE 12.21** An instance of the aperture problem.

A local brightness measurement can identify the constraint line, but not a specific point on the line. Note that this problem cannot really be solved via, e.g., adding an additional constraint. It is a fundamental aspect of the image data. A true solution cannot be guaranteed, but a solution can be found.

To view this limitation in another way, the constraint equation can be rewritten in vector form, as

$$(f_x, f_y) \cdot (r_x, r_y) = -f_t \tag{12.47}$$

so that the component of optical flow in the direction of the intensity gradient $(f_x, f_y)^T$ is

$$\frac{f_t}{\sqrt{f_x^2 + f_y^2}} \tag{12.48}$$

However, the component of the optical flow perpendicular to the gradient (along isointensity contours) cannot be determined. This is a manifestation of the aperture problem. If the motion of an oriented element is detected by a unit that is small compared with the size of the moving element, the only information that can be extracted is the component of motion perpendicular to the local orientation of the element. For example, looking at a moving edge through a small aperture (Figure 12.21), it is impossible to tell whether the actual motion is in the direction of *a* or *b*.

One way to work around this limitation is to impose an explicit smoothness constraint. Motion was implicitly assumed smooth earlier, when a Taylor's expansion was used and when the higher-order terms were ignored. Following this approach, an iterative scheme for finding the optical flow for the image sequence can be formulated:

$$\begin{aligned} r_x(k,l)^{n+1} &= \overline{r_x(k,l)}^n - \frac{\lambda f_x^2 \overline{r_x(k,l)}^n + \lambda f_x f_y \overline{r_y(k,l)}^n + \lambda f_x f_t}{1 + \lambda(f_x^2 + f_y^2)} \\ &= \overline{r_x(k,l)}^n - \lambda f_x \frac{f_x \overline{r_x(k,l)}^n + f_y \overline{r_y(k,l)}^n + f_t}{1 + \lambda(f_x^2 + f_y^2)} \end{aligned} \tag{12.49}$$

and

$$r_y(k,l)^{n+1} = \overline{r_y(k,l)}^n - \lambda f_y \frac{f_x \overline{r_x(k,l)}^n + f_y \overline{r_y(k,l)}^n + f_t}{1 + \lambda(f_x^2 + f_y^2)} \tag{12.50}$$

where the superscripts $n$ and $n+1$ indicate the iteration number, $\lambda$ is a parameter allowing a trade-off between smoothness and errors in the flow constraint equation, and $\overline{r_x(k,l)}$ and $\overline{r_y(k,l)}$ are local averages of $r_x$ and $r_y$. The updated estimates are thus the average of the surrounding values, minus an adjustment (which in velocity space is in the direction of the intensity gradient).

The previous discussion relied heavily on smoothness of the flow field. However, there are places in image sequences where discontinuities should occur. In particular, the boundaries of moving objects should exhibit discontinuities in optical flow. One approach taking advantage of smoothness but allowing

discontinuities is to apply segmentation to the flow field. In this way, the boundaries between regions with smooth optical flow can be found, and the algorithm can be prevented from smoothing over these boundaries. Because of the "chicken-and-egg" nature of this method (a good segmentation depends on a good optical flow estimate, which depends on a good segmentation, etc.), it is best applied iteratively.

#### 12.5.3.3 Spatiotemporal-Frequency-Based Methods

It was shown in Section 12.2.3 that motion can be considered in the frequency domain, as well as in the spatial domain. A number of motion estimation methods have been developed with this in mind. If the sequence to be analyzed is very simple (has only a single motion component, for example) or if motion detection alone is required, the Fourier transform can be used as the basis for motion analysis, as examined in Refs. [23–25]. However, owing to the global nature of the Fourier transform, it cannot be used to determine the location of the object in motion. It is also poorly suited for cases in which multiple motions exist (i.e., when the scene of interest consists of more than one object moving independently), since the signatures of the different motions are difficult (impossible, in general) to separate in the Fourier domain. As a result, although Fourier analysis can be used to illustrate some interesting phenomena, it cannot be used as the basis of motion analysis methods for the majority of sequences of practical interest.

To identify the locations and motions of objects, frequency analysis localized to the neighborhoods of the objects is required. Windowed Fourier analysis has been proposed for such cases [26]. However, the accuracy of a motion analysis method of this type is highly dependent on the resolution of the underlying transform, in both the spatiotemporal and spatiotemporal-frequency domains. It is known that the windowed Fourier transform does not perform particularly well in this regard. Filterbank-based approaches to this problem have also been proposed, as in Ref. [27]. The methods examined below each exploit the frequency domain characteristics of motion, and provide spatiotemporally localized motion estimates.

#### 12.5.3.4 Optical Flow via the 3-D Wigner Distribution

Jacobson and Wechsler [28] proposed an approach to spatiotemporal-frequency-based derivation of optical flow using the 3-D WD. Extending the 2-D definition given earlier, the 3-D WD can be written as

$$W_f(x, y, t, \xi_x, \xi_y, \xi_t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f\left(x + \frac{\alpha}{2}, y + \frac{\beta}{2}, t + \frac{\tau}{2}\right) \cdot f^*\left(x - \frac{\alpha}{2}, y - \frac{\beta}{2}, t - \frac{\tau}{2}\right) \cdot e^{-j2\pi(\alpha\xi_x + \beta\xi_y + \tau\xi_t)} d\alpha\, d\beta\, d\tau \tag{12.51}$$

It can be shown that the WD of a linearly translating image with velocity $\vec{r} = (r_x, r_y)$ is

$$W_f(x, y, t, \xi_x, \xi_y, \xi_t) = \delta(r_x\xi_x + r_y\xi_y + \xi_t) \cdot W_f(x - r_x t, y - r_y t, \xi_x, \xi_y) \tag{12.52}$$

which is nonzero only when $r_x\xi_x + r_y\xi_y + \xi_t = 0$.

For a linearly translating image, then, the local spectra $W_{f_{x,y,t}}(\xi_x, \xi_y, \xi_t)$ contain energy only in a plane (as in the Fourier case) the slope of which is determined by the velocity. Jacobson and Wechsler [28] proposed to find this plane by integrating over the possible planar regions in these local spectra (via a so-called velocity polling function), using the plane of maximum energy to determine the velocity.

#### 12.5.3.5 Optical Flow Using 3-D Gabor Filters

Heeger [29] proposed the use of 3-D Gabor filters to determine this slope. Following the definition discussed for 2-D, a 3-D Gabor filter has the impulse response

$$g(x, y, t) = \hat{g}(x, y, t) e^{i2\pi[\xi_{x_0}(x - x_0) + \xi_{y_0}(y - y_0) + \xi_{t_0}(t - t_0)]} \tag{12.53}$$

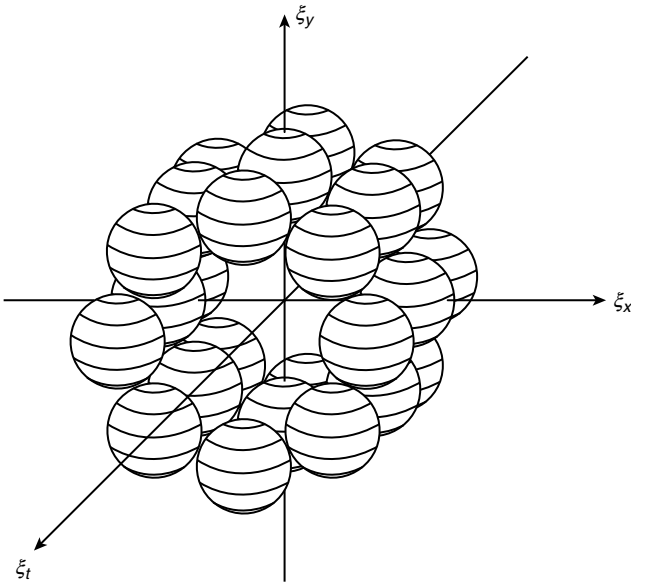



**FIGURE 12.22** The (stylized) power spectra of a set of 3-D Gabor filters.

where

$$\hat{g}(x, y, t) = \frac{1}{(2\pi)^{\frac{3}{2}}\sigma_x\sigma_y\sigma_t} e^{-\frac{1}{2}\left[\left(\frac{x-x_0}{\sigma_x}\right)^2+\left(\frac{y-y_0}{\sigma_y}\right)^2+\left(\frac{t-t_0}{\sigma_t}\right)^2\right]} \tag{12.54}$$

To detect motion in different directions, a family of these filters is defined, as shown in Figure 12.22.

To capture velocities at different scales (high velocities can be thought of as occurring over large scales, since a large distance is covered per unit time), these filters are applied to a Gaussian pyramidal decomposition of the sequence. Given the energies of the outputs of these filters, which can be thought of as sampling spatiotemporal/spatiotemporal-frequency space, the problem is analogous to that shown in Figure 12.23. The slope of the line (corresponding to the slope of the plane that characterizes motion) must be found via a finite set of observations. In this method, this problem is solved under the assumption of a random texture input (the plane in the frequency domain consists of a single constant value).

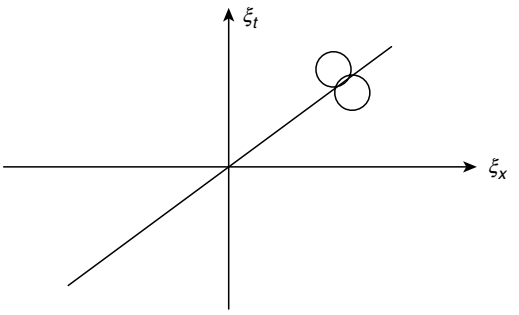



**FIGURE 12.23** Velocity estimation in the frequency domain via estimation of the slope of the spectrum.

#### 12.5.3.6 Optical Flow via the 3-D Gabor Transform

One shortcoming of a filterbank approach (if the filters are not orthogonal or do not provide a complete basis) is the possibility of loss. Using the 3-D Gabor functions as the basis of a transform resolves this problem. A sequence of dimension $N \times M \times P$ can then be expressed at each discrete point $(x_m, y_n, t_p)$ as

$$f(x_m, y_n, t_p) = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} \sum_{l=0}^{L-1} \sum_{q=0}^{Q-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} c_{x_q, y_r, t_s, \xi_{x_j}, \xi_{y_k}, \xi_{t_l}} \cdot g_{x_q, y_r, t_s, \xi_{x_j}, \xi_{y_k}, \xi_{t_l}}(x_m, y_n, t_p) \tag{12.55}$$

where $JKLQRS = NMP$ for completeness, the functions $g_{x_q, y_r, t_s, \xi_{x_j}, \xi_{y_k}, \xi_{t_l}}(x_m, y_n, t_p)$ denote the Gabor basis functions with spatiotemporal and spatiotemporal-frequency centers of $(x_q, y_r, t_s)$ and $(\xi_{x_j}, \xi_{y_k}, \xi_{t_l})$, respectively, and $c_{x_q, y_r, t_s, \xi_{x_j}, \xi_{y_k}, \xi_{t_l}}$ are the associated coefficients. Note that these coefficients are not found by convolving with the Gabor functions, since the functions are not orthogonal. See Ref. [30] for a survey and comparison of methods for computing this transform.

In the case of uniform translational motion, the slope of the planar spectrum is sought, yielding the optical flow vector $\vec{r}$. A straightforward approach to estimating the slope of the local spectra [31,32] is to form vectors of the $\xi_x$, $\xi_y$, and $\xi_t$ coordinates of the basis functions that have significant energy for each point in the sequence at which basis functions are centered. From Equation 12.20, the optical flow vector and the coordinate vectors $\vec{\xi}_x$, $\vec{\xi}_y$, and $\vec{\xi}_t$ at each point are related as

$$\vec{\xi}_t = -\left(r_x \vec{\xi}_x + r_y \vec{\xi}_y\right) = -\mathbf{S}\vec{r} \tag{12.56}$$

where $\mathbf{S} = (\vec{\xi}_x | \vec{\xi}_y)$. An LMS estimate of the optical flow vector at a given point can then be found using the pseudoinverse of $\mathbf{S}$:

$$\vec{r}_{\text{est}} = -(\mathbf{S}^T \mathbf{S})^{-1} \mathbf{S}^T \vec{\xi}_t \tag{12.57}$$

In addition to providing a means for motion estimation, this approach has also proven useful in predicting the apparent motion reversal associated with temporal aliasing [33]. It has also been extended to provide dense [34] and rotational [35] motion estimates.

#### 12.5.3.7 Wavelet-Based Methods

A number of wavelet-based approaches to this problem have also been proposed. In Refs. [36–39], 2-D wavelet decompositions are applied frame-by-frame to produce multiscale feature images. This view of motion analysis exploits the multiscale properties of wavelets, but does not seek to exploit the frequency domain properties of motion. In Ref. [40], a spatiotemporal (3-D) wavelet decomposition is employed, so that some of these frequency domain aspects can be utilized. Leduc et al. explore the estimation of translational, accelerated, and rotational motion via spatiotemporal wavelets in Refs. [41–45]. Decompositions designed and parameterized specifically for the motion of interest (e.g., rotational motion) are tuned to the motion to be estimated.

## 12.6 Image Sequence Compression

Image sequences represent an enormous amount of data (e.g., a 2 h movie at the U.S. HDTV resolution of $1280 \times 720$ pixels, 60 frames/s progressive, with 24 bits/pixel results in 1194 GB of data). This data is highly redundant, and much of it has minimal perceptual relevance. One approach to reducing this volume of data is to apply still image compression to each frame in the sequence (generally referred to as intraframe coding). For example, the JPEG still image compression algorithm can be applied frame by frame (sometimes referred to as motion-JPEG or M-JPEG). This method, however, does not take

advantage of the substantial correlation which typically exists between frames in a sequence. Compression techniques that seek to exploit this temporal redundancy are referred to as interframe coding methods.

### 12.6.1 Motion Compensated Prediction/Transform Coders

Predictive coding is based on the idea that to the degree that all or part of a frame in a sequence can be predicted, that information need not be transmitted. As a result, it is usually the case that the better the prediction, the better the compression that can be achieved. The simplest possible predicator is to assume that successive frames are identical (differential coding). However, the optical flow, which indicates the motion of intensity patterns in the image sequence, can be used to improve the predictor. Motion compensated prediction uses optical flow information, together with a reconstruction of the previous frame, to predict the content of the current frame.

Quantization (and the attendant loss of information) is inherent to lossy compression techniques. This loss, if introduced strategically, can be exploited to produce a highly compressed sequence, with good visual quality. Transforms (e.g., the DCT), followed by quantization, provide a convenient mechanism to introduce (and control) this loss. Following this approach, a hybrid (motion compensated prediction/transform) encoder and decoder are shown in Figures 12.24 and 12.25. This hybrid algorithm (with the addition of entropy coders and decoders) is the essence of the H.261, MPEG-1, MPEG-2, MPEG-4, and U.S. HDTV compression methods [46].

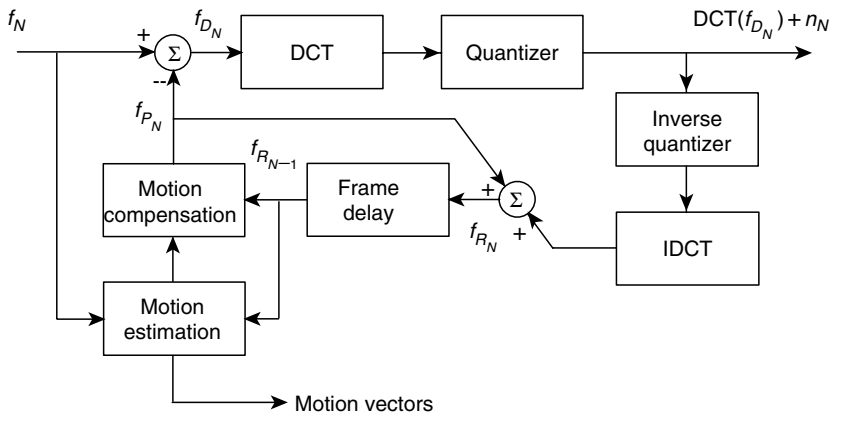



**FIGURE 12.24** An hybrid (predictive/transform) encoder.

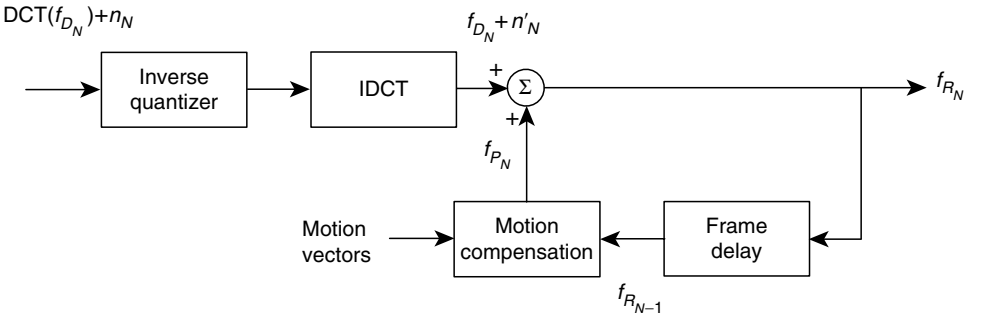



**FIGURE 12.25** A predictive/transform decoder.

## 12.6.2 Perceptually-Based Methods

Although algorithms such as MPEG exploit the properties of visual perception (principally in the formulation of quantization matrices), it is not especially central to the structure of the algorithm. There is, for example, no explicit model of vision underlying the MPEG-1 and -2 algorithms. In perceptually-based (sometimes called second-generation) methods, knowledge of the HVS takes a much more central role. This view of the problem is particularly effective (and necessary) when designing compression algorithms intended to operate at very high compression ratios (e.g., over 200:1).

The methods in this subsection are inspired by specific models of visual perception. The first is an approach based on a very comprehensive vision model, performing spatial and temporal frequency decomposition via filters designed to reflect properties of the HVS. The second and third are techniques using visually relevant transforms (the Gabor and derivative of Gaussian transforms, respectively) in an otherwise conventional hybrid (predictive/transform) framework. Finally, one of the earlier developed methods based on spatiotemporal segmentation (following the contour/texture model of vision) is discussed. Certain aspects of the segmentation-based coding idea are supported in the MPEG-4 standard.

### 12.6.2.1 Perceptual Components Architecture

The perceptual components architecture [47] is a framework for the compression of color image sequences based on the processing thought to take place in the early HVS. It consists of the following steps. The input RGB image sequence is converted into an opponent color space (white/black [WB], red/green [RG], and blue/yellow [BY]). The sequence is filtered spatially with a set of frequency and orientation selective filters, inspired by the frequency and orientation selectivity of the HVS. Filters based on the temporal frequency response of the visual system are applied along the temporal dimension. The filtered sequences are then subsampled using a hexagonal grid, and subsampled by a factor of two in the temporal dimension. Uniform quantization is applied within each subband, with higher frequency subbands quantized more coarsely. The WB (luminance) component is quantized less coarsely overall than the RG and BY (chrominance) components. The first-order entropy of the result provides an estimate of the compression ratio.

Note that there is no prediction or motion compensation. This is a 3-D subband coder, where temporal redundancy is exploited via the temporal filters. For a $256 \times 256$, 8 frame segment of the "football" sequence (a widely used test sequence depicting a play from an American football game), acceptable image quality was achieved for about 1 bit/pixel (from 24 bits/pixel). Although this is not very high compression, the sequence used is more challenging than most. Another contributing factor is that the subsampled representation is eight-thirds the size (in terms of bits) of the original, which must be overcome before any compression is realized.

### 12.6.2.2 Very Low Bit-Rate Coding Using the Gabor Transform

In discussing the Gabor transform previously, it was stated that the basis functions of this transform are optimally (jointly) local. In the context of coding, there are three mechanisms that can be exploited to achieve compression, all of which depend on locality: the local correlation between pixels in the sequence; the bounded frequency response of the human visual system (as characterized by the CSF); and visual masking (the decrease in visual sensitivity near spatial and temporal discontinuities). To take advantage of local spatial correlation, the image representation upon which a compression method is based must be spatially local (which is why images are partitioned into blocks in JPEG, MPEG-1 and -2, most implementations of MPEG-4, H.261-4, etc.). If the CSF is to be exploited (e.g., by quantizing high frequency coefficients coarsely), localization in the spatial-frequency domain is required. To exploit visual masking, spatial locality (of a fairly high degree) is required.

Since the Gabor transform is inherently local in space, the partitioning of the image into blocks is not required (hence no blocking artifacts are observed at high compression ratios). Its spatial locality also provides a mechanism for exploiting visual masking, while its spatial-frequency locality allows the band-limited nature of the HVS to be utilized.

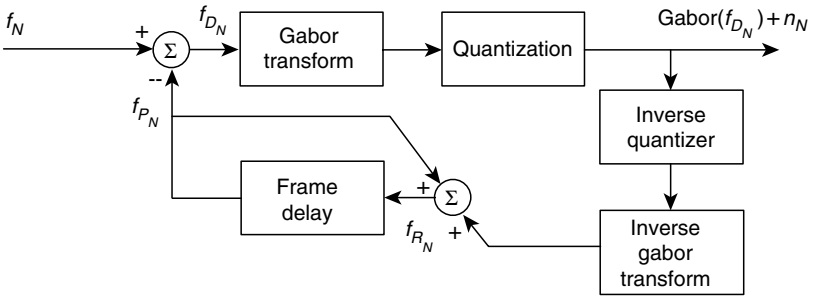



**FIGURE 12.26** A Gabor transform-based video encoder.

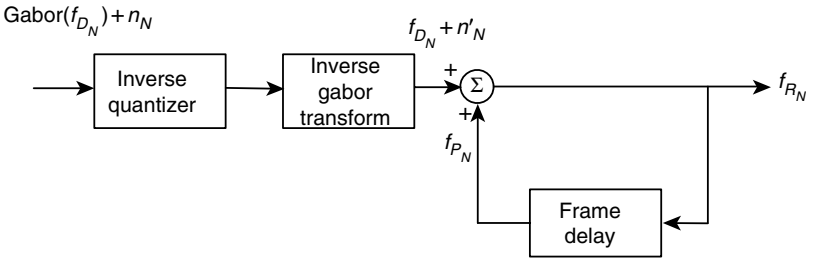



**FIGURE 12.27** Associated Gabor transform-based decoder.

An encoder and decoder based on this transform are shown in Figures 12.26 and 12.27 [48]. Note that they are in the classic hybrid (predictive/transform) form. This codec does not include motion compensation and is for monochrome image sequences.

Applying this method to a 128-by-128, 8 bit/pixel version of the Miss America sequence resulted in reasonable image quality at a compression ratio of approximately 335:1.* At 24 frames/s, the associated bit rate is 9.4 Kbits/s (a bitrate consistent, e.g., with wireless videotelephony).

### 12.6.2.3 Video Coding Using the Derivative of Gaussian Transform

As mentioned previously, the DGT has properties similar to the Gabor transform, but with the practical advantage that it is real valued. This makes it particularly well suited to video compression. In Ref. [49] the hybrid codec structure shown in Figures 12.24 and 12.25 is adapted to the DGT, replacing the DCT (and IDCT), and adapting the quantization scheme to fit the visibility of the DGT basis, via a simple quantization mask.

Comparable results to those of the standard H.261 (DCT-based) codec are obtained for bitrates around 320 Kbits/s (five channels in the $p * 64$ model).

### 12.6.2.4 Object-Based Coding by Split and Merge Segmentation

Object-based coding reflects the fact that scenes are largely composed of distinct objects, and that these objects are perceived as boundaries surrounding fields of shading or texture (the contour/texture theory of vision). Encoding an image or sequence in this way requires segmentation to identify the constituent objects. This view of compression, which also facilitates interaction and editing, underlies the MPEG-4 video compression standard [50]. Although the method that will be described is different in detail from MPEG-4, as one of the earliest documented object-based systems, it illustrates many important aspects of such systems.

In this approach [51], 3-D (spatiotemporal) segmentation is used to reduce the redundant information in a sequence (essentially identifying objects within the sequence), while retaining information

*Not including the initial frame, which is intracoded to 9.1 Kbits (a compression ratio of 14).

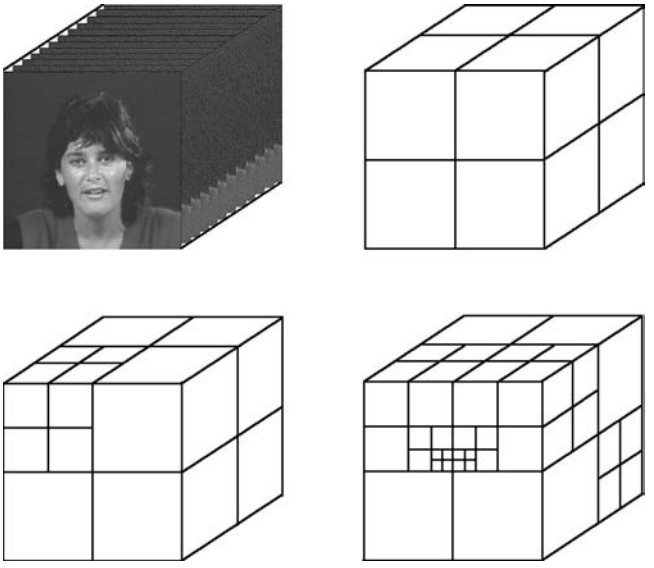


**FIGURE 12.28** Split phase.

critical to the human observer. The sequence is treated as a single 3-D data volume, the voxels of which are grouped into regions via split and merge. The uniformity criterion used for the segmentation is the goodness of fit to a 3-D polynomial. The sequence is then encoded in terms of region boundaries (a binary tree structure) and region interior intensities (the coefficients of the 3-D polynomial).

The data volume is first split such that each region is a parallelepiped over which the gray level variation can be approximated within a specified mean squared error (Figure 12.28). Regions are split by quadrants, following the octree strategy. A region adjacency graph is constructed, with nodes corresponding to each region and links between the nodes assigned a cost indicating the similarity of the regions. A high cost indicates low similarity. Regions are merged, starting with regions with the lowest cost, and the region adjacency graph is updated. The resulting regions are represented using a pyramidal (binary tree) structure, with the regions labeled so that adjacent regions have different labels.

Using 16 frames from the Secretary sequence, the compression ratio achieved was 158:1 (a bitrate of 83 Kbits/s). A total of 5740 parallelepipeds (1000 regions) were used.

## 12.7 Conclusions

In this chapter, we examined some of the fundamental aspects and algorithms in the processing of digital video. Continued improvements in computing performance make many methods that previously required specialized platforms (or were primarily of research interest due to computational requirements) practical. In addition to bringing high-end applications to the desktop, numerous new applications are thus enabled, in areas as diverse as medical imaging, entertainment, and human–computer interaction.

## References

1. J. Wyver. *The Moving Image—An International Histoy of Film, Television and Video.* BFI Publishing, London, 1989.
2. A.B. Watson and A.J. Ahumada. A look at motion in the frequency domain. SIGGRAPH/SIGART Interdisciplinary Workshop MOTION: Representation and Perception, pp. 1–10, Toronto, Canada, April 4–6,1983.

3. A.T. Smith and R.J. Snowden (Eds.). *Visual Detection of Motion*. Academic Press, San Diego, CA, 1994.
4. K. Nakayama. Biological image motion processing: A review. *Vision Research,* 25(5):625–660, 1985.
5. A.B. Watson and A.J. Ahumada. Model of human visual-motion sensing. *Journal of the Optical Society of America A,* 2(2):322–342, 1985.
6. L.B. Stelmach, W.J. Tam, and P. Hearty. Static and dynamic spatial resolution in image coding: An investigation of eye movements. *Proceedings of the SPIE/SPSE Symposium on Electronic Imaging Science and Technology*, vol. 1453, pp. 147–152, San Jose, CA, 1995.
7. T.R. Reed. Local frequency representations for image sequence processing and coding. In A.B. Watson (Ed.), *Digital Images and Human Vision*. MIT Press, Cambridge, MA, 1993.
8. L. Cohen. *Time–Frequency Analysis*. Prentice Hall PTR, Englewood Cliffs, NJ, 1995.
9. R. Tolimieri and M. An. *Time–Frequency Representations*. Birkhäuser, Boston, MA, 1998.
10. D. Gabor. Theory of communication. *Proceedings of the Institute of Electrical Engineers,* 93(26):429–457, 1946.
11. J.G. Daugman. Complete discrete 2-D Gabor transforms by neural networks for image analysis and compression. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 36(7):1169–1179, 1988.
12. H.G. Feichtinger and T. Strohmer (Eds.). *Gabor Analysis and Algorithms*. Birkhäuser, Boston, MA, 1998.
13. R.A. Young. The Gaussian derivative model for spatial vision: I. retinal mechanisms. *Spatial Vision*, 2:273–293, 1987.
14. R.A. Young, oh say can you see? the physiology of vision. *Proceedings of the SPIE-Human Vision, Visual Processing, and Digital Display II*, vol. 1453, pp. 92–123, 1991.
15. J.A. Bloom and T.R. Reed. A Gaussian derivative-based transform. *IEEE Transactions on Image Processing,* 5(3):551–553, 1996.
16. E. Wigner. On the quantum correction for thermodynamic equilibrium. *Physical Review,* 40:749–759, June 1932.
17. M. Vetterli and J. Kovačvic. *Wavelets and Subband Coding*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
18. G. Strang and T. Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, Wellesley, MA, 1996.
19. S. Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, San Diego, CA, 1998.
20. B.K.P. Horn. *Robot Vision*. MIT Press, Cambridge, MA, 1986.
21. G. Tziritas and C. Labit. *Motion Analysis for Image Sequence Coding*. Elsevier, Amsterdam, 1994.
22. B.K.P. Horn and B.G. Shunck. Determining optical flow. *Artificial Intelligence,* 17(1–3):185–203, 1981.
23. H. Gafni and Y.Y. Zeevi. A model for separation of spatial and temporal information in the visual system. *Biological Cybernetics*, 28:73–82, 1977.
24. A. Kojima, N. Sakurai, and J. Kishigami. Motion detection using 3D-FFT spectrum. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 5, pp. 213–216, Minneapolis, MN, April 27–30, 1993.
25. B. Porat and B. Friedlander. A frequency domain algorithm for multiframe detection and estimation of dim targets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(4):398–401, 1990.
26. H. Gafni and Y.Y. Zeevi. A model for processing of movement in the visual system. *Biological Cybernetics*, 32:165–173, 1979.
27. D.J. Fleet and A.D. Jepson. Computation of component image velocity from local phase information. *International Journal of Computer Vision,* 5(1):77–104, 1990.
28. L. Jacobson and H. Wechsler. Derivation of optical flow using a spatiotemporal-frequency approach. *Computer Vision, Graphics, and Image Processing,* 38:29–65, 1987.
29. D.J. Heeger. Optical flow using spatiotemporal filters. *International Journal of Computer Vision,* 1:279–302, 1988.
30. T.T. Chinen and T.R. Reed. A performance analysis of fast Gabor transforms. *Graphical Models and Image Processing,* 59(3):117–127, 1997.
31. T.R. Reed. The analysis of motion in natural scenes using a spatiotemporal/spatiotemporal-frequency representation. *Proceedings of the IEEE International Conference on Image Processing*, pp. I–93–I–96, Santa Barbara, CA, October 26–29, 1997.

32. T.R. Reed. On the computation of optical flow using the 3-D Gabor transform. *Multidimensional Systems and Signal Processing*, 9(4):115–120, 1998.
33. T.R. Reed. A spatiotemporal/spatiotemporal-frequency interpretation of apparent motion reversal. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, vol. 2, pp. 1140–1145, Stockholm, Sweden, July 31–August 6, 1999.
34. J. Magarey and N. Kingsbury. Motion estimation using a complex-valued wavelet transform. *IEEE Transactions on Signal Processing,* 46(4):1069–1084, 1998.
35. Y.-T. Wu, T. Kanade, J. Cohn, and C.-C. Li. Optical flow estimation using wavelet motion model. *Proceedings of the IEEE International Conference on Computer Vision*, pp. 992–998, Bombay, India, January 4–7, 1998.
36. G. Van der Auwera, A. Munteanu, G. Lafruit, and J. Cornelis. Video coding based on motion estimation in the wavelet detail image. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 5, pp. 2801–2804, Seattle, WA, May 12–15, 1998.
37. C.P. Bernard. Discrete wavelet analysis: A new framework for fast optic flow computation. *Proceedings of the fifth European Conference on Computer Vision*, vol. 2, pp. 354–368, Freiburg, Germany, June 2–6, 1998.
38. T.J. Burns, S.K. Rogers, M.E. Oxley, and D.W. Ruck. Discrete, spatiotemporal, wavelet multiresoltion analysis method for computing optical flow. *Optical Engineering,* 33(7):2236–2247, 1994.
39. J.-P. Leduc. Spatio-temporal wavelet transforms for digital signal analysis. *Signal Processing,* 60(1):23–41, 1997.
40. J.-P. Leduc, J. Corbett, M. Kong, V. Wickerhauser, and B. Ghosh. Accelerated spatio-temporal wavelet transforms: An iterative trajectory estimation. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 5, pp. 2781–2784, Seattle, WA, May 12–15, 1998.
41. J.-P. Leduc, J.R. Corbett, and M.V. Wickerhauser. Rotational wavelet transforms for motion analysis, estimation and tracking. *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, pp. 195–199, Chicago, IL, October 4–7, 1998.
42. J.-P. Leduc and J.R. Corbett. Spatio-temporal continuous wavelets for the analysis of motion on manifolds. *Proceedings of the IEEE-SP International Symposium on Time–Frequency and Time-Scale Analysis*, pp. 57–60, Pittsburgh, PA, October 6–9, 1998.
43. J.-P. Leduc, J.R. Corbett, and M.V. Wickerhauser. Rotational wavelet transforms for motion analysis, estimation and tracking. *Proceedings of the IEEE International Conference on Image Processing*, vol. 2, pp. 195–199, Chicago, IL, October 4–7, 1998.
44. M. Kong, J.-P. Leduc, B.K. Ghosh, J. Corbett, and V.M. Wickerhauser. Wavelet based analysis of rotational motion in digital image sequences. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 5, pp. 2777–2780, Seattle, WA, May 12–15, 1998.
45. J. Corbett, J.-P. Leduc, and M. Kong. Analysis of deformational transformations with spatio-temporal continuous wavelet transforms. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 6, pp. 3189–3192, Phoenix, AZ, March 15–19, 1999.
46. A.N. Netravali and B.G. Haskell. *Digital Pictures—Representation, Compression, and Standards.* Plenum Press, New York, 1995.
47. A.B. Watson and C.L.M. Tiana. Color motion video coded by perceptual components. SID '92 Digest of Technical Papers, vol. 23, pp. 314–317, 1992.
48. T.R. Reed and A.E. Soohoo. Very-low-bit-rate coding of image sequences using the gabor transform. *Journal of the Society for Information Display,* 3(2):77–81, 1995.
49. J.A. Bloom and T.R. Reed. On the compression of video using the derivative of Gaussian transform. *Proceedings of the Thirty Second Annual Asilomar Conference on Signals, Systems, and Computers*, vol. 1, pp. 865–869, Pacific Grove, CA, November 1–4, 1998.
50. R. Koenen (Ed.). MPEG-4 Overview. ISO/IEC JTC1/SC29/WG11 N3747, La Baule, October 2000.
51. P. Willemin, T.R. Reed, and M. Kunt. Image sequence coding by split and merge. *IEEE Transactions on Communications,* 39(12):1845–1855, 1991.

# 13

# Low-Power Digital Signal Processing




Alice Wang
*Texas Instruments*

Thucydides Xanthopoulos
*Cavium Networks*


## 13.1 Introduction

During the last few years, signal processing integrated circuits have always introduced the newest low-power digital design techniques. Two main factors motivate this design trend. The first is the abundant proliferation and market penetration of cellular phones. During the first phase in the life cycle of cellular systems, programmable digital signal processors (DSPs) were used to implement the voice coding component. As DSPs became more powerful and flexible, they took over most of the baseband tasks within a cellular handset such as channel coding (convolutional coding and decoding), encryption/decryption, and demodulation/equalization [1]. Consumer preferences placed significant importance on handset size and battery life and this in turn created pressure in the design community to produce higher performance and lower power signal processors. The second motivating factor is the introduction of a new breed of consumer electronic devices such as digital cameras, portable digital video and audio players, and wireless-enabled personal digital assistants that require substantial signal processing capability and at the same time are battery-powered and can benefit substantially from reduced energy consumption.

The low-power trends in the signal processing domain has continued with the introduction of 3G wireless and the corresponding wideband code-division multiple-access (WCDMA) physical transmission channel. Another emerging wireless computing platform for low-power DSPs is the sensor data processing system [2].

Low-power design involves a vertical design process and a global optimization across algorithmic, architectural, circuit, and physical design boundaries. The best algorithm must be selected, which minimizes a weighted average of the number of arithmetic operations, memory accesses, on-chip communication, and silicon area. The right boundary must be achieved between programmability and predefined functionality. Architectural, circuit, and physical design techniques that fully support

the algorithmic selection must then be applied, but at the same time should be allowed to influence such selection in order to achieve optimum results.

A few case studies demonstrate concurrent optimization across all design phases and as a result have achieved impressive results in a few key DSP application areas.

- A low-power chipset for a portable multimedia terminal is power optimized from a system perspective and performs a number of functions such as protocol conversion, synchronization, and video decompression, among others, while consuming under 5 mW of power [3].
- An ultra-low-power programmable DSP for physiological monitoring (heartbeat detection and classification) [4]. The techniques demonstrated include algorithmic design, a balanced hybrid architecture containing both customized and programmable units, and appropriate circuits supporting such architectural choices. The DSP consumes 220 nW at 1.0 V, 1.2 kHz and includes embedded support for harvesting energy from ambient sources.
- A low-power single chip video encoder with embedded dynamic memory uses wavelet filtering and a combination of zero-tree and arithmetic coding of filter coefficients. Hooks for motion estimation are also provided [5]. The chip dissipates on the order of 0.5 mW while compressing an 8-bit gray scale 30 frames/s, $128 \times 128$ video stream.
- A programmable reconfigurable public key processor demonstrates 2–3 orders of magnitude of power reduction compared to software and programmable-logic based implementations while providing similar flexibility and freedom in algorithm selection [6].
- A DSP for a hearing aid chipset featuring 77 MOPS/mW [7]. Algorithmic and architectural optimizations were heavily employed to achieve such a result.

In this chapter, an overview of commonly used low-power techniques in programmable digital signal processors, as well as embedded DSP subsystems for custom applications, are presented.

## 13.2 Power Dissipation in Digital Circuits

This section provides a brief overview of power dissipation basics to render this document self-contained. Five major sources of power dissipation are present in digital circuits:

- Switching or dynamic power ($P_{sw}$)
- Short-circuit or direct-path power ($P_{sc}$)
- Leakage power ($P_{leak}$)
- Gate induced drain leakage ($P_{GIDL}$)
- Static power ($P_{stat}$)

The total chip power is given by the following equation:

$$P_{total} = P_{sw} + P_{sc} + P_{leak} + P_{GIDL} + P_{stat} \tag{13.1}$$

The switching power dissipation is the dominant component and is because of the charging and discharging of all capacitive nodes in the circuit. It is given by the following equation:

$$P_{sw} = aCV_{DD}^2 f \tag{13.2}$$

where

$a$ is the switching activity ($0 \leq a \leq 1$)
$C$ is the total capacitance of all capacitive nodes in the circuit
$V_{DD}$ is the supply voltage
$f$ is the clock frequency

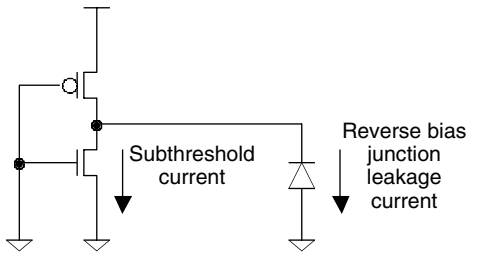



**FIGURE 13.1** Leakage current components.

Short-circuit power ($P_{sc}$) is dissipated when there is a transient direct path from $V_{DD}$ to ground during switching: During the rising (or falling) transition of static CMOS gates from $V_{TN}$ to ($V_{DD} - V_{TP}$) ($V_{DD} - V_{TP}$ to $V_{TN}$) a direct path from $V_{DD}$ to ground exists through a PMOS and NMOS stack that are both in their ON region. If the rise and fall times of the digital circuit are kept well under control (a small fraction of the period), short-circuit power is rarely a design issue. A comprehensive analysis on short-circuit power in static CMOS circuits can be found in Ref. [8].

There are three main types of leakage powers. The first is subthreshold leakage and involves finite channel conductance in the NMOS and PMOS OFF regions. The second is reverse bias junction leakage and it involves source and drain-to-substrate PN junction leakage. The third is gate-induced drain leakage that starts at the drain and terminates in the body. It occurs for large drain-to-source voltage and small gate-to-source voltage [9]. Figure 13.1 shows the subthreshold and junction leakage components.

The subthreshold current is typically the dominant component of leakage power. A simplified subthreshold leakage power model is given by

$$P_{leak} = V_{DD} I_0 \exp\left(\frac{V_{GS} - V_T}{n V_{th}}\right)\left(1 - \frac{-V_{DS}}{V_{th}}\right) \tag{13.3}$$

where

$I_0$ is the drain current when $V_{GS} = V_T$
$V_T$ is the transistor threshold voltage
$n$ is the subthreshold slope factor and
$V_{th}$ is the thermal voltage

Low-voltage process technologies that rely on reduced threshold voltages to maintain performance are especially susceptible to increased subthreshold leakage. Leakage currents can be especially important in low activity embedded DSP systems that are mostly in standby mode. In such cases, system battery life is mainly dependent on $P_{leak}$. Techniques for reducing $P_{leak}$ include the use of multiple $V_T$ devices (MTCMOS) [10] and substrate bias control variable threshold CMOS [11]. Commercial signal processors use such techniques for leakage reduction [12].

## 13.3 Low-Power Design in Programmable DSPs

Over the last 20 years, integrated circuits have experienced substantial power reductions as technology scaled to deep submicron dimensions. Figure 13.2 plots the operating voltage of microprocessors, DSPs, and other specialized ICs from 1985 to 2005 that were published in the International Solid-State Circuits Conference or the *Journal of Solid-State Circuits* [13]. To prevent large power-density problems as more and more transistors were packed onto one die, the voltage started dropping in the sub-1 μm process technologies. Because DSP applications tend to be highly energy constrained, many innovative power reduction techniques were first introduced in DSPs. In this section, we examine several design trends that have contributed to increased power efficiency.

### 13.3.1 Voltage Scaling

As shown in Figure 13.2, the design supply voltage of programmable DSPs has been reduced substantially during the last 20 years. DSP performance is mostly driven by the sample data rate on which

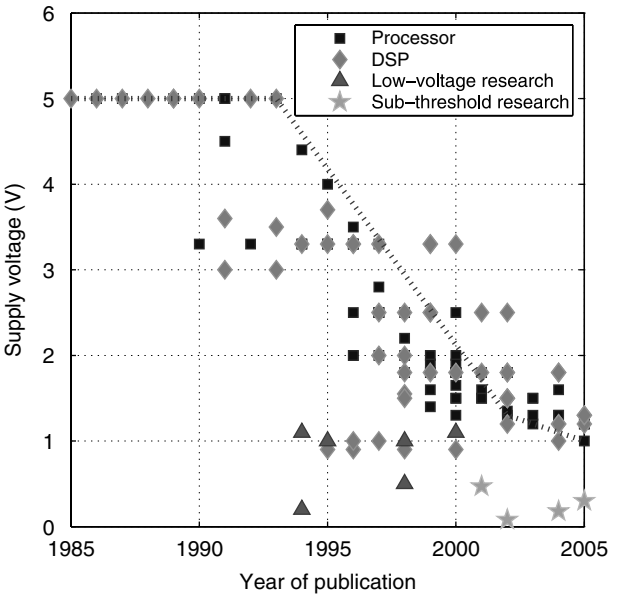



**FIGURE 13.2** Supply voltage trends of microprocessors, DSPs, and research from ISSCC and JSSC publications. (From Wang, A., Calhoun, B., and Chandrakasan, A., *Sub-Threshold Design for Ultra Low-Power Systems*, Springer, Norwell, MA, 2006. With permission.)

the processor operates as opposed to pure clock frequency, as is the case in the general-purpose microprocessor world. Adding multiple execution units in parallel to speedup DSP computational kernels with small code dependencies permits designers to reduce the required clock frequencies and in response reduce the supply voltage with important power benefits [13–15]. Voltage scaling may also be implemented without proportional performance degradation if the threshold voltages are scaled accordingly. Voltage scaling has a dramatic effect on power efficiency due to the square law dependence on $V_{DD}$ in Equation 13.2.

## 13.3.2 Architectural Power Optimizations

Traditionally, programmable DSP chips have included support for frequently used DSP operations and addressing modes. Such operations include parallel multiply-accumulate operations used to implement efficiently computation kernels such as FIR/IIR filters and linear transformation operations [16]. In the last few years, we have seen programmable DSPs that include native support for Viterbi decoding [13]. The Viterbi algorithm [17] is a computationally efficient maximum likelihood estimator for convolutional decoding used in cellular phone and modem applications. Programmable DSPs usually include hardware support for an add compare select (ACS) operation used to eliminate the nonoptimal trellis paths during the decoding process [16]. The TI TMS320C54x architecture includes an explicit compare, select, and store unit (CSSU), which decouples the path metric computation from the path selection process.

Maximum datapath efficiency and minimum control overhead have become of paramount importance in DSPs because of reduced power dissipation in addition to performance benefits [18]. Increased datapath parallelism can allow the DSP designer to reduce the clock frequency and power supply for additional power benefits. Efficient hardwired instructions can reduce control overhead and minimize communication among functional units thus reducing the switched capacitance term in Equation 13.2.

DSP algorithms usually involve the repetitive execution of a small set of instructions (kernel). Most programmable DSPs include hardware support for tight loops. A standard software loop implementation requires the maintenance and update of a loop index, a compare instruction, and a conditional

branch to the beginning of the loop. The loop overhead can easily slow down a DSP kernel by a substantial factor. DSPs include hardware support for both single and multiple instruction loops (i.e., REPEAT instruction) [18]. A single instruction loop repeats a single instruction multiple times without maintaining a loop index and by fetching it only once from memory. A multiple instruction loop on the other hand must repeatedly fetch the instructions from memory each time the processor executes the loop. Addition of a small decoded instruction buffer (DIB) to store decoded instructions during the first iteration into the loop is used in DSPs to lower the power dissipation. Subsequent iterations do not engage the instruction memory and decode unit, but fetch the decoded instructions from the DIB. Case studies have indicated 40% power savings when a DIB is implemented in a DSP for certain multimedia applications [19].

The Berkeley Pleiades project [20] introduced a 1 V heterogeneous reconfigurable DSP targeted for wireless baseband processing. The architecture consists of multiple satellite arithmetic processors, on-chip FPGA sections, on-chip memory banks, address generators, and an embedded ARM core. All these heterogeneous units are interconnected with a hierarchical reconfigurable network. The ARM core is responsible for the online reconfiguration through a dedicated bus. According to the Pleiades computation model, the embedded microprocessor core executes the high-level control and spawns arithmetic-intensive DSP kernels to the satellites. The flow of control is returned to the ARM core when all the satellite operations have completed. Run-time reconfiguration makes such an architecture very power-efficient compared to conventional programmable DSPs. A Pleiades silicon implementation is reported to implement baseband wireless functions at 10–100 MOPS/mW.

### 13.3.3 Low-Power Standby Modes

General-purpose DSPs typically include instructions that place them in multiple levels of standby modes [21]. As an example, the OMAP2 application processor, used to run cell phone applications is designed to have various low-power modes such as idle (clock stopped), retention for low leakage, and fast restart and power off mode for ultra low leakage. In its ultra low leakage state, leakage was reduced by up to 40×.

Fine-grain standby is achieved by partitioning the SoC into multiple power domains that can individually be powered down to reduce leakage power dissipation [21]. Examples of power domains in an application processor include the processor core, graphics core, always-on core, and peripherals. To quickly power down a domain, a grid of power switches is inserted between a global and local power mesh. The global mesh is always on power, whereas the local power mesh is switched on and off quickly. Other power management cells are also needed such as isolation cells between domains and always-on buffers to route signals through a domain that is powered off. Retention flip-flops and RAM is also used to save state during low-power modes.

### 13.3.4 Circuit Power Optimizations

Most of the DSPs available in the market today include some form of fine-grain, clock-gating mechanism for power reduction. The regular datapath structure and the small control structures of DSPs make them well suited for clock-gating. A typical datapath pipeline stage employing clock-gating is shown in Figure 13.3.

Signals EN0 and EN1 are the stage clock enables that are latched 180° ahead of time and computed by the control section. A master clock is distributed to the gating clock drivers, which are typically amortized across the entire datapath width of the pipeline. Clock-gating not only saves clock and flip-flop power but also prevents the combinational logic between pipeline stages from switching. Clock-gating reduces the switching activity factor $a$ in Equation 13.2.

On-chip memory blocks (SRAMs and ROMs) are typically optimized for low power. Memory blocks are partitioned in multiple banks so that a small fraction of the total memory array is activated during a memory access [12,15]. Moreover, address bits are typically allocated in such a fashion among row decoders and column decoders such that sequential memory accesses do not activate the row decoders during each cycle [15].

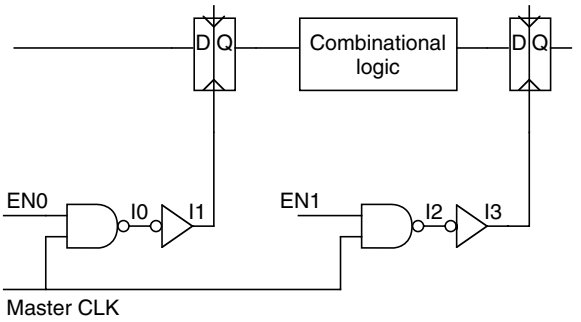



**FIGURE 13.3** Clock-gating.

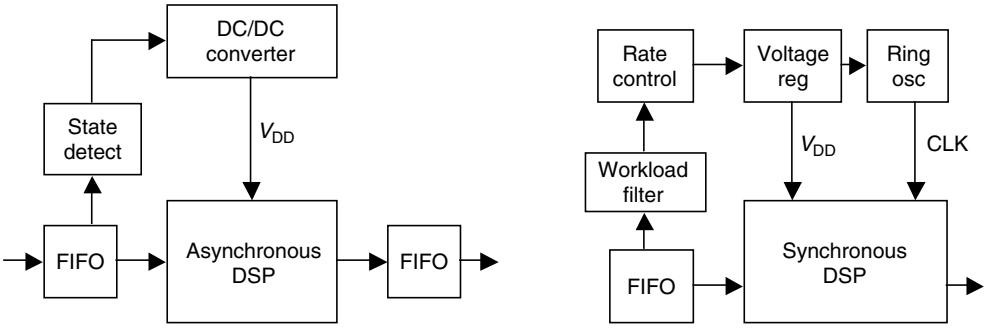
## 13.4 Low-Power Design in Application-Specific DSPs

Low-power design approaches in application-specific DSPs exhibit more breadth and innovation owing to the fact that such designs target a well-defined problem as opposed to a wide range of possible applications. Classification of such design techniques can by no means be complete due to continuous novelties in circuit and system designs improving DSP system power performance. In this section, technology and low-level circuit issues are not addressed as they were briefly addressed earlier and are also relevant to general-purpose DSP systems. Instead, the focus is on unique application-specific power reduction techniques that have been reported in the literature during the last few years.

### 13.4.1 Variable Supply Voltage Schemes

Embedded adaptive supply scaling has been the focus of multiple investigators due to the potential for substantial power savings in both fixed and variable throughput systems.

Nielsen et al. [22] have demonstrated a self-timed adaptive supply voltage system that takes advantage of variable computational loads (Figure 13.4a). The self-timed system operates in a synchronous environment and is enclosed between rate-matching FIFO buffers. The state detecting circuit monitors the state of the input FIFO, which is an indicator of remaining workload. If the buffer is relatively full, the supply voltage is increased and the circuit operates faster to keep up with the load. If the FIFO is relatively empty, the supply voltage is reduced because the circuit is operating too quickly. In this way, the supply voltage is optimally adjusted to the actual workloads maintaining the throughput requirements at all times. Wei and Horowitz [23] have investigated techniques for low-power switching supplies for similar applications.




(a) Asynchronous adaptive supply scheme

(b) Synchronous adaptive supply scheme

**FIGURE 13.4** Adaptive supply voltage schemes. (From Gutnik, V. and Chandrakasan, A., *IEEE Transactions on VLSI Systems*, pp. 425–435, December 1997. With permission.)

Gutnik [24] has demonstrated a synchronous implementation of a variable supply voltage scheme that uses FIFO state to generate both a supply voltage and a corresponding variable clock using a closed-loop ring oscillator (Figure 13.4b). As the FIFO fills up, the clock speed increases to sustain the higher workload and as the FIFO empties, the clock slows and the supply voltage decreases for quadratic power reduction. Power savings are higher than simple clock-gating mechanisms due to the square law dependence of power dissipation on supply voltage. Buffering and workload averaging makes this scheme applicable to fixed throughput but variable algorithmic load applications (i.e., video compression/decompression and digital communication applications).

Goodman and Dancy [25] have demonstrated a low-power encryption processor with an embedded high-efficiency DC–DC converter that takes advantage of the time-varying data rates found in wireless encryption applications. Power reduction varying from 1× up to 5.33× has been reported depending on data throughput variations.

### 13.4.2 Optimum Energy and Subthreshold Circuits

In highly energy-constrained DSP applications, minimizing energy consumption is the principal metric, and often in these applications performance is sacrificed to optimize for power. Subthreshold circuits that operate at supply voltages below the device threshold voltage are ideal for extremely energy-starved applications [13]. Examples of areas where subthreshold circuits are targeted are RFID, microsensor networked nodes, and systems that operate off of the energy-harvested sources.

Optimizing for minimum energy consumption often means that slower is better. As we reduce the supply voltage, switching power is decreased quadratically as seen in Equation 13.2. At the same time, lower supply voltage means reduced transistor drive current leading to slower circuit speeds. Therefore, if an application can run at a slower speed, it trades-off speed for switching power. There is a limit to how slow the circuit can run before it either fails functionally or power dissipation increases due to leakage current. Below the threshold voltage, both delay increases and the amount of leakage power begins to surpass the switching power dissipation. There is an optimum supply voltage level where energy is minimized [13,26]. Figure 13.5 shows a simulation of switching, leakage, and total power of a

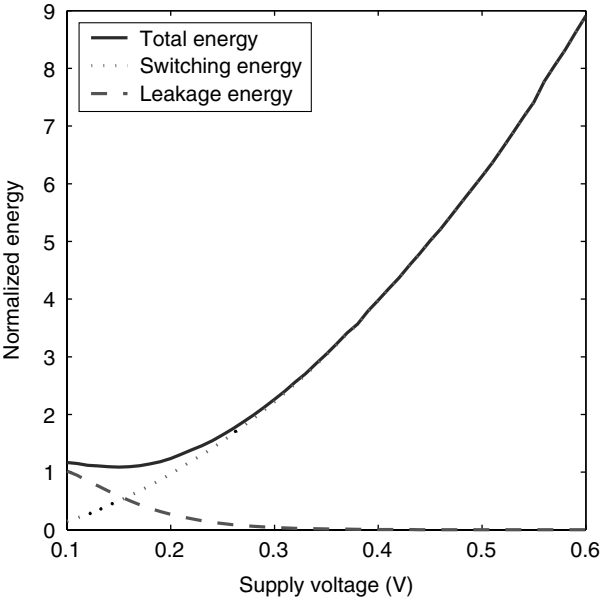



**FIGURE 13.5** Minimum energy point of a 2-input NAND circuit. (From Wang, A., Calhoun, B., and Chandrakasan, A., *Sub-Threshold Design for Ultra Low-Power Systems*, Springer, Norwell, MA, 2006. With permission.)

nand circuit. The minimum operating point occurs at ∼150 mV, which is much lower than the threshold voltages of a typical device.

The minimum energy point occurrence depends highly on the ratio of switching energy to leakage energy. The amount of switching depends highly on the activity factor, which is either related to the workload or the duty cycle. The leakage energy depends on the process conditions such as the threshold voltage of the devices or the operating conditions such as the temperature. The optimal supply voltage to minimize energy has been analytically derived from switching and subthreshold current models [13,27] and confirmed using curve-fitting [28].

Subthreshold digital circuits are designed to operate below the threshold voltage. A study of the minimum voltage operation of an inverter shows that operation is possible at as low as 55 mV, if sized properly [13]. However, across process variations, the minimum operation increases to 200 mV. Figure 13.6 shows the minimum voltage for which the inverter maintains 10%–90% output voltage swing. The upper bound on size occurs because the subthreshold leakage through a large pMOS device limits the extent to which the smaller nMOS can pull down the voltage at the output. The curve denoted by diamonds ($W_p$[max]) shows the maximum pMOS width for which the output-low voltage of the inverter achieves 10% or less of $V_{DD}$. Similarly, the lower bound on size occurs because the subthreshold leakage through a large nMOS device limits the extent to which the smaller pMOS can pull up the voltage at the output. The curve marked with circles ($W_p$[min]) shows the minimum pMOS width for which the output-high voltage achieves 90% of $V_{DD}$. $W_p$(max) in Figure 13.6 is defined at the weak nMOS, strong pMOS (WS) corner, where the nMOS is much weaker than the pMOS devices to show the worst-case sizing for this condition. Likewise, $W_p$(min) is defined at the strong nMOS, weak pMOS (SW) corner to provide the worst-case for pull-up.

This pair of curves essentially gives the worst-case bounds for the process. On the basis of this analysis, the worst-case minimum supply voltage is $V_{DD} = 200$ mV. The pMOS/nMOS sizing ratio ($W_p = W_n$) to achieve this minimum voltage is 12. Since minimum voltage operation occurs for symmetrical pMOS and nMOS currents, this optimum ratio tells us that the pMOS transistor in the inverter needs 12 times the width of the nMOS to equalize the subthreshold currents in this technology.

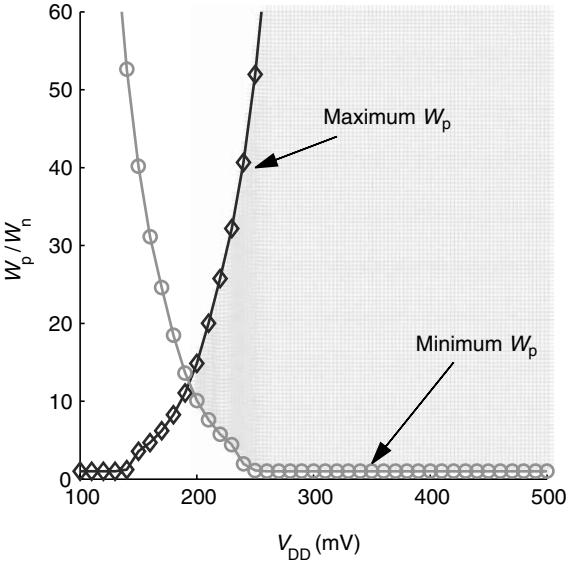



**FIGURE 13.6** Minimum achievable voltage retaining 10%–90% output swing for inverter across worst-case process corners (simulation). (From Wang, A., Calhoun, B., and Chandrakasan, A., *Sub-Threshold Design for Ultra Low-Power Systems*, Springer, Norwell, MA, 2006. With permission.)

Studies on CMOS logic families have shown certain logic families are more suitable for subthreshold operation. For example, ratioed logic flip-flops have shown to fail at 450 mV due to process variations and the inability of the devices to overcome the feedback associated with ratioed logic [13]. Upsizing the transistors may help the circuit to function at lower voltages, but a better approach is to use CMOS flip-flops, which eliminate the feedback path when writing to the cell. Logic family analysis in light of process variations shows that transmission gate logic and static CMOS logic are robust to variation while in subthreshold, whereas dynamic logic and psuedo-NMOS logic are not energy efficient and fail at low voltage [13].

Subthreshold SRAMs are also necessary to optimize energy dissipation. With technology scaling to smaller dimensions, the amount of RAM integration is also increasing. 6T SRAM bit-cell dimensions are often smaller than the standard cell dimensions leading to worsening effect of process variations on the SRAM performance and functionality. Also, the 6T SRAM bit-cell suffers from the same problems as ratioed logic at low voltage. It is difficult to write the cell due to the feedback of the cross-coupled inverters. Read stability is compromised at low voltage due to large bit-line leakage. The bit-line leakage is acerbated in deep submicron technologies.

One way to further scale the voltage to subthreshold levels is to replace SRAMs with register files [30]. Register files use static CMOS registers that are able to operate at as low as 200 mV. Analysis of bit-lines showed that increasing the number of bits on a bit-line increases the minimum voltage operation of the SRAM. Partitioning the bit-line hierarchically using 2-to-1 muxes extends the bit-line operation down to 100 mV in simulation.

However, this RAM design is equivalent to an 18T design and not suitable for large designs where large areas are not acceptable. A subthreshold 10T SRAM bit-cell design is shown in Figure 13.7 [29,13]. The 10T bit-cell consists of a 6T traditional bit-cell with a 4T read buffer. This scheme separates the read port from the write port, and the read buffer is both low-leakage and does not compromise read static noise margin (SNM). Results from a 65 nm testchip of the 10T SRAM showed functionality down to 300 mV making it suitable for subthreshold designs. At 300 mV, the 10T cell saves 2.25× leakage power relative to the 6T at 0.6 V [13].

Many of these subthreshold techniques were used to implement a subthreshold fast Fourier transform (FFT) processor [30]. The variable bit-precision FFT processor was designed to operate as low as 100 mV. The FFT has two operating modes, 8-bit and 16-bit precision. For 8-bit precision, the minimum energy operating point occurred at 350 mV; and for 16-bit precision at 400 mV [13]. The FFT energy versus voltage is shown in Figure 13.8.

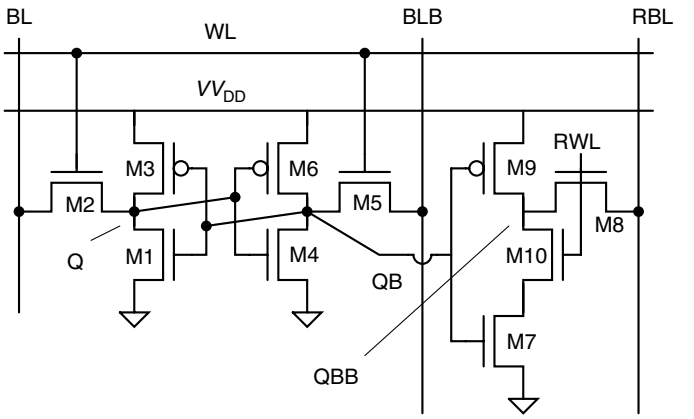



**FIGURE 13.7** 10T SRAM bit-cell schematic. (From Calhoun, B.H. and Chandrakasan, A., *IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, 49, 628, 2006. With permission.)

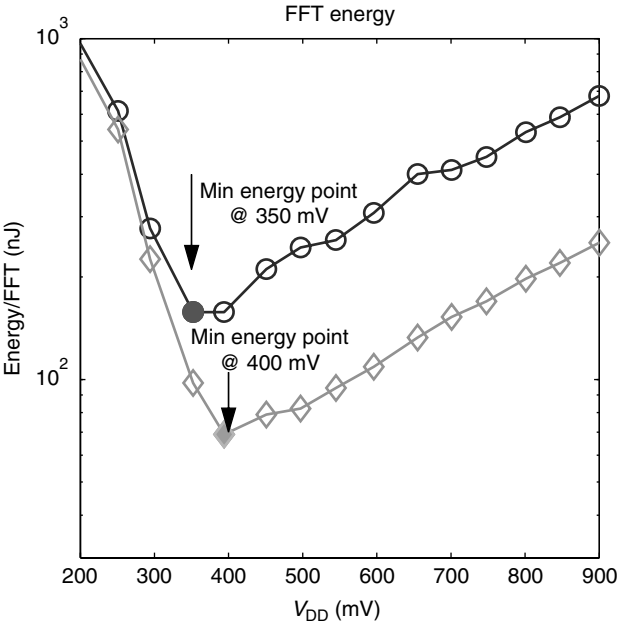



**FIGURE 13.8** FFT measured minimum energy point for 8-bit and 16-bit processing. (From Wang, A., Calhoun, B., and Chandrakasan, A., *Sub-Threshold Design for Ultra Low-Power Systems*, Springer, Norwell, MA, 2006. With permission.)

### 13.4.3 Nonstandard Arithmetic Structures

Fixed-function DSP VLSI implementations (i.e., digital filters with constant coefficients, frequency domain data transformations such as FFT and DCT) can reduce power significantly when designed with hardwired arithmetic structures, which are different from standard multipliers and adders. One such important structure is the distributed arithmetic (DA) implementation [31,32].

DA [33,34] is a bit-serial operation that computes the inner product of two vectors (one of which is a constant) in parallel. In the DSP domain, this operation finds applications in FIR computations, linear transform computation, and any other DSP kernel, which involves dot products. Its main advantage is the efficiency of storing precomputed values in a ROM and the fact that no power-hungry multiplications are necessary. DA has an inherent bit serial nature but this additional latency can be hidden if the number of bits in each variable vector element is equal or similar to the number of elements in each vector. In other words, DA is very efficient in computing long dot products of relatively low precision numbers.

Figure 13.9 shows a detailed example of a DA computation. The structure shown computes the dot product of a 4-element vector **X** and a constant vector **A**. All 16 possible linear combinations of the constant elements (**Ai**) are stored in a ROM. The variable vector **X** is forming the ROM address, MSB-first. The figure assumes that the **Xi** elements are 4-bit 2's complement integers (bit 3 is the sign bit). Every clock cycle, the RESULT register adds 2× its previous value to the currently addressed ROM contents. In addition, the 4-shift registers that hold the variable vector **X** are shifted to the right. The sign timing pulse $\mathbf{T_s}$ is activated when the ROM is addressed by the sign bit (bit 3) of the vector elements. In this case, the accumulator subtracts the addressed ROM contents to implement the first negative term of the dot product. After four cycles, the dot product has been produced within the RESULT register.

The power advantages of DA versus multiply-accumulate can be summarized as follows:

1. ROM accesses can be more energy efficient than multiplications.
2. A ROM and accumulator (RAC) structure can be much more area efficient than a multiplier and accumulator (MAC) structure. In such case, wires tend to be shorter and less capacitive.

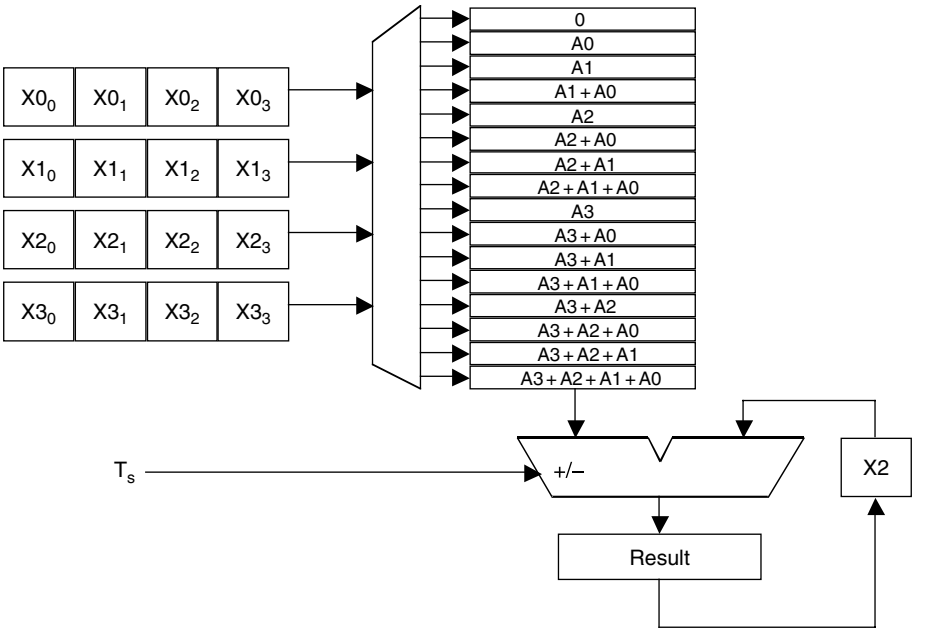



**FIGURE 13.9** Distributed arithmetic ROM and accumulator (RAC) structure. (From Xanthopoulos, T. and Chandrakasan, A., *IEEE Journal of Solid-State Circuits*, 35(5), 740, May 2000. With permission.)

3. If the number of elements in the vectors forming the dot product is greater than the bit precision of the variable vector, then DA structure can be clocked slower than the sample rate and advantage of voltage scaling techniques can be taken. Essentially, such a configuration is an interesting form of parallelism.
4. A DA RAC structure is an ideal arithmetic unit for approximate processing (trading-off power dissipation vs. output quality).

### 13.4.4 Algorithmic/Architectural Exploitation of Data Distribution Properties

Fixed-function DSP systems typically operate on data streams that exhibit common distribution properties. The following are some examples:

1. Data streams related to human aural and visual perception (uncompressed audio and video samples): Such streams typically exhibit large spatial and temporal correlation and reduced dynamic range.
2. Data streams of compressed video or image data in the frequency domain: Typically, such streams contain large numbers of zero-valued coefficients indicating the lack of high spatial frequencies in natural images.

A priori knowledge of data stream distribution can be exploited at the algorithmic and architectural level for computation minimization and power reduction.

It has been observed that 16-bit sampled speech data samples exhibit significant correlation, in addition to a predominance of small signal values [35]. As a result, a sliced datapath for a digital hearing aid filter bank is used to exploit the small magnitude of the input samples. The arithmetic datapath is partitioned into an MSB and an LSB slice. The MSB slice is only engaged when the input bit-width requires it. The slices are activiated by using special data tags that indicate the presence of sign extension bits in the MSB input slice. Additional circuit overhead is required for the computation and update of the tags. Dynamic bit-width adaptation is coarse and can only be performed on a per-slice basis. This scheme results in data-dependent power reduction and processing time.

A DA-based discrete cosine transform (DCT) architecture that exploits correlation in the incoming image or video samples for computation minimization and power reduction has been demonstrated [36]. DCT [37] is a frequency domain data transform widely used in video and still image compression standards such as MPEG [38] and JPEG [39]. The 8-point one-dimensional DCT transform is defined as follows:

$$X[u] = \frac{c[u]}{2} \sum_{i=0}^{7} X[i] \cos\left(\frac{(2i+1)u\pi}{16}\right) \tag{13.4}$$

where $c[u] = 1/\sqrt{2}$ if $u = 0$ and 1 otherwise.

Image pixels are locally well correlated and exhibit a certain number of common most significant bits. These bits constitute a common-mode DC offset that only affects the computation of the DC DCT coefficient ($X[0]$ in Equation 13.4) and is irrelevant for the computation of the higher spectral (AC) coefficients ($X[1] \ldots X[7]$ in Equation 13.4). The DCT chip in Ref. [36] includes adaptive bit-width DA computation units that reject common most significant bits for all AC coefficient computations resulting in arithmetic operations with reduced bit-width operands, thus reducing switching activity. The bit-serial nature of the DA operation allows very fine-grain (1-bit) adaptation to the input dynamic range as opposed to the coarse slice-level adaptation [35].

An interesting algorithmic adaptation to data distribution properties has been demonstrated in Ref. [40]. The chip computes the inverse discrete cosine transform (IDCT) and is targeted to MPEG-compressed video data. The 8-point, one-dimensional IDCT is defined as follows:

$$X[i] = \sum_{u=0}^{7} \frac{c[u]}{2} X[u] \cos\left[\frac{(2i+1)u\pi}{16}\right] \tag{13.5}$$

where $c[u] = 1/\sqrt{2}$ if $u = 0$ and 1 otherwise.

Numerous fast IDCT algorithms can minimize the number of multiplications and additions implied by Equation 13.5 [41,42]. Yet, the statistical distribution of the input DCT coefficients possesses unique properties that can affect IDCT algorithmic design. Typically, 64-coefficient DCT blocks of MPEG-compressed video sequences have only 5–6 nonzero coefficients, mainly located in the low spatial frequency positions due to the low pass characteristics of frame sequences [40]. The histogram of Figure 13.10 shows the frequency of 64-coefficient block occurrence plotted versus the number of nonzero coefficient content for a typical MPEG sequence. The mode of such distributions is invariably blocks with a single nonzero spectral coefficient (typically the DC).

Given such input data statistics, we observe that direct application of Equation 13.5 will result in a small average number of operations since multiplication and accumulation with a zero-valued

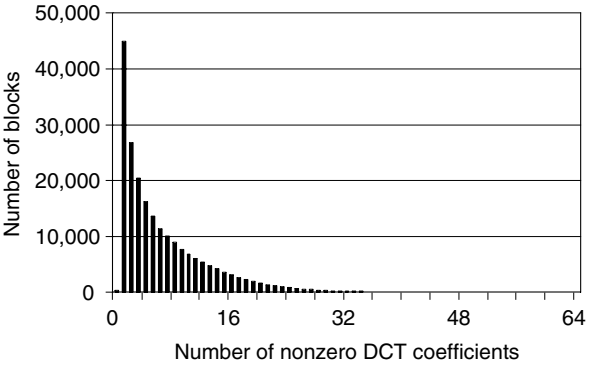



**FIGURE 13.10** Histogram of Nonzero DCT coefficients in sample MPEG stream. (From Xanthopoulos, T. and Chandrakasan, A., *IEEE Journal of Solid-State Circuits*, 34, 693, 1999. With permission.)

coefficient $X[k]$ constitutes a NOP [43]. The chip in Ref. [40] uses such a direct coefficient-by-coefficient algorithm coupled with extensive clock-gating techniques to implement the implied NOPs. IDCT computation power of 4.5 mW for MPEG-2 sample rates has been reported.

### 13.4.5 Approximate Processing

In many DSP applications, lower quality in visual or audio output can be tolerated for reduced power dissipation. Recently, a number of researchers have resorted to approximate processing as a method for reducing average system power. An approximate filtering technique has been demonstrated to reduce the filter order dynamically based on the input data characteristics [44]. More specifically, the number of taps of a frequency-selective FIR filter is dynamically varied based on the estimated stopband energy of the input signal. The resulting stopband energy of the output signal is always kept under a predefined threshold. This technique results in power savings of a factor of six for speech inputs. An adaptive scheme dynamically reduces the input amplitude of a booth-encoded multiplier to the lowest acceptable precision level in an adaptive digital equalizer [45]. The scheme simply involves an arithmetic shift (multiplication/division by a power of two) of the multiplier input depending on the value of the error at the equalizer output. They report power savings of 20%.

When the DA operation is performed MSB-first, it exhibits stochastically monotonic successive approximation properties. In other words, each successive intermediate value is closer to the final value in a stochastic sense. An analytical derivation is presented in Ref. [46]. As an example, let us assume that we have a DA structure computing the dot product of two vectors. Each vector element is 8-bits 2's complement integer. If we clock the DA structure of Figure 13.9 for eight full cycles, the full precision value of the dot product will form into the RESULT register. If instead we clock the DA structure for four cycles and perform a 4-bit arithmetic left shift of the output in the RESULT register (multiplication by $2^4$), we obtain an approximation of the actual dot product. If we clock the structure once more (total of five cycles) and then perform a 3-bit arithmetic left shift of the output (multiplication by $2^3$), we obtain a better approximation. In this way, a DA structure can implement a fine-grain trade-off between power and precision.

A DCT application can use this property extensively for power reduction [36]. In image and video compression applications not all spectral coefficients have the same visual significance. Typically, a large number of high spatial frequencies are quantized to zero in a lossy image/video compression environment (i.e., JPEG and MPEG) with no significant change in visual quality. The DCT processor in Ref. [36] exploits such different precision requirements on a coefficient basis by reducing the number of iterations of the DA units that compute the visually insignificant spectral coefficients in a user-programmable fashion. Figure 13.11 plots average power chip dissipation versus compressed image quality in terms of the image peak SNR (PSNR), a widely used quality measure in image processing. The data points in the graph have been obtained by chip power measurements at different RAC maximum iteration settings.

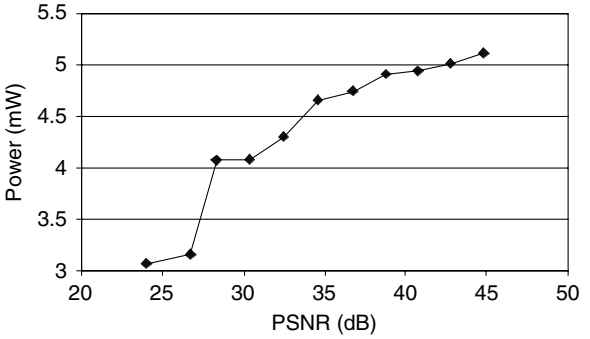



**FIGURE 13.11** DCT chip average power versus compressed image quality. (From Xanthopoulos, T. and Chandrakasan, A., *IEEE Journal of Solid-State Circuits*, 35(5), 740, May 2000. With permission.)


**FIGURE 13.12** Compressed image quality and power. (From Xanthopoulos, T. and Chandrakasan, A., *IEEE Journal of Solid-State Circuits*, 35(5), 740, May 2000. With permission.)

The data implies that the chip can produce on average 10 additional decibels of image quality per milliwatt of power dissipation. Figure 13.12 displays the actual compressed images for three (power, PSNR) data points of Figure 13.11 for visual appreciation.

A similar DA-based approximate processing technique is used in a programmable ultra-low-power DSP targeted to physiological monitoring [47,4]. Reduced RAC iterations reduce the signal-to-noise ratio of the input signal (effectively increasing the quantization noise) and result in less reliable heartbeat detection. Yet, the reduced performance results in linear power savings, which may be desirable in certain situations.

Variable bit-precision was explored in conjunction with a real-valued FFT design for sensor networks [13]. Energy-aware computing meant that when energy resources were plentiful, the sensor could compute a highly accurate FFT. However, when energy resources were low and being depleted, the sensor can compute a lower bit-precision FFT, sacrificing accuracy for less power dissipated. Variable bit-precision allowed for graceful degradation of sensor signal processor over time.

Figure 13.13 shows a block diagram of a variable bit-precision Baugh–Wooley (BW) multiplier [30,13]. The multiplier design partitions the 16-bit BW multiplier into quadrants. The MSB quadrant is an 8-bit BW multiplier, thus during 8-bit processing, inputs are directly fed into the MSB quadrant.

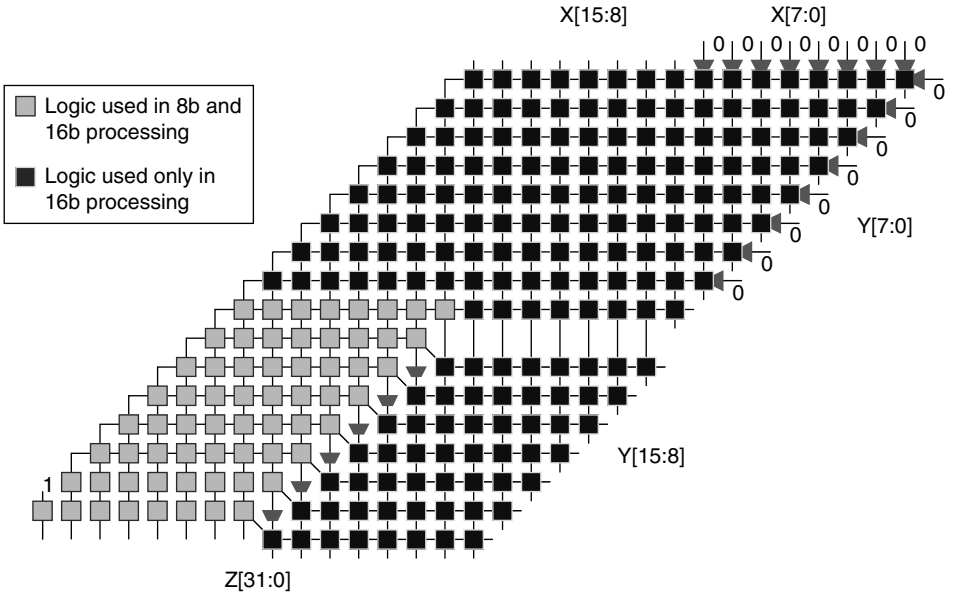



**FIGURE 13.13** Variable bit-precision Baugh–Wooley multiplier allows for energy-scalable 8-bit and 16-bit processing. (From Wang, A. and Chandrakasan, A., *IEEE Journal of Solid-State Circuits*, 40(1), 310, January 2005. With permission.)

The inputs to remaining quadrants and outputs from the 8-bit BW are gated, to prevent additional switching power overhead. When 16-bit processing is required, the entire BW multiplier is used. In traditional nonscalable techniques, one multiplier is created that operates over the largest required bit-precision and an 8-bit multiplication would have switching power overhead.

## References


1. A. Gatherer, T. Stetzler, M. McMahan, and E. Auslander. DSP-based architectures for mobile communications: Past, present and future, *IEEE Communications Magazine*, Vol. 38, No. 1, Jan. 2000, pp. 84–90.
2. B. Calhoun et al., Design considerations for ultra-low energy wireless microsensor nodes, *IEEE Transactions on Computers*, 54(6), Jun 2005, pp. 727–740.
3. A. Chandrakasan, A. Burstein, and R. Brodersen. A low-power chipset for a portable multimedia I/O terminal, *IEEE Journal of Solid-State Circuits*, 29(12), Dec. 1994, 1415–1428.
4. R. Amirtharajah. Design of low power VLSI systems powered by ambient mechanical vibration. PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1999.
5. T. Simon and A. Chandrakasan, An Ultra low power adaptive wavelet video encoder with integrated memory, *IEEE Journal of Solid-State Circuits*, 35(4), April 2000, 572–582.
6. J. Goodman and A. Chandrakasan, An energy-efficient IEEE 1363-based reconfigurable public-key cryptography processor, In *2001 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb. 2001, pp. 330–331.
7. P. Mosch et al., A 660-$\mu$ W 50-Mops 1-V DSP for a hearing aid chip set, *IEEE Journal of Solid-State Circuits*, 35(11), Nov. 2000, 1705–1712.
8. H. Veendrick, Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits, *IEEE Journal of Solid-State Circuits*, SC-19(4), Aug. 1984, 468–473.
9. K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits, *Proceedings of the IEEE*, 91(2), Feb. 2003, 305–327.
10. S. Mutoh et al., 1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS, *IEEE Journal of Solid-State Circuits*, 30(8), Aug. 1995, 847–854.
11. T. Kuroda et al., A 0.9-V, 150-MHz, 10-mW, 4 mm$^2$, 2-D discrete cosine transform core processor with variable threshold-voltage (VT) scheme, *IEEE Journal of Solid-State Circuits*, 31(11), Nov. 1996, 1770–1779.
12. W. Lee et al., A 1-V programmable DSP for wireless communications. *IEEE Journal of Solid-State Circuits*, 32(11) Nov. 1997, 1766–1776.
13. A. Wang, B. Calhoun, and A. Chandrakasan, *Sub-Threshold Design for Ultra Low-Power Systems*, Springer, Norwell, MA, 2006.
14. K. Ueda et al., A 16 b low-power-consumption digital signal processor, In *1993 IEEE International Solid State Circuits Conference Digest of Technical Papers*, Feb. 1993, pp. 28–29.
15. T. Shiraishi et al., A 1.8 V 36 mW DSP for the half-rate speech CODEC, In *Proceedings of the 1996 IEEE Custom Integrated Circuits Conference*, May 1996, pp. 371–374.
16. I. Verbauwhede and C. Nikol, Low power DSPs for wireless communications, In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, July 2000, pp. 303–310.
17. B. Sklar. *Digital Communications*. Prentice-Hall, Englewood Cliffs, NJ, 1988.
18. P. Lapsley, J. Bier, A. Shoham, and E. Lee. *DSP Processor Fundamentals*. IEEE Press, Piscataway, NJ, 1997.
19. M. Hiraki et al., Stage-skip pipeline: A low power processor architecture using a decoded instruction buffer, In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, Aug. 1996, pp. 353–358.
20. H. Zhang et al., A 1-V Heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing, *IEEE Journal of Solid-State Circuits*, 35(11), Nov. 2000, 1697–1704.

21. P. Royannez et al., 90 nm low leakage SoC design techniques for wireless applications, Technical. Digest ISSCC, Feb. 2005, pp. 138–139.
22. L. Nielsen et al., Low-power operation using self-timed circuits and adaptive scaling of the supply voltage, *IEEE Transactions on VLSI Systems*, Dec. 1994, pp. 391–397.
23. G. Wei and M. Horowitz, A low power switching power supply for self-clocked systems, In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, Aug. 1996, pp. 313–318.
24. V. Gutnik and A. Chandrakasan, Embedded power supply for low-power DSP, *IEEE Transactions on VLSI Systems*, Dec. 1997, pp. 425–435.
25. J. Goodman, A. Dancy, and A. Chandrakasan, An energy/security scalable encryption processor using an embedded variable voltage DC/DC converter, *IEEE Journal of Solid-State Circuits*, 33(11), Nov. 1998, 1799–1809.
26. A. Wang, A. Chandrakasan, and S. Kosonocky, Optimal supply and threshold scaling for sub-threshold CMOS circuits, In *IEEE Computer Society Annual Symposium on VLSI*, Apr. 2002, pp. 7–11.
27. B. Calhoun, A. Wang, and A. Chandrakasan, Modeling and sizing for minimum energy operation in subthreshold circuits, *IEEE Journal of Solid-State Circuits*, 40(9), Sept. 2005, 1778–1786.
28. B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner, Theoretical and practical limits of dynamic voltage scaling, In *ACM/IEEE Design Automation Conference (DAC) Digest of Technical Papers*, 2004, pp. 868–873.
29. B.H. Calhoun and A. Chandrakasan, A 256 kb Sub-threshold SRAM in 65nm CMOS, In *IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, Vol. 49, 2006, pp. 628–629.
30. A. Wang and A. Chandrakasan, A 180-mV subthreshold FFT processor using a minimum energy design methodology, *IEEE Journal of Solid-State Circuits*, 40(1), Jan 2005, 310–319.
31. M. Sun, T. Chen, and A. Gottlieb, VLSI Implementation of a $16 \times 16$ Discrete Cosine Transform, *IEEE Transactions on Circuits and Systems*, 36(4), April 1989, 610–617.
32. S. Uramoto et al., A 100 MHz 2-D discrete cosine transform core processor, *IEEE Journal of Solid-State Circuits*, 36(4), April 1992, pp. 492–499.
33. A. Peled and B. Liu, A new hardware realization of digital filters, *IEEE Transactions on Acoustics Speech and Signal Processing*, ASSP-22(6), Dec. 1974, 456–462.
34. S. White. Applications of distributed arithmetic to digital signal processing: A tutorial review, *IEEE ASSP Magazine*, July 1989.
35. L. Nielsen and J. Sparso, An 85 μW asynchronous filter bank for a digital hearing aid, In *1998 IEEE International Solid State Circuits Conference Digest of Technical Papers*, Feb. 1998, pp. 108–109.
36. T. Xanthopoulos and A. Chandrakasan, A low-power DCT core using adaptive bitwidth and arithmetic activity exploiting signal correlations and quantization, *IEEE Journal of Solid-State Circuits*, 35(5), May 2000, 740–750.
37. K. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications*. Academic Press, San Diego, 1990.
38. D. LeGall, MPEG: A video compression standard for multimedia applications, *Communications of the ACM*, 34(4), April 1991, 46–58.
39. G. Wallace, The JPEG still picture compression standard, *Communications of the ACM*, 34(4), April 1991, 30–44.
40. T. Xanthopoulos and A. Chandrakasan, A low-power IDCT macrocell for MPEG-2 MP@ML exploiting data distribution properties for minimal activity, *IEEE Journal of Solid-State Circuits*, 34(5), May 1999, 693–703.
41. W. Chen, C. Smith, and S. Fralick, A fast computational algorithm for the discrete cosine transform, *IEEE Transactions on Communications*, 25(9), Sept. 1977, 1004–1009.
42. E. Feig, and S. Winograd, Fast algorithms for the discrete cosine transform, *IEEE Transactions on Signal Processing*, 40(9), Sept. 1992, 2174–2193.

43. L. McMillan and L. Westover, A forward-mapping realization of the inverse discrete cosine transform, In *Proceedings of the 1992 Data Compression Conference*, IEEE Computer Society Press, March 1992, pp. 219–228.
44. J. Ludwig, S. Nawab, and A. Chandrakasan, Low-power digital filtering using approximate processing, *IEEE Journal of Solid-State Circuits*, 31(3), March 1996, 395–400.
45. C. Nikol, P. Larsson, K. Azadet, and N. O'Neill, A low-power 128-tap digital adaptive equalizer for broadband modems, *IEEE Journal of Solid-State Circuits*, 32(11), Nov. 1997, 1777–1789.
46. R. Amirtharajah, T. Xanthopoulos, and A. Chandrakasan, Power scalable processing using distributed Arithmetic, In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, Aug. 1999, pp. 170–175.
47. R. Amirtharajah, S. Meninger, O. Mur-Miranda, A. Chandrakasan, and J. Lang, A micropower programmable DSP powered using a MEMS-based vibration-to-electric energy converter, In *2000 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb. 2000, pp. 362–363.

# IV

# Communications and Networks

# 14

# Communications and Computer Networks




Anna Hać
*University of Hawaii*


## 14.1 Architecture

The set of layers and the corresponding set of protocols are called the architecture of a network. In designing a layered architecture, there are problems that must be solved in several layers. These problems include addressing and connection establishment, connection termination, nature of channel (e.g., full or half-duplex), error control and sequencing, flow control, and multiplexing.

### 14.1.1 OSI Reference Model

Open systems interconnect (OSI) reference model consists of seven layers: physical, data link, network, transport, session, presentation, and application.

The physical layer is responsible for transmitting bits over a communication channel.

The data link layer is responsible for providing an error-free line to the higher layers. It provides error and sequence checking, and implements a system of time-outs and acknowledgments that enables a transmitter to determine the frames that need to be retransmitted due to error or dropout. In addition, flow control is provided in data link layer.

The network layer provides routing and congestion control services to higher layers. The network accounting function is obtained in this layer.

The transport layer accepts data in message form from the session layer above it, breaks it into smaller pieces, usually called packets, and passes the packets to the network layer. It must then ensure that the packets arrive correctly at the destination host. The transport layer is an end-to-end protocol, as opposed to layers below it, which are chained. The transport layer may multiplex several sessions over a single network connection, or it may utilize several connections to provide a high data-rate for a session that requires it. The transport layer may provide either virtual circuit or datagram service to the session layer. In a virtual circuit, messages are delivered in the order in which they were sent, whereas in a datagram service there is no guarantee concerning order of delivery. The transport layer also has the

responsibility for establishing and terminating connections between hosts across the network and for providing host-to-host flow control.

The session layer is responsible for establishing and managing connections between two processes. Session establishment typically requires authentication, billing authorization, and agreement on a set of parameters that will be in effect for the session. The session layer is also responsible for recovery from a transport failure, and for providing virtual circuit service if the transport layer does not do so.

The presentation layer performs services that are commonly requested by users, such as text compression, code conversion, file formatting, and encryption.

The application layer contains routines specific to a particular application.

## 14.1.2 Networks

### 14.1.2.1 LAN

Local area network (LAN) is a privately owned network of up to a few kilometers in size. LANs are used to connect computers, workstations, and file servers, and attach printers and other devices. The restricted size of LANs allows for prediction of transmission time, and simplifies network management. Traditional LANs run at speeds of 10–100 Mbps and have low delay of tens of milliseconds.

Broadcast topologies include bus and ring. In a bus network, at any instant one machine is the master and is allowed to transmit. At the same time, all other machines are required to refrain from sending. IEEE 802.3, which is the Ethernet, is a bus-based broadcast network with decentralized control operating at 10 or 100 Mbps. Computers on an Ethernet can transmit at any time. If two or more packets collide, each computer waits for a random time and tries again later.

In a ring network, each bit propagates around on its own, not waiting for the rest of the packet to which it belongs. IEEE 802.5, which is the IBM token ring, is a ring-based LAN operating at 4 and 16 Mbps.

### 14.1.2.2 WAN

Wide area network (WAN) covers a large area, a country, or a continent. The hosts in WAN are used to run application programs and are connected by a communication subnet, which consists of transmission lines and switching elements. A switching element, also called router, is used to forward packets to their destinations.

### 14.1.2.3 Cellular Network

The most widely employed wireless network topology is the cellular network. This network architecture is used in cellular telephone networks, personal communication networks, mobile data networks, and wireless local area networks (WLAN). In this network configuration, a service area, usually over a wide geographic area, is partitioned into smaller areas called cells. Each cell, in effect, is a centralized network, with a base station (BS) controlling all the communications to and from each mobile user in the cell. Each cell is assigned a group of discrete channels from the available frequency spectrum (usually radio frequency). These channels are in turn assigned to each mobile user, when needed.

Typically, BSs are connected to their switching networks using landlines through switches. The BS is the termination point of the user-to-network interface of a wireless cellular network. In addition, the BS also provides call setups, cell handoffs, and various network management tasks, depending on the type of network.

## 14.1.3 TCP/IP

Transmission control protocol/Internet protocol (TCP/IP) suite is used in the network and transport layers. TCP/IP is a set of protocols allowing computers to share resources across the network. Although the protocol family is referred to as TCP/IP, user datagram protocol (UDP) is also a member of this protocol suite.

TCP/IP protocol suite used as network and transport layers has the following advantages:

1. It is not vendor-specific.
2. It has been implemented on most systems from personal computers to the largest supercomputers.
3. It is used for both LANs and WANs.

Using TCP/IP also makes the network system portable, and program portability is one of the system design goals.

The network layer uses IP. IP is responsible for routing individual datagrams and getting datagrams to their destination. The IP layer provides a connectionless and unreliable delivery system. It is connectionless because it considers each IP datagram independent of all others and any association between datagrams must be provided by the upper layers. Every IP datagram contains the source address and the destination address so that each datagram can be delivered and routed independently. The IP layer is unreliable because it does not guarantee that IP datagrams ever get delivered or that they are delivered correctly. Reliability must be provided by the upper layers.

Transport layer uses UDP and TCP. TCP is a connection-oriented protocol that provides a reliable, full-duplex, byte stream for the multimedia communication process. TCP is responsible for breaking up the message into datagrams, reassembling them at the other end, resending anything that got lost, and putting everything back in the right order. TCP handles the establishment and termination of connections between processes, the sequencing of data that might be received out of order, the end-to-end reliability (checksums, positive acknowledgments, timeouts), and the end-to-end flow control.

UDP is a connectionless protocol for user processes. Unlike TCP, which is a reliable protocol, there is no guarantee that UDP datagrams ever reach their intended destination. UDP is less reliable than TCP but transfers data faster as they are not held up by earlier messages awaiting retransmission. TCP protocol is used for file transfer that requires reliable, sequenced delivery, where real-time delivery may not be of utmost importance.

### 14.1.4 Mobile IPs

In TCP/IP an application is connected with another application through a router. Each host in Internet has a unique address. An IP address is a 32-bit binary number that can also be used in a dotted notation. IP addresses contain two parts: network address, which identifies the network to which the host is attached and local address, which identifies the host. Local address can be separated into two parts, subnet address and local address.

The hierarchical address makes routing simple. A host that wants to send packets to another host only needs to send packets to the network to which the target host is attached. The host does not need to know the inside of the network; however, a computer's IP address cannot be changed during connection and communication. If the user wants to move the computer to the other area while using it, this will be difficult because the physical IP address of the computer must be changed in a different subnet. To solve this problem, a number of protocols have been proposed: virtual IP (VIP), loose source routing IP (LSRIP), and Internet engineering task force mobile IP (IETF-MIP).

#### 14.1.4.1 VIP

VIP uses two 32-bit IP-style addresses to identify mobile hosts: one is named virtual IP (VIP) address and the other temporary IP (TIP) address. VIP address is the IP address that mobile hosts get from their home network. Mobile hosts always use VIP as their source address inside IP packet. When mobile hosts move to another network, they get another IP address from the foreign network, it is a TIP address. Each VIP packet contains information to combine VIP and TIP, so the packet target to VIP can be routed through general Internet to its temporary network by reading its TIP. VIP uses additional space inside packet to carry this information: a new IP option to identify VIP while original address fields carry TIP.

When a mobile host moves to a foreign network, the information about mobile host's current location is sent to the mobile host's home network. During the transmission, each intermediate router that supports VIP protocol can receive this information and update this router's cache. This cache is a database that stores information about mobile hosts' current location.

If a host wants to send a packet to a mobile host, it only knows the VIP address of the mobile host and does not know its TIP, which is its current location. The packet will be sent to the mobile host's home area. If any intermediate router that supports VIP receives this packet, it will modify this packet according to its cache so that the new packet will include information of TIP and can be routed to the mobile host's current location. If the packet does not reach any intermediate router that supports VIP and has cache information about the mobile host, the packet will be routed to mobile host's home network. The gateway in mobile host's home network can modify this packet according to this gateway's cache and route the packet to the mobile host's current network. This gateway always has the mobile host's current location, because each time when the mobile host moves to a new network, the mobile host notifies its home network about the mobile host's current location. The optimized path in VIP is mainly based on the number of intermediate VIP routers. If many intermediate routers support VIP, the optimized routing path should be obtained. The option VIP uses to carry information of VIP is an option of IP, and not all of the routers will support this option. Some computers even discard all of the options IP packets carry. VIP also needs many extra IP addresses for foreign network to assign to the mobile hosts.

#### 14.1.4.2 LSRIP

LSRIP uses one of the IP options to cause the IP packets to be routed through a series of intermediate routers to the destination. This IP option is loose source and record route (LSRR). For example, if host A wants to send a packet to host C with LSRR option, the packet can reach host B through general IP routing. Then host B replaces the destination IP address in IP header with the first IP address C in LSRR option, and routes the packet to a new destination C. Also, host B puts the pointer to the second IP address in LSRR option. Host C can perform the same procedure: replace destination IP address with next IP address in LSRR option, increase pointer to next IP address in LSRR option, and reroute the packet to a new destination. Until the pointer points to the last IP address in LSRR option, the packet is sent to its destination address. LSRIP uses LSRR option to carry the information of a mobile host.

When a mobile host moves to a foreign network, the information about the mobile host's current location is sent to the mobile host's home network. During the transmission, each intermediate router that supports LSRIP protocol can receive this information and update this router's cache.

If a host wants to send a packet to a mobile host, which is not at its home area, the packet will be first sent to the mobile host's home network. During the path to the home network, if an intermediate router has the cache of the mobile host, this router can put the LSRIP option into the packet and cause it to be routed to the current network of the mobile host. The gateway of current network reads information from LSRIP option and can determine that the mobile host is the destination of this packet's destination. Thus, the gateway can route the packet to mobile host that is connected to current network.

If the packet does not reach any intermediate router that supports LSRIP and has cache information about the mobile host, the packet will be routed to mobile host's home network. The gateway in mobile host's home network can add the LSRIP option to this packet according to the gateway's cache and route the packet to the mobile host's current network. This gateway always has the mobile host's current location because each time when the mobile host moves to a new network, the mobile host notifies its home network about the mobile host's current location.

LSRIP needs more intermediate routers to achieve optimized routing path. Also, the option used by LSRIP is not compatible with current routers. This can be tested by sending a packet using traceroute, a tool to check the path of a passed packet. Traceroute uses LSRR option to record the path of the packet and the transmission time to each intermediate host. If any intermediate host does not support LSRR option, traceroute bypasses this host. After sending a packet, it can be found from the messages sent back that some sites are not displayed correctly, which means that these sites do not support LSRR option.

#### 14.1.4.3 IETF-MIP

IETF-MIP is the most usable protocol for mobile IP in the Internet. The basic idea is to use two agents to handle the job related to the mobile host. When the mobile host moves to the other networks, it will notify foreign network's agent, foreign agent, and its home agent about its current location. Then, when the packet to mobile host is sent to the home agent using general IP, the home agent will modify the header of IP packet: change the destination address to foreign agent's address and add some fields to the packet including the mobile host's permanent address. When the foreign agent receives this packet, it will know this is for one mobile host that is now at its location, the foreign agent will modify this packet again and send it directly to the mobile host through the local network. In IETF-MIP, there are cache agents to optimize the performance. A cache agent is a host that can maintain a database that stores the mobile hosts' current location. This database can be changed according to the location change of the mobile host.

In this protocol, it is difficult to achieve the optimized routing path, especially in a WAN. For example, a user in London wants to send a packet to a mobile host in London, whose home network is in New York. This packet will be first sent to New York, then modified by the home agent, and then sent back. This takes about one half of the circle of the whole Earth. The optimized path can only be about 100 ft. Thus, the only solution for IETF-MIP is to set up as many cache agents as possible in the entire Internet. When the cache agents receive a packet, they can modify the packet instead of sending it to the home agent of the mobile host.

Mobile IP should be compatible with current IPs; that means the current protocols and applications do not need to be changed. Mobile IP also needs to have optimized routing path that means the protocol should be efficient in routing packets. VIP and LSRIP use the option of IP to carry the information of mobile hosts. But some of the current hosts do not support an IP option. When these hosts receive a packet that includes options, they discard the options of the IP packet. Meanwhile, IETF-MIP only uses basic IP header and packet, and does not use any IP option. From the compatibility point, IETF-MIP is the best out of these three protocols. From the point of optimized routing path, all three protocols depend on intermediate cache hosts. If enough intermediate cache hosts are inside the Internet, the three protocols can find optimized routing path. The IETF-MIP is the best out of the three mobile IPs considering both the compatibility and optimized routing path. IETF-MIP is also the mobile IP protocol that is used in the Internet.

## 14.2 Technology

### 14.2.1 Broadband Networks

Broadband integrated services digital network (B-ISDN) based on asynchronous transfer mode (ATM) is used for transport of information from multimedia services and applications.

#### 14.2.1.1 ATM

ATM is a cell-based, high-bandwidth, low-delay switching and multiplexing technology that is designed to deliver a variety of high-speed digital communication services. These services include LAN interconnection, imaging, and multimedia applications as well as video distribution, video telephony, and other video applications.

ATM standards define a fixed-size cell with a length of 53 bytes comprised of a 5-byte header and a 48-byte payload.

The virtual path identifiers (VPIs) and virtual channel identifiers (VCIs) are the labels to identify a particular virtual path (VP) and virtual channel (VC) on the link. The switching node uses these values to identify a particular connection and then uses the routing table established at connection setup to route the cells to the appropriate output port. The switch changes the value of the VPI and VCI fields to the new values that are used on the output link.

#### 14.2.1.2 SONET

Synchronous optical network (SONET) is used for framing and synchronization at the physical layer. The basic time unit of a SONET frame is 125 μs. The basic building block of SONET is synchronous transport signal level 1 (STS-1) with a bit rate of 51.84 Mbps. Higher-rate SONET signals are obtained by byte-interleaving *n* frame-aligned STS-1s to form an STS-*n* (e.g., STS-3 has a bit rate of 155.52 Mbps).

Owing to physical layer framing overhead, the transfer capacity at the user-network interface (UNI) is 155.52 Mbps with a cell-fill capacity of 149.76 Mbps. Because the ATM cell has 5 bytes of overhead, the 48 bytes information field allows for a maximum of 135.631 Mbps of actual user information. A second UNI interface is defined at 622.08 Mbps with the service bit rate of approximately 600 Mbps. Access at these rates requires a fiber-based loop.

#### 14.2.1.3 ATM Services

Users request services from the ATM switch in terms of destinations, traffic types, bit rates, and quality of service (QoS). These requirements are usually grouped together and categorized in different ATM traffic classifications. The ATM services are categorized as follows:

- Constant bit rate (CBR): Connection-oriented constant bit rate service such as digital voice and video traffic.
- Real-time variable bit rate (rt-VBR): Intended for real-time traffic from bursty sources such as compressed voice or video transmission.
- Non-real-time variable bit rate (nrt-VBR): Intended for applications that have bursty traffic but do not require tight delay guarantee. This type of service is appropriate for connectionless data traffic.
- Available bit rate (ABR): Intended for sources that accept time-varying available bandwidth. Users are only guaranteed a minimum cell rate (MCR). An example of such traffic is LAN emulation traffic.
- Unspecified bit rate (UBR): Best effort service that is intended for noncritical applications. It does not provide traffic-related service guarantees.

ATM networks are fixed (optical) point-to-point networks with high bandwidth and low error rates. These attributes are not associated with the limited bandwidth and error prone radio medium. Although increasing the number of cables (copper or fiber optics) can increase the bandwidth of wired networks, wireless telecommunications networks experience a more difficult task. Owing to limited usable radio frequency, a wireless channel is an expensive resource in terms of bandwidth. For wireless networks to support high-speed networks such as ATM, a multiple access approach is needed for sharing this limited medium in a manner different from the narrowband, along with the means of supporting mobility and maintaining QoS guarantees.

### 14.2.2 Wireless Networks

Media access control (MAC) is a set of rules that attempt to efficiently share a communication channel among independent competing users. Each MAC uses a different media (or multiple) access scheme to allocate the limited bandwidth among multiple users. Many multiple access protocols have been designed and analyzed both for wired and wireless networks. Each has its advantages and limitations based on the network environment and traffic. These schemes can be classified into three categories: fixed assignments, random access, and demand assignment. The demand assignment scheme is the most efficient access protocol for traffic of varying bit rate in the wireless environment.

Because of the limited radio frequencies available for wireless communication, wireless networks have to maximize the overall capacity attainable within a given set of frequency channel. Spectral efficiency describes the maximum number of calls that can be served in a given service area. To achieve high spectral efficiency, cellular networks are designed with frequency reuse. If a channel with a specific frequency covers an area of a radius *R*, the same frequency can be reused to cover another area. A service

area is divided into seven cell clusters. Each cell in the cluster, designated one through seven, uses a different set of frequencies. The same set of frequencies in each cell can be reused in the same service area if it is sufficiently apart from the current cell. Cells using the same frequency channels are called co-cells. In principle, by using this layout scheme, the overall system capacity can be increased as large as desired by reducing the cell size, while controlling power levels to avoid co-channel interference. Co-channel interference is defined as the interference experienced by users operating in different cells using the same frequency channel. Smaller size cells called microcells are implemented to cover areas about the size of a city block. Research has been done on even smaller cells called picocells.

#### 14.2.2.1 TDMA

Time-division multiple access (TDMA) and frequency-division multiple access (FDMA) are fixed assignment techniques that incorporate permanent subchannel assignments to each user. These traditional schemes perform well with stream-type traffic, such as voice, but are inappropriate for integrated multimedia traffic because of the radio channel spectrum utilization. In a fixed assignment environment, a subchannel is wasted whenever the user has nothing to transmit. It is widely accepted that most services in the broadband environment are VBR service (bursty traffic). Such traffic wastes a lot of bandwidth in a fixed assignment scheme.

#### 14.2.2.2 ALOHA

Typical random assignment protocols like ALOHA and carrier sense multiple access with collision detection (CSMA/CD) schemes are more efficient in servicing bursty traffic. These techniques allocate the full channel capacity to a user for short periods, on a random basis. These packet-oriented techniques dynamically allocate the channel to a user on a per-packet basis.

Although a few versions of the ALOHA protocol are used, in its simplest form it allows the users to transmit at will. Whenever two or more user transmissions overlap, a collision occurs and users have to retransmit after a random delay. The ALOHA protocol is inherently unstable due to the random delay, i.e., it is possible that a transmission may be delayed for an infinite time. Various collision resolution algorithms were designed to stabilize and reduce contention in this scheme.

Slotted ALOHA is a simple modification of the ALOHA protocol. After a collision, instead of retransmitting at a random time, slotted ALOHA retransmits at a random time slot. Transmission can only be made at the beginning of a time slot. Obviously, this protocol is implemented in time-slotted systems. Slotted ALOHA is proven to be twice as efficient as a regular or pure ALOHA protocol.

#### 14.2.2.3 CSMA/CD

CSMA/CD, taking advantage of the short propagation delays between users in a typical LAN, provides a very high throughput protocol. In a plain CSMA protocol, users will not transmit unless it senses that the transmission channel is idle. In CSMA/CD, the user also detects any collision that happens during a transmission. The combination provides a protocol that has high throughput and low delay; however, carrier sensing is a major problem for radio networks. The signal from the local transmitter will overload the receiver, disabling any attempts to sense remote transmission efficiently. Despite some advances in this area, sensing still poses a problem due to severe channel fading in indoor environments. Similarly, collision detection proves to be a difficult task in wireless networks. Although it can be easily done on a wired network by measuring the voltage level on a cable, sophisticated devices are required in wireless networks. Radio signals are dominated by the terminal's own signal over all other signals in the vicinity preventing any efficient collision detection. To avoid this situation, a terminal transmitting antenna pattern has to be different from its receiving pattern. This requires sophisticated directional antennas and expensive amplifiers for both the BS and the mobile station (MS). Such requirements are not feasible for the low-powered mobile terminal end.

#### 14.2.2.4 CDMA

Code-division multiple access (CDMA) is a combination of both fixed and random assignment. CDMA has many advantages such as near zero channel access delay, bandwidth efficiency, and excellent

statistical multiplexing, but it suffers from significant limitations such as limited transmission rate, complex BS, and problems related to the power of its transmission signal. The limitation in transmission rate is a significant drawback touse CDMA for integrated wireless networks.

#### 14.2.2.5 Demand Assignment

In demand assignment protocol, channel capacity is assigned to users on demand basis, as needed. Demand assignment protocols typically involve two stages: a reservation stage where the user requests access and a transmission stage where the actual data is transmitted. A small portion of the transmission channel, called the reservation subchannel, is used solely for users requesting permission to transmit data. Short reservation packets are sent to request channel time by using some simple multiple access schemes, typically, TDMA or slotted ALOHA. Once channel time is reserved, data can be transmitted through the second subchannel contention-free. Unlike a random access protocol where collisions occur in the data transmission channel, in demand assignment protocols, collisions occur only in the small-capacity reservation subchannel.

This reservation technique allows demand assignment protocols to avoid bandwidth waste due to collisions. In addition, unlike fixed assignment schemes no channels are wasted whenever a VBR user enters an idle period. The assigned bandwidth will simply be allocated to another user requesting access. Owing to these features, protocols based on demand assignment techniques are most suitable for integrated wireless networks.

Demand assignment protocols can be classified into two categories based on the control scheme of the reservation and transmission stages. They can be either centralized or distributed. An example of a centralized controlled technique in demand assignment is polling. Each user is sequentially queried by the BS for transmission privileges. This scheme, however, relies heavily on the reliability of the centralized controller.

An alternative approach is to use distributed control, where MSs transmit based on information received from all the other MSs. Network information is transmitted through broadcast channels. Every user listens for reservation packets and performs the same distributed scheduling algorithm based on the information provided by the MS in the network. Requests for reservation are typically made using contention or fixed assignment schemes.

#### 14.2.2.6 Network Support for Wireless Sensors

The essence of ubiquitous computing is the creation of environments saturated with computing and communication in an unobtrusive way. WWRF (Wireless world research forum) and ISTAG (Information society technologies advisory group) envision a vast number of various intelligent devices, embedded in the environment, sensing, monitoring, and actuating the physical world, communicating with each other and with the humans.

The main features of the IEEE 802.15.4 standard are network flexibility, low cost, and low power consumption. This standard is suitable for many applications in the home requiring low-data-rate communications in an ad hoc self-organizing network.

Wireless sensor networks are used in a wide range of different applications where numerous sensor nodes are linked to monitor and report distributed event occurrences. In contrast to traditional communication networks, the single major resource constraint in sensor networks is power, due to the limited battery life of sensor devices. Data-centric methodologies can be used to solve this problem efficiently. In data-centric storage (DCS), data dissemination framework, all event data is stored by type at designated nodes in the network and can later be retrieved by distributed mobile access points in the network. Resilient data-centric storage (R-DCS) is a method to achieve scalability and resilience by replicating data at strategic locations in the sensor network.

This scheme leads to significant energy savings in reasonably large-sized networks and scales well with increasing node-density and query rate. R-DCS realizes graceful performance degradation in the presence of clustered as well as isolated node failures, hence making the sensornet data robust.

Wireless sensor networks require low-power, low-cost devices that accommodate powerful processor, a sensing unit, wireless communication interface, and power source in a robust and tiny package. These

devices have to work autonomously, to require no maintenance and to adapt to environment. For example, the MEMS (micro electro mechanical systems) technology enables production of very small sensing units with low power consumption.

Sensor network management protocol has to support control of individual nodes, network configuration updates, location information data exchange, network clustering, and data aggregation rules.

Sensor network gateway has to provide tools and functions for presentation of network topology, services, and characteristics to the users and to connect the network to other networks and users.

The IEEE 802.15.4 standard defines a low-rate wireless personal area network (LR-WPAN), which has ultra-low complexity, cost, and power for low-data-rate wireless connectivity among inexpensive fixed, portable, and moving devices. The IEEE 802.15.4 standard defines the physical (PHY) layer and media access control (MAC) layer specifications.

The IEEE 802.15.4 standard targets the residential and industrial market. LR-WPAN is designed as an enabler technology. The IEEE 802.15.4 is complementary to the other wireless networking technologies by occupying the lower end of the power consumption and data throughput space.

Data-centric storage (DCS) is a data-dissemination paradigm for sensor networks. In DCS, data is stored, according to event type, at corresponding sensornet nodes. All data of a certain event type (e.g., humidity measurements) is stored at the same node. A significant benefit of DCS is that the queries for data of a certain type can be sent directly to the node storing data of that type, rather than flooding the queries throughout the network (unlike data-centric routing proposals). DCS is based on the low-level routing functionality provided by the GPSR (greedy perimeter stateless routing) geographic routing algorithm, and on distributed hash-table functionality provided by peer-to-peer lookup algorithms. DCS offers reduced total network load and very good network usage.

## 14.3 Routing

### 14.3.1 Routing in Terrestrial Networks

Routing refers to the determination of a set of paths to be used for carrying messages from a source node to all destination nodes. It is important that the routes used for such communications consume a minimal amount of resources. In order to use network resources as little as possible while meeting the network service requirements, the most popular solution involves the generation of a tree spanning the source and destination nodes.

Routing algorithms for constructing trees have been developed with two optimization goals in mind. Two measures of the tree quality are in terms of the tree delay and tree cost, and are defined as follows:

1. The first measure of efficiency is in terms of the cost of the tree, which is the sum of the costs on the edges in the tree.
2. The second measure is the minimum average path delay, which is the average of minimum path delays from the source to each of the destinations in the group.

Optimization objectives are to minimize the cost and delay; however, the two measures are individually insufficient to characterize a good routing tree. For example, when the optimization objective is only to minimize the total cost of the tree, a minimum cost tree is built. Although total cost as a measure of bandwidth efficiency is certainly an important parameter, it is not sufficient to characterize the quality of the tree, because networks, especially those supporting real-time traffic, need to provide certain QoS guarantees in terms of the end-to-end delay along the individual paths from source to destination node. Therefore, both cost and delay optimization goals are important for the routing tree construction. The performance of such a route is determined by two factors:

1. Bounded delay along the path from source to destination
2. Minimum cost of the tree, for example, in terms of network bandwidth utilization

The goal of the routing algorithm is to construct a delay constrained minimum cost tree. In order to provide a certain quality of service to guarantee end-to-end delay along the path from source to destination node, the algorithm sets the delay constraint on the path, instead of trying to minimize the average path delay. The two measures of the tree quality, the tree edge delay and tree edge cost, can be described by different functions. For example, edge cost can be a measure of the amount of buffer space or channel bandwidth, and edge delay can be a combination of propagation, transmission, and queuing delay.

The shortest path algorithm can be used to generate the shortest paths from the source to destination nodes; this provides the optimal solution for delay optimization. Routing algorithms that perform cost optimization have been based on computing the minimum Steiner tree, which is known to be an NP-complete problem.

#### 14.3.1.1 DDBMA

A heuristic algorithm called DDBMA (dynamic delay bounded multicasting algorithm) is used for constructing minimum-cost multicast trees with delay constraints. The algorithm sets variable delay bounds on destinations and can be used to handle the network cost optimization goal: minimizing the total cost (total bandwidth utilization) of the tree. The algorithm can also be used to handle a dynamic delay-bounded minimum Steiner tree, which is accomplished by updating the existing multicast tree when destinations need to be added or deleted.

During the network connection establishment, DDBMA can be used to construct a feasible tree for a given destination set. For certain applications, however, nodes in the network may join or leave the initial multicast group during the lifetime of the multicast connection. Examples of these applications such as teleconferencing, mobile communication, etc., allow each user in the network to join or leave the connection at any time without disrupting network services to other users.

The DDBMA is based on a feasible search optimization method, which starts with the minimum delay tree and monotonically decreases the cost by iterative improvement of the delay-bounded tree. Then the algorithm starts to update the existing tree when nodes in the network request to join or leave. The algorithm will stay steady when there is no leaving or joining requests from nodes in the network.

Multimedia, multiparty communication services are supported by networks having the capability to setup/modify the following five basic types of connections: point-to-point, point-to-multipoint (also called multicast), multipoint-to-point (also called concast), multipoint-to-multipoint, and point-to-allpoint (also called broadcast).

Many types of communication require transmission of certain information from the source to a selected set of destinations. This could be the cast of multipoint video conference, the distribution of a document to a selected number of persons via a computer network or the request for certain information from a distributed database.

### 14.3.2 Routing in Wireless Networks

Wireless personal communication networks use a general routing procedure, a rerouting procedure, and a handoff. Along with the features of wireless communication, the user mobility control function tracking locations of networks subscribers should be associated with routing schemes during communication connection. In wireless communications networks, the network topology is established by virtual paths. Virtual paths are logical direct radio links between all switch nodes. The bandwidth of virtual path can consist of a number of virtual channels. Because of the features of wireless communications networks, the network topology is highly dynamic. The bandwidth of wireless communications networks is limited, the traffic increases quickly, and it is hard to schedule incoming traffic on time in the centralized approaches, which are not efficient when network size increases and the network services are enhanced.

The subscribers in wireless communications networks roam. To create connections between all communication parties to deliver incoming and outgoing calls, the first consideration is the current

location of mobile users and hosts. In wireless communications networks, the key service for providing seamless connectivity to mobile hosts is creation and maintenance of a message forwarding the path between two known locations of calling and called mobile hosts. A routing decision in wireless communications networks is made using not only the states of paths and internal switching nodes but also the location of available information.

Geographical area covered by wireless communications networks is partitioned into a set of cells. A routing path may be inefficient whereas a mobile host hands off to another cell coverage area. The connection paths need to be reestablished each time to continue communication. As a result, the network call processor can become involved many times during the lifetime of mobile connection. When wireless communications networks move toward smaller size cells to accommodate more mobile hosts or to provide higher capacity, the handoff becomes a more frequent part of communications. Conventional routing procedures for connecting mobile hosts fail due to frequent handoff when the network call processor becomes a bottleneck. Hence, routing efficiency in wireless communications networks depends critically on the propagation of location information into the network; however, excessive information propagation can waste network resources, while insufficient location information leads to inefficient routing.

Wireless communications networks can provide different personal communications services, which have different transmission time delay requirements. For cellular telephone communication, the shortest time delay or strict time delay to transmit voice message is required. In portable computer communications or other data communications, the requirements of transmitted time delay are not very strict. Transmitted data can be stored in buffers and be transmitted later when channels are available; however, to provide a high QoS, transmission time delay is an important factor in wireless communications networks. Routing procedure in wireless communications networks should depend on different requirements of transmission time delay, and on how to balance transmission load and find minimum cost transmission paths.

### 14.3.3 Routing in Wireless Sensor Networks

Networking of a large number of low power mobile nodes involves routing, addressing, and support for different classes of service at the network layer. Energy aware routing (EAR) protocol is built on the principle of attribute-based addressing. EAR and directed diffusion belong to the class of reactive routing protocols (RRP). In RRP, the routing information between nodes is set up only on demand and maintained as long as it is needed. This eliminates the need to maintain permanent routing tables. This way, before any communication can take place, a route discovery has to be performed.

In RRP, the consumers of data (called sinks) initiate the route discovery.

The database generic query interface for data aggregation can be applied to ad hoc networks of sensor devices. Aggregation is used as a data reduction tool. Networking approaches have focused on application-specific solutions. The network aggregation approach is driven by a general-purpose, SQL (structured query language) style interface that can execute queries over any type of sensor data while providing opportunities for significant optimization.

The topology discovery algorithm for wireless sensor networks selects a set of designated nodes, called cluster heads. The algorithm then constructs a reachability map based on the cluster heads information. The cluster heads reply back to the topology discovery probes, thereby minimizing the communication overhead. The topology discovery algorithm logically organizes the network in the form of clusters. A tree of clusters rooted at the monitoring node is built. This organization is used for efficient data dissemination and aggregation, duty cycle assignments, and network state retrieval. The topology discovery algorithm is distributed, uses only local information, and is highly scalable.

The vision of ubiquitous computing is based on the idea that future computers merge with their environment until they become completely invisible to the user. Distributed wireless microsensor networks are an important component of the ubiquitous computing. Small dimensions are a design goal for microsensors. The energy supply of the sensors is a main constraint of the intended

miniaturization process. It can be reduced only to a specific degree since energy density of conventional energy sources increases slowly. In addition to improvements in energy density, energy consumption can be reduced. This approach includes the use of energy-conserving hardware. Moreover, a higher lifetime of sensor networks can be accomplished through optimized applications, operating systems, and communication protocols. Particular modules of the sensor hardware are turned off when they are not needed. Routing and data dissemination in sensor networks requires a simple and scalable solution.

## 14.4 Applications Support

### 14.4.1 Multimedia

Multimedia communications is the field referring to the representation, storage, retrieval, and dissemination of machine-processable information expressed in multimedia, such as voice, image, text, graphics, and video. With high-capacity storage devices, powerful, and yet economical, computer workstations and high-speed integrated services digital networks, providing a variety of multimedia communication services, is becoming not only technically but also economically feasible. Multimedia conference systems can help people to interact with each other from their homes or offices while they work as teams by exchanging information in several media, such as voice, text, graphics, and video. Multimedia conference system allows a group of users to conduct a meeting in real time. The participants can jointly view and edit relevant multimedia information, including text, graphics, and still images distributed throughout the LAN. Participants can also communicate simultaneously by voice to discuss the information they are sharing. This multimedia conference system can be used in a wide variety of cooperative work environment, such as distributed software development, joint authoring, and group decision support.

Multimedia is the integration of information that may be represented by several media types, such as audio, video, text, and still images. The diversity of media involved in a multimedia communication system imposes strong requirements on the communication system. The media used in the multimedia communications can be classified into two categories: discrete media and continuous media.

Discrete media are those media that have time-independent values, such as text, graphics, or numerical data, bit mapped images, geometric drawings, or any other non-time-dependent data format. Capture, storage, transmission, and display of non-real-time media data does not require that it happen at some predictable and fixed time or within some fixed period.

Continuous media data may include sound clips, video segments, animation, or timed events. Real-time data requires that any system that is recording or displaying be able to process the appropriate data within a predictable and specified time. In addition, the display of real-time data may need to be synchronized with other data or some external (real-world) event.

Multimedia data can be accessed by the user either locally or remotely during multimedia communication. Locally stored data typically resides in conventional mass storage systems such as hard disk, CD-ROMs, optical disk, or high-density magnetic tape. It can also be stored to and recalled from analog devices that are under the control of the system, video tape disks, videodisk players, CD-audio disks, image scanners, and printers. In addition, media data can be synthesized locally by the systems or its peripherals. Multimedia data is typically recorded and edited on local systems for distribution on some physical media and is later played back using local devices.

Remotely stored multimedia data is accessed via a network connection to a remote system. The data is stored on that remote server and recalled over the network for viewing, editing, or storage on the user's system.

Multimedia communications cover a large set of domains including office, electronic publishing, medicine, and industry. Multimedia communication can be classified into real-time applications and non-real-time applications.

Multimedia conferencing represents a typical real-time multimedia communication. In general, high conductivity is needed for real-time multimedia communications. A guaranteed bandwidth is required

to ensure real-time consistency, and to offer the throughput required by the different media. This bandwidth varies depending on the media involved in the application. There is also a need for synchronization between different users and between different flows of data at a user workstation.

Non-real-time multimedia communications, such as multimedia mail, are less demanding than real-time applications in terms of throughput and delay, but edition tools, exchange formats, and exchange protocols are essential. Multicast service and synchronization at presentation time has to be offered.

Local non-real-time multimedia characterizes most typical personal computer applications, such as word processing, and still image editing. Typical text-based telecommunications can be described as remote non-real-time. Database of text and still image may be interactively viewed and searched, and audio or video data (perhaps included in mail messages) can be downloaded for display locally.

Multimedia workstations are generally characterized by local real-time applications. Data from video and audio editing and annotations, interactive animated presentations, and music recording are stored on local devices and are distributed on physical media for use locally.

Networks that can provide real-time multimedia communication via a high-speed network connection enable the new generation of multimedia applications. Real-time remote workstation-based multimedia conferencing, video and audio remote database browsing, and viewing of movies or other video resources on demand are typical for these systems.

### 14.4.2 Mobile and Wireless

The support for bandwidth intensive (multimedia) services in mobile cellular networks increases the network congestion and requires the use of micro/picocellular architectures in order to provide higher capacity in regard to radio spectrum. Micro/pico cellular architectures introduce the problem of frequent handoffs and make resource allocation difficult. As a result, availability of wireless network resources at the connection setup time does not necessarily guarantee that wireless resources are available throughout the lifetime of a connection. Multimedia traffic imposes the need to guarantee a predefined QoS to all calls serviced by the network.

In microcellular networks supporting multimedia traffic, the resource allocation schemes have to be designed such that a call can be assured a certain QoS once it is accepted into the network. The resource allocation for multimedia traffic becomes quite complex for different classes of traffic comprising multimedia traffic. These classes of traffic have different delay and error rate requirements. Resource allocation schemes must be sensitive to traffic characteristics and adapt to rapidly changing load conditions. From a service point of view, multimedia traffic can be categorized into two main categories: real-time traffic with stringent time delays and relaxed error rates, and non-real-time traffic with relaxed time delays and stringent error rates.

It is important to note that provisioning of QoS to different classes of traffic necessitates a highly reliable radio link between the mobile terminal and its access point. This requires efficient communication techniques to mitigate the problems of delay sensitivity, multipath fading, shadow fading, and co-channel interference. Some methods such as array antennas and optimal combining can be used to combat these problems.

#### 14.4.2.1 CAC

Schemes have been proposed to address the problem of resource allocation for multimedia traffic support in microcellular networks. In these schemes, real-time traffic being more delay sensitive is given priority over non-real-time traffic.

In these schemes, the central approach used is call admission control (CAC). CAC imposes a limit on the number of calls accepted into the network. Each cell site only supports a predetermined number of call connections. This call threshold is periodically calculated depending on the number of existing calls in the cell in which the call arrives and its adjoining cells and the resources utilized by all calls in the cell. Once the threshold is reached, all subsequent requests for new call connections are refused.

#### 14.4.2.2 AT

In an admission threshold (AT)-based scheme, resource management is done by periodically calculating the admission threshold and by blocking all new call connection requests once the threshold is reached. The call admission decision is made in a distributed manner whereby each cell site makes a decision by exchanging state information with adjoining cells periodically. A cell with a base station and a control unit is referred to as cell site.

#### 14.4.2.3 RS

In a resource sharing (RS)-based scheme, to support traffic classes with different delay and error requirements, resource sharing provides a mechanism to ensure a different grade of service to each class of traffic. This scheme employs a resource sharing mechanism that reacts to rapidly changing traffic conditions in a cell. An adaptive call admission control policy that reacts to changing new call arrival rates can be used to keep the handoff dropping rate and forced call termination rate acceptably low.

The call admission control scheme differentiates the new call on the basis of its traffic class and a decision is based on traffic class of the new call connection request and number of call connections of each class already being serviced in the cell cluster.

For real-time call connections, a new call is blocked if no bandwidth is available to service the request. A similar algorithm is used to service a handoff request. The QoS metrics for real-time calls are handoff dropping probability and forced call termination probability. For non-real-time calls, the available bandwidth is shared equally among all non-real-time call connections in the cell. Handoff queuing or delaying is not used in this scheme. CAC keeps the probability of a call being terminated before its lifetime is acceptably low. Resource sharing algorithms provide better performance for a particular class of traffic.

#### 14.4.2.4 RRN

Resource reservation and renegotiation (RRN) scheme provides QoS guarantee to real-time traffic and at the same time guarantees a better performance to non-real-time traffic. The resource allocation scheme uses resource reservation in surrounding cells for real-time calls and renegotiation of bandwidth assigned to non-real-time calls. The resource allocation scheme is simple enough and can be implemented in a distributed manner to ensure fast decision making.

In RRN scheme, for service applications requiring smaller bandwidths, a shared pool of bandwidth is used for reservation. For applications requiring greater bandwidth, the largest of requested bandwidth is reserved. This helps in keeping the call blocking rate low and does not affect the handoff dropping rate.

In microcellular networks, calls require handoffs at much faster rates in comparison to networks with larger cells. On the other hand, microcellular networks provide a higher system capacity. The RRN scheme supports real-time calls and non-real-time calls along with a variety of service type for each class. Real-time calls are delay sensitive and hence cannot be queued or delayed. Resources must be available when a handoff is requested. To guarantee that real-time calls are not forced to terminate at the time of handoff, a resource reservation mechanism is used. Resource reservation guarantees acceptably low handoff dropping rate and forced call termination rate for real-time traffic.

For a real-time call, bandwidth is reserved in all cells adjacent to the cell in which the call arrives. When a call hands off to another cell, if enough bandwidth is not available to service the handoff, it uses the bandwidth reserved in the target cell and thus the likelihood that a call will be dropped is reduced. When a call is successfully handed off to another cell, the bandwidth of old cell is released and reserved in the cell cluster of new cell.

Non-real-time calls are more tolerant to delay as compared to real-time calls. Delay tolerance is equivalent to accepting variable service rate. This property of data traffic makes resource renegotiation possible in microcellular networks. Non-real-time calls receive higher service rates under low traffic conditions whereas, under heavy traffic conditions, the service rate available to them is kept at a minimum. Thus, the resource renegotiation scheme adapts to changing traffic conditions in the network.

### 14.4.3 Sensor Networks

Wireless sensor nodes are deployed in areas and environments where they may be hard to access, yet those nodes need to provide information about measurements of temperature, humidity, biological agents, seismic activity, pictures, and many other activities. Macrosensor nodes usually provide accurate information about measured activity. The accuracy of individual microsensor nodes is lower, yet a network of hundreds or thousands of nodes deployed in an area enables to achieve fault-tolerant, high-quality measurements.

Wireless sensor nodes are designed by using microelectromechanical systems (MEMS) technology and its associated interfaces, signal processing, and radio frequency (RF) circuitry. Communication occurs within a wireless microsensor network, which aggregates the data to provide information about the observed environment.

Low energy dissipation is particularly important for wireless microsensor nodes, which are deployed in hundreds or thousands, and are often hard to read in inhospitable terrain. A power-aware system design employs a system whose energy consumption adapts to constraints and changes in the environment. These power-aware design methods offer scalable energy savings in wireless microsensor environment.

There is a trade-off between battery lifetime and quality-performance of data collection and transmission. A scalable system enables a user to trade-off system performance parameters as opposed to hard wiring them. Scalability allows the end-user to implement operational policy, which may change over the system lifetime. Power-awareness allows a well-designed system to gracefully degrade its quality and performance as energy resources are depleted.

Activity in the observed environment may lead to tremendous measurement diversity in the sensor node microprocessor. Node functionality may also vary, for instance, a sensor networking protocol may request the node to act as a data gatherer, aggregator, relay, or any combination of these. This way, the microprocessor can adjust the energy consumption depending on the activity in the measured environment.

Several devices have been built to perform sensor node functions. A software and hardware framework includes a microprocessor, low power radio, battery, and sensors. Data aggregation and network protocols are processed by using a micro operating system.

Data aggregation is used as a data reduction tool. Aggregates summarize current sensor values in sensor network. Computing aggregates in sensor network preserves network performance and saves energy by reducing the amount of data routed through the network.

The computation of aggregates can be optimized by using SQL (structured query language). The data are extracted from the sensor network by using declarative queries.

Examples of database aggregates (COUNT, MIN, MAX, SUM, and AVERAGE) can be implemented in a sensor network. Aggregation in SQL-based database systems is defined by an aggregate function and a grouping predicate. The aggregate function specifies how to compute an aggregate.

Aggregation can be implemented in a centralized network by using a server-based approach where all sensor readings are sent to the host PC (personal computer), which computes the aggregates. A more efficient approach is distributed, in-network computing of aggregates where the readings are routed through the network to the host PC.

Lifetime of sensor networks is defined by using the following three metrics: first node dies (FND), half of the nodes alive (HNA), and last node dies (LND). FND denotes an estimated lifetime for this event. In this case, adjacent sensors can take over the functions, and the quality of network service may not be diminished. HNA denotes an estimated half-lifetime for the sensor network. LND gives a value of sensor network lifetime.

A topology discovery algorithm can be used to find a set of nodes to construct the network topology. Those nodes reply to the topology discovery probes, thereby minimizing the wireless communication overhead. A tree of clusters rooted at the monitoring node is built.

The small battery-powered sensor devices have limited computational and communication resources. This makes it impractical to use secure algorithms designed for powerful workstations. A sensor node

memory is not capable of holding the variables required in asymmetric cryptographic algorithms, and perform operations by using these variables.

The sensor nodes communicate by using RF (radio frequency), thus trust assumptions and minimal use of energy are important for network security. The sensor network communication patterns include sensor readings, which involve node to base station communication, specific requests from the base station to the node, and routing or queries from the base station to all sensor nodes.

We assume that the sensor nodes are not trusted, but the base stations belong to the trusted computing base. The senor nodes trust the base station and are given a master key that is shared with the base station. The possible threats to network communication security are an insertion of malicious code, an interception of the messages, and injecting false messages.

Wireless technologies enabling communications with sensor nodes include Bluetooth and LR-WPAN (low-rate wireless personal area network).

Bluetooth enables seamless voice and data communication via short-range radio links. Bluetooth provides a nominal data-rate of 1 Mbps for a piconet, which consists of one master and up to seven slaves. The master defines and synchronizes the frequency hop pattern in its piconet. Bluetooth operates in the 2.4 GHz ISM (industrial, scientific, and medical) band.

Low-rate wireless personal area network (LR-WPAN) is defined by the IEEE 802.15.4 standard. This network has ultra-low complexity, cost, and power for low data-rate sensor nodes. The IEEE 802.15.4 offers two physical layer options, the 2.4 GHz physical layer, and the 868/915 MHz physical layer. The 2.4 GHz physical layer specifies operation in the 2.4 GHz ISM band. The 868/915 MHz physical layer specifies operation in the 868 MHz band in Europe, and in 915 MHz band in the United States.

The main features of the IEEE 802.15.4 standard are network flexibility, low cost, and low power consumption. This standard is suitable for many applications in the home requiring low-data-rate communications in an ad hoc self-organizing network.

The major resource constraint in sensor networks is power, due to the limited battery life of sensor devices. Data-centric methodologies can be used to solve this problem efficiently. Data-centric storage (DCS) is used as a data-dissemination paradigm for sensor networks. In DCS, data is stored, according to event type, at corresponding sensornet nodes. All data of a certain event type is stored at the same node. A significant benefit of DCS is that queries for data of a certain type can be sent directly to the node storing data of that type. Resilient data-centric storage (R-DCS) is a method to achieve scalability and resilience by replicating data at strategic locations in the sensor network.

This scheme leads to significant energy savings in networks and scales well with increasing node-density and query rate.

Sensor network management protocol has to support control of individual nodes, network configuration updates, location information data exchange, network clustering, and data aggregation rules.

Sensor network gateway has to provide tools and functions for presentation of network topology, services, and characteristics to the users and to connect the network to other networks and users.

## Further Reading

1. E. Callaway, P. Gorday, L. Hester, J.A. Gutierrez, M. Naeve, B. Heile, and V. Bahl, Home networking with IEEE 802.15.4: A developing standard for low-rate wireless personal area networks, *IEEE Communications Magazine*, Vol. 40, No. 8, August 2002, pp. 70–77.
2. A. Ghose, J. Grossklags, and J. Chuang, Resilient data-centric storage in wireless ad-hoc sensor networks, *Proceedings of the 4th International Conference on Mobile Data Management (MDM 2003)*, Melbourne, Australia, January 21–24, 2003, Lecture Notes in Computer Science 2574 Springer 2003, pp. 45–62.
3. A. Hać, Network centric designs in sensor networks, *Proceedings of the AIAA Infotech@Aerospace Conference*, Arlington, VA, September 26–29, 2005, pp. 1–4.

4. A. Hać, Embedded systems and sensors in wireless networks, *Proceedings of the International IEEE Conference on Wireless Networks, Communications, and Mobile Computing WirelessCom 2005*, Maui, Hawaii, June 13–16, 2005, pp. 330–335.
5. A. Hać, *Multimedia Applications Support for Wireless ATM Networks*, Prentice-Hall, Englewood Cliffs, NJ, 2000.
6. A. Hać, *Wireless Sensor Network Designs*, John Wiley & Sons, New York, 2003.
7. W.B. Heinzelman, A. Chandrakasan, and H. Balakrishnan, Energy-efficient communication protocol for wireless microsensor networks, *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS)*, Maui, Hawaii, January 2000, pp. 3005–3014.
8. W.B. Heinzelman, A.P. Chandrakasan, and H. Balakrishnan, An application-specific protocol architecture for wireless microsensor networks, *IEEE Transactions on Wireless Networking*, 1(4): 660–670, October 2002.
9. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, System architecture directions for networked sensors, *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-IX*, Cambridge, MA, November 2000, pp. 93–104.
10. IEEE 802.15 Working Group for Wireless Personal Area Networks (WPANs). URL: http://www.ieee802.org/15.
11. R. Jain, A. Puri, and R. Sengupta, Geographical routing using partial information for wireless ad hoc networks, *IEEE Personal Communications*, 8(1): 48–57, February 2001.
12. P.-A. Larson, Data reduction by partial preaggregation, *Proceedings of the International Conference on Data Engineering*, San Jose, CA, 2002 pp 706–715.
13. S. Lindsey, C. Raghavendra, and K.M. Sivalingam, Data gathering algorithms in sensor networks using energy metric, *IEEE Transactions on Parallel and Distributed Systems*, 13(9): 924–935, September 2002.
14. V. Raghunathan, C. Schurgers, S. Park, and M.B. Srivastava, Energy-aware wireless microsensor networks, *IEEE Signal Processing Magazine*, Vol. 19, No. 2, March 2002, pp. 40–50.
15. R.C. Shah and J.M. Rabaey, Energy aware routing for low energy ad hoc sensor networks, *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, Vol. 1, Orlando, FL, March 2002, pp. 350–355.
16. S. Singh, M. Woo, and C.S. Raghavendra, Power aware routing in mobile ad hoc networks, *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 1998)*, Dallas, TX, October 1998, pp. 181–190.
17. A.S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

# V

# Input/Output

# 15

# Circuits for High-Performance I/O




Chik-Kong Ken Yang
*University of California*


The speed of off-chip I/O circuits plays a significant role in the overall performance of a computer system. To keep up with the increasing clock rates in processors, designers target I/O data rates that are exceeding gigabits per second per pin for memory busses [26], peripheral connections [29], and multiprocessor interconnection networks [17]. This chapter examines the issues and challenges in the design of these high-performance I/O subsystems.

As illustrated in Fig. 15.1, an I/O subsystem consists of four components: a transmitter, a transmission medium, a receiver, and a timing-recovery circuit. A transmitter converts the binary sequential bit stream into a stream of analog voltages properly sequenced in time. The medium such as a cable or PCB trace delays and filters the voltage waveform. The receiver recovers the binary values from the output of the medium. As part of the receiver, a clock samples the data to recover the bit sequence compensating for the arbitrary delay of the medium.

This chapter focuses on the transmission over an electrical medium* and begins by reviewing the electrical characteristics of transmission lines. The design issues and design techniques for each link component will be described, beginning with the transmitter (Section 15.2) continuing with the receiver (Section 15.3), and ending with the timing-recovery circuits (Section 15.4).

## 15.1 Transmission Lines

A transmission medium confines the energy of a signal and propagates it [33]. The energy is stored as the electric and magnetic field between two conductors, the transmission line. The geometric

*Optical local interconnects are emerging as an alternative for short haul systems.

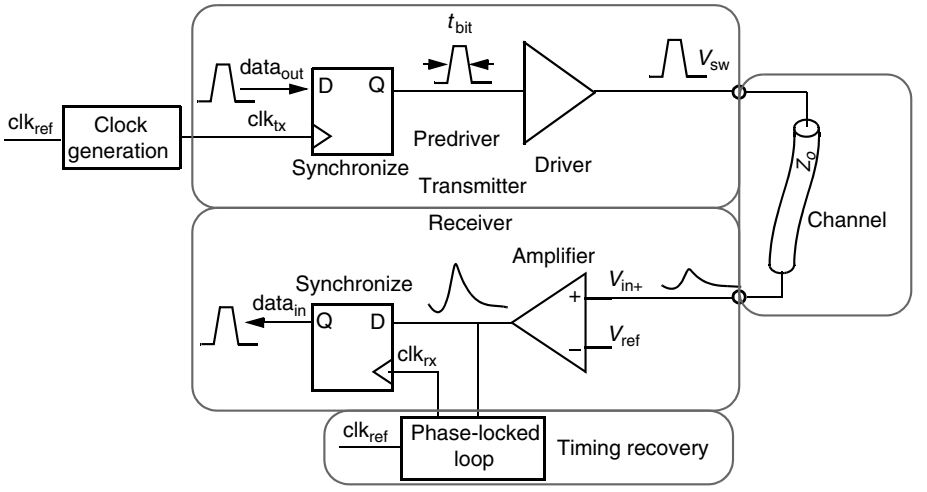



**FIGURE 15.1** Components of an I/O subsystem.

configuration of the conductors for a segment of the transmission line determines the voltage and current relationship in that segment defining an effective impedance ($Z_o = V/I$). A signal source driving the segment only sees this impedance as a sink of the signal energy and has no immediate knowledge of other parts of the line.

The two conductors can be either two signals wires driven differentially or a single signal wire over a reference plane where an image current* flows in the plane coupled to the signal. A coaxial cable has a center conductor for the signal and the outer shield as the reference (Fig. 15.2a). Similarly a PCB trace forms a microstrip line with a ground plane as the reference (Fig. 15.2b).

An effective way to model a transmission line is to use capacitances and inductances to represent the electrical and magnetic energy storage and propagation. The entire line is modeled using multiple LC segments as illustrated in Fig. 15.2c.† The impedance of the line and the propagation velocity can be represented as $Z = \sqrt{L/C}$ and $\nu = 1/\left(\sqrt{LC}\right)$. An ideal transmission line propagates a signal with no added noise or attenuation. Imperfections in the construction of the line such as varying impedance or neglecting the image current path cause noise in the signal transmission.

## 15.1.1 Reflections, Termination, and Crosstalk

When a signal wave encounters a segment with a different impedance, a portion of the signal power reflects back to the transmitter and can interfere with future transmitted signals. The reflection occurs because the boundary condition at a junction of two impedances must be preserved such that (1) the voltage is the same on both sides of the junction and (2) the signal energy into and out of the junction is

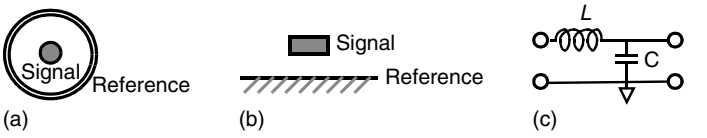



**FIGURE 15.2** Cross-sectional view of transmission line: (a) coaxial, (b) microstrip (PCB), and (c) LC model of a transmission line.

*Image current, also called return current, is equal to the signal current.

†The *L*'s and *C*'s are per unit length.

conserved. For instance, if a lower impedance is seen by a signal, a lower voltage must be propagated along the new segment so that the propagated power is less than the original power. The lower voltage at the junction implies that a negative voltage wave is propagated in the reverse direction. Similarly, at the end of a transmission line, the receiver appears as an open circuit (high impedance) and would cause a positive reflection of the entire signal energy. The equation

$$V_{\text{reflect}} = \Gamma V_{\text{in}} = \frac{Z_{\text{out}} - Z_{\text{in}}}{Z_{\text{out}} + Z_{\text{in}}} V_{\text{in}}$$

represents the reflected wave, where $\Gamma$ is the reflection coefficient.

Using a termination resistance at the end of the line that is equal to that of the transmission line impedance eliminates the reflection by dissipating the signal power. A reflection only poses a problem if the transmitter reflects the wave again causing the old signal energy from a previous bit to add to the signal of a newly transmitted data bit.* Allowing the entire signal to reflect at the receiver is acceptable as long as the line is properly terminated at the transmitter. So proper termination of a transmission line includes matching the resistance either at the receiver (end termination) or at the transmitter (source termination). Noise in the signal results either from imperfect termination at the ends of the line or from variations in the impedance along the line.†

A second source of noise is due to the leakage of signal energy from other transmission lines (aggressors) known as crosstalk. Improper design of transmission line often neglects the image current path. Image current can flow on closely routed signal traces on a PCB or nearby signal pins on a connector instead of the reference plane. The coupling appears as noise when the nearby signal transitions. The worst often occurs in the chip carrier where a reference plane is not readily available.

The noise source is modeled as either mutual inductance or capacitance in the LC model. The amount of noise is proportional to the aggressor's signal amplitude. Because the coupling is reactive, the noise is proportional to the frequency of the aggressor signal. This motivates the design of transmitters and receivers to filter frequencies above the data bandwidth, as will be discussed.

### 15.1.2 Frequency Response and ISI

An ideal transmission-line segment delays a signal perfectly; however, real transmission media attenuate the signal because of the line conductor's resistance and the loss in the dielectric between the conductors. Both loss mechanisms of wires increase at higher signal frequencies. Hence, a wire low-pass filters and the filtering increases with distance. The transfer function of a 6-m and 12-m cable, shown in Fig. 15.3a, illustrates increasing attenuation with frequency and distance. Figure 15.3b illustrates the effects of that frequency-dependent attenuation in the time-domain. The signal amplitudes are reduced and the energy of each bit is spread in time. If the bit time is short, the spreading causes interference between subsequent bits known as inter-symbol interference (ISI). As data rates increase beyond gigabits per second, the design of transmitters and receivers must incorporate additional filtering to compensate for this low-pass filtering.

### 15.1.3 Methods of Signaling

The characteristics of the transmission medium influence the trade-off between various signaling methods. In DRAM or backplane applications where a data word connects between multiple chips, multi-drop busses save considerable pins over point-to-point connections; however, each drop of a bus structure introduces a splitting of a transmission line that causes reflections and increases noise. A similar trade-off exists with differential and single-ended signaling. Differential signaling is more

*This noise can be compensated if the length of the delay and amount of reflection can be measured, but it adds significant complexity to the system.

†Impedance variations can be due to vias, changing of reference plane, connectors, etc.

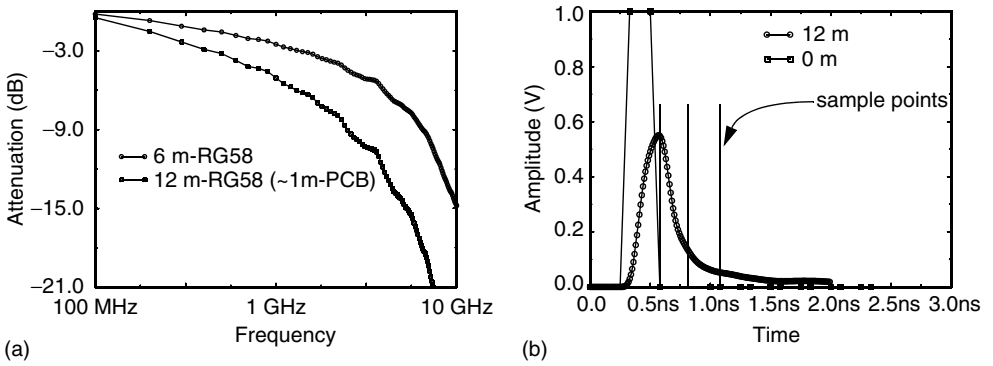



**FIGURE 15.3** Frequency (a) and time (b) domain of an RG-55U cable illustrating filtering and ISI.

robust to common-mode noise by using the second wire as the explicit image current path. A third trade-off involves whether or not signaling occurs in both directions on a pin simultaneously (full-duplex). Typically, an I/O link contains both a transmitter and a receiver on each end. Only one pair is operating at one time (half-duplex). Operating full-duplex halves the number of pins but degrades the signal amplitude and increases noise because the receiver must now compensate for the transmitted values. All three of these common choices are trading between the number of I/O pins and the signal-to-noise ratio (SNR). For high performance, system designers often opt for the more expensive options of point-to-point and differential links that are half-duplex. Some designers use single-ended signaling that has an explicit and dedicated ground pin for a signal's image current since perfectly differential structures are difficult to maintain in a PCB environment.

The following sections focus on the design of high-performance link circuitry in a high-performance system of point-to-point links. Many of the design techniques are applicable to busses and bidirectional links as well.

## 15.2 Transmitters

Transmitters convert the digital bits into analog voltages. Figure 15.4 illustrates the major pieces of a transmitter. Prior to the conversion by the output driver, transmitters commonly synchronize the data to that of a stable, noise-free clock so the resulting waveform has well-defined timing. Because I/Os often operate at a higher rate than the on-chip clock, the synchronization also multiplexes the data. The

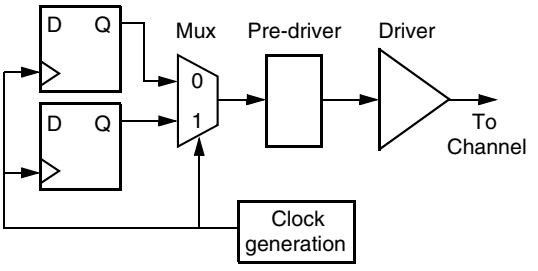



**FIGURE 15.4** Transmitter components.

simplest and most commonly used is 2:1 multiplexing, using each half-cycle of the clock to transmit a data bit.* A pre-driver follows the multiplexing and provides any pre-conditioning of the data signal.

The output voltage range depends on the signaling specification. If the voltage range nears or exceeds that of the on-chip supply voltage, the design must convert the voltage and ensure the reliability to the over-voltage. In addition to protection against electrostatic discharge (ESD), transistors that are not built to handle large voltages across the gate oxide or source/drain junction must be appropriately protected by cascoding and well-biasing. This section begins with discussing these large output-swing transmitters. The section follows with low-swing transmitters, which are more common for high-performance designs.

Noise is the primary challenge. Techniques that reduce noise such as impedance matching, swing control, and slew-rate control are described next. The last part addresses techniques that can be used to reduce intersymbol interference due to a band-limiting transmission channel.

### 15.2.1 Large-Swing Output Drivers

A simple push-pull architecture, as shown in Fig. 15.5, can drive a signal as large as the voltage provided for the I/O, $V_s$. When driving a transmission line, the initial output voltage is the result of a voltage division

$$V_o = V_s\left(\frac{R_o}{R_{\text{drv}} + R_o}\right)$$

where $R_{\text{drv}}$ is the on-resistance of the driving device. The initial voltage is also the final voltage if the line is terminated appropriately at the receiver. In which case, the driver draws continuous current even with the absence of signal transitions. With only source termination ($R_{\text{drv}}$ equal to $R_o$), the line voltage settles to $V_s$. The power dissipation is less since no current flows when the signal is constant. If the line is unterminated on either end, the signal will reflect several times before settling to $V_s$. Because the bit period must be long enough for the signal to settle, high-performance links avoid this penalty.

Impedance matching at the transmitter is challenging because (1) process, voltage, and temperature (PVT) varies, and (2) the impedance changes significantly as the device is switched from *on* to *off*. To minimize the net variation, designers over-design the size of the device for an impedance much lower than $R_o$. And then, by adding an external but constant resistance $R_{\text{ext}} = R_o - R_{\text{drv}}$, the net impedance varies within an acceptable tolerance.

Many chips are required to interface with chips that operate at different power supply voltages. As on-chip supplies lower with CMOS technology scaling, the disparity between on-chip and off-chip voltages increases. Unfortunately, for high reliability, the on-chip devices cannot tolerate excessive over-voltage. Catastrophic breakdown of gate oxide occurs at 12 MV/cm of oxide thickness.

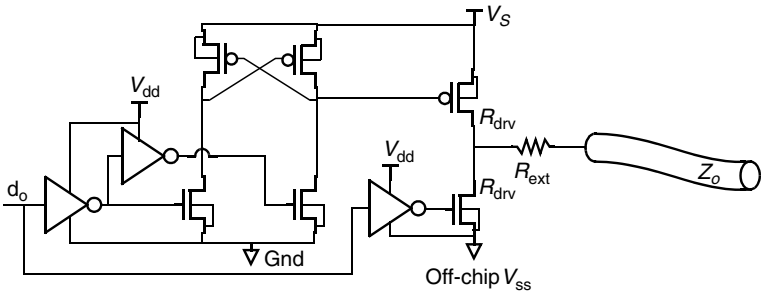



**FIGURE 15.5** Push-pull I/O driver with level shifting pre-drivers.

*For memories, the 2:1 multiplexing is known as double-data rate (DDR). The duty cycle of the clock is critical in guaranteeing a constant width of each bit. Even higher multiplexing has been demonstrated using multiple clock phases [52].

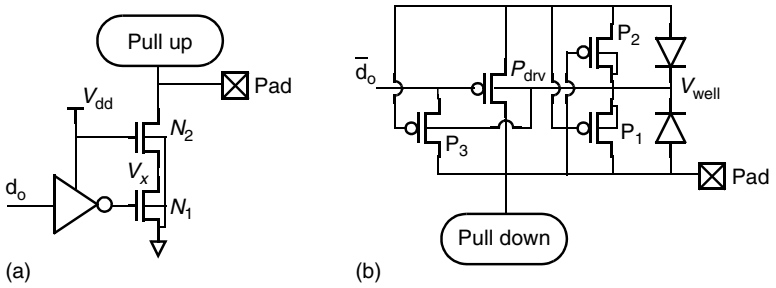



**FIGURE 15.6** Cascoding (a) and well-biasing (b) to protect driving devices.

Device technologists address the issue by providing transistors that are slower but high-voltage tolerant. One of the tasks of the pre-driver is to shift the level of the input so that the output-driver devices are fully turned *off*. Figure 15.5 illustrates an example of level-shifting using cross-coupled PMOS devices in the pre-driver.

To avoid over-voltage, circuit designers add a cascode transistor in series with the output switch to reduce the voltage drop [39]. Figure 15.6a shows a bottom device that switches with the data. The upper cascoding device uses a constant *high* gate voltage that is commonly the core $V_{dd}$. As long as the output voltage does not exceed $V_{dd} + V_{oxide(max)}$, the gate oxide is preserved. $V_x$ remains below $V_{dd} - V_{T(eff)}$, hence avoiding source/drain punchthrough of $N_1$.* To avoid a source/drain punchthrough of $N_2$ during an output *high-low* transition, the size of the cascode device needs to be large enough so that $V_x$ does not fall too quickly.†

PMOS devices for the pull-up pose an additional challenge. In a half-duplex configuration, the system tri-states the transmitter by pulling the gate of the driving device to the I/O supply voltage, $V_s$; however, with reflections and inductive ringing, line voltages can exceed $V_s$. To avoid forward biasing the drain–well junction, designers leave the well floating, as shown in Fig. 15.6b [8]. Transistors $P_1$ and $P_2$ allow the well to be charged up to either the pad voltage or $V_s$ depending on which is higher. To avoid conduction of the driving device when pad voltage is *high*, $P_{drv}$, transistor $P_3$ pulls the gate input to the pad voltage.

## 15.2.2 Small-Swing Output Drivers

I/O standards are migrating toward smaller output voltage swings due to several advantages. There is less concern regarding over-voltages on I/O devices. Using smaller devices and fewer over-voltage protection devices reduces output capacitance and improves bandwidth. The device stays in a single region of operation (either in triode or saturation) reducing impedance mismatches. The transmitter also dissipates less power because of the lower-swing and smaller drive devices; however, reducing signal swing directly reduces the SNR making the designs more sensitive to noise.‡ The following describes two commonly used driver architectures: low-impedance and high-impedance drivers.

A simple extension of the large-swing push-pull driver to low-swing is shown in Fig. 15.7, where $V_s$ is a low voltage that determines the signal swing. The transistors operate in the linear region of their I–V curve appearing as a low-impedance signal source. With signal swings under 1 V, a smaller NMOS device can have the same pull-up resistance as PMOS devices. The impedance matching is better than the large signal driver because the device impedance varies less with $V_{ds}$ [19].

However, with low-impedance drivers, power-supply noise appears directly on the signal. By connecting the power supply as the signal's return connection, the noise would appear as common-mode.

*Feedthrough from output to the $V_{gate}$ of $N_2$ can dynamically elevate $V_x$ so $N_2$ cannot be excessively large.

†$N_2$ can often be a size 4× larger than $N_1$.

‡Fortunately, many noise sources are proportional to the signal swing, so the SNR degradation is not overly severe.

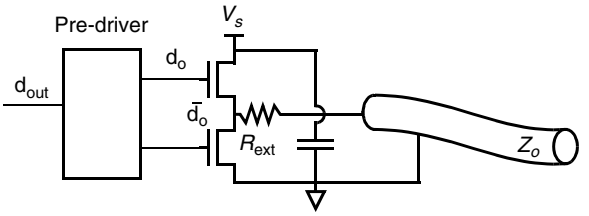



**FIGURE 15.7** Low-swing, push-pull driver with supply bypassing.

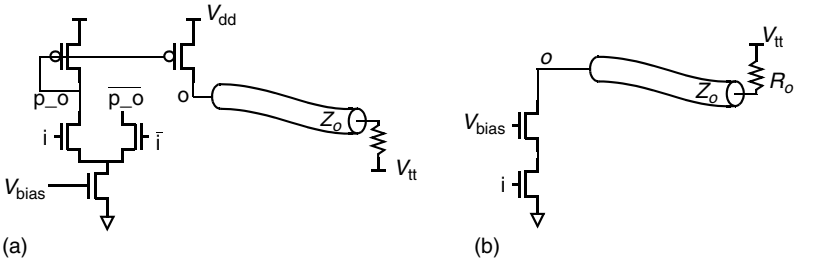



**FIGURE 15.8** High-impedance drivers: PECL (a) and open-drain (b).

Unfortunately, the connection is difficult because multiple I/Os (and sometimes the core logic) share the ground to minimize cost. Furthermore, signal current flows through two supplies. To reduce noise, designers trade-off area and pin by (1) bypassing $V_s$ to ground with a large capacitance, (2) limiting the number of I/Os sharing a single ground, and (3) carefully minimizing the inductive loop formed by the current return path (ground connection).

A second style of drivers, high-impedance drivers, switch currents instead of voltages. By keeping transistors in the saturated-current region, the devices appear as current sources. The current can be switched either differentially in PECL type drivers (Fig. 15.8a) or single-ended in open-drain type drivers (Fig. 15.8b). To provide source termination, a resistor ($R_o$) can be placed in parallel with the output. These drivers have several advantages over their low-impedance counterparts. The outputs have less noise because the high-impedance isolates the output from one of the power supplies, but it is critical for the current to remain constant. The output bandwidth is higher because the saturated device (with a higher $V_{ds}$) is smaller in size, for a given current than a triode device (with low $V_{ds}$); however, because of the higher $V_{ds}$, these drivers dissipate more power, $I \cdot V_s$.

For both high- and low-impedance drivers, switching currents inject noise onto the supply via di/dt. Instead of using purely single-ended drivers, complementary single-ended drivers approximates a constant current and reduces the noise. Differential drivers such as PECL force a constant current over time and eliminate the problem.*

### 15.2.3 Impedance, Current, and Slew-Rate Control

Process, voltage, and temperature (PVT) variations can cause drive resistance (of low-impedance drivers) and currents (of high-impedance drivers) to deviate from the design target causing offsets and noise. For robust operation, control loops are often used to dynamically maintain the proper impedance or current. To minimize coupled and reflected noise, designers also limit the high-frequency spectral content of the output signal. This section describes these noise reduction methods.

*The drawback is that differential drivers have slightly larger output capacitance because the differential input devices have smaller $V_{gs}$ and need to be larger to switch the output current.

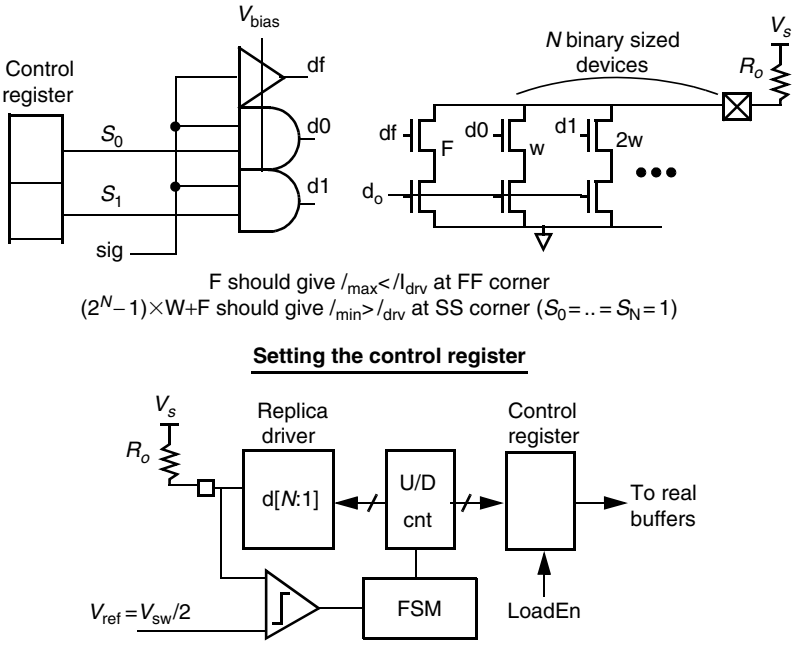



**FIGURE 15.9** Current-control feedback loop.

Figure 15.9 illustrates the block diagram of a loop that controls the current of a high-impedance driver [28]. The output driver device is divided into binary-weighted segments. A digital control word, stored in a register, sets the number of transistors used by the driver. A replica driver determines the control word. The replica drives half the output impedance. A comparator compares the output voltage with a reference voltage set at $V_s - V_{sw}/2$, where $V_{sw}$ is the desired voltage swing of the output. The comparison result increments or decrements the control word until reaching the desired output current. A similar loop can control the output impedance by adjusting the resistance of driving devices using binary-weighted segments [7].

As mentioned earlier, filtering the high-frequency spectral content of the output signal reduces coupling noise. This is equivalent to limiting the output slew rate, but an excessively low slew rate may filter the signal's desired spectral frequencies and cause ISI. The difficulty arises when the slew rate is designed for the fastest operating condition. The slowest operating condition would cause excessive ISI. Early designs of drivers use devices that correlate inversely with transistor speed. In the example shown in Fig. 15.10a, an output device can be broken into segments and each segment turns on sequentially [41]. The delay can be introduced using polysilicon resistors, which are not very sensitive to PVT. More recent methods (Fig. 15.10b) control the rate at which the pre-driver turns on the output device. By using a control voltage that tracks PVT,* the pre-driver resistance or current stays constant and consequently the slew-rate.

## 15.2.4 Transmitter Pre-Emphasis

When the data rate exceeds the channel bandwidth, designers compensate for the filtering by equalization. Because of the ease of implementation, many high-speed links equalize at the transmitter by pre-distorting the signal to emphasize higher frequencies [6,14,50]. Early pre-emphasis designs were known as advanced pull up/down (APU/D) [12], which were applied to driving large capacitances. The

*The control voltage can be the voltage of a VCO whose frequency is locked to an external reference clock via a PLL [49].

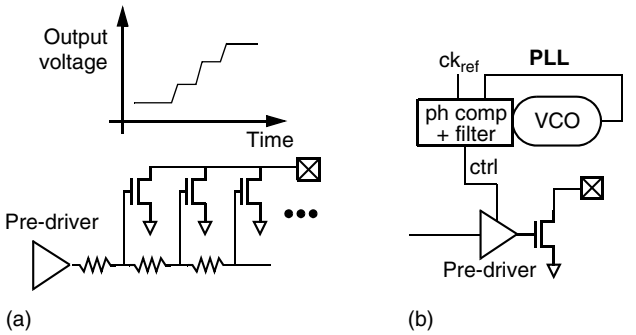



**FIGURE 15.10** Slew-rate control using resistors (a) and controlled pre-driver (b).

technique turns on the driver more strongly for a period immediately after a data transition so that the transmitter drives higher frequency components with more signal power. For more complex channel responses, a programmable filter precedes the actual line driver and inverts the effect of the channel. Figure 15.11a illustrates an example of an analog filter. The length of the optimal filter depends on the tail of the pulse response. For many cables (less than 10 m) one or two taps is sufficient [6,13,14]. Figure 15.11b shows the effect of transmitter equalization. The small, negative pulses before and after the original pulse eliminates the tails of the pulse response.

A digital-FIR filter would output a quantized word that represents the output voltage. Instead of transmitting two levels, the driver is a high-speed D/A converter. Because current or impedance control uses binary-weighted driver segments, designs for a D/A converter are not significantly different; however, a design with linearity of >6 bits at multi-GSamples/sec faces challenging issues: device mismatches limit linearity, transmit clock jitter limits the SNR, and the switching of output transitions induce glitches. Thermal noise for a 50-$\Omega$ environment is approximately 1 nV/$\sqrt{\text{Hz}}$ and only limits resolution at very high resolution. Recent research has demonstrated this potential with <6-bit D/A converters [10,13].

Two system issues must be considered when implementing transmitter pre-distortion. First, transmit power is limited, so the low-frequency signal energy must be attenuated to that of the worst-case attenuation of the channel. This leads to significant loss of SNR. Second, the channel characteristic is not known to the transmitter. Accurate filter coefficients are dynamically trained with loopback information sent from the receiver, which adds complexity to the system.

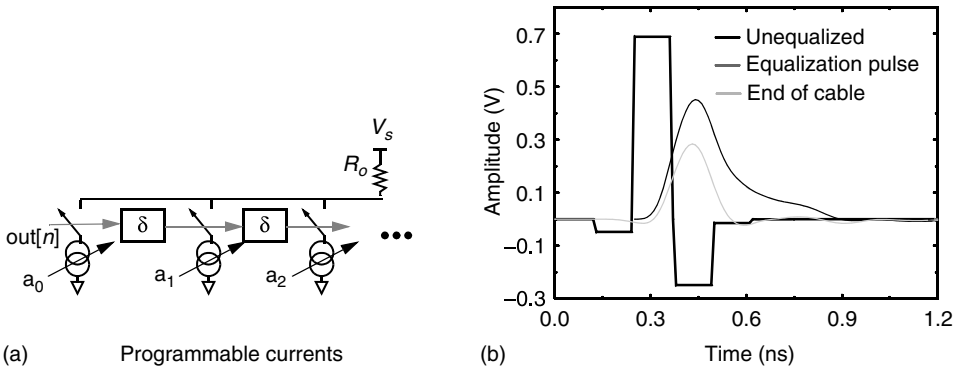



**FIGURE 15.11** Transmitter pre-distorted waveform (a) and implementation (b).

## 15.3 Receivers

The task of the receiver is to convert the analog waveform from the channel into a sequence of binary data. Figure 15.12 illustrates the common components of a receiver. First, an input amplifier conditions the signal. A sampling circuit follows and captures the analog value of each bit. A comparator amplifies the sampled value to digital values. Similar to the transmitter, the sampling block often demultiplexes the data so that the on-chip clock rate can be slower than the off-chip data rate. The most simple and common design uses two samplers operating on opposite edges of a digital clock for 2:1 demultiplexing.

The primary difficulty in high-performance receiver design is maintaining low noise, both static and dynamic. The noise of a signal at the receiver can be illustrated by an eye diagram (Fig. 15.13), which overlays the waveform of each bit of a random sequence. The transmitter design and the channel contributes the majority of the signal's amplitude and timing noise. The receiver should compare the signal with a proper reference voltage. Static offsets reduce the effective signal amplitude reducing the SNR. To minimize dynamic noise, the receiver should reject supply and common-mode noise, filter high-frequency input noise, and avoid any bandwidth limitation and ISI. Sampling the data at the optimal point will be addressed in Section 15.4. This section describes several examples of high-performance receiver designs. Then techniques to reduce noise and ISI are addressed.

### 15.3.1 Receiver Designs

Figure 15.14 illustrates an example of a receiver design. The first stage performs several tasks: (1) filtering the noise, (2) level-shifting the output, and (3) amplifying the signal. An amplifier with appropriate bandwidth can filter input noise frequencies above the data bandwidth. Furthermore, using a differential

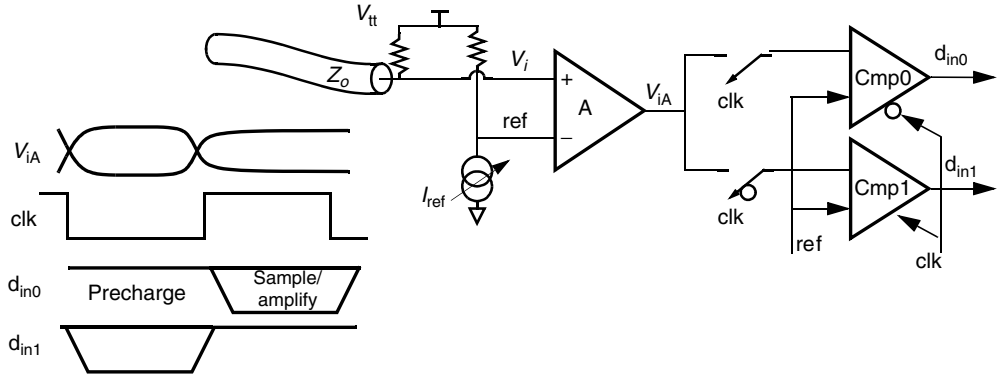



**FIGURE 15.12** Receiver components.

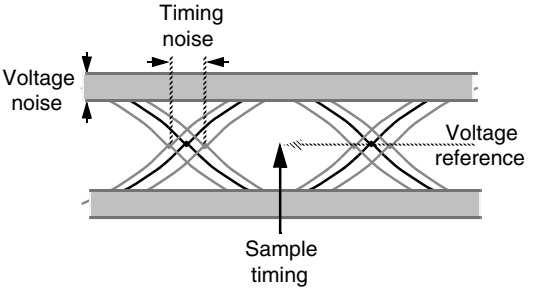



**FIGURE 15.13** Eye diagram at the receiver.

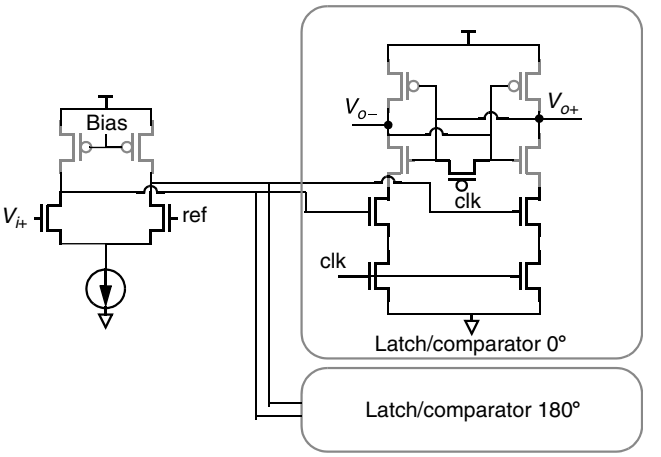



**FIGURE 15.14** Receiver design with receiving amplifier.

structure improves the common-mode and supply noise rejection even though the input may be single-ended. The outputs of the first stage [5,9] are differential for good supply noise sensitivity and are level-shifted* to accommodate the clocked comparator that follows [34]. A *high* clock level resets the comparator shown in Fig. 15.14. The negative clock edge samples the data and starts a positive feedback that regeneratively amplifies the sampled value to digital values during the *low* clock phase. To demultiplex the data, the comparators operate on different clock phases. The amplification is exponentially dependent on the duration of the *low* phase. Because the comparator has high gain, the first stage does not need significant gain. Some gain reduces the effective input offset voltage since the contribution of the comparator's offset is divided by the gain. Mismatch in the feedback devices and clock coupling of the comparators can introduce significant offsets. For very high data rates, the drawback of the design is that the first stage must have sufficient bandwidth to minimize ISI. Furthermore, delay variation of the first stage can add timing noise.

A simple design can avoid ISI by eliminating the first stage and sampling/demultiplexing the input with comparators directly [24,52]. Because the comparators are reset before each sample, no signal energy from previous bits remains hence removing ISI; however, direct sampling is noisier and has larger static offsets. Figure 15.15 illustrates an alternate design that clocks the first stage to remove ISI but still conditions the signal [26,43]. During the low phase of the clock, the amplifier output is reset. During the high phase of the clock, the amplifier conditions the data. For demultiplexing, two clocked amplifiers loads the input. A comparator samples the amplifier output to further amplify to digital levels. The clock used for the clocked amplifier must be timed with the arriving signal to amplify the proper bit. The timing issue will be discussed in Section 15.4.

### 15.3.2 dc Offsets

Random dc offsets limit the voltage resolution of the receiver. These offset are due to random mismatches in the devices and scales inversely with the size of the device [35]. Because minimum size devices are often used to minimize pin capacitance and power dissipation, input-referred offset of amplifiers and comparators can be tens of millivolts.

To compensate for the error, devices are added that can create an offset in either the first amplifier or the comparator. The control can be open-loop where the compensation value is determined with an initial calibration [10]. Figure 15.16a shows a comparator with digitally controllable switches that

*The input common-mode voltage depends on the transmitter and the I/O specification.

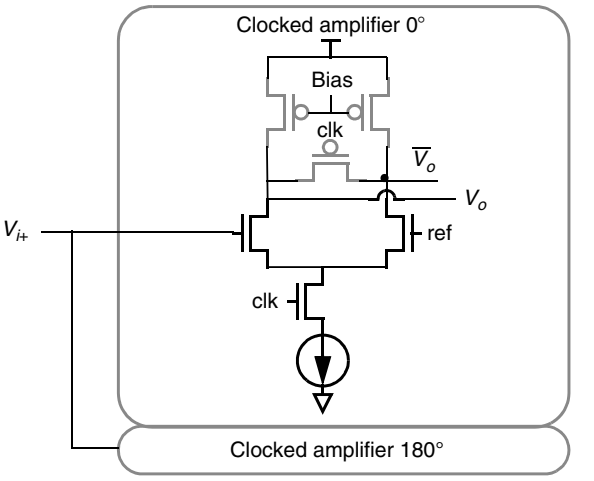



**FIGURE 15.15** Receiver design using clocked amplifier/sampler as first stage.

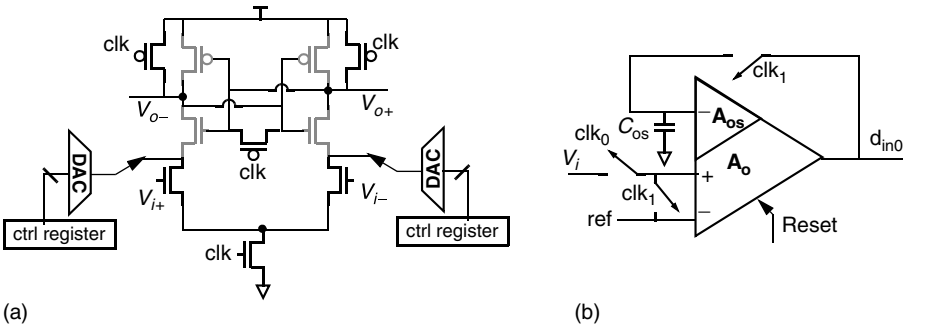



**FIGURE 15.16** Offset cancellation using digital controllable switches (a) and using feedback control (b).

differentially inject an error current. The open-loop compensation is commonly digital so the value does not drift in time. Alternatively, the control can be continuously operating and closed-loop [51]. As shown in Figure 15.16b, a third nonoverlapping clock phase, $clk_1$, is added to the reset and amplify ($clk_0$) phases of operation. $Clk_1$ reconfigures the amplifier to short the inputs and to store the value of the offset on capacitors, $C_{os}$. If data is encoded so that averaged dc is a constant (dc-balanced), a similar technique finds the offset by averaging of the received data [22] instead of wasting a phase to short the inputs.

### 15.3.3 Noise

The main sources of noise for a receiver are mutual coupling between I/O signals and differences between chip ground and board ground. Large image currents that flow through the supply pins to support the output drivers cause significant voltage differences.* Some of the supply noise inevitably appears at the input to the receiver. Signaling differentially and carefully routing the two signals together can effectively reduce noise to the order of tens of millivolts. Supply noise couple capacitively as common-mode noise. Furthermore, mutual coupling from other signals is at least partially compensated by coupling from its complement.

*On-chip bypass capacitance only reduces chip $V_{DD}$ to chip ground noise, and has no effect on the noise between chip ground and board ground.

Single-ended signaling can achieve nearly the same performance if the return current supply connection is brought on-chip, tightly coupled to the signal through a separate pin. The receiver's reference can be derived from the return connection, but this requires the same number of pins as differential signaling. To save pins, most single-ended systems use the chip supplies ($V_{dd}$ and ground) to derive the reference. Or, several receivers share a single return current connection. Unfortunately, since the reference signal is shared, the capacitance between the supplies to the input pad and to the reference voltage differ. The larger capacitance to the reference couples more high-frequency supply noise [26,42,46]. Single-ended systems typically require larger input swings than differential systems for the same performance.

A band-limited receiving amplifier can filter some of the noise. One approach to control the bandwidth is to bias the effective load transistors with a control signal that tracks the bit time.* To maintain constant output swing, the bias current of the differential amplifier must also track. An ideal filter for square-wave inputs averages the input signal over the bit time with an integrator† [43]. An integrating receiver replaces the load elements with capacitors. The capacitors integrate the current that is switched by the input value. At the end of the bit time, a comparator samples and compares the values on the capacitors before the integrator is reset.

### 15.3.4 Receiver Equalization

With data rates above the bandwidth of the channel, an alternative to transmitter pre-emphasis is to build the inverse channel filter at the receiver. Designers can increase the gain of the first amplifier at high-frequencies to flatten the system response [45]. The required high-pass filter can also be implemented digitally by first feeding the input to an analog-to-digital converter (ADC) and digitally post-processing the ADC's output. An ADC is commonly used in disk-drive read channels since it also allows one to implement more complex nonlinear receive filters. Although this approach works well at frequencies lower than 1 GHz, it is very challenging with gigahertz signals because of the required GSamples/sec converters. Recent research demonstrated a multi-GSamples/sec 4-bit ADC [10] (1 W of power), which indicates the potential of high data rate conversion albeit with high power dissipation. Instead of a digital implementation, for less area and power overhead at these high bit rates, a simple 1-tap FIR filter ($1 - \alpha D$) has been implemented as a switched-current filter [13] or a switched capacitor filter [47].

## 15.4 Timing Generation and Recovery

The task of timing recovery essentially determines the timing relationship between the transmitter and the receiver so that the data can be received with minimal error. Typically, the burden of adjusting the timing relationship falls on the receiver. Transmitter clocking is much easier where one primarily needs a low-jitter clock source.‡ The receiver has a more difficult task of recovering the timing from the received signals.

The prior receiver discussion does not address how to generate the clock for the amplifiers and samplers. Recovering a clock signal with low timing noise (jitter) and with accurate phase position is the most difficult challenge for high data rates. The same eye diagram in Fig. 15.13 illustrates the timing margin of a receiver. To maximize the timing margin, the receiver should sample the data in the middle of the data-eye.§ If clocked amplifiers are used, the clock should be in-phase with the data to maximize the settling time of the amplifier. Furthermore, designs should minimize the jitter of both the sampling clock and the clock used at the transmitter. Almost all clock recovery circuits use a feedback loop known

*Similar to transmitter slew-rate control, one can leverage the fact that buffers in the clock generator have been adjusted to have a bandwidth related to the bit rate [49].

†Most signals are not perfect square waves. In addition to finite signal slew rate, bit boundaries contain timing uncertainty. Integrating over a portion of the bit-time ("window") can reduce noise.

‡If data is multiplexed, clock phases must be properly positioned. For 2:1 multiplexing, the duty cycle needs to be 50%.

§The eye may not be symmetric. Off-center sampling may increase the amplitude of the sampled signal.

as a phase-locked loop (PLL) to adjust the clock phase position and minimize jitter. This section discusses different PLL architectures and methods to reduce offsets from the ideal sampling position (static phase offsets) and jitter.

### 15.4.1 Architectures

A PLL is often used to synchronize the transmitter clock's phase and frequency* to that of a system clock. In order to transmit phase information along with the data, two methods are commonly used. For short distances of a wide data bus, source synchronous clocking is a method that transmits a clock in-phase with the data. Otherwise, prior to transmission, data is encoded to contain periodic data transitions that can be used to align the receive clock [13,15]. In some systems, the receiver and the transmitter use clocks with slightly different frequencies. Then the timing recovery PLL has the additional task of recovering the frequency from the data transitions.

Figure 15.17 shows the architecture of a PLL. Two basic approaches are used: oscillator-based PLLs, and delay-line-based PLLs or delay-locked loops (DLLs). Both systems are similar feedback loops where a control voltage ($V_{ctl}$) adjust the phase of the periodic output signal ($clk_{int}$) to have a fixed phase relationship with the input signal ($inp_{ref}$). To distribute the clock to many receivers, a buffer chain drives the clock line, $clk_{samp}$.

DLLs control the output phase by directly adjusting the delay of a voltage-controlled delay line (VCDL) [25]. The control loop integrates the control voltage to drive the phase error to zero. This feedback loop is a first-order loop and is guaranteed stability, but it is constrained in that the frequency of the input clock ($clk_{ext}$ or $inp_{ref}$) determines the frequency of the output signal. Furthermore, the delay elements limit the maximum and minimum delay of the line. Designing the range to be large enough for all PVT and starting the loop at the correct delay often require auxiliary circuits. Using an oscillator-based PLL provides more flexible in frequency and phase. The oscillator is often implemented using a ring of controllable delay elements,† but an oscillator-based system is more complex to design. The phase of the output signal is adjusted by integrating the change in frequency of the oscillator. Thus, an oscillator-based PLL is a higher-order control system that has stability constraints [3,38].

Phase detector designs vary depending on whether the input reference is a clock or a data sequence. Recovering the phase from an input clock is easier because a transition is guaranteed every cycle. An example shown in Fig. 15.18a is an SR-latch where the Q and $\overline{Q}$ outputs have equal pulse widths when the input clocks are spaced 180° apart [44]. When the phases deviate from 180°, the difference in pulse width indicates the phase difference. For data input, the added difficulty is recovering the transitions. A common design technique shown in Fig. 15.18b uses XORs to compare consecutive bits [20]. When

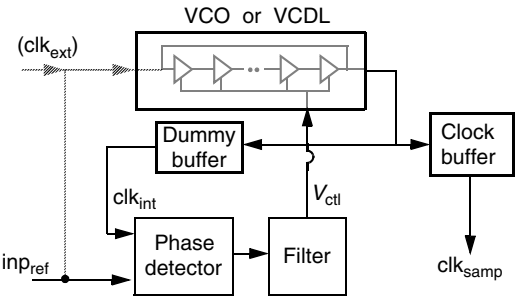



**FIGURE 15.17** Phase-locked loop architecture using oscillator (a) and delay-line (b).

*PLLs are often used to generate a multiplied frequency.

†With the availability of on-chip inductors, LC-type oscillators often used in RF applications are being considered in large digital ICs.

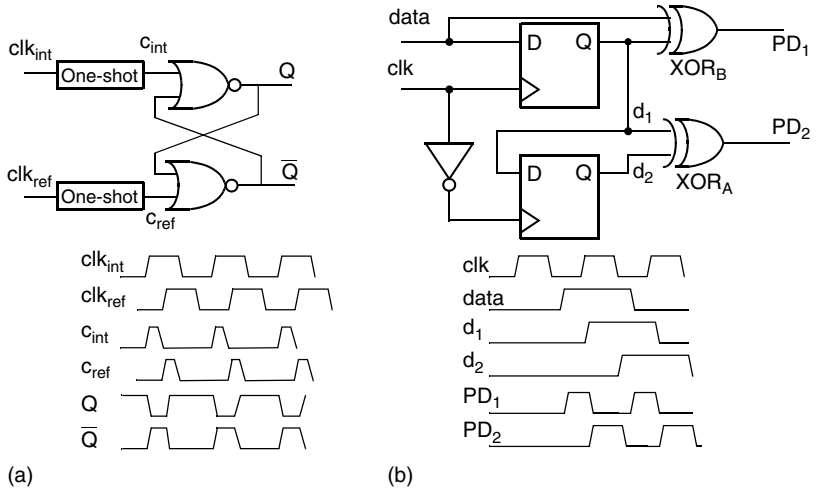



**FIGURE 15.18** Phase-detection using SR-latch (a) or XOR (b) to detect data transition.

the XOR output is *high*, a phase difference is present. $PD_1$ is high starting on a transition of the input to the rising edge of the clock. $PD_2$ is high for half the clock period whenever data transitions. The phase difference is the difference between the pulse width of $PD_1$ and $PD_2$.*

## 15.4.2 Minimizing Jitter

Jitter in the sampling clock is primarily due to the sensitivity of the loop elements to supply noise. Although the feedback system can correct for noise with frequencies below the bandwidth of the loop, high-frequency noise can appear as jitter on the output clocks. Loop elements, especially oscillator or delay-line buffer elements, are often differential and have high common-mode and supply rejection to minimize the noise. Oscillators in particular are carefully designed because noise causes errors in frequency [32]. Phase error accumulates because it is the integral of the frequency error.

Many clocks drive large capacitances. Clock buffers are typically CMOS inverters for power efficiency, but they have much higher supply sensitivity than the delay buffers† and cause over half of the total jitter of the output clock. Dummy clock buffers are often included in the feedback of the PLL (Fig. 15.17) to use the feedback loop to track out the low frequency portion of the noise [1,21]. A well-designed loop in a system with 5% supply noise will often have a jitter roughly 0.5 of the delay of a FO-4 inverter‡ of the clock period. Intrinsic jitter without supply noise can be more than three times less.

## 15.4.3 Phase Detection and Static Phase Offsets

In addition to the jitter, dc phase offsets are equally important in maximizing the timing margin. Using a loop that integrates the phase error helps reduce any inherent offsets. The offset primarily depends on any errors in the time spacing between sampling clocks when demultiplexing, and the mismatch between the phase detector and the receiver.

In a 1:2 demultiplexing receiver, the clock (0°) and its complement (180°) are used. Duty cycle errors can cause one receiver to not sample at its optimal location. Typically, a correction loop is added to the

*In order to recover frequency where the input data frequency is significantly different from the oscillator natural frequency, phase detection alone is often not sufficient. An entire class of circuits aids frequency acquisition [13,36,40].

†A 1% change in supply yields roughly a 1% change in delay, which can be 10× that of delay buffers.

‡For a figure relatively insensitive to PVT, the time can be normalized relative to the delay of a FO-4 inverter.

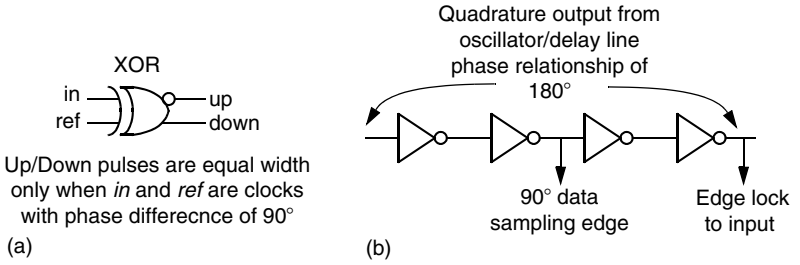



**FIGURE 15.19** 90° locking using XOR (a) and ring-oscillator (b).

PLL output to guarantee 50% duty cycle* [31]. The loop averages a clock waveform to determine the duty cycle. Using the information, the duty cycle can be adjusted by changing the logical threshold of a clock buffer.

To sample at the middle of a data bit, a clock must be 90° shifted with respect to the data. This shift can be achieved by either (1) using a phase detector that indicates zero error when the difference is 90° [42], or (2) locking to 0° and shifting the clock by 90°. The first method employs XORs in the design of the phase detector. Figure 15.19a illustrates a simple case, when the reference input and loop output are two clock waveforms. The XOR output has equal high and low durations when the clocks are 90° apart. In the second method shown in Fig. 15.19b, reference and loop output are locked in-phase. Using a ring-oscillator with even number of stages, an internal clock phase in the ring can be tapped for the 90° clock [27,49].

A common error in phase locking is that the receiving comparator has a nonzero setup time. To optimally sample the data, the clock position should be adjusted for the setup time.† An additional delay at the input of the phase detector can compensate for the setup delay. The most accurate compensation is to use a replica of the receiver as the phase detector since the setup time is inherent to the receiver; however, this poses challenge because a receiver does not give phase information proportional to the phase difference. The output only indicates that the loop clock is either earlier or later than an reference clock transition. An oscillator-based loop is not stable with this type of bang-bang control so only DLLs can be built. In order to also lock to the input frequency, a clever design uses a dual-loop architecture that locks to the input frequency using a core loop [21,27,44]. Coarsely-spaced clock phases from the core loop are interpolated‡ to generate a clock phase that can be finely controlled. This loop clock is locked to the input phase using a receiver replica phase detector. Using these techniques, phase offsets can be smaller than 2% of the bit time.

## 15.5 Conclusion

This chapter has described the design goals and challenges for high-performance I/O. Performance using 2:1 multiplexing of greater than 5 Gb/s has been demonstrated using a 0.18-$\mu$m CMOS technology [16,18,40]. Higher bit-rates have been shown using higher degree of multiplexing and demultiplexing. Because transistor speeds will scale with technology, link speeds are expected to scale as well. Unfortunately, noise coupling due to parasitic capacitances and inductances increases with frequency requiring designs to be even more robust to noise. Designs employ many of the noise

*A higher degree of demultiplexing requires multiple phases to be generated and tuning of each phase position [52].

†An error that is not easily dealt with is any data dependent setup time variations. This can be minimized by designing the receiver for low input-offset voltage and hysteresis.

‡Interpolation takes two clock phases and performs a weighted average to generate an intermediate clock phase [32,52].

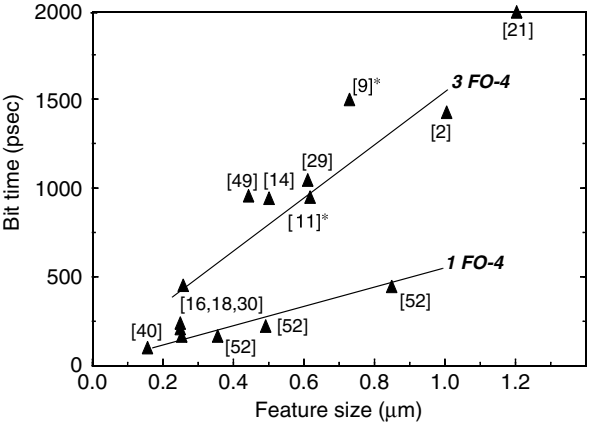



**FIGURE 15.20** Scaling of link performance with process technology.

reductions techniques described in this chapter and have continued to scale. Figure 15.20 illustrates the scaling so far. Future designs will need to improve these noise reducing and filtering techniques. Furthermore, wire bandwidth does not scale with technology scaling so the compensating for the low-pass filtering will be even more important.

Methods are being researched that can squeeze more bits into existing bandwidth. Given an SNR, Shannon's limit shows the maximum channel capacity to be Capacity$/f_{bw} = \log(1 + \text{SNR})$. Researchers are beginning to show that multilevel $(4 + \text{PAM})$ can be encoded in each bit period at the gigabits per second broadband data rate [10,13]. This and many techniques [37] that have been demonstrated in phone modems [4,23] can dramatically increase capacity to 10 bits/Hz in extremely noisy conditions, but all require accurate A/D and D/A converters. Research has shown that they are feasible but require extremely accurate timing. Low-jitter PLLs that lock accurately to the data phase are critical in maintaining the resolution at the gigahertz sampling rates.

Power of these links is becoming an important issue. For many digital systems, the aggregate off-chip bandwidth is expected to exceed terabits per second in 2010. The data rate is not expected of a single link but over hundreds of I/Os. Each I/O cannot afford power more than a few tens of milliwatts.

These issues challenge the next generation of higher-performance link designs. The availability of faster and more abundant transistor as CMOS technology scales will help designers face the challenges.

## References

1. Alvarez, J. et al., "A wide-bandwidth low-voltage PLL for PowerPC microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 4, pp. 383–391, April 1995.
2. Banu, M., and A. Dunlop, "A 660Mb/s CMOS clock recovery circuit with instantaneous locking for NRZ data and burst-mode transmission," in *1993 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, San Francisco, CA, pp. 102–103, Feb. 1993.
3. Best, R., *Phase-Locked Loops*, 3rd ed., McGraw Hill, New York, 1997.
4. Boxho, J. et al., "An analog front end for multi-standard power line carrier modem," in *1997 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 84–85.
5. Chappell, B. et al., "Fast CMOS ECL receivers with 100-mV worst case sensitivity," *IEEE Journal of Solid State Circuits*, vol. 23, no. 1, pp. 59–67, Feb. 1988.
6. Dally, W.J. et al., "Transmitter equalization for 4-Gbps signaling," *IEEE Micro*, vol. 17, no. 1, pp. 48–56, Jan.–Feb. 1997.
7. DeHon, A. et al., "Automatic impedance control," in *1993 IEEE International Solid-State Circuits Conference. Digest of Technical*, pp. 164–165, Feb. 1993.

8. Dobberpuhl, D. et al., "A 200-MHz 64 b dual-issue microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, p. 1555, Nov. 1992.
9. Donnelly, K.S. et al., "A 660 MB/s interface megacell portable circuit in 0.3 $\mu$m–0.7 $\mu$m CMOS ASIC," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 12, pp. 1995–2003, Dec. 1996.
10. Ellersick, W. et al., "A serial-link transceiver based on 8 GSa/s A/D and D/A converters in 0.25-$\mu$m CMOS," in *2001 IEEE International Solid-State Circuits Conference. Digest of Technical*, pp. 58–59, Feb. 2001.
11. Enam, S.K., and A.A. Abidi, "NMOS ICs for clock and data regeneration in gigabit-per-second optical-fiber receivers," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 12, pp. 1763–1774, Dec. 1992.
12. Esch, G.L., Jr. et al., "Theory and design of CMOS HSTL I/O pads," *Hewlett-Packard Journal Hewlett-Packard*, vol. 49, no. 3, pp. 46–52, Aug. 1998.
13. Farjad-Rad, R. et al., "A 0.3-$\mu$m CMOS 8-GS/s 4-PAM serial link transceiver," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 5, pp. 757–764, May 2000.
14. Fiedler, A. et al., "A 1.0625 Gbps transceiver with 2x-oversampling and transmit signal preemphasis," in *1997 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 238–239.
15. Franaszek, P., A. Widmar, "Byte oriented DC balanced (0,4) 8B/10B partitioned block transmission code," US Patent 4486739, Dec. 04, 1984.
16. Fukaishi, M. et al., "A 4.25-Gb/s CMOS fiber channel transceiver with asynchronous tree-type demultiplexer and frequency conversion architecture," *IEEE Journal of Solid-State Circuits*, vol. 33, pp. 2139–2147, Dec. 1998.
17. Galles, M. et al., "Spider: a high-speed network interconnect," *IEEE Micro*, vol. 17, no. 1, pp. 34–39, Jan.–Feb. 1997.
18. Gu, R. et al., "A 0.5–3.5 Gb/s low-power low-jitter serial data CMOS transceiver," in *1999 International Solid-State Circuits Conference. Digest of Technical Papers*, San Francisco, CA, pp. 352–353, Feb. 1999.
19. Gunning, B. et al., "A CMOS low-voltage-swing transmission-line transceiver," in *1992 International Solid-State Circuits Conference. Digest of Technical Papers*, San Francisco, CA, Feb. 1992.
20. Hogge, Jr., C.R., "A self-correcting clock recovery circuit," *IEEE Transation on Electron Devices*, vol. ED-32, pp. 2704–2706, Dec. 1985.
21. Horowitz, M. et al., "PLL design for a 500 MB/s interface," in *1993 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 160–161.
22. Hu, T.H. et al., "A monolithic 480 Mb/s parallel AGC/decision/clock-recovery circuit in 1.2-$\mu$m CMOS," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 12, pp. 1314–1320, Dec. 1993.
23. Ishida, H. et al., "A single-chip V.32 bis modem," in *1994 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 66–67.
24. Johansson, H.O. et al., "Time resolution of NMOS sampling switches used on low-swing signals," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 2, pp. 237–245, Feb. 1998.
25. Johnson, M.G. et al., "A variable delay line PLL for CPU-coprocessor synchronization," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 5, pp. 1218–1223, Oct. 1988.
26. Kushiyama, N. et al., "A 500-megabyte/s data rate 4.5 M DRAM," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 4, pp. 490–498, April 1993.
27. Larsson, P. et al., "A 2–1600 MHz 1.2–2.5 V CMOS clock-recovery PLL with feedback phase-selection and averaging phase-interpolation for jitter reduction," in *1999 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 356–357, Feb. 1999.
28. Lau, B. et al., "A 2.6 GB/s multi-purpose chip to chip interface," in *1998 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 162–163.
29. Lee, K. et al., "A jitter-tolerant 4.5 Gb/s CMOS interconnect for digital display," in *1998 IEEE International Solid-State Circuits Conference. Digest of Technical*, pp. 310–311, Feb. 1998.
30. Lee, M.J., W. Dally, and P. Chang, "Low-power area-efficient high-speed I/O circuit techniques," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1591–1599, Nov. 2000.

31. Lee, T.H. et al., "A 2.5 V CMOS delay-locked loop for 18 Mbit, 500 megabyte/s DRAM," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 12, pp. 1491–1496, Dec. 1994.
32. Maneatis, J.G. et al., "Low-jitter process-independent DLL and PLL based on self-biased techniques," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1723–1732, Nov. 1996.
33. Matick, R., "Transmission lines for digital and communication networks," 3rd ed., IEEE Press, 1997.
34. Montanaro, J. et al., "A 160-MHz 32-b 0.5-W CMOS RISC microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1703–1714, Nov. 1996.
35. Pelgrom, M.J., "Matching properties of MOS transistors," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 10, p. 1433, Dec. 1989.
36. Pottbacker, A. et al., "A Si-bipolar phase and frequency detector IC for clock extraction up to 8Gb/s," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 1747–1751, Dec. 1992.
37. Proakis, J. and Salehi, M., *Communication Systems Engineering*, Prentice-Hall, 1994.
38. Razavi, B. Editor, *Monolithic Phase Locked Loops and Clock Recovery Circuits*, IEEE Press, 1996.
39. Sanchez, H., et al., "A versatile 3.3 V/2.5 V/1.8 V CMOS I/O driver built in a 0.2-mm 3.5 nm tox 1.8 V technology," in *1999 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, p. 276.
40. Savoj, J., B. Razavi, "A 10 Gb/s CMOS clock data recovery circuit," *Symposium on VLSI Circuits (IEEE/JSAP). Digest of Technical Papers*, Honolulu, HI, pp. 136–139, June 2000.
41. Senthinathan, R. et al., "Application specific CMOS output driver circuit design techniques to reduce simultaneous switching noise," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 12, pp. 1383–1388, Dec. 1993.
42. Sidiropoulos, S. et al., "A CMOS 500-Mbps/pin synchronous point to point link interface," in *Proceedings of 1994 IEEE Symposium on VLSI Circuits. Digest of Technical Papers*, pp. 43–44.
43. Sidiropoulos, S. et al., "A 700-Mb/s/pin CMOS signaling interface using current integrating receivers," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 5, pp. 681–690, May 1997.
44. Sidiropoulos, S. et al., "A semi-digital DLL with unlimited phase shift capability and 0.08–400 MHz operating range," in *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*, pp. 332–333.
45. Song, B.S. et al., "NRZ timing recovery technique for band limited channels," in *1996 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 194–195.
46. Takahashi, T. et al., "A CMOS gate array with 600 Mb/s simultaneous bidirectional I/O circuits," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 12, Dec. 1995.
47. Tamura, H. et al., "Partial response detection technique for driver power reductio in high speed memory-to-processor communications," in *1997 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 241–248.
48. Van de Plassche, R., *Integrated Analog-to-Digital and Digital-to-Analog Converters*, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1994.
49. Wei, G. et al., "A variable frequency parallel I/O interface with adaptive supply regulation," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1600–1610, Nov. 2000.
50. Widmer, A.X. et al., "Single-chip 4 * 500-MBd CMOS transceiver," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 12, pp. 2004–2014, Dec. 1996.
51. Wu, J.-T. et al., "A 100-MHz pipelined CMOS comparator," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 6, pp. 1379–1385.
52. Yang, C.K.K. et al., "A 0.5-$\mu$m CMOS 4-Gbps serial link transceiver with data recovery using oversampling," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 5, pp. 713–722, May 1998.

# 16

# Algorithms and Data Structures in External Memory




Jeffrey Scott Vitter
*Purdue University*


## 16.1 Introduction

In large applications, data sets are often too massive to fit completely inside the computer's internal memory. The resulting input/output (I/O) communication between fast internal memory and slower external memory (such as disks) can be a major performance bottleneck. For example, loading a register

takes on the order of a nanosecond ($10^{-9}$ s), and accessing internal memory takes tens of nanoseconds, but the latency of accessing data from a disk is several milliseconds ($10^{-3}$ s), which is about one million times slower.

Many computer programs exhibit some degree of locality in their pattern of memory references: Certain data are referenced repeatedly for a while, and then the program shifts attention to other sets of data. Substantial gains in performance may be possible by incorporating locality directly into the algorithm design and by explicit management of the contents of each level of the memory hierarchy, thereby bypassing the virtual memory system.

### 16.1.1 Overview of the Chapter

This chapter discusses the I/O communication between the random access internal memory and the magnetic disk external memory, where the relative difference in access speeds is most apparent. It surveys several paradigms for how to exploit locality and thereby reduce I/O costs when solving problems in external memory. The problems that are considered fall into two general categories:

1. Batched problems: No preprocessing is done and the entire file of data items must be processed, often by streaming the data through the internal memory in one or more passes.
2. Online problems: Computation is done in response to a continuous series of query operations and updates.

The approach is based upon the parallel disk model (PDM), which provides an elegant model for analyzing the relative performance of external memory (EM) algorithms and data structures. The three main performance measures of PDM are the number of I/O operations, the disk space usage, and the CPU time. For reasons of brevity, we focus on the first two measures. Most of the algorithms we consider are also efficient in terms of CPU time. In Section 16.4, we list four fundamental I/O bounds that pertain to most of the problems considered in this chapter. In Section 16.5, we discuss an automatic load balancing technique called disk striping for using multiple disks in parallel.

Section 16.6 examines canonical batched EM problem of external sorting and the related problems of permuting and fast Fourier transform (FFT). In Section 16.7, we discuss grid and linear algebra batched computations.

For most problems, parallel disks can be utilized effectively by means of disk striping or the parallel disk techniques of Section 16.6, and hence we restrict ourselves starting in Section 16.8 to the conceptually simpler single-disk case. In Section 16.8, we mention several effective paradigms for batched EM problems in computational geometry. In Section 16.9, we look at EM algorithms for combinatorial problems on graphs, such as list ranking, connected components, topological sorting, and finding shortest paths.

In Sections 16.10 through 16.12 we consider data structures based on hash tables and search trees in the online setting. We discuss some additional EM approaches useful for dynamic data structures, and we also consider kinetic data structures, in which the data items are moving. Section 16.13 deals with EM data structures for manipulating and searching text strings. In Section 16.14, we list several programming environments and tools that facilitate high-level development of efficient EM algorithms. In Section 16.15, we discuss EM algorithms that adapt optimally to dynamically changing internal memory allocations.

## 16.2 Parallel Disk Models

EM algorithms explicitly control data placement and movement, and thus it is important for algorithm designers to have a simple but reasonably accurate model of the memory system's characteristics. Magnetic disks consist of one or more rotating platters and one read/write head per platter surface. The data are stored on the platters in concentric circles called tracks. To read or write a data item at a certain address on disk, the read/write head must mechanically seek to the correct track and then wait

for the desired address to pass by. The seek time to move from one random track to another is often on the order of 3–10 milliseconds, and the average rotational latency, which is the time for half a revolution, has the same order of magnitude. In order to amortize this delay, it pays to transfer a large contiguous group of data items, called a block. Similar considerations apply to all levels of the memory hierarchy.

Even if an application can structure its pattern of memory accesses to exploit locality and take full advantage of disk block transfer, there is still a substantial access gap between internal memory performance and external memory performance. In fact, the access gap is growing because the latency and bandwidth of memory chips are improving more quickly than those of disks. Use of parallel processors further widens the gap. Storage systems such as RAID deploy multiple disks in order to get additional bandwidth [52,104].

The main properties of magnetic disks and multiple disk systems can be captured by the commonly used PDM introduced by Vitter and Shriver [200]:

$N$ = problem size (in units of data items)
$M$ = internal memory size (in units of data items)
$B$ = block transfer size (in units of data items)
$D$ = number of independent disk drives
$P$ = number of CPUs

where $M < N$ and $1 \leq DB \leq M/2$. The data items are assumed to be of fixed length. In a single I/O, each of the $D$ disks can simultaneously transfer a block of $B$ contiguous data items. When the problem involves queries, two more performance parameters are needed:

$Q$ = number of input queries (for a batched problem)
$Z$ = query output size (in units of data items)

It is convenient to refer to some of the above PDM parameters in units of disk blocks rather than in units of data items:

$$n = \frac{N}{B}, m = \frac{M}{B}, q = \frac{Q}{B}, z = \frac{Z}{B} \tag{16.1}$$

It is assumed that the input data are initially "striped" across the $D$ disks, in units of blocks, as illustrated in Figure 16.1, and we require the output data to be similarly striped. Striped format allows a file of $N$ data items to be read or written in $O(N/DB) = O(n/D)$ I/Os, which is optimal.

The following are the three primary measures of performance in PDM:

1. The number of I/O operations performed
2. The amount of disk space used
3. The internal (sequential or parallel) computation time

| | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|
| Stripe 0 | 0 1 | 2 3 | 4 5 | 6 7 | 8 9 |
| Stripe 1 | 10 11 | 12 13 | 14 15 | 16 17 | 18 19 |
| Stripe 2 | 20 21 | 22 23 | 24 25 | 26 27 | 28 29 |
| Stripe 3 | 30 31 | 32 33 | 34 35 | 36 37 | 38 39 |

**FIGURE 16.1** Initial data layout on the disks, for $D = 5$ disks and block size $B = 2$. The input data items are initially striped block-by-block across the disks. For example, data items 16 and 17 are stored in the second block (i.e., in stripe 1) of disk $D_3$.

For reasons of brevity, this chapter focuses on only the first two measures. The reader can refer to Ref. [197] for discussion and references on more complex and precise disk models.

## 16.3 Related Memory Models, Hierarchical Memory, and Caches

The study of problem complexity and algorithm analysis when using EM devices began more than 40 years ago with Demuth's Ph.D. thesis on sorting [70,122]. In the early 1970s, Knuth [122] did an extensive study of sorting using magnetic tapes and, to a lesser extent, magnetic disks. At about the same time, Floyd [86,122] considered a disk model akin to PDM for $D = 1$, $P = 1$, $B = M/2 = \Theta(N^c)$, for constant $c > 0$, and developed optimal upper and lower I/O bounds for sorting and matrix transposition. Hong and Kung [107] developed a pebbling model of I/O for straight-line computations, and Savage and Vitter [175] extended the model to deal with block transfer. Aggarwal and Vitter [15] generalized Floyd's I/O model to allow $D$ simultaneous block transfers, but the model was unrealistic in that the $D$ simultaneous transfers were allowed to take place on a single disk. They developed matching upper and lower I/O bounds for all parameter values for a host of problems. Because the PDM model can be thought of as a more restrictive (and more realistic) version of Aggarwal and Vitter's model, their lower bounds apply to PDM as well. Section 16.6.3 discusses a recent simulation technique due to Sanders et al. [174]; the Aggarwal–Vitter model can be simulated probabilistically by PDM with only a constant factor more I/Os, thus making the two models theoretically equivalent in the randomized sense. Deterministic simulations on the other hand require a factor of $\log(N/D)/\log[\log(N/D)]$ more I/Os [28].

Surveys of I/O models, algorithms, and challenges appear in Refs. [179,192,197,216,227,293]. Several versions of PDM have been developed for parallel computation [69,133,183]. Models of active disks augmented with processing capabilities to reduce data traffic to the host, especially during streaming applications, are given in Refs. [3,165]. Models of microelectromechanical systems (MEMS) for mass storage appear in Ref. [99].

Some authors have studied problems that can be solved efficiently by making only one pass (or a small number of passes) over the data [80,105]. In such data streaming applications, one approach to reduce the internal memory requirements is to require only an approximate answer to the problem; the more the memory available the better the approximation. A related approach to reducing I/O costs for a given problem is to use random sampling or data compression in order to construct a smaller version of the problem whose solution approximates the original. These approaches are highly problem dependent and somewhat orthogonal to our focus in this chapter.

The same type of bottleneck that occurs between internal memory (DRAM) and external disk storage can also occur at other levels of the memory hierarchy, such as between registers and level 1 cache, between level 1 and level 2 cache, between level 2 cache and DRAM, and between disk storage and tertiary devices. The PDM model can be generalized to model the hierarchy of memories ranging from registers at the small end to tertiary storage at the large end. Optimal algorithms for PDM often generalize in a recursive fashion to yield optimal algorithms in the hierarchical memory models [12,13,199,201]. Conversely, the algorithms for hierarchical models can be run in the PDM setting.

Frigo et al. [88] introduce the important notion of cache-oblivious algorithms, which require no knowledge of the storage parameters, like $M$ and $B$. They develop optional cache-oblivious algorithms for merge sort and distribution sort. Bender et al. [42] and Bender et al. [300] develop cache-oblivious versions of B-trees. We refer the reader to Ref. [237] for a survey of cache-oblivious algorithms and data structures.

The match between theory and practice is harder to establish for hierarchical models and caches than for disks. Generally, the most significant speedups come from optimizing the I/O communication between internal memory and the disks. For reasons of focus, such hierarchical models and caching issues are not considered in this chapter. The reader is referred to the discussion and references in Ref. [197].

**TABLE 16.1** I/O Bounds for the Four Fundamental Operations

| Operation | I/O Bound, $D = 1$ | I/O Bound, General $D \geq 1$ |
|---|---|---|
| *Scan* ($N$) | $\Theta\left(\frac{N}{B}\right) = \Theta(n)$ | $\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$ |
| *Sort* ($N$) | $\Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) = \Theta(n\log_m n)$ | $\Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right) = \Theta\left(\frac{n}{D}\log_m n\right)$ |
| *Search* ($N$) | $\Theta(\log_B N)$ | $\Theta(\log_{DB} N)$ |
| *Output* ($Z$) | $\Theta\left(\max\left\{1, \frac{Z}{B}\right\}\right) = \Theta(\max\{1, z\})$ | $\Theta\left(\max\left\{1, \frac{Z}{DB}\right\}\right) = \Theta\left(\max\left\{1, \frac{z}{D}\right\}\right)$ |

*Note*: The PDM parameters are defined in Section 16.2.

## 16.4 Fundamental I/O Operations and Bounds

The I/O performance of many algorithms and data structures can be expressed in terms of the bounds for the following four fundamental operations:

1. *Scanning* (also known as *streaming* or *touching*) a file of $N$ data items, which involves the sequential reading or writing of the items in the file
2. *Sorting* a file of $N$ data items, which puts the items into sorted order
3. *Searching* online through $N$ sorted data items
4. *Outputting* the $Z$ answers to a query in a blocked "output-sensitive" fashion

The I/O bounds for these four operations are given in Table 16.1. The special case of a single disk ($D = 1$) is emphasized, because the formulas are simpler and many of the discussions in this chapter are restricted to the single-disk case. The searching lower bounds assume the comparison model of computation.

## 16.5 Disk Striping for Multiple Disks

It is conceptually much simpler to program for the single-disk case ($D = 1$) than for the multiple-disk case ($D \geq 1$). Disk striping [120,167] is a practical paradigm that can ease the programming task with multiple disks: I/Os are permitted only on entire stripes, one stripe at a time. For example, in the data layout in Figure 16.1, data items 20–29 can be accessed in a single I/O step because their blocks are grouped into the same stripe. The net effect of striping is that the $D$ disks behave as a single logical disk, but with a larger logical block size $DB$.

Therefore, the paradigm of disk striping can be applied automatically to convert an algorithm designed to use a single disk with block size $DB$ into an algorithm for use on $D$ disks each with block size $B$: In the single-disk algorithm, each I/O step transmits one block of size $DB$; in the $D$-disk algorithm, each I/O step consists of $D$ simultaneous block transfers of size $B$ each. The number of I/O steps in both algorithms is the same; in each I/O step, the $DB$ items transferred by the two algorithms are identical. Of course, in terms of wall clock time, the I/O step in the multiple-disk algorithm will be $\Theta(D)$ times faster than in the single-disk algorithm due to parallelism.

Disk striping can be used to get optimal multiple-disk algorithms for three of the four fundamental operations of Section 16.4—streaming, online search, and output reporting—but it is nonoptimal for sorting. If $D$ is replaced by 1 and then $B$ by $DB$ in the sorting bound $Sort(N)$ given in Section 16.4, an expression is obtained that is larger than $Sort(N)$ by a multiplicative factor of

$$\frac{\log(n/D)}{\log n}\,\frac{\log m}{\log(m/D)} \approx \frac{\log m}{\log(m/D)} \tag{16.2}$$

When $D$ is on the order of $m$, the $\log(m/D)$ term in the denominator is small, and the resulting value of Equation 16.2 is in the order of $\log m$, which can be significant in practice.

It follows that the only way to theoretically attain the optimal sorting bound $Sort(N)$ is to forsake disk striping and to allow the disks to be controlled independently, so that each disk can access a different

stripe in the same I/O step. In the next section, algorithms for sorting with multiple independent disks are discussed. The techniques that arise can be applied to many of the batched problems addressed later in the chapter. Two such sorting algorithms—distribution sort with randomized cycling and simple randomized merge sort—have relatively low overhead and will outperform disk-striped approaches.

## 16.6 External Sorting and Related Problems

The problem of external sorting (or sorting in external memory) is a central problem in the field of EM algorithms, partly because sorting and sorting-like operations account for a significant percentage of computer use [122], and also because sorting is an important paradigm in the design of efficient EM algorithms, as shown in Section 16.9. With some technical qualifications, many problems that can be solved easily in linear time in internal memory, such as permuting, list ranking, expression tree evaluation, and finding connected components in a sparse graph, require the same number of I/Os in PDM as does sorting.

THEOREM 16.1 [15,155] *The average-case and worst-case number of I/Os required for sorting $N = nB$ data items using $D$ disks is*

$$Sort(N) = \Theta\left(\frac{n}{D}\log_m n\right) \tag{16.3}$$

From Section 16.5, efficient sorting algorithms can be constructed for multiple disks by applying the disk-striping paradigm to an efficient single-disk algorithm. But in the case of sorting, the resulting multiple-disk algorithm does not meet the optimal $Sort(N)$ bound of Theorem 16.1. In Sections 16.6.1 and 16.6.2, we discuss some recently developed external sorting algorithms that use disks independently. The algorithms are based on the important distribution and merge paradigms, which are two generic approaches to sorting.

### 16.6.1 Sorting by Distribution

Distribution sort [122] is a recursive process in which we use a set of $S-1$ partitioning elements to partition the items into $S$ disjoint buckets. All the items in one bucket precede all the items in the next bucket. We complete the sort by recursively sorting the individual buckets and concatenating them together to form a single fully sorted list.

One requirement is that we choose the $S-1$ partitioning elements so that the buckets are of roughly equal size. When that is the case, the bucket sizes decrease from one level of recursion to the next by a relative factor of $\Theta(S)$, and thus there are $O(\log_S n)$ levels of recursion. During each level of recursion, we scan the data. As the items stream through internal memory, they are partitioned into $S$ buckets in an online manner. When a buffer of size $B$ fills for one of the buckets, its block is written to the disks in the next I/O, and another buffer is used to store the next set of incoming items for the bucket. Therefore, the maximum number of buckets (and partitioning elements) is $S = \Theta(M/B) = \Theta(m)$, and the resulting number of levels of recursion is $\Theta(\log_m n)$.

It seems difficult to find $S = \Theta(m)$ partitioning elements using $\Theta(n/D)$ I/Os and guarantee that the bucket sizes are within a constant factor of one another. Efficient deterministic methods exist for choosing $S = \sqrt{m}$ partitioning elements [15,154,200], which has the effect of doubling the number of levels of recursion. Probabilistic methods based upon random sampling can be found in Ref. [82]. A deterministic algorithm for the related problem of (exact) selection (i.e., given $k$, find the $k$th item in the file in sorted order) appears in Ref. [182].

To meet the sorting bound (2), the buckets at each level of recursion must be formed using $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each read step and each write step during the bucket formation must involve on the average $\Theta(D)$ blocks. The file of items being partitioned was itself one of the buckets formed in the previous level of recursion. To read that file efficiently, its blocks must be spread uniformly among the disks, so that no single disk is a

bottleneck. The challenge in distribution sort is to write the blocks of the buckets to the disks in an online manner and achieve a global load balance by the end of the partitioning, so that the bucket can be read efficiently during the next level of the recursion.

Vitter and Shriver [200] develop two complementary randomized online techniques for the partitioning so that with high probability, each bucket will be well balanced across the $D$ disks. Putting the methods together, they got the first provably optimal randomized method for sorting with parallel disks.

DeWitt et al. [71] present a randomized distribution sort algorithm in a similar model to handle the case when sorting can be done in two passes. They use a sampling technique to find the partitioning elements and route the items in each bucket to a particular processor. The buckets are sorted individually in the second pass.

An even better way to do distribution sort, and deterministically at that, is the BalanceSort method developed by Nodine and Vitter [154]. During the partitioning process, the algorithm keeps track of how evenly each bucket has been distributed so far among the disks. It maintains an invariant that guarantees good distribution across the disks for each bucket.

The distribution sort methods that we mentioned above for parallel disks perform write operations in complete stripes, which make it easy to write parity information for use in error correction and recovery. But since the blocks written in each stripe typically belong to multiple buckets, the buckets themselves will not be striped on the disks, and we must use the disks independently during read operations. In the write phase, each bucket must therefore keep track of the last block written to each disk so that the blocks for the bucket can be linked together.

An orthogonal approach is to stripe the contents of each bucket across the disks so that read operations can be done in a striped manner. As a result, the write operations must use disks independently, since during each write, multiple buckets will be writing to multiple stripes. Error correction and recovery can still be handled efficiently by devoting to each bucket one block-sized buffer in internal memory. The buffer is continuously updated to contain the exclusive-or (parity) of the blocks written to the current stripe, and after $D-1$ blocks have been written, the parity information in the buffer can be written to the final ($D$th) block in the stripe.

Under this new scenario, the basic loop of the distribution sort algorithm is, as before, to read one memory load at a time and partition the items into $S$ buckets; however, unlike before, the blocks for each individual bucket will reside on the disks in contiguous stripes. Each block therefore has a predefined place where it must be written. If we choose the normal round-robin ordering for the stripes (namely, $\ldots, 1, 2, 3, \ldots, D, 1, 2, 3, \ldots, D, \ldots$), the blocks of different buckets may "collide," meaning that they need to be written to the same disk, and subsequent blocks in those same buckets will also tend to collide. Vitter and Hutchinson [198] solve this problem by the technique of randomized cycling. For each of the $S$ buckets, they determine the ordering of the disks in the stripe for that bucket via a random permutation of $\{1, 2, \ldots, D\}$. The $S$ random permutations are chosen independently. If two blocks (from different buckets) happen to collide during a write to the same disk, one block is written to the disk and the other is kept on a write queue. With high probability, subsequent blocks in those two buckets will be written to different disks and thus will not collide. As long as there is a small pool of available buffer space to temporarily cache the blocks in the write queues, Vitter and Hutchinson [198] show that with high probability, the writing proceeds optimally.

The randomized cycling method or the related merge sort methods discussed at the end of Section 16.6.2 will be the methods of choice for sorting with parallel disks. Distribution sort algorithms may have an advantage over the merge approaches in that they typically make better use of lower levels of cache in the memory hierarchy of real systems, based on analysis of distribution sort and merge sort algorithms on models of hierarchical memory, such as the RUMH model of Vitter and Nodine [199].

### 16.6.2 Sorting by Merging

The merge paradigm is somewhat orthogonal to the distribution paradigm of the previous section. A typical merge sort algorithm works as follows [122]: In the "run formation" phase, the $n$ blocks of data

are scanned, one memory load at a time; each memory load is sorted into a single "run," which is then output onto a series of stripes on the disks. At the end of the run formation phase, there are $N/M = n/m$ (sorted) runs, each striped across the disks. (In actual implementations, we can use the "replacement-selection" technique to get runs of $2M$ data items, on the average, when $M \gg B$ [122].) After the initial runs are formed, the merging phase begins. In each pass of the merging phase, we merge together groups of $R$ runs. For each merge, we scan the $R$ runs and merge the items in an online manner as they stream through internal memory. Double buffering is used to overlap I/O and computation. At most, $R = \Theta(m)$ runs can be merged at a time, and the resulting number of passes is $O(\log_m n)$.

To achieve the optimal sorting bound (Equation 16.3), each merging pass must be performed in $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each parallel read operation during the merging must on an average bring in the next $\Theta(D)$ blocks needed for the merging. The challenge is to ensure that those blocks reside on different disks so that they can be read in a single I/O (or a small constant number of I/Os). The difficulty lies in the fact that the runs being merged were themselves formed during the previous merge pass. Their blocks were written to the disks in the previous pass without knowledge of how they would interact with other runs in later merges.

For the binary merging case, $R = 2$ can be devised a perfect solution, in which the next $D$ blocks needed for the merge are guaranteed to be on distinct disks, based on the Gilbreath principle [91,122]. The first run is striped into ascending order by disk number, and the other run is striped into descending order. Regardless of how the items in the two runs interleave during the merge, it is always the case that we can access the next $D$ blocks needed for the output via a single I/O operation, and thus the amount of internal memory buffer space needed for binary merging is minimized. Unfortunately, there is no analog to the Gilbreath principle for $R > 2$, and as we have seen above, we need the value of $R$ to be large in order to get an optimal sorting algorithm.

The Greed Sort method of Nodine and Vitter [155] was the first optimal deterministic EM algorithm for sorting with multiple disks. Each merge is done approximately so that items go relatively closely to their final destinations. A final application of Columnsort [131], using $O(n)$ extra I/Os, completes the merge.

Aggarwal and Plaxton [14] developed an optimal deterministic merge sort based on the Sharesort hypercube parallel sorting algorithm [66]. To guarantee even distribution during the merging, it employs two high-level merging schemes in which the scheduling is almost oblivious. Similar to Greed Sort, the Sharesort algorithm is theoretically optimal (i.e., within a constant factor of optimal), but the constant factor is larger than the distribution sort methods.

One of the most practical methods for sorting is based on the simple randomized merge sort (SRM) algorithm of Barve et al. [33,35], referred to as randomized striping by Knuth [122]. Each run is striped across the disks, but with a random starting point (the only place in the algorithm where randomness is utilized). During the merging process, the next block needed from each disk is read into memory, and if there is not enough room, the least needed blocks are "flushed" (without any I/Os required) to free up space. Barve et al. [33] derive an asymptotic upper bound on the expected I/O performance, with no assumptions on the input distribution. A more precise analysis, which is related to the so-called cyclic occupancy problem, is an interesting open problem. The expected performance of SRM is not optimal for some parameter values, but it significantly outperforms the use of disk striping for reasonable values of the parameters, as shown in Table 16.2. Experimental confirmation of the speedup was obtained on a 500 MHz CPU with six fast disk drives, as reported by Barve and Vitter [35].

Further improvements can be obtained in merge sort by a more careful prefetching schedule for the runs. Barve et al. [34], Kallahalla and Varman [112,113], Shah et al. [297], and Hon et al. [285] have developed competitive and optimal methods for prefetching blocks in parallel I/O systems. Hutchinson et al. [110] have demonstrated a powerful duality between parallel writing and parallel prefetching, which gives an easy way to compute optimal prefetching and caching schedules for multiple disks. More significantly, they show that the same duality exists between distribution and merging, which they exploit to get a provably optimal and very practical parallel disk merge sort. Rather than using random starting points and round-robin stripes as in SRM, Hutchinson et al. [110] order the stripes for each run independently, based upon the randomized cycling strategy discussed in Section 16.6.1 for distribution sort.

**TABLE 16.2** Ratio of the Number of I/Os Used by Simple Randomized Merge Sort (SRM) to the Number of I/Os Used by Merge Sort with Disk Striping, during a Merge of $kD$ Runs, Where $kD \approx M/2B$

| | $D=5$ | $D=10$ | $D=50$ |
|---|---|---|---|
| $k=5$ | 0.56 | 0.47 | 0.37 |
| $k=10$ | 0.61 | 0.52 | 0.40 |
| $k=50$ | 0.71 | 0.63 | 0.51 |

*Note*: The figures were obtained by simulation.

### 16.6.3 A General Simulation

Sanders et al. [174] and Sanders [173] give an elegant randomized technique to simulate the Aggarwal–Vitter model of Section 16.3, in which $D$ simultaneous block transfers are allowed regardless of where the blocks are located on the disks. On the average, the simulation realizes each I/O in the Aggarwal–Vitter model by only a constant number of I/Os in PDM. One property of the technique is that the read and write steps use the disks independently. Armen [28] had earlier shown that deterministic simulations resulted in an increase in the number of I/Os by a multiplicative factor of $\log(N/D)/\text{loglog}(N/D)$.

### 16.6.4 Handling Duplicates: Bundle Sorting

Arge et al. [22] describe a single-disk merge sort algorithm for the problem of duplicate removal, in which a total of $K$ distinct items are among the $N$ items. When duplicates get grouped together during a merge, they are replaced by a single item and a count of the occurrences. The algorithm runs in $O(n \max\{1, \log_m(K/B)\})$ I/Os, which is optimal in the comparison model. The algorithm can be used to sort the file, assuming that a group of equal items can be represented by a single item and a count.

A harder instance of sorting called bundle sorting arises when $K$ distinct key values are among the $N$ items, but all the items have different secondary information. Abello et al. [2] and Matias et al. [143] develop optimal distribution sort algorithms for bundle sorting using

$$BundleSort(N, K) = \mathrm{O}(n \ \max\{1, \log_m \min\{K, n\}\})$$

I/Os, and Matias et al. [143] prove the matching lower bound. Matias et al. [143] also show how to do bundle sorting (and sorting in general) in place (i.e., without extra disk space). In distribution sort, for example, the blocks for the subfiles can be allocated from the blocks freed up from the file being partitioned; the disadvantage is that the blocks in the individual subfiles are no longer consecutive on the disk. The algorithms can be adapted to run on $D$ disks with a speedup of $O(D)$, using the techniques described in Sections 16.6.1 and 16.6.2.

### 16.6.5 Permuting and Transposition

Permuting is the special case of sorting in which the key values of the $N$ data items form a permutation of $\{1, 2, \ldots, N\}$.

THEOREM 16.2 [15] *The average-case and worst-case number of I/Os required for permuting N data items using D disks is*

$$\Theta\left(\min\left\{\frac{N}{D}, Sort(N)\right\}\right) \tag{16.4}$$

The I/O bound (Equation 16.4) for permuting can be realized by one of the sorting algorithms from Section 16.6 except in the extreme case of $B \log m = o(\log n)$, where it is faster to move the data items

one by one in a nonblocked way. The one-by-one method is trivial if $D = 1$, but with multiple disks there may be bottlenecks on individual disks; one solution for doing the permuting in $O(N/D)$I/Os is to apply the randomized balancing strategies given in Ref. [200].

Matrix transposition is the special case of permuting in which the permutation can be represented as a transposition of a matrix from row-major order into column-major order.

THEOREM 16.3 [15] *With D disks, the number of I/Os required to transpose a $p \times q$ matrix from row-major order to column-major order is*

$$\Theta\left(\frac{n}{D}\log_m \min\{M, p, q, n\}\right) \tag{16.5}$$

*where*
$N = pq$
$n = N/B$

When $B$ is relatively large (for instance, $1/2\ M$) and $N$ is $O(M^2)$, matrix transposition can be as hard as general sorting; but for smaller $B$, the special structure of the transposition permutation makes transposition easier. In particular, the matrix can be broken up into square submatrices of $B^2$ elements such that each submatrix contains $B$ blocks of the matrix in row-major order and also $B$ blocks of the matrix in column-major order. Thus, if $B^2 < M$, the transpositions can be done in a simple one-pass operation by transposing the submatrices one-at-a-time in internal memory.

Matrix transposition is a special case of a more general class of permutations called bit-permute/complement (BPC) permutations, which in turn is a subset of the class of bit-matrix-multiply/complement (BMMC) permutations. BMMC permutations are defined by a $\log N \times \log N$ nonsingular 0–1 matrix $A$ and a ($\log N$)-length 0–1 vector $c$. An item with binary address $x$ is mapped by the permutation to the binary address given by $Ax \oplus c$, where $\oplus$ denotes bitwise XOR. BPC permutations are the special case of BMMC permutations in which $A$ is a permutation matrix, that is, each row and each column of $A$ contain a single 1. BPC permutations include matrix transposition, bit-reversal permutations (which arise in the FFT), vector-reversal permutations, hypercube permutations, and matrix reblocking. Cormen et al. [61] characterize the optimal number of I/Os needed to perform any given BMMC permutation solely as a function of the associated matrix $A$, and they give an optimal algorithm for implementing it.

THEOREM 16.4 [61] *With D disks, the number of I/Os required to perform the BMMC permutation defined by matrix A and vector $\boldsymbol{c}$ is*

$$\Theta\left(\frac{n}{D}\left(1 + \frac{\text{rank}(\gamma)}{\log m}\right)\right) \tag{16.6}$$

*where $\gamma$ is the lower-left $\log n \times \log B$ submatrix of A.*

An interesting theoretical question is to determine the I/O cost for each individual permutation, as a function of some simple characterization of the permutation, like number of inversions.

### 16.6.6 Fast Fourier Transform and Permutation Networks

Computing the FFT in external memory consists of a series of I/Os that permit each computation implied by the FFT directed graph (or butterfly) to be done while its arguments are in internal memory. A permutation network computation consists of an oblivious (fixed) pattern of I/Os such that any of the $N!$ possible permutations can be realized; data items can only be reordered when they are in internal memory. A permutation network can be realized by a series of three FFTs [211].

THEOREM 16.5 *With D disks, the number of I/Os required for computing the N-input FFT digraph or an N-input permutation network is Sort(N).*

Cormen and Nicol [60] give some practical implementations for one-dimensional (1-D) FFTs based on the optimal PDM algorithm given in Ref. [200]. The algorithms for FFT are faster and simpler than

for sorting because the computation is nonadaptive in nature, and thus the communication pattern is fixed in advance.

### 16.6.7 Lower Bounds on I/O

The most trivial batched problem is that of scanning (also known as streaming or touching) a file of $N$ data items, which can be done in a linear number $O(N/DB) = O(n/D)$ of I/Os. Permuting is one of several simple problems that can be done in linear CPU time in the (internal memory) RAM model, but require a nonlinear number of I/Os in PDM because of the locality constraints imposed by the block parameter $B$.

The proof of the permutation lower bound (Equation 16.4) of Theorem 16.2 is due to Aggarwal and Vitter [15]. A stronger lower bound is obtained from a more refined argument that counts input operations separately from output operations [110]. For the typical case in which $B \log m = \omega(\log N)$, the I/O lower bound, up to lower order terms, is $2n \log_m n$. For the pathological, in which $B \log m = o(\log N)$, the I/O lower bound, up to lower order terms, is $N/D$. Permuting is a special case of sorting, and hence, the permuting lower bound applies also to sorting. In the unlikely case that $B \log m = o(\log n)$, the permuting bound is only $\Omega(N/D)$, and the comparison model must be used to get the full lower bound (Equation 16.3) of Theorem 16.1 [15]. The reader is referred to Ref. [197] for further discussion and references on lower bounds for sorting and related problems.

## 16.7 Matrix and Grid Computations

Dense matrices are generally represented in memory in row-major or column-major order. Matrix transposition, which is the special case of sorting that involves conversion of a matrix from one representation to the other, was discussed in Section 16.6.5. For certain operations such as matrix addition, both representations work well; however, for standard matrix multiplication (using only semiring operations) and LU decomposition, a better representation is to block the matrix into square $\sqrt{B} \times \sqrt{B}$ submatrices, which gives the upper bound of the following theorem:

THEOREM 16.6 [107,175,200,210] *The number of I/Os required for standard matrix multiplication of two $K \times K$ matrices or to compute the LU factorization of a $K \times K$ matrix is $\Theta(K^3/\min\{K, \sqrt{M}\}DB)$.*

Hong and Kung [107] and Nodine et al. [153] give optimal EM algorithms for iterative grid computations, and Leiserson et al. [132] reduce the number of I/Os of naive multigrid implementations by a $\Theta(M^{1/5})$ factor. Gupta et al. [101] show how to derive efficient EM algorithms automatically for computations expressed in tensor form.

If a $K \times K$ matrix $A$ is sparse, that is, if the number $N_z$ of nonzero elements in $A$ is much smaller than $K^2$, then it may be more efficient to store only the nonzero elements. Each nonzero element $A_{i,j}$ is represented by the triple $(i, j, A_{i,j})$. Unlike the dense case, in which transposition can be easier than sorting (e.g., see Theorem 16.3 when $B^2 \leq M$), transposition of sparse matrices is as hard as sorting.

THEOREM 16.7 *For a matrix stored in sparse format and containing $N_z$ nonzero elements, the number of I/Os required to convert the matrix from row-major order to column-major order, and vice-versa, is $\Theta(Sort(N_z))$.*

The lower bound follows by reduction from sorting. If the $i$th item in the input of the sorting instance has key value $x \neq 0$, there is a nonzero element in matrix position $(i, x)$.

For further discussion of numerical EM algorithms, the reader is referred to the surveys by Toledo [188] and Kowarschik and Weiß [309]. Some issues regarding programming environments are covered in Ref. [58] and in Section 16.14.

## 16.8 Batched Problems in Computational Geometry

Problems involving massive amounts of geometric data are ubiquitous in spatial databases [129,170, 171], geographic information systems (GIS) [129,170,192], constraint logic programming [117,118], object-oriented databases [213], statistics, virtual reality systems, and computer graphics [89]. NASA's Earth Observing System project, the core part of the Earth Science Enterprise (formerly Mission to

Planet Earth), produces petabytes ($10^{15}$ bytes) of raster data per year [75]. The Microsoft TerraServer [186] and Google Earth [304] online databases of satellite images are multiple terabytes in size. Wal-Mart's sales data warehouse contains over a half petabyte of data. A major challenge is to develop mechanisms for processing the data, or else much of the data will be useless.*

For systems of this size to be efficient, fast EM algorithms and data structures are needed for basic problems in computational geometry. Luckily, many problems on geometric objects can be reduced to a small core of problems, such as computing intersections, convex hulls, or nearest neighbors. Useful paradigms have been developed for solving these problems in external memory.

THEOREM 16.8 *The following batched problems involving $N = nB$ input items, $Q = qB$ queries, and $Z = zB$ output items can be solved using*

$$O((n+q)\log_m n + z) \tag{16.7}$$

I/Os with a single disk:

1. Computing the pairwise intersections of $N$ segments in the plane and their trapezoidal decomposition
2. Finding all intersections between $N$ nonintersecting red line segments and $N$ nonintersecting blue line segments in the plane
3. Answering $Q$ orthogonal 2-D range queries on $N$ points in the plane (i.e., finding all the points within the $Q$ query rectangles)
4. Constructing the 2-D and 3-D convex hull of $N$ points
5. Voronoi diagram and triangulation of $N$ points in the plane
6. Performing $Q$ point-location queries in a planar subdivision of size $N$
7. Finding all nearest neighbors for a set of $N$ points in the plane
8. Finding the pairwise intersections of $N$ orthogonal rectangles in the plane
9. Computing the measure of the union of $N$ orthogonal rectangles in the plane
10. Computing the visibility of $N$ segments in the plane from a point
11. Performing $Q$ ray-shooting queries in 2-D constructive solid geometry (CSG) models of size $N$: The parameters $Q$ and $Z$ are set to 0 if they are not relevant for the particular problem

Goodrich et al. [96], Zhu [215], Arge et al. [23,26], and Crauser et al. [63,64] develop EM algorithms for those problems using the following EM paradigms for batched problems:

*Distribution sweeping*: A generalization of the distribution paradigm of Section 16.6 for "externalizing" plane sweep algorithms.

*Persistent B-trees*: An offline method for constructing an optimal-space persistent version of the $B$-tree data structure (see Sections 16.11 and 16.11.1), yielding a factor of $B$ improvement over the generic persistence techniques of Driscoll et al. [73].

*Batched filtering*: A general method for performing simultaneous EM searches in data structures that can be modeled as planar layered directed acyclic graphs; it is useful for 3-D convex hulls and batched point location. Multisearch on parallel computers is considered in Ref. [72].

*External fractional cascading*: An EM analog to fractional cascading on a segment tree, in which the degree of the segment tree is $O(m^{\alpha})$ for some constant $0 < \alpha \leq 1$. Batched queries can be performed efficiently using batched filtering; online queries can be supported efficiently by adapting the parallel algorithms of work of Tamassia and Vitter [185] to the I/O setting.

---

*For brevity, in the remainder of this chapter, only with the single-disk case $D = 1$ is presented. The single-disk I/O bounds for the batched problems can often be cut by a factor of order $D$ for the case $D \geq 1$ by using the load balancing techniques of Section 16.6. In practice, disk striping (cf., Section 16.5) may be sufficient. For online problems, disk striping will convert optimal bounds for the case $D = 1$ into optimal bounds for $D \geq 1$.

*External marriage-before-conquest*: An EM analog to the technique of Kirkpatrick and Seidel [121] for performing output-sensitive convex hull constructions.

*Batched incremental construction*: A localized version of the randomized incremental construction paradigm of Clarkson and Shor [55], in which the updates to a simple dynamic data structure are done in a random order, with the goal of fast overall performance on the average. The data structure itself may have bad worst-case performance, but the randomization of the update order makes worst-case behavior unlikely. The key for the EM version so as to gain the factor of $B$ I/O speedup is to batch together the incremental modifications.

Other batched geometric problems studied in the PDM model include range counting queries [287], constrained Delauney triangulation [222], and a host of problems on terrains and grid-based GIS models [7,217,223,230,231,306]. Breimann and Vahrenhold [254] survey several EM problems in computational geometry.

## 16.9 Batched Problems on Graphs

We adopt the convention that the edges of the input graph, each of the form $(u, v)$ for some vertices $u$ and $v$, are given in arbitrary order in list form. We denote the number of vertices by $V$ and the number of edges by $E$. We also adopt the lower case notation $v = V/B$ and $e = E/B$ to denote the number of blocks of vertices and edges. We can convert the input graph to adjacency list format via a sort operation in $Sort(E)$ I/Os. Table 16.3 gives the best known I/O bounds (with appropriate corrections made for errors in the literature) for several graph problems. For simplicity of notation, we assume that $E \geq V$. The best known I/O lower bound for these problems is $\Omega((E/V)Sort(V)) = \Omega(e \log_m v)$.

The first work on EM graph algorithms was by Ullman and Yannakakis [190] for the problem of transitive closure. Chiang et al. [53] consider a variety of graph problems, several of which have upper

**TABLE 16.3** Best Known I/O Bounds for Batched Graph Problems for the Single-Disk Case $D = 1$

| Graph Problem | I/O Bound, $D = 1$ |
|---|---|
| List ranking, Euler tour of a tree, centroid decomposition, expression tree evaluation | $\Theta(Sort(V))$ [53] |
| Connected components, Biconnected components, minimum spanning forest (MSF), bottleneck MSF, ear decomposition | $O(\min\{Sort(V^2), \max\{1, \log\frac{V}{M}\}\frac{E}{V}Sort(V), \max\{1, \log\log\frac{V}{e}\}Sort(E), (\log\log B)\frac{E}{V}Sort(V)\})$ [2,19,53,76,126,149] (deterministic)<br>$\Theta\left(\frac{E}{V}Sort(V)\right)$ [53,76] (randomized) |
| Maximal matching | $O(Sort^*(E))$ [313] (deterministic)<br>$O\left(\frac{E}{V}\text{Sort(V)}\right)$ [53] (randomized) |
| Undirected breadth-first search | $\sqrt{Ve} + Sort(E) + SF(V,E)$ [290] |
| Undirected single-source shortest paths | $O\left(\min\left\{V + e\log V, \sqrt{Ve\log\left(1+\frac{W}{w}\right)} + MSF(V,E), \sqrt{Ve}\log V + MSF(V,E)\right\}\right)$ [126,291,292] |
| Directed and undirected depth-first search, topological sorting, directed breadth-first search, directed single-source shortest paths | $O\left(\min\left\{\frac{ve}{m} + V + \text{Sort(E)}, (V+e)\log V\right\}\right)$ [48,53,126] |
| Transitive closure | $O\left(Vv\sqrt{\frac{e}{m}}\right)$ [53] |
| Undirected all-pairs shortest paths | $O(V\sqrt{Ve} + Ve\log e)$ [267] |
| Diameter and undirected unweighted all-pairs shortest paths | $O(V\,\text{Sort}(E))$ [236,267] |

*Note*: The number of vertices is denoted by $V = v\,B$ and the number of edges by $E = e\,B$. For simplicity in notation, we assume that $E \geq V$. The term $Sort(N)$ is the I/O bound for sorting defined in Section 16.4. The terms $SF(V,E)$ and $MSF(V,E)$ represent the I/O bounds for finding a spanning forest and minimum spanning forest, respectively. We use $w$ and $W$ to denote the minimum and maximum weight in a weighted graph.

and lower I/O bounds related to sorting and permuting. Abello et al. [2] formalize a functional approach to EM graph problems, in which computation proceeds in a series of scan operations over the data; the scanning avoids side effects and thus permits checkpointing to increase reliability. Kumar and Schwabe [126], followed by Buchsbaum et al. [48], develop graph algorithms based upon amortized data structures for binary heaps and tournament trees. Munagala and Ranade [149] give improved graph algorithms for connectivity and undirected breadth-first search (BFS). Their approach is extended by Arge et al. [19] to compute the minimum spanning forest (MSF) and by Mehlhorn and Meyer [290] for undirected BFS. Arge [17] gives efficient algorithms for constructing ordered binary decision diagrams.

Techniques for storing graphs on disks for efficient traversal and shortest path queries are discussed in Refs. [7,95,109,152]. Computing wavelet decompositions and histograms [203,204,206] is an EM graph problem related to transposition that arises in online analytical processing (OLAP). Wang et al. [205] give an I/O-efficient algorithm for constructing classification trees for data mining. Further surveys of EM graph algorithms appear in Refs. [308,310].

A sparsification technique [76] can often be applied to convert I/O bounds of the form $O(Sort(E))$ to the improved form $O((E/V)\ Sort(V))$. For example, the actual I/O bound for MSF derived by Arge et al. [19] is $O(\max\{1, \text{loglog}(V/e)\}\ Sort(E))$. For the MSF problem, we can partition the edges of the graph into $E/V$ sparse subgraphs, each with $V$ edges on the $V$ vertices, and then apply the algorithm given in Ref. [19] to each subproblem to create $E/V$ spanning forests in a total of $O(\max\{1, \text{loglog}(V/v)\}\ (E/V)\ Sort(V)) = O(\text{(loglog } B)\ (E/V)\ Sort(V))$ I/Os. We can then merge the $E/V$ spanning forests, two at a time, in a balanced binary merging procedure by repeatedly applying the algorithm given in Ref. [19]. After the first level of binary merging, the spanning forests collectively have at most $E/2$ edges; after two levels, they have at most $E/4$ edges, and so on in a geometrically decreasing manner. The total I/O cost for the final spanning forest is thus $O((\text{loglog } B)\ (E/V)\ Sort(V))$. The reason why sparsification works is that the spanning forest output by each binary merge is only $O(V)$ in size, yet it preserves the necessary information needed for the next merge step. That is, the MSF of the merge of two graphs $G$ and $G'$ is the MSF of the merge of the MSFs of $G$ and $G'$.

The same sparsification approach can be applied to connectivity, biconnectivity, and maximal matching. For example, to apply sparsification to finding biconnected components, we modify the merging process by first replacing each biconnected component by a cycle that contains the vertices in the biconnected component. The resulting graph has $O(V)$ size and contains the necessary information for computing the biconnected components of the merged graph.

In the case of semi-external graph problems [2], in which the vertices fit in internal memory but not the edges (i.e., $V \leq M < E$), several of the problems in Table 16.3 can be solved optimally in external memory. For example, finding connected components, biconnected components, and MSFs can be done in $O(e)$ I/Os when $V \leq M$.

The I/O complexities of several problems in the general case remain open, including connected components, biconnected components, and MSFs in the deterministic case, as well as breadth-first search, topological sorting, shortest paths, depth-first search, and transitive closure. It may be that the I/O complexity for several of these latter problems is $\Theta((E/V)\ Sort(V)) + V)$. For special cases, such as trees, planar graphs, outerplanar graphs, and graphs of bounded tree width, several of these problems can be solved substantially faster in $O(Sort(E))$ I/Os [53,139,140,245,289,312]. Other algorithms for planar and near-planar graphs appear in Refs. [146,238,241,242,246].

Chiang et al. [53] exploit the key idea that efficient EM algorithms can often be developed by a sequential simulation of a parallel algorithm for the same problem. The intuition is that each step of a parallel algorithm specifies several operations and the data they act upon. If the data arguments for each operation are brought together, which can be done by two applications of sorting, the operations can be performed by a single linear scan through the data. After each simulation step, sorting is again performed in order to reblock the data into the linear order required for the next simulation step. In list ranking, which is used as a subroutine in the solution of several other graph problems, the number of working processors in the parallel algorithm decreases geometrically with time, so the number of I/Os for the entire simulation is proportional to the number of I/Os used in the first phase of the simulation,

which is $Sort(N) = \Theta(n \log_m n)$. The optimality of the EM algorithm given in Ref. [53] for list ranking assumes that $\sqrt{m} \log\ m = \Omega(\log\ n)$, which is usually true in practice. That assumption can be removed by use of the buffer tree data structure [16] (see Sections 16.11 and 16.11.1). A practical, randomized implementation of list ranking appears in Ref. [181]. Dehne et al. [68,69] and Sibeyn and Kaufmann [183] use a related approach and get efficient I/O bounds by simulating coarse-grained parallel algorithms in the BSP parallel model.

## 16.10 External Hashing for Online Dictionary Search

This section focuses on online data structures for supporting the dictionary operations of insert, delete, and lookup. Given a value $x$, the lookup operation returns the items, if any, in the structure with key value $x$. The two main types of EM dictionaries are hashing, which we discuss in this section, and tree-based approaches, which is deferred until Section 16.11. The advantage of hashing is that the expected number of probes per operation is a constant, regardless of the number $N$ of items. The common element of all EM hashing algorithms is a predefined hash function:

$$\text{hash: } \{\text{all possible keys}\} \rightarrow \{0, 1, 2, \ldots, K-1\}$$

that assigns the $N$ items to $K$ address locations in a uniform manner. Hashing algorithms differ from one another in how they resolve the collision that results when there is no room to store an item at its assigned location.

The goals in EM hashing are to achieve an average of $O(Output(Z)) = O(z+1)$ I/Os per lookup, where $Z = zB$ is the number of items output, $O(1)$ I/Os per insert and delete, and linear disk space. Most traditional hashing methods use a statically allocated table and are thus designed to handle only a fixed range of $N$. The challenge is to develop dynamic EM structures that can adapt smoothly to widely varying values of $N$.

EM hashing methods fall into one of two categories: directory methods and directoryless methods. Fagin et al. [78] proposed a directory scheme called extendible hashing, illustrated in Figure 16.2. The directory, for a given $d \geq 0$, consists of a table (array) of $2^d$ pointers. Each item is assigned to the table location corresponding to the $d$ least significant bits of its hash address. The value of $d$, called the global depth, is set to the smallest value for which each table location has at most $B$ items assigned to it. Each table location contains a pointer to a block where its items are stored. Thus, a lookup takes two I/Os: one to access the directory and one to access the block storing the item. If the directory fits in internal memory, only one I/O is needed. Several table locations may have many fewer than $B$ assigned items, and for purposes of minimizing storage utilization, they can share the same disk block for storing their items by using a local depth smaller than the global depth. When new items are inserted and deleted, the blocks can overflow or underflow, and the local depth and global depth are changed accordingly.

The expected number of disk blocks required to store the data items is asymptotically $n/\ln 2 \approx n/0.69$; that is, the blocks tend to be about 69% full [145]. At least $\Omega(n/B)$ blocks are needed to store the directory. Flajolet [85] showed on the average that the directory uses $\Theta(N^{1/B} n/B) = \Theta(N^{1+1/B} B^2)$ blocks, which can be superlinear in $N$ asymptotically; however, for practical values of $N$ and $B$, the $N^{1/B}$ term is a small constant, typically less than 2, and directory size is within a constant factor of optimal.

The resulting directory is equivalent to the leaves of a perfectly balanced tree [122], in which the search path for each item is determined by its hash address, except that hashing allows the leaves of the trie to be accessed directly in a single I/O. Any item can thus be retrieved in a total of two I/Os. If the directory fits in internal memory, only one I/O is needed.

A disadvantage of directory schemes is that two I/Os rather than one I/O are required when the directory is stored in external memory. Litwin [134] and Larson [128] developed a directoryless method called linear hashing that expands the number of data blocks in a controlled regular fashion. In contrast to directory schemes, the blocks in directoryless methods are chosen for splitting in a predefined order.

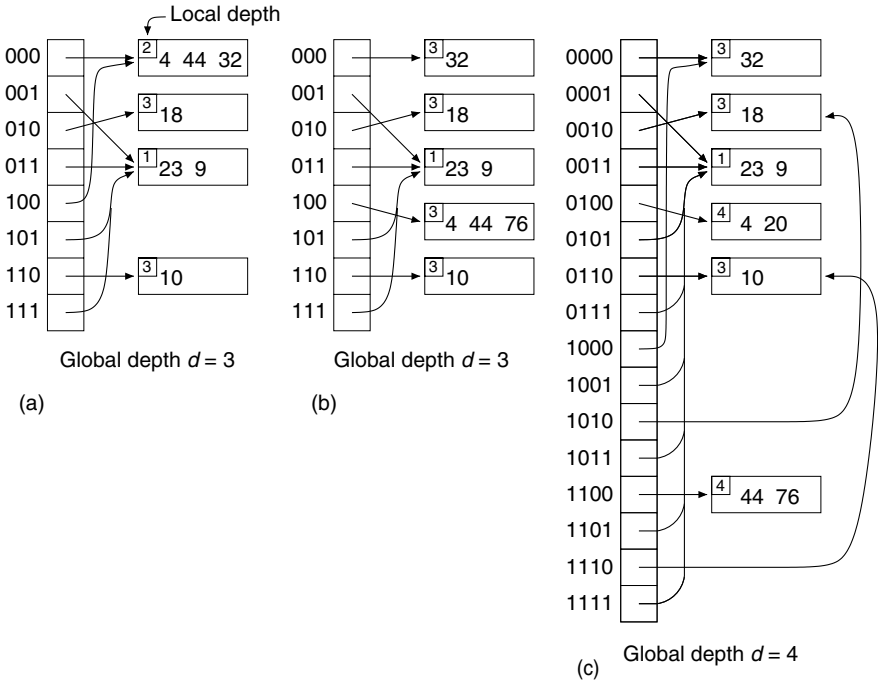



**FIGURE 16.2** Extendible hashing with block size $B = 3$. The keys are indicated in italics. For convenience of exposition, the hash address of a key consists of its binary representation. For example, the hash address of key 4 is "...000100" and the hash address of key 44 is "...0101100." (a) The hash table after insertion of the keys 4, 23, 18, 10, 44, 32, 9. (b) Insertion of the key 76 into table location 100 causes the block with local depth 2 split into two blocks with local depth 3. (c) Insertion of the key 20 into table location 100 causes a block with local depth 3 to split into two blocks with local depth 4. The directory doubles in size and the global depth $d$ is incremented from 3 to 4.

Thus, the block that splits is usually not the block that has overflowed, so some of the blocks may require auxiliary overflow lists to store items assigned to them. On the other hand, directoryless methods have the advantage that there is no need for access to a directory structure, and thus searches often require only one I/O.

## 16.11 Multiway Tree Data Structures

An advantage of search trees over hashing methods is that the data items in a tree are sorted, and thus the tree can be used readily for 1-D range search. The items in a range $[x, y]$ can be found by searching for $x$ in the tree and then performing an inorder traversal in the tree from $x$ to $y$. In this section, we explore some important search-tree data structures in external memory.

### 16.11.1 B-Trees and Variants

Tree-based data structures arise naturally in the online setting, in which the data can be updated and queries must be processed immediately. Binary trees have a host of applications in the (internal memory) RAM model. To exploit block transfer, trees in external memory generally use a block for each node, which can store $\Theta(B)$ pointers and data values.

The well-known balanced multiway *B-tree* due to Bayer and McCreight [38,57,122] is the most widely used nontrivial EM data structure. The degree of each node in the B-tree (with the exception of the root) is required to be $\Theta(B)$, which guarantees that the height of a B-tree storing $N$ items is roughly $\log_B N$. B-trees support dynamic dictionary operations and 1-D range search optimally in linear space,

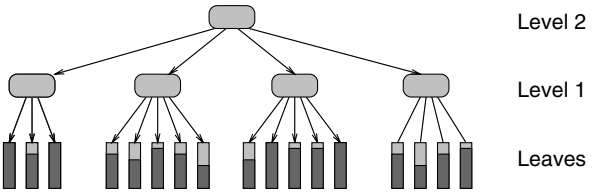



**FIGURE 16.3** B+-tree multiway search tree. Each internal and leaf node corresponds to a disk block. All the items are stored in the leaves; the darker portion of each leaf block indicates its relative fullness. The internal nodes store only key values and pointers, $\Theta(B)$ of them per node. Although not indicated here, the leaf blocks are linked together sequentially.

$O(\log_B N)$ I/Os per insert or delete, and $O(\log_B N + z)$ I/Os per query, where $Z = zB$ is the number of items output. When a node overflows during an insertion, it splits into two half-full nodes, and if the splitting causes the parent node to overflow, the parent node splits, and so on. Splittings can thus propagate up to the root, which is how the tree grows in height. Deletions are handled in a symmetric way by merging nodes. Franceschini et al. [279] show how to achieve the same I/O bounds without space for pointers.

In the *B+-tree* variant, pictured in Figure 16.3, all the items are stored in the leaves, and the leaves are linked together in symmetric order to facilitate range queries and sequential access. The internal nodes store only key values and pointers and thus can have a higher branching factor. In the most popular variant of B+-trees, called *B∗-trees*, splitting can usually be postponed when a node overflows, by sharing the node's data with one of its adjacent siblings. The node needs to be split only if the sibling is also full; when that happens, the node splits into two, and its data and those of its full sibling are evenly redistributed, making each of the three nodes about two-thirds full. This local optimization reduces how often new nodes must be created and thus increases the storage utilization. Because fewer nodes are in the tree, search I/O costs are lower. When no sharing is done (as in B+-trees), Yao [212] shows that nodes are roughly $\ln 2 \approx 69\%$ full on the average, assuming random insertions. With sharing (as in B*-trees), the average storage utilization increases to about $2 \ln(3/2) \approx 81\%$ [31,127]. Storage utilization can be increased further by sharing among several siblings, at the cost of more complicated insertions and deletions. Some helpful space-saving techniques borrowed from hashing are partial expansions [32] and use of overflow nodes [184].

A cross between B-trees and hashing, where each subtree rooted at a certain level of the B-tree is instead organized as an external hash table, was developed by Litwin and Lomet [135] and further studied in Ref. [29,136]. O'Neil [156] proposed a B-tree variant called the SB-tree that clusters together on the disk symmetrically ordered nodes from the same level so as to optimize range queries and sequential access. Rao and Ross [163,164] use similar ideas to exploit locality and optimize search-tree performance in internal memory. Reducing the number of pointers allows a higher branching factor and thus faster search.

Partially persistent versions of B-trees have been developed by Becker et al. [40], Varman and Verma [193], and Arge et al. [234]. By persistent data structure, we mean that searches can be done with respect to any timestamp $y$ [73,74]. In a partially persistent data structure, only the most recent version of the data structure can be updated. In a fully persistent data structure, any update done with timestamp $y$ affects all future queries for any time after $y$. An interesting open problem is whether B-trees can be made fully persistent. Salzberg and Tsotras [169] survey work done on persistent access methods and other techniques for time-evolving data. Lehman and Yao [130], Mohan [147], Lomet and Salzberg [138], and Bender et al. [253] explore mechanisms to add concurrency and recovery to B-trees.

Arge and Vitter [27] introduce a powerful variant of B-trees called weight-balanced B-trees, with the property that the weight of any subtree at level $h$ (i.e., the number of nodes in the subtree rooted at a node of height $h$) is $\Theta(a^h)$, for some fixed parameter $a$ of order $B$. By contrast, the sizes of subtrees at level $h$ in a regular B-tree can differ by a multiplicative factor that is exponential in $h$.

It is sometimes useful to augment B-trees with parent pointers. Order queries, such as Does leaf $x$ precede leaf $y$ in the total order represented by the tree, can be answered using $O(\log_B N)$ I/Os by following parent pointers starting at $x$ and $y$. The update operations insert, delete, cut, and concatenate can be done in $O((1+ (b/B) \log_m n) \log_b N)$ I/Os amortized, for any $2 \leq b \leq B/2$, which is never worse than $O((\log_B N)^2)$ by appropriate choice of $b$.

Agarwal et al. [4] apply level-balanced B-trees in a data structure for point location in monotone subdivisions, which supports queries and (amortized) updates in $O((\log_B N)^2)$ I/Os. They also use it to dynamically maintain planar $s,t$-graphs using $O((1 + (b/B) \log_m n) \log_b N)$ I/Os (amortized) per update, so that reachability queries can be answered in $O(\log_B N)$ I/Os (worst-case). (Planar $s,t$-graphs are planar directed acyclic graphs with a single source and a single sink.) An interesting open question is whether level-balanced B-trees can be implemented in $O(\log_B N)$ I/Os per update. Such an improvement would immediately give an optimal dynamic structure for reachability queries in planar $s,t$-graphs.

Arge [16] developed the elegant buffer tree data structure to support batched dynamic operations, such as in sweep line applications, where the queries do not have to be answered right away or in any particular order. The buffer tree is a balanced multiway tree, but with degree $\Theta(m)$ rather than degree $\Theta(B)$, except possibly for the root. Its key distinguishing feature is that each node has a buffer that can store $\Theta(M)$ items (i.e., $\Theta(m)$ blocks of items). Items in a node are pushed down to the children when the buffer fills. Emptying a full buffer requires $\Theta(m)$ I/Os, which amortizes the cost of distributing the $M$ items to the $\Theta(m)$ children. Each item, thus, incurs an amortized cost of $O(m/M) = O(1/B)$ I/Os per level, and the resulting cost for queries and updates is $O((1/B) \log_m n)$ I/Os amortized.

Buffer trees provide a natural amortized implementation of priority queues for time-forward processing applications like discrete event simulation, sweeping, and list ranking [53]. Govindrajan et al. [98] use time-forward processing to construct a well-separated pair decomposition of $N$ points in $d$ dimensions in $O(Sort(N))$ I/Os, and they apply it to the problems of finding the $K$ nearest neighbors for each point and the $K$ closest pairs. Brodal and Katajainen [47] provide a worst-case optimal priority queue, in the sense that every sequence of $B$ *insert* and *delete_min* operations requires only $O(\log_m n)$ I/Os. Practical implementations of priority queues based on these ideas are examined in Refs. [46,172]. Brodal and Fagerberg [258] examine I/O trade-offs between update and search for comparison-based EM dictionaries. Matching upper bounds for several cases can be achieved with a truncated version of the buffer tree. Further experiments on buffer trees appear in Ref. [108].

## 16.12 Spatial Data Structures and Range Search

A fundamental database primitive in spatial databases and GIS is range search, which includes dictionary lookup as a special case. An orthogonal range query, for a given $d$-dimensional rectangle, returns all the points in the interior of the rectangle. Various forms of 2-D orthogonal range search are pictured in Figure 16.4. Other types of spatial queries include point-location queries, ray-shooting queries,

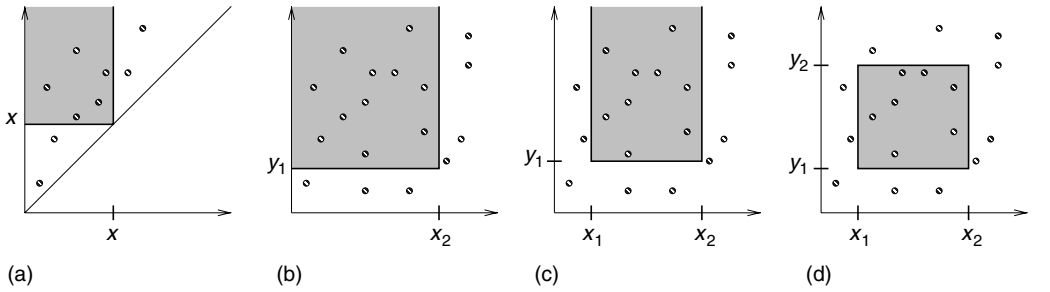



**FIGURE 16.4** Different types of 2-D orthogonal range queries: (a) diagonal corner two-sided 2-D query equivalent to a stabbing query (cf., Section 16.12.3), (b) two-sided 2-D query, (c) three-sided 2-D query, and (d) general four-sided 2-D query.

nearest-neighbor queries, and intersection queries, but for brevity we restrict our attention primarily to range searching.

Two types of spatial data structures are used: data-driven and space-driven. R-trees and kd-trees are data-driven since they are based on a partitioning of the data items themselves, whereas space-driven methods like quad trees and grid files are organized by a partitioning of the embedding space, akin to order-preserving hash functions. In this section, primarily data-driven data structures are discussed. The goal is generally to perform queries in $O(\log_B N + z)$ I/Os, use linear storage space (namely, $O(n)$ disk blocks), and support dynamic updates in $O(\log_B N)$ I/Os.

### 16.12.1 Linear-Space Spatial Structures

Grossi and Italiano [100] construct an elegant multidimensional version of the B-tree called the cross tree. Using linear space, it combines the data-driven partitioning of weight-balanced B-trees at the upper levels of the tree with the space-driven partitioning of methods like quad trees at the lower levels of the tree. Cross trees can be used to construct dynamic EM algorithms for MSF and 2-D priority queues (in which the *delete_min* operation is replaced by $delete\_min_x$ and $delete\_min_y$). For $d > 1$, $d$-dimensional orthogonal range queries can be done in $O(n^{1-1/d} + z)$ I/Os, and inserts and deletes take $O(\log_B N)$ I/Os. The O-tree of Kanth and Singh [119] provides similar bounds. Cross trees also support the dynamic operations of cut and concatenate in $O(n^{1-1/d})$ I/Os. In some restricted models for linear-space data structures, the 2-D range search query performance of cross trees and O-trees can be considered to be optimal, although it is much larger than the logarithmic bound of Criterion 1.

One way to get multidimensional EM data structures is to augment known internal memory structures, such as quad trees and kd-trees, with block-access capabilities. Examples include kd-B-trees [166], buddy trees [178], hB-trees [77,137], and Bkd-trees [294]. Grid files [106,125,150] are a flattened data structure for storing the cells of a 2-D grid in disk blocks. Another technique is to linearize the multidimensional space by imposing a total ordering on it (a so-called space-filling curve), and then the total order is used to organize the points into a B-tree [92,115,158]. Linearization can also be used to represent nonpoint data, in which the data items are partitioned into one or more multidimensional rectangular regions [1,157]. All the methods described in this paragraph use linear space, and they work well in certain situations; however, their worst-case range query performance is no better than that of cross trees, and for some methods, like grid files, queries can require $\Theta(n)$ I/Os, even if there are no points satisfying the query. We refer the reader to Refs. [10,90,151] for a broad survey of these and other interesting methods. Space-filling curves arise again in connection with R-trees, which we describe next.

### 16.12.2 R-Trees

The R-tree of Guttman [103] and its many variants are a practical multidimensional generalization of the B-tree for storing a variety of geometric objects, such as points, segments, polygons, and polyhedra, using linear disk space. Internal nodes have degree $\Theta(B)$ (except possibly the root), and leaves store $\Theta(B)$ items. Each node in the tree has associated with it a bounding box (or bounding polygon) of all the items in its subtree. A big difference between R-trees and B-trees is that in R-trees the bounding boxes of sibling nodes are allowed to overlap. If an R-tree is being used for point location, for example, a point may lie within the bounding box of several children of the current node in the search. In that case, the search must proceed to all such children.

In the dynamic setting, several popular heuristics are used to determine to insert new items into an R-tree and how to rebalance it; see Refs. [10,90,98] for a survey. The R*-tree variant of Beckmann et al. [41] seems to give best overall query performance. New R-tree partitioning methods by de Berg et al. [67], Agarwal et al. [9], and Arge et al [233] provide some provable bounds on overlap and query performance.

In the static setting, in which there are no updates, constructing the R*-tree by repeated insertions, one by one, is extremely slow. A faster alternative to the dynamic R-tree construction algorithms mentioned above is to bulk-load the R-tree in a bottom–up fashion [1,114,157]. The quality of the

bottom–up R-tree in terms of query performance is generally not as good as that of an R*-tree, especially for higher-dimensional data [44,116].

In order to get the best of both worlds—the query performance of R*-trees and the bulk construction efficiency of Hilbert R-trees—Arge et al. [21] and van den Bercken et al. [191] independently devised fast bulk loading methods based on the buffer trees that do top–down construction in $O(n \log_m n)$ I/Os, which matches the performance of the bottom–up methods within a constant factor. The former method is especially efficient and supports dynamic batched updates and queries.

### 16.12.3 Specialized Structures for 2-D Orthogonal Range Search

Diagonal corner two-sided queries (see Figure 16.4a) are equivalent to stabbing queries, which have the following form: Given a set of 1-D intervals, report all the intervals "stabbed" by the query value $x$ (i.e., report all intervals that contain $x$). A diagonal corner query $x$ on a set of 2-D points $\{(a_1, b_2), (a_2, b_2), \ldots\}$ is equivalent to a stabbing query $x$ on the set of closed intervals $\{[a_1, b_2], [a_2, b_2], \ldots\}$. Arge and Vitter [27,197] introduced a new paradigm we call bootstrapping to support such queries in optimal I/O bounds and space: The data structure uses $O(n)$ disk blocks, queries use $O(\log_B N + z)$ I/Os, and updates take $O(\log_B N)$ I/Os. In another example of bootstrapping, Arge et al. [24] achieve the same bounds for three-sided orthogonal 2-D range searching (see Figure 16.4c). The data structures can be applied to yield indexes for a variety of probabilistic queries [265,298]. Range-max and stabbing-max queries are studied in Refs. [220,221].

The dynamic data structure for three-sided range searching can be generalized using the filtering technique of Chazelle [50] to handle general four-sided queries with optimal I/O query bound $O(\log_B N + z)$ and optimal disk space usage $O(n(\log n)/\log(\log_B N + 1))$ [24]. The update bound becomes $O((\log_B N)(\log n)/\log(\log_B N + 1))$, which may not be optimal.

### 16.12.4 Other Types of Range Search

Govindarajan et al. [281] develop data structures for 2-D range-count and range-sum queries. For other types of range searching, such as in higher dimensions and for nonorthogonal queries, different filtering techniques are needed. So far, relatively little work has been done, and many open problems remain.

Vengroff and Vitter [194] develop the first theoretically near-optimal EM data structure for static 3-D orthogonal range searching. They create a hierarchical partitioning in which all the points that dominate a query point are densely contained in a set of blocks. Compression techniques are needed to minimize disk storage. By using $(B \log n)$-approximate boundaries rather than $B$-approximate boundaries [202], $(3 + k)$-sided 3-D range queries, where $k$ of the dimensions $(0 \leq k \leq 3)$ have finite ranges, can be done in $O(\log_B N + z)$ I/Os, which is optimal, and the space usage is $O(n(\log n)^{k+1}/(\log(\log_B N + 1))^k)$. The result also provides optimal $O(\log N + Z)$-time query performance for three-sided 3-D queries in the (internal memory) RAM model, but using $O(N \log N)$ space.

By the reduction in the date structure given in Ref. [51], a data structure for three-sided 3-D queries also applies to 2-D homothetic range search, in which the queries correspond to scaled and translated (but not rotated) transformations of an arbitrary fixed polygon. An interesting special case is "fat" orthogonal 2-D range search, where the query rectangles are required to have bounded aspect ratio. For example, every rectangle with bounded aspect ratio can be covered by a constant number of overlapping squares. An interesting open problem is to develop linear-sized optimal data structures for fat orthogonal 2-D range search. By the reduction, one possible approach would be to develop optimal linear-sized data structures for three-sided 3-D range search.

Agarwal et al. [6] consider half-space range searching, in which a query is specified by a hyperplane and a bit indicating one of its two sides, and the output of the query consists of all the points on that side of the hyperplane. They give various data structures for half-space range searching in two, three, and higher dimensions, including one that works for simplex (polygon) queries in two dimensions, but with a higher query I/O cost. They have subsequently improved the storage bounds for half-space range queries in two dimensions to obtain an optimal static data structure satisfying criteria 1 and 2 of Section 16.12.

The number of I/Os needed to build the data structures for 3-D orthogonal range search and half-space range search is rather large (more than $\Omega(N)$). Still, the structures shed useful light on the complexity of range searching and may open the way to improved solutions. An open problem is to design efficient construction and update algorithms and to improve upon the constant factors.

Callahan et al. [49] develop dynamic EM data structures for several online problems in $d$ dimensions. For any fixed $e > 0$, they can find an approximately nearest neighbor of a query point (within a $1 + e$ factor of optimal) in $O(\log_B N)$ I/Os; insertions and deletions can also be done in $O(\log_B N)$ I/Os. They use a related approach to maintain the closest pair of points; each update costs $O(\log_B N)$ I/Os. Govindrajan et al. [97] achieve the same bounds for closest pair by maintaining a well-separated pair decomposition. For finding nearest neighbors and approximate nearest neighbors, two other approaches are partition trees [5,6] and locality-sensitive hashing [94]. Planar point location is studied in Refs. [234,299], and the dual problem of planar point enclosure is studied in Ref. [241]. Numerous data structures and lower bounds have been developed for range queries and related problems on spatial data. We refer to Refs. [10,90,151,197,227] for a broad survey.

### 16.12.5 Dynamic and Kinetic Data Structures

The preceding sections have outlined cases of data structures in which the data items change dynamically. The bootstrapping paradigm discussed in the two previous subsections is a very useful approach for converting static data structures that are efficient in internal memory into dynamic ones that are efficient for external memory.

In another approach to dynamic data, Arge and Vahrenhold [25] obtain I/O bounds for dynamic point location in general planar subdivisions similar to those given in Ref. [4], but without use of level-balanced trees. Their method uses a weight-balanced base structure at the outer level and a multislab structure for storing segments similar to that of Arge and Vitter [27]. They use an externalization of Bentley's logarithmic method [43,159] to construct a data structure to answer vertical ray-shooting queries in the multislab structures. Agarwal et al. [8] apply the logarithmic method (in both the binary form and B-way variant) to get EM versions of kd-trees, BBD trees, and BAR trees.

In some applications, the data items are moving and their spatial coordinates change in a regular manner. Early work on temporal data generally concentrated on time-series data or multiversion data [169]. A question of growing interest in this mobile age is how to store and index continuously moving items, such as mobile telephones, cars, and airplanes (see Refs. [111,168,209]). Two main approaches are used for storing moving items. The first approach is to use the same sort of data structure as for nonmoving data, but to update it whenever items move sufficiently so far as to trigger important combinatorial events that are relevant to the application at hand [37]. A different approach is to store each item's location and speed trajectory, so that no updating is needed as long as the item's trajectory plan does not change. Such an approach requires fewer updates, but the representation for each item generally has higher dimension, and the search strategies are therefore less efficient.

Kollios et al. [124] developed a linear-space indexing scheme for moving points along a (1-D) line, based upon the notion of partition trees. Their structure supports a variety of range search and approximate nearest neighbor queries. For example, given a range and time, the points in that range at the indicated time can be retrieved in $O(n^{1/2+e} + k)$ I/Os, for arbitrarily small $e > 0$. Updates require $O((\log n)^2)$ I/Os. Agarwal et al. [5] extend the approach to handle range searches in two dimensions, and they improve the update bound to $O((\log_B n)^2)$ I/Os. They also propose an event-driven data structure with the same query times as the range search data structure of Arge et al. [24] discussed earlier, but with the potential need to do many updates. A hybrid data structure combining the two approaches permits a trade-off between query performance and update frequency.

R-trees offer a practical generic mechanism for storing multidimensional points and are thus a natural alternative for storing mobile items. One approach is to represent time as a separate dimension and to cluster trajectories using the R-tree heuristics. However, the orthogonal nature of the R-tree does not lend itself well to diagonal trajectories. For the case of points moving along linear trajectories,

Šaltenis et al. [168] build the R-tree upon only the spatial dimensions, but parameterize the bounding box coordinates to account for the movement of the items stored within. They maintain an outer approximation of the true bounding box, which they periodically update to refine the approximation. Agarwal et al. [11] show how to maintain a provably good approximation of the minimum bounding box with need for only a constant number of refinement events. Agarwal et al. [219] develop persistent data structures where query time degrades the further the time frame of the query is from current time.

## 16.13 String and Text Algorithms

The simplest and most commonly used method to index text in large documents or collections of documents is the inverted file, which is analogous to the index at the back of a book. The words of interest in the text are sorted alphabetically, and each item in the sorted list has a list of pointers to the occurrences of that word in the text. In an EM setting, it makes sense to use a hybrid approach, in which the text is divided into large chunks (consisting of one or more blocks) and an inverted file is used to specify the chunks containing each word; the search within a chunk can be carried out by using a fast sequential method, such as the Knuth–Morris–Pratt [123] or Boyer–Moore [45] method. This particular hybrid method was introduced as the basis of the widely used GLIMPSE search tool [142]. Another way to index text is to use hashing to get small signatures for portions of text. The reader is referred to Refs. [30,87] for more background on the above methods.

In a conventional B-tree, $\Theta(B)$ unit-sized keys are stored in each internal node to guide the searching, and thus the entire node fits into one or two blocks; however, if the keys are variable-sized text strings, the keys can be arbitrarily long, and there may not be enough space to store $\Theta(B)$ strings per node. Pointers to $\Theta(B)$ strings could be stored instead in each node, but access to the strings during search would require more than a constant number of I/Os per node. To save space in each node, Bayer and Unterauer [39] investigated the use of prefix representations of keys. Ferragina and Grossi [82,83] recently developed an elegant generalization of the B-tree called the string B-tree or simply SB-tree (not to be confused with the SB-tree [156] mentioned in Section 16.11). The query time to search in an SB-tree for a string of $k$ characters is $O(\log_B N + k/B)$, which is optimal. Insertions and deletions can be done in the same I/O bound. The space used is linear (optimal). Bender et al. [300] show that the cache-oblivious B-tree data structures can be competitive with those developed with explicit knowledge of the parameters in the PDM model.

Ferragina and Grossi [82,83] apply SB-trees to the problems of string matching, prefix search, and substring search. Ferragina and Luccio [84] apply SB-trees to get new results for dynamic dictionary matching; their structure even provides a simpler approach for the (internal memory) RAM model. Eltabakh et al. [272] use string B-trees and three-sided structures to index strings compressed by run-length encoding. Hon et al. [284] use SB-trees to externalize approximate string indexes.

Ciriani et al. [269] construct a randomized EM data structure that exhibits static optimality, in a similar way as splay trees do in the internal memory model. In particular, they show that $Q$ search queries on a set of $N$ strings $s_1, s_2, \ldots, s_N$ of total length $L$ can be done in $O(L/B + \sum_{1 \le i \le N} n_i \log_B (Q/n_i))$, where $n_i$ is the number of times $s_i$ is queried. Insertion or deletion of a string can be done in the same bounds as given for SB-trees.

Tries and Patricia tries are commonly used as internal memory data structures for storing sets of strings. One particularly interesting application of Patricia tries is to store the set of suffixes of a text string. The resulting data structure, called a suffix tree [144,208], can be built in linear time and supports search for an arbitrary substring of the text in time linear in the size of the substring. A more compact (but static) representation of a suffix tree, called a suffix array [141], consisting of the leaves of the suffix tree in symmetric traversal order, can also be used for fast searching. (See Ref. [101] for general background.) Farach et al. [79] show how to construct SB-trees, suffix trees, and suffix arrays on strings of total length $N$ using $O(n \log_m n)$ I/Os, which is optimal. Clark and Munro [54] give a practical implementation of dynamic suffix trees that use about five bytes per indexed suffix. Crauser and

Ferragina [62] present an extensive set of experiments on various text collections in which they compare the practical performance of some novel and known suffix array construction algorithms. Ferragina et al. [275] give algorithms for 2-D indexing. Ghanem et al. [280] use buffer techniques to index suffix trees and unbalanced search trees. Kärkkäinen and Rao [307] survey several aspects of EM text indexing.

Arge et al. [20] consider several models for the problem of sorting $K$ strings of total length $N$ in external memory. They develop efficient sorting algorithms in these models, making use of the SB-tree, buffer tree techniques, and a simplified version of the SB-tree for merging called the lazy trie.

THEOREM 16.9 [20] *The number of I/Os needed to sort K strings of total length N, where there are $K_1$ short strings of total length $N_1$ and $K_2$ long strings of total length $N_2$ (i.e., $N = N_1 + N_2$ and $K = K_1 + K_2$) is*

$$O\left(\min\left\{\frac{N_1}{B}\log_m\left(\frac{N_1}{B}+1\right), K_1\log_M(K_1+1)\right\} + K_2\log_M(K_2+1) + \frac{N}{B}\right) \tag{16.8}$$

Further work appears in Ref. [303]. Lower bounds for various models of how strings can be manipulated are given in Ref. [20]. There are gaps in some cases between the upper and lower bounds for sorting.

## 16.14 TPIE External Memory Programming Environment

Three basic approaches are used for supporting development of I/O-efficient code, which we call access-oriented, array-oriented, and framework-oriented approaches. TPIE falls primarily into the framework-oriented category with some elements of the access-oriented category. Access-oriented systems preserve the programmer abstraction of explicitly requesting data transfer. They often extend the read–write interface to include data type specifications and collective specification of multiple transfers, sometimes involving the memories of multiple processing nodes. Examples of access-oriented systems include the UNIX file system at the lowest level, higher-level parallel file systems such as Whiptail [180], Vesta [59], PIOUS [148], and the high-performance storage system [207], and I/O libraries MPI-IO [58] and LEDA-SM [65].

Array-oriented systems access data stored in external memory primarily by means of compiler-recognized data types (typically arrays) and operations on those data types. The external computation is directly specified via iterative loops or explicitly data-parallel operations, and the system manages the explicit I/O transfers. Array-oriented systems are effective for scientific computations that make regular strides through arrays of data and can deliver high-performance parallel I/O in applications such as computational fluid dynamics, molecular dynamics, and weapon system design and simulation. Array-oriented systems are generally ill-suited to irregular or combinatorial computations. Examples of array-oriented systems include PASSION [187], Panda [177] (which also has aspects of access orientation), PI/OT [162], and ViC* [56].

TPIE [21,189,195,240] provides a framework-oriented interface for batched computation as well as an access-oriented interface for online computation. Instead of viewing batched computation as an enterprise in which code reads data, operates on it, and writes results, a framework-oriented system views computation as a continuous process during which a program is fed streams of data from an outside source and leaves trails of results behind. TPIE programmers do not need to worry about making explicit calls to I/O routines. Instead, they merely specify the functional details of the desired computation, and TPIE automatically choreographs a sequence of data movements to feed the computation. The reader is referred to Ref. [197] for further discussion of TPIE and some examples of timing experiments in TPIE.

## 16.15 Dynamic Memory Allocation

The amount of internal memory allocated to a program may fluctuate during the course of execution because of demands placed on the system by other users and processes. EM algorithms must be able to

adapt dynamically to whatever resources are available so as to preserve good performance [160]. The algorithms in the previous sections assume a fixed memory allocation; they must resort to virtual memory if the memory allocation is reduced, often causing a severe degradation in performance.

Barve and Vitter [36] discuss the design and analysis of EM algorithms that adapt gracefully to changing memory allocations. In their model, without loss of generality, an algorithm (or program) $\mathcal{P}$ is allocated internal memory in phases: During the $i$th phase, $\mathcal{P}$ is allocated $m_i$ blocks of internal memory, and this memory remains allocated to $\mathcal{P}$ until $\mathcal{P}$ completes $2m_i$ I/O operations, at which point the next phase begins. The process continues until $\mathcal{P}$ finishes execution. We say that $\mathcal{P}$ is dynamically optimal, no other algorithm can perform more than a constant number of sorts in the worst-case for the same sequence of memory allocations.

Barve and Vitter [36] define a precise model and give dynamically optimal strategies for sorting, matrix multiplication, and buffer tree operations. Previous work was done on memory-adaptive algorithms for merge sort [160,214] and hash join [161], but the algorithms handle only special cases and can be made to perform nonoptimally for certain patterns of memory allocation.

## 16.16 Conclusions

In this chapter, several useful paradigms for the design and implementation of efficient external memory (EM) algorithms and data structures were described. The problem domains we have considered include sorting, permuting, FFT, scientific computing, computational geometry, graphs, databases, geographic information systems, and text and string processing. Interesting challenges remain in virtually all these problem domains. One difficult problem is to prove lower bounds for permuting and sorting without an item indivisibility assumption. Another promising area is the design and analysis of EM algorithms for efficient use of multiple disks. Optimal bounds have not yet been determined for several basic EM graph problems like topological sorting, shortest paths, breadth-first and depth-first searches, and connected components. There is an intriguing connection between problems that have good I/O speedups and problems that have fast and work-efficient parallel algorithms. Several problems remain open in the dynamic and kinetic settings, such as range searching, ray shooting, point location, and finding nearest neighbors.

A continuing goal is to develop optimal EM algorithms and to translate theoretical gains into observable improvements in practice. For some of the problems that can be solved optimally up to a constant factor, the constant overhead is too large for the algorithm to be of practical use, and simpler approaches are needed. In practice, algorithms cannot assume a static internal memory allocation; they must adapt in a robust way when the memory allocation changes.

Many interesting challenges and opportunities in algorithm design and analysis arise from new architectures being developed, such as networks of workstations and hierarchical storage devices. Active (or intelligent) disks, in which disk drives have some processing capability and can filter information sent to the host, have recently been proposed to further reduce the I/O bottleneck, especially in large database applications [3,165]. MEMS-based nonvolatile storage has the potential to serve as an intermediate level in the memory hierarchy between DRAM and disks. It could ultimately provide better latency and bandwidth than disks, at less cost per bit than DRAM [176,196].

### Acknowledgments

The author thanks his colleagues, especially Pankaj Agarwal, Lars Arge, Jeff Chase, Wing-Kai Hon, David Hutchinson, Rahul Shah, and Norbert Zeh, for helpful comments and suggestions. This work was supported in part by Army Research Office MURI Grant DAAH04-96-1-0013 and by National Science Foundation Research Grants CCR-9522047, EIA-9870734, CCR-9877133, CCR-0082986, IIS-0415097, and CCF-0621457.

## References


1. D.J. Abel. A $B^+$-tree structure for large quadtrees. *Computer Vision, Graphics, and Image Processing,* 27(1), 19–31, July 1984.
2. J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica,* 32(3), 437–458, 2002.
3. A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices,* 33(11), 81–91, Nov. 1998.
4. P.K. Agarwal, L. Arge, G.S. Brodal, and J.S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 10, 11–20, Baltimore, MD, 1999.
5. P.K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *Journal of Computer and System Sciences,* 66(1), 207–243, 2003.
6. P.K. Agarwal, L. Arge, J. Erickson, P.G. Franciosa, and J.S. Vitter. Efficient searching with linear constraints. *Computational Geometry,* 24(3), 179–195, 2003.
7. P.K. Agarwal, L. Arge, T.M. Murali, K. Varadarajan, and J.S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 9, 117–126, 1998.
8. P.K. Agarwal, L. Arge, O. Procopiuc, and J.S. Vitter. A framework for index bulk loading and dynamization. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, volume 2076 of *Lecture Notes in Computer Science,* 115–127, Springer-Verlag, Berlin, Germany, July 2001.
9. P.K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H.J. Haverkort. Box-trees and R-trees with near-optimal query times. *Discrete and Computational Geometry,* 28(3), 291–312, 2002.
10. P.K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J.E. Goodman, and R. Pollack (Eds.), *Advances in Discrete and Computational Geometry,* volume 23 of *Contemporary Mathematics,* 1–56. American Mathematical Society Press, Providence, RI, 1999.
11. P.K. Agarwal, S. Har-Peled, and K.R. Varadarajan. Approximating extent measures of points. *Journal of the ACM,* 51(4), 606–635, 2004.
12. A. Aggarwal, B. Alpern, A.K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the ACM Symposium on Theory of Computing*, volume 19, 305–314, New York, 1987.
13. A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 28, 204–216, Los Angeles, CA, 1987.
14. A. Aggarwal and C.G. Plaxton. Optimal parallel sorting in multilevel storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 5, 659–668, 1994.
15. A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM,* 31(9), 1116–1127, 1988.
16. L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithima,* 37(1), 1–24, 2003.
17. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 1004 of *Lecture Notes in Computer Science,* 82–91. Springer-Verlag, Berlin, 1995.
18. L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer (Eds.), *Algorithmic Foundations of GIS,* volume 1340 of *Lecture Notes in Computer Science*, 213–254. Springer-Verlag, Berlin, 1997.
19. L. Arge, G.S. Brodal, and L. Toma. On external-memory MST, SSSP and multiway planar graph separation. *Journal of Algorithms*, 53(2), 186–206, 2004.
20. L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proceedings of the ACM Symposium on Theory of Computing,* volume 29, 540–548, 1997.
21. L. Arge, K.H. Hinrichs, J. Vahrenhold, and J.S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1), 104–128, 2002.

22. L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, 83–94. Springer-Verlag, Berlin, 1993.
23. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J.S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 9, 685–694, 1998.
24. L. Arge, V. Samoladas, and J.S. Vitter. Two-dimensional indexability and optimal range search indexing. In *Proceedings of the ACM Conference Principles of Database Systems*, volume 18, 346–357, Philadelphia, PA, May–June 1999.
25. L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry*, 29(2), 147–162, 2004.
26. L. Arge, D.E. Vengroff, and J.S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica,* 47(1), 1–25, Jan. 2007.
27. L. Arge and J.S. Vitter. Optimal external memory. *SIAM Journal on Computing*, 32(6), 1488–1508, 2002.
28. C. Armen. Bounds on the separation of two parallel disk models. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, volume 4, 122–127, Philadelphia, PA, May 1996.
29. R. Baeza-Yates. Bounded disorder: The effect of the index. *Theoretical Computer Science,* 168, 21–38, 1996.
30. R. Baeza-Yates and B. Ribeiro-Neto (Eds.). *Modern Information Retrieval.* Addison Wesley Longman, Reading MA, 1999, Chapter 8.
31. R.A. Baeza-Yates. Expected behaviour of $B^+$-trees under random insertions. *Acta Informatica,* 26(5), 439–472, 1989.
32. R.A. Baeza-Yates and P.-A. Larson. Performance of $B^+$-trees with partial expansions. *IEEE Transactions on Knowledge and Data Engineering,* 1(2), 248–257, June 1989.
33. R.D. Barve, E.F. Grove, and J.S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing,* 23(4), 601–631, 1997.
34. R.D. Barve, M. Kallahalla, P.J. Varman, and J.S. Vitter. Competitive analysis of buffer management algorithms. *Journal of Algorithms,* 36(2), 152–181, 2000.
35. R.D. Barve and J.S. Vitter. A simple and efficient parallel disk mergesort. *Theory of Computing Systems,* 35(2), 189–215, 2002.
36. R.D. Barve and J.S. Vitter. A theoretical framework for memory-adaptive algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 40, 273–284, New York, Oct. 1999.
37. J. Basch, L.J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31, 1–28, 1999.
38. R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Informatica,* 1, 173–189, 1972.
39. R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1), 11–26, March 1977.
40. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4), 264–275, Dec. 1996.
41. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 322–331, 1990.
42. M.A. Bender, E.D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing,* 35(2), 341–358, 2005.
43. J.L. Bentley and J.B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *Journal of Algorithms,* 1(4), 301–358, Dec. 1980.
44. S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proceedings of the International Conference on Extending Database Technology*, volume 5, 216–230, 1998.

45. R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM,* 20(10), 762–772, Oct. 1977.
46. K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues in external memory. *Journal of Experimental Algorithmics,* 5(17), 2000.
47. G.S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, volume 1432 of *Lecture Notes in Computer Science,* 107–118, Stockholm, Sweden, July 1998. Springer-Verlag.
48. A.L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J.R. Westbrook. On external memory graph traversal. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 11, 859–860, 2000.
49. P. Callahan, M.T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science,* 381–392. Springer-Verlag, 1995.
50. B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing,* 15, 703–724, 1986.
51. B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete and Computational Geometry*, 2, 113–126, 1987.
52. P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 145–185, June 1994.
53. Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 6, 139–149, Jan. 1995.
54. D.R. Clark and J.I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 7, 383–391, Atlanta, GA, June 1996.
55. K.L. Clarkson and P.W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry,* 4, 387–421, 1989.
56. A. Colvin and T.H. Cormen. ViC*: A compiler for virtual-memory C*. In *Proceedings of the International Workshop on High-Level Programming Models and Supportive Environments*, volume 3, 23–33, 1998.
57. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys,* 11(2), 121–137, 1979.
58. P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In R. Jain, J. Werth, and J.C. Browne (Eds.), *Input/Output in Parallel and Distributed Computer Systems,* volume 362 of *The Kluwer International Series in Engineering and Computer Science*, Chapter 5, pp. 127–146. Kluwer Academic Publishers, 1996.
59. P.F. Corbett and D.G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems,* 14(3), 225–264, Aug. 1996.
60. T.H. Cormen and D.M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing,* 24(1), 5–20, Jan. 1998.
61. T.H. Cormen, T. Sundquist, and L.F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal on Computing,* 28(1), 105–136, 1999.
62. A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica,* 32(1), 1–35, 2002.
63. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E.A. Ramos. Randomized external-memory algorithms for line segment intersection and other geometric problems. *International Journal of Computational Geometry and Applications*, 11(3), 305–337, 2001.
64. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E.A. Ramos. I/O-optimal computation of segment intersections. In J. Abello and J.S. Vitter (Eds.), *External Memory Algorithms and Visualization, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 131–138. American Mathematical Society Press, Providence, RI, 1999.

65. A. Crauser and K. Mehlhorn. LEDA-SM: Extending LEDA to secondary memory. In J.S. Vitter and C. Zaroliagis (Eds.), *Proceedings of Workshop on Algorithm Engineering Vol. 1668 of Lecture Notes in Computer Science*, 228–242, London, July 1999. Springer-Verlag.
66. R. Cypher and G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Journal of Computer and System Sciences,* 47(3), 501–548, 1993.
67. M. de Berg, J. Gudmundsson, M. Hammar, and M. Overmars. On R-trees with low query complexity. *Computational Geometry,* 24(3), 179–195, 2003.
68. F.K.H.A. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica,* 36(2), 97–122, 2003.
69. F. Dehne, D. Hutchinson, and A. Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proceedings of the International Parallel Processing Symposium*, volume 13, 14–20, April 1999.
70. H.B. Demuth. *Electronic Data Sorting.* Ph.D. Thesis, Stanford University, 1956. A shortened version appears in *IEEE Transactions on Computing,* C-34(4), 296–310, April 1985, special issue on sorting, E.E. Lindstrom, C.K. Wong, and J.S. Vitter (Eds.).
71. D.J. DeWitt, J.F. Naughton, and D.A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, volume 1, 280–291, Dec. 1991.
72. W. Dittrich, D. Hutchinson, and A. Maheshwari. Blocking in parallel multisearch problems. *Theory of Computing Systems,* 34(2), 145–189, 2001.
73. J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences,* 38, 86–124, 1989.
74. M.C. Easton. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development,* 30, 230–241, 1986.
75. NASA's Earth Observing System (EOS) Web page, NASA Goddard Space Flight Center, http://eospso.gsfc.nasa.gov/.
76. D. Eppstein, Z. Galil, G.F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM,* 44(5), 669–696, 1997.
77. G. Evangelidis, D.B. Lomet, and B. Salzberg. The hBΠ-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal,* 6, 1–25, 1997.
78. R. Fagin, J. Nievergelt, N. Pippinger, and H.R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems,* 4(3), 315–344, 1979.
79. M. Farach, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM,* 47(6), 987–1011, 2000.
80. J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams. *SIAM Journal on Computing,* 32(1), 131–151, 2002.
81. W. Feller. *An Introduction to Probability Theory and Its Applications,* volume 1, 3rd ed. John Wiley & Sons, New York, 1968.
82. P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 7, 373–382, Atlanta, GA, June 1996.
83. P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM,* 46(2), 236–280, March 1999.
84. P. Ferragina and F. Luccio. Dynamic dictionary matching in external memory. *Information and Computation,* 146(2), 85–99, Nov. 1998.
85. P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica,* 20(4), 345–369, 1983.
86. R.W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher (Eds.), *Complexity of Computer Computations,* pp. 105–109. Plenum, New York, 1972.
87. W. Frakes and R. Baeza-Yates (Eds.). *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

88. M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 40, 285–298, 1999.
89. T.A. Funkhouser, C.H. Sequin, and S.J. Teller. Management of large amounts of data in interactive building walkthroughs. Symposium on Interactive 3D Graphics, 11–20, 1992.
90. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys,* 30(2), 170–231, June 1998.
91. M. Gardner. *Magic Show,* Chapter 7. Knopf, New York, 1977.
92. I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM,* 25(12), 905–910, Dec. 1982.
93. G.A. Gibson, J.S. Vitter, and J. Wilkes. Report of the working group on storage I/O issues in large-scale computing. *ACM Computing Surveys,* 28(4), 779–793, Dec. 1996.
94. A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Databases*, volume 25, 78–89, Edinburgh, Scotland, 1999, Morgan Kaufmann Publishers.
95. R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proceedings of the International Conference on Very Large Databases*, volume 24, 26–37, Aug. 1998.
96. M.T. Goodrich, J.-J. Tsay, D.E. Vengroff, and J.S. Vitter. External-memory computational geometry. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 34, 714–723, Palo Alto, CA, Nov. 1993.
97. S. Govindarajan, T. Lukovszki, A. Maheshari, and N. Zeh. I/O-efficient well-separated pair decomposition and its applications. *Algorithmica*, 220–231, 2006.
98. D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of IEEE International Conference on Data Engineering*, volume 5, 606–615, 1989.
99. J.L. Griffin, S.W. Schlosser, G.R. Ganger, and D.F. Nagle. Modeling and performance of MEMS-based storage devices. In *Proceedings of ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 56–65, Santa Clara, CA, 2000.
100. R. Grossi and G.F. Italiano. Efficient splitting and merging algorithms for order decomposable problems. *Information and Computation,* 154(1), 1–33, 1999.
101. S.K.S. Gupta, Z. Li, and J.H. Reif. Generating efficient programs for two-level memories from tensor-products. In *Proceedings of the IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, volume 7, 510–513, Washington, DC, Oct. 1995.
102. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, Cambridge, 1997.
103. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 47–57, 1984.
104. L. Hellerstein, G. Gibson, R.M. Karp, R.H. Katz, and D.A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica,* 12(2–3), 182–208, 1994.
105. M.R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In J. Abello and J.S. Vitter (Eds.), *External Memory Algorithms and Visualization, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 107–118. American Mathematical Society Press, Providence, RI, 1999.
106. K.H. Hinrichs. The grid file system: Implementation and case studies of applications. Ph.D. Dissertation, Department of Information Science, ETH, Zürich, 1985.
107. J.W. Hong and H.T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the ACM Symposium on Theory of Computing*, volume 13, 326–333, May 1981.
108. D. Hutchinson, A. Maheshwari, J.-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. In *Proceedings of the Workshop on Algorithm Engineering*, 92–103, 1997.
109. D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics,* 126(1), 55–82, 2003.

110. D.A. Hutchinson, P. Sanders, and J.S. Vitter. Duality between prefetching and queued writing with parallel disks. *SIAM Journal on Computing,* 34(6), 1443–1463, 2005.
111. D. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proceedings of the International Conference on Very Large Databases*, 395–406, Cairo, 2000.
112. M. Kallahalla and P.J. Varman. Optimal read-once parallel disk scheduling. *Algorithmica,* 43(4), 309–343, 2005.
113. M. Kallahalla and P.J. Varman. PC-OPT: Optimal prefetching and caching for parallel I/O systems. *IEEE Transactions on Computers,* 51(11), 1333–1344, 2002.
114. I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the International ACM Conference on Information and Knowledge Management*, volume 2, 490–499, 1993.
115. I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the International Conference on Very Large Databases*, volume 20, 500–509, 1994.
116. I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In *Proceedings of the International Symposium on Spatial Data Handling*, volume 4, 3B, 31–42, 1996.
117. P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences,* 51(1), 26–52, 1995.
118. P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, and J.S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences,* 52(3), 589–612, 1996.
119. K.V.R. Kanth and A.K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proceedings of the International Conference on Database Theory,* volume 1540 of *Lecture Notes in Computer Science*, 257–276. Springer-Verlag, Jan. 1999.
120. M.Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers,* 35(11), 978–988, Nov. 1986.
121. D.G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing,* 15, 287–299, 1986.
122. D.E. Knuth. *Sorting and Searching,* volume 3 of *The Art of Computer Programming,* 2nd ed. Addison-Wesley, Reading, MA, 1998.
123. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing,* 6, 323–350, 1977.
124. G. Kollios, D. Gunopulos, and V.J. Tsotras. On indexing mobile objects. In *Proceedings of the ACM Symposium on Principles of Database Systems*, volume 18, 261–272, 1999.
125. R. Krishnamurthy and K.-Y. Wang. Multilevel grid files. Technical Report, IBM T.J. Watson Center, Yorktown Heights, New York, Nov. 1985.
126. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, volume 8, 169–176, Oct. 1996.
127. K. Küspert. Storage utilization in B*-trees with a generalized overflow technique. *Acta Informatica,* 19, 35–55, 1983.
128. P.-A. Larson. Performance analysis of linear hashing with partial expansions. *ACM Transactions on Database Systems,* 7(4), 566–587, Dec. 1982.
129. R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems.* Academic Press, London, 1992.
130. P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-Trees. *ACM Transactions on Database Systems,* 6(4), 650–570, Dec. 1981.
131. F.T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers,* C-34(4), 344–354, April 1985. Special issue on sorting, E.E. Lindstrom, C.K. Wong, and J.S. Vitter (Eds.).
132. C.E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *Journal of Computer and System Sciences,* 54(2), 332–344, 1997.

133. Z. Li, P.H. Mills, and J.H. Reif. Models and resource metrics for parallel and distributed computation. *Parallel Algorithms and Applications,* 8, 35–59, 1996.
134. W. Litwin. Linear hashing: A new tool for files and tables addressing. In *Proceedings of the International Conference on Very Large Databases*, volume 6, 212–223, Montreal, Quebec, Canada, Oct. 1980.
135. W. Litwin and D. Lomet. A new method for fast data searches with keys. *IEEE Software,* 4(2), 16–24, March 1987.
136. D. Lomet. A simple bounded disorder file organization with good performance. *ACM Transactions on Database Systems,* 13(4), 525–551, 1988.
137. D.B. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems,* 15(4), 625–658, 1990.
138. D.B. Lomet and B. Salzberg. Concurrency and recovery for index trees. *VLDB Journal,* 6(3), 224–240, 1997.
139. A. Maheshwari and N. Zeh. I/O-optimal algorithms for outerplanar graphs. *Journal of Graph Algorithms and Applications,* 8, 47–87, 2004.
140. A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 89–90, 2001.
141. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing,* 22(5), 935–948, Oct. 1993.
142. U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In USENIX Association (Ed.), *Proceedings of the Winter USENIX Conference*, 23–32, San Francisco, CA, Jan. 1994. USENIX.
143. Y. Matias, E. Segal, and J.S. Vitter. Efficient bundle sorting. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 11, 839–848, San Francisco, CA, Jan. 2000.
144. E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM,* 23(2), 262–272, 1976.
145. H. Mendelson. Analysis of extendible hashing. *IEEE Transactions on Software Engineering,* SE–8, 611–619, Nov. 1982.
146. U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 12, 87–88, Washington, DC, Jan. 2001.
147. C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions on B-tree indices. In *Proceedings of the International Conference on Very Large Databases*, 392–405, Brisbane, Australia, Aug. 1990.
148. S.A. Moyer and V. Sunderam. Characterizing concurrency control performance for the PIOUS parallel file system. *Journal of Parallel and Distributed Computing,* 38(1), 81–91, Oct. 1996.
149. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 10, 687–694, Baltimore, MD, Jan. 1999.
150. J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multi-key file structure. *ACM Transactions on Database Systems,* 9, 38–71, 1984.
151. J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer (Eds.), *Algorithmic Foundations of GIS,* volume 1340 of *Lecture Notes in Computer Science*, 153–197. Springer-Verlag, 1997.
152. M.H. Nodine, M.T. Goodrich, and J.S. Vitter. Blocking for external graph searching. *Algorithmica,* 16(2), 181–214, Aug. 1996.
153. M.H. Nodine, D.P. Lopresti, and J.S. Vitter. I/O overhead and parallel VLSI architectures for lattice computations. *IEEE Transactions on Communications,* 40(7), 843–852, July 1991.
154. M.H. Nodine and J.S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 5, 120–129, Velen, Germany, June–July 1993.
155. M.H. Nodine and J.S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM,* 42(4), 919–933, July 1995.

156. P.E. O'Neil. The SB-tree. An index-sequential structure for high-performance sequential access. *Acta Informatica,* 29(3), 241–265, June 1992.
157. J.A. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 294–305, Portland, OR, June 1989.
158. J.A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Proceedings of the ACM Conference Principles of Database Systems*, volume 3, 181–190, 1984.
159. M.H. Overmars. *The Design of Dynamic Data Structures. Vol. 156 of Lecture Notes in Computer Science.* Springer-Verlag, 1983.
160. H. Pang, M. Carey, and M. Livny. Memory-adaptive external sorts. In *Proceedings of the International Conference on Very Large Databases*, volume 19, 618–629, Dublin, 1993.
161. H. Pang, M.J. Carey, and M. Livny. Partially preemptive hash joins. In P. Buneman and S. Jajodia (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 59–68, Washington, DC, May 1993.
162. I. Parsons, R. Unrau, J. Schaeffer, and D. Szafron. PI/OT: Parallel I/O templates. *Parallel Computing,* 23(4), 543–570, June 1997.
163. J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. In M. Atkinson et al. (Eds.), *Proceedings of the International Conference on Very Large Databases*, volume 25, 78–89, Los Altos, CA 1999, Morgan Kaufmann Publishers.
164. J. Rao and K.A. Ross. Making $B^+$-trees cache conscious in main memory. In W. Chen, J. Naughton, and P.A. Bernstein (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 475–486, Dallas, TX, 2000.
165. E. Riedel, G.A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the International Conference on Very Large Databases*, volume 22, 62–73, Aug. 1998.
166. J.T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM Conference Principles of Database Systems*, volume 1, 10–18, 1981.
167. K. Salem and H. Garcia-Molina. Disk striping. In *Proceedings of IEEE International Conference on Data Engineering*, volume 2, 336–242, Los Angeles, CA, 1986.
168. S. Šaltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez. Indexing the positions of continuously moving objects. In W. Chen, J. Naughton, and P.A. Bernstein (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 331–342, Dallas, TX, 2000.
169. B. Salzberg and V.J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys,* 31, 158–221, June 1999.
170. H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1989.
171. H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1989.
172. P. Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithms,* 5(7), 1–25, 2000.
173. P. Sanders. Reconciling simplicity and realism in parallel disk models. *Parallel Computing,* 28(5), 705–723, 2002.
174. P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. *Algorithmica,* 35(1), 21–55, 2003.
175. J.E. Savage and J.S. Vitter. Parallelism in space-time tradeoffs. In F.P. Preparata (Ed.), *Advances in Computing Research,* volume 4, pp. 117–146. JAI Press, Greenwich, CT, 1987.
176. S.W. Schlosser, J.L. Griffin, D.F. Nagle, and G.R. Ganger. Designing computer systems with MEMS-based storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 9, 1–12, Nov. 2000.
177. K.E. Seamons and M. Winslett. Multidimensional array I/O in Panda 1.0. *Journal of Supercomputing,* 10(2), 191–211, 1996.
178. B. Seeger and H.-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proceedings of the International Conference on Very Large Databases*, 590–601, 1990.

179. E.A.M. Shriver and M.H. Nodine. An introduction to parallel I/O models and algorithms. In R. Jain, J. Werth, and J.C. Browne (Eds.), *Input/Output in Parallel and Distributed Computer Systems,* Chapter 2, pp. 31–68. Kluwer Academic Publishers, Norwell, MA, 1996.
180. E.A.M. Shriver and L.F. Wisniewski. An API for choreographing data accesses. Technical Report PCS-TR95—267, Department of Computer Science, Dartmouth College, Nov. 1995.
181. J.F. Sibeyn. From parallel to external list ranking. Technical Report MPI–I–97–1–021, Max-Planck-Institute, Sept. 1997.
182. J.F. Sibeyn. External selection. *Journal of Algorithms,* 58(2), 104–117, 2006.
183. J.F. Sibeyn and M. Kaufmann. BSP-like external-memory computation. In *Proceedings of the Italian Conference on Algorithms and Complexity*, volume 3, 229–240, 1997.
184. B. Srinivasan. An adaptive overflow technique to defer splitting in b-trees. *The Computer Journal,* 34(5), 397–405, 1991.
185. R. Tamassia and J.S. Vitter. Optimal cooperative search in fractional cascaded data structures. *Algorithmica,* 15(2), 154–171, Feb. 1996.
186. Microsoft's TerraServer online database of satellite images, available on the World Wide Web at http://terraserver.microsoft.com/.
187. R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer,* 29(6), 70–78, June 1996.
188. S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In J. Abello and J.S. Vitter (Eds.), *External Memory Algorithms and Visualization, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 161–179. American Mathematical Society Press, Providence, RI, 1999.
189. TPIE User Manual and Reference, 1999. The manual and software distribution are available on the Web at http://www.cs.duke.edu/TPIE/.
190. J.D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence,* 3, 331–360, 1991.
191. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of the International Conference on Very Large Databases*, volume 23, 406–415, 1997.
192. M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer (Eds.). *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
193. P.J. Varman and R.M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering,* 9(3), 391–409, May–June 1997.
194. D.E. Vengroff and J.S. Vitter. Efficient 3-d range searching in external memory. In *Proceedings of the ACM Symposium on Theory of Computing*, volume 28, 192–201, Philadelphia, PA, May 1996.
195. D.E. Vengroff and J.S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of NASA Goddard Conference on Mass Storage Systems*, volume 5, II, 553–570, Sept. 1996.
196. P. Vettiger, M. Despont, U. Drechsler, U. Dürig, W. Häberle, M.I. Lutwyche, E. Rothuizen, R. Stutz, R. Widmer, and G.K. Binnig. The "Millipede"—more than one thousand tips for future AFM data storage. *IBM Journal of Research and Development,* 44(3), 323–340, 2000.
197. J.S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys,* 33(2), 209–271, June 2001. Updated version available via the author's web page http://www.vitter.org/jsv/.
198. J.S. Vitter and D.A. Hutchinson. Distribution sort with randomized cycling. *Journal of the ACM,* 53(7), 2006.
199. J.S. Vitter and M.H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing,* 17, 107–114, 1993.
200. J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica,* 12(2 & 3), 110–147, 1994.
201. J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica,* 12(2 & 3), 148–169, 1994.

202. J.S. Vitter and D.E. Vengroff. Notes, 1999.
203. J.S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 193–204, Philadelphia, PA, June 1999.
204. J.S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the International ACM Conference on Information and Knowledge Management*, volume 7, 96–104, Washington, Nov. 1998.
205. M. Wang, B. Iyer, and J.S. Vitter. Scalable mining for classification rules in relational databases. In *Herman Rubin Festschrift*, *Lecture Notes Monograph Series*, 45, Institute of Mathematical Statistics, Hayward, CA, Fall 2004.
206. M. Wang, J.S. Vitter, L. Lim, and S. Padmanabhan. Wavelet-based cost estimation for spatial queries, *Proceedings of the 7th International Symposium on Spatial and Temporal Databases*, 175–196, July 2001.
207. R.W. Watson and R.A. Coyne. The parallel I/O architecture of the high-performance storage system (HPSS). In *Proceedings of the IEEE Symposium on Mass Storage Systems,* volume 14, 27–44, Sept. 1995.
208. P. Weiner. Linear pattern matching algorithm. In *Proceedings of the IEEE Symposium on Switching and Automata Theory*, volume 14, 1–11, Washington, DC, 1973.
209. O. Wolfson, P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. DOMINO: Databases for MovINg Objects tracking. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 547–549, May 1999.
210. D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the DAGS Symposium on Parallel Computation*, volume 2, 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.
211. C. Wu and T. Feng. The universality of the shuffle-exchange network. *IEEE Transactions on Computers,* C-30, 324–332, May 1981.
212. A.C. Yao. On random 2–3 trees. *Acta Informatica,* 9, 159–170, 1978.
213. S.B. Zdonik and D. Maier (Eds.). *Readings in Object-Oriented Database Systems.* Morgan Kauffman, San Mateo, CA, 1990.
214. W. Zhang and P.-A. Larson. Dynamic memory adjustment for external mergesort. In *Proceedings of the International Conference on Very Large Databases,* volume 23, 376–385, Athens, Greece, 1997.
215. B. Zhu. Further computational geometry in secondary memory. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 834 of *Lecture Notes in Computer Science*, 514–522. Springer-Verlag, 1994.
216. J. Abello, P.M. Pardalos, and M.G. Resende (Eds.). *Handbook of Massive Data Sets,* Kluwer, Norwell, MA, 2002.
217. P.K. Agarwal, L. Arge, and A. Danner. From LIDAR to GRID DEM: A scalable approach. In *Proceedings of the International Symposium on Spatial Data Handling.* 2006.
218. P.K. Agarwal, L. Arge, O. Procopiuc, and J.S. Vitter. A framework for index bulk loading and dynamization. In *Proceedings of the International Colloquium on Automata, Languages and Programming, Vol. 2076 of Lecture Notes in Computer Science*, 115–127. Springer-Verlag, 2001.
219. P.K. Agarwal, L. Arge, and J. Vahrenhold. Time responsive external data structures for moving points. In *Proceedings of the Workshop on Algorithms and Data Structures*, 50–61, 2001.
220. P.K. Agarwal, L. Arge, J. Yang, and K. Yi. I/O-efficient structures for orthogonal range-max and stabbing-max queries. In *Proceedings of the European Symposium on Algorithms*, 7–18, 2003.
221. P.K. Agarwal, L. Arge, and K. Yi. An optimal dynamic interval stabbing-max data structure? In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 803–812, 2005.
222. P.K. Agarwal, L. Arge, and K. Yi. I/O-efficient construction of constrained delaunay triangulations. In *Proceedings of the European Symposium on Algorithms*, 355–366, 2005.
223. P.K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain anaylsis. In *Proceedings of the Symposium on Computational Geometry*, 2006.

224. D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory BFS algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 601–610, 2006.
225. S. Albers and M. Buttner. Integrated prefetching and caching with read and write requests. In *Proceedings of the Workshop on Algorithms and Data Structures*, 162–173, 2003.
226. S. Albers and M. Buttner. Integrated prefetching and caching in single and parallel disk systems. *Information and Computation* 198(1), 24–39, 2005.
227. L. Arge. External memory data structures. Chapter 9 in J. Abello, P.M. Pardalos, and M.G. Resende (Eds.), *Handbook of Massive Data Sets*, pp. 313–358, Kluwer, Norwell, MA, 2002.
228. L. Arge, M.A. Bender, E.D. Demaine, B. Holland-Minkley, and J.I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the ACM Symposium on Theory of Computing*, 268–276, 2002.
229. L. Arge, G.S. Brodal, R. Fagerberg, and M. Lausten. Cache-oblivious planar orthogonal range searching and counting. In *Proceedings of the Symposium on Computational Geometry*, 160–169, 2005.
230. L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J.S. Vitter, and R. Wickremesinghe. Flow computation on massive grid terrains. *GeoInformatica* 7(4), 283–313, 2003.
231. L. Arge, A. Danner, H.J. Haverkort, and N. Zeh. I/O-efficient hierarchical watershed decomposition of grid terrains models. In *Proceedings of the International Symposium on Spatial Data Handling*, 2006.
232. L. Arge, M. de Berg, and H.J. Haverkort. Cache-oblivious R-trees. In *Proceedings of the Symposium on Computational Geometry*, 170–179, 2005.
233. L. Arge, M. de Berg, H.J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 347–358, 2004.
234. L. Arge, A. Danner, and S. He. I/O-efficient point location using persistent B-trees. In *Proceedings of the Workshop on Algorithm Engineering and Experimentation*, 2003.
235. L. Arge, D. Eppstein, and M.T. Goodrich. Skip-Webs: Efficient distributed data structures for multi-dimensional data sets. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 69–76, 2005.
236. L. Arge, U. Meyer, and L. Toma. External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 146–157, 2004.
237. L. Arge, G. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni (Eds.), *Handbook on Data Structures and Algorithms*, CRC Press, Boca Raton, FL, 2005.
238. L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. *Journal of Graph Algorithms and Applications*, 7(2), 105–129, 2003.
239. L. Arge and J. Pagter. I/O-space tradeoffs. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, 448–461, 2000.
240. L. Arge, O. Procopiuc, and J.S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proceedings of the European Symposium on Algorithms*, 88–100, 2002.
241. L Arge, V. Samoladas, and K. Yi. Optimal external memory planar point enclosure. In *Proceedings of the European Symposium on Algorithms*, 40–52, 2004.
242. L Arge and L. Toma. Simplified external memory algorithms for planar DAGs. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, 493–503, 2004.
243. L. Arge and L. Toma. External data structures for shortest path queries on planar digraphs. In *Proceedings of the International Symposium on Algorithms and Computation*, LNCS 3827, 328–338, 2005.
244. L. Arge, L. Toma, and J.S. Vitter, I/O-efficient algorithms for problems on grid-based terrains. *Journal of Experimental Algorithmics*, 6(1), 2001.
245. L. Arge, L. Toma, and N. Zeh. I/O-efficient topological sorting of planar DAGs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 85–93, 2003.

246. L. Arge and N. Zeh. I/O-efficient strong connectivity and depth-first search for directed planar graphs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 261–270, 2003.
247. M.J. Atallah and S. Prabhakar. Almost optimal parallel block access for range queries. *Information Sciences*, 157, 21–31, 2003.
248. M.A. Bender, G.S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. Lopez-Ortiz. The cost of cache-oblivious searching. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 271–282, 2003.
249. M.A. Bender, R. Cole, E.D. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proceedings of the European Symposium on Algorithms*, 139–151, 2002.
250. M.A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 195–207, 2002.
251. M.A. Bender, E.D. Demaine, and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *Proceedings of the European Symposium on Algorithms*, 165–173, 2002.
252. M.A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms,* 53(2), 115–136, 2004.
253. M.A. Bender, J.T. Fineman, S. Gilbert, and B.C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 228–237, 2005.
254. C. Breimann and J. Vahrenhold. External memory computational geometry revisited. Chapter 6 in U. Meyer, P. Sanders, and J. Sibeyn (Eds.), *Algorithms for Memory Hierarchies, Lecture Notes in Computer Science*, pp. 110–148. Springer Verlag, Berlin, Germany, 2002.
255. G.S. Brodal. Cache-oblivious algorithms and data structures. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, 3–13, 2004.
256. G.S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 39–48, 2002.
257. G.S. Brodal and R. Fagerberg. Funnel heap—a cache oblivious priority queue. In *Proceedings of the International Symposium on Algorithms and Computation*, 219–228, 2002.
258. G.S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 546–554, 2003.
259. G.S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the ACM Symposium on Theory of Computing*, 307–315, 2003.
260. G.S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 581–590, 2006.
261. G.S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 39–48, 2002.
262. G.S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proceedings of the Scandinavian Workshop on Algorithmic Theory*, 480–492, 2004.
263. G.S. Brodal, R. Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 576–588, 2005.
264. G. Chaudhry and T.H. Cormen. Oblivious vs distribution-based sorting: An experimental evaluation. In *Proceedings of the European Symposium on Algorithms*, 317–328, 2005.
265. R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J.S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of the 30th International Conference on Very Large Databases*, 876–887, Aug. 2004.
266. R.A. Chowdhury and V. Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 245–254, 2004.

267. R.A. Chowdhury and V. Ramachandran. External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 735–744, 2005.
268. R.A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 591–600, 2006.
269. V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 219–227, 2002.
270. F.K.H.A. Dehne, W. Dittrich, D.A. Hutchinson, and A. Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems* 35(6), 567–597, 2002.
271. R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 138–148, 2003.
272. M. Eltabakh, W.K. Hon, R. Shah, W. Aref, and J.S. Vitter. SVC-tree: Efficient indexing for RLE-compressed strings. Submitted.
273. J. Erickson. Lower bounds for external algebraic decision trees. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 755–761, 2005.
274. A. Farzan, P. Ferragina, G. Franceschini, and J.I. Munro. Cache-oblivious comparison-based algorithms on multisets. In *Proceedings of the European Symposium on Algorithms*, 305–316, 2005.
275. P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Strivastava. Two-dimensional substring indexing. *Journal of Computer and System Sciences*, 66(4), 763–774, 2003.
276. G. Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 291–299, 2004.
277. G. Franceschini and R. Grossi. Optimal cache-oblivious implicit dictionaries. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 316–331, 2003.
278. G. Franceschini and R. Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proceedings of the Workshop on Algorithm Engineering,* 114–126, 2003.
279. G. Franceschini, R. Grossi, J.I. Munro, and L. Pagli. Implicit B-trees: A new data structure for the dictionary problem. *Journal of Computer and System Sciences,* 68(4), 788–807, 2004.
280. T.M. Ghanem, R. Shah, M.F. Mokbel, W.G. Aref, and J.S. Vitter. Bulk operations for space-partitioning trees. In *Proceedings of the 20th Annual IEEE International Conference on Data Engineering*, 29–41, March–April 2004. (this Indexes suffix tree as well as more general unbalanced search trees)
281. S. Govindarajan, P.K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *Proceedings of the International Conference on Fuzzy Systems*, 143–157, 2003.
282. M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 1076–1088, 2005.
283. H.J. Haverkort and L. Toma. I/O-efficient algorithms on near-planar graphs. In *Proceedings of the LATIN*, 580–591, 2006.
284. W.K. Hon, T.W. Lam, R. Shah, S.L. Tam, and J.S. Vitter. Cache-oblivious index for approximate string matching. In *Proceedings of the Conference on Combinatorial Pattern Matching*, July 2007.
285. W.K. Hon, R. Shah, P.J. Varman, and J.S. Vitter. Tight competitive ratios for parallel disk prefetching and caching. Submitted.
286. H. Jampala and N. Zeh. Cache-oblivious planar shortest paths. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 563–575, 2005.
287. T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient batched range counting and its applications to proximity problems. In *Proceedings of the Foundations of Software Technology and Theoretical Computer Science*, 244–255, 2001.
288. A. Maheshwari, M.H.M. Smid, and N. Zeh. I/O-efficient shortest path queries in geometric spanner. In *Proceedings of the Workshop on Algorithms and Data Structures*, 287–299, 2001.

289. A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 372–381, 2002.
290. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the European Symposium on Algorithms*, 723–735, 2002.
291. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the European Symposium on Algorithms*, 435–445, 2003.
292. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths with unbounded weights. In *Proceedings of the European Symposium on Algorithms,* 2006.
293. U. Meyer, P. Sanders, and J. Sibeyn (Eds.). *Algorithms for Memory Hierarchies,* Springer-Verlag, Berlin, Germany, 2003.
294. O. Procopiuc, P.K. Agarwal, L. Arge, and J.S. Vitter. BKD-tree: A dynamic scalable KD-tree. In *Proceedings of the International Symposium of Spatial and Temporal Databases,* volume 2750 of *Lecture Notes in Computer Science*, 46–65, 2003.
295. N. Rahman and R. Raman. Analysing the cache behaviour of non-uniform distribution sorting algorithms. In *Proceedings of the European Symposium on Algorithms*, 380–391, 2000.
296. P. Sanders. Asynchronous scheduling of redundant disk arrays. *IEEE Transactions on Computers,* 52 (9), 1170–1184, 2003.
297. R. Shah, P.J. Varman, and J.S. Vitter. Online algorithms for prefetching and caching in parallel disks. In *Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms and Architectures*, Barcolona, Spain, 255–264, 2004.
298. R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J.S. Vitter. Efficient join processing over uncertain-valued attributes. In *Proceedings of the 2006 ACM Conference on Information and Knowledge Management,* 876–887, Nov. 2006.
299. J. Vahrenhold and K. Hinrichs. Planar point location for large data sets: To seek or not to seek. *ACM Journal of Experimental Algorithmics*, 7, Aug. 2002.
300. M. Bender, M. Farach-Colton, and S. Muthukrishnan. Cache-oblivious string B-trees. In *Proceedings of the ACM Symposium on Principles of Distributed Systems,* 233–242, 2006.
301. M. Berger, E.R. Hansen, R. Pagh, P. Patrascu, M. Ruzic, and P. Tiedmann. Deterministic load balancing and dictionaries in the parallel disk model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 299–307, 2006.
302. M. Dietzfelberger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proceedings of the International Conference on Automata, Languages, and Programming,* 235–246, 1992.
303. R. Fagerberg, A. Pagh, and R. Pagh. External string searching: Faster and cache oblivious. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science,* 68–79, 2006.
304. Google Earth, earth.google.com.
305. J. Gudmundsson and J. Vahrenhold. I/O-efficiently pruning dense spanners. In *Revised Selected Papers of the Japanese Conference on Discrete and Computational Geometry,* volume 3742 of *Lecture Notes in Computer Science*, 106–116. Springer Verlag, 2005.
306. T. Hazel, L. Toma, R. Wickremesinghe, and J. Vahrenhold. TerraCost: A versatile and scalable approach to computing least-cost-path surfaces for massive grid-based terrains. In *Proceedings of the Symposium on Applied Computing,* 52–57, 2006.
307. J. Kärkkäinen and S.S. Rao, Full-text indexes in external memory. Chapter 7 in U. Meyer, P. Sanders, and J. Sibeyn (Eds.), *Algorithms for Memory Hierarchies,* pp. 149–170, Springer-Verlag, Berlin, Germany, 2003.
308. I. Katriel and U. Meyer. Elementary graph algorithms in external memory. Chapter 4 in U. Meyer, P. Sanders, and J. Sibeyn (Eds.), *Algorithms for Memory Hierarchies,* pp. 62–84, Springer-Verlag, Berlin, Germany, 2003.
309. M. Kowarschik and C. Weiß, An overview of cache optimization techniques and cache-aware numerical algorithms. Chapter 10 in U. Meyer, P. Sanders, and J. Sibeyn (Eds.), *Algorithms for Memory Hierarchies,* pp. 213–232, Springer-Verlag, Berlin, Germany, 2003.

310. A. Maheshwari and N. Zeh. A survey of techniques for designing I/O-efficient algorithms. Chapter 3 in U. Meyer, P. Sanders, and J. Sibeyn (Eds.), *Algorithms for Memory Hierarchies,* pp. 36–61, Springer-Verlag, Berlin, Germany, 2003.
311. R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms,* 51(2), 122–144, 2004.
312. L. Toma and N. Zeh. I/O efficient algorithms for sparse graphs. Chapter 5 in U. Meyer, P. Sanders, and J. Sibeyn (Eds.), *Algorithms for Memory Hierarchies,* pp. 85–109, Springer-Verlag, Berlin, Germany, 2003.
313. N. Zeh. I/O efficient algorithms for shortest path related problems, Ph.D. Dissertation, School of Computer Science, Carleton University, 2002.

# 17

# Parallel I/O Systems




Peter J. Varman
*Rice University*


## 17.1 Introduction

The I/O system is a critical bottleneck for many modern data-intensive applications. The demand for greater storage capacity and high-speed access to stored data is growing rapidly. Disks, the most common secondary-storage medium in use today, have shown remarkable improvements in capacity and performance over the past decade. Innovations in disk technology have resulted in higher recording densities, smaller form factors, increased spindle speeds, and the increased availability of multi-zoned disks with variable transfer rates. Nonetheless, the storage requirements of modern applications is growing at an even faster rate, exceeding the impressive capacity of modern disk drives, and necessitating the use of multiple storage devices. Simultaneously the I/O rates required by these applications has outstripped the data rates that can be provided by single disks, despite the very significant improvements that have been made.

Consider as one example the growing use of digital multimedia in diverse applications ranging from entertainment and education to medicine and commerce. Multimedia or multimedia-enhanced applications routinely manipulate digitized images and video and audio data, requiring tremendous amounts of storage capacity, and placing stringent real-time constraints on the delivery rate to ensure smooth playback. A single hour-long MPEG-compressed video stream recorded at a rate of 4 Mbits/s, would require almost 2 GB of storage. A storage system with hundred or thousands of such clips would require several storage devices, perhaps a combination of disks to keep the more popular clips online, and slower tertiary tape storage to archive less popular video streams. The data transfer rate of a single disk is able to support the real-time retrieval of at most a few tens of concurrent streams, and the capacity decreases with increased video resolution and playback speeds. Analogous issues arise in other applications like real-time databases [47] where large numbers of sensory inputs need to be continually monitored and logged in an event database; critical events in turn may trigger the execution of data analysis routines

that need to be complete within stipulated time bounds, placing a tremendous strain on the I/O subsystem. Spatial databases in geographic information systems [7], temporal and kinetic databases that track the evolution or movement of objects in time [2,47], Web and application servers, graphics and visualization, and data mining systems, are other examples of the growing list of data-centric applications requiring the use of parallel I/O [1]. Even in compute-intensive domains like scientific computing applications, the scale of problems being addressed necessitates the use of advanced data management techniques, including the use of concurrent I/O to achieve acceptable performance [41].

## 17.2 Parallel I/O Organization

In this chapter, a parallel I/O system will refer to a disk-based I/O subsystem, made up of multiple disk drives that can access their data in parallel. Within this broad framework, different parallel I/O organizations are conceivable and supported by different vendors. RAID (an acronym that now stands for redundant array of independent disks) systems provide increased storage capacity and bandwidth by incorporating multiple disk drives within a single storage unit, and employ fault-tolerance mechanisms to cope with the increased failure probability stemming from the use of multiple devices [15]. Different RAID organizations (traditionally referred to as RAID levels) using different redundancy techniques to achieve fault tolerance have been proposed. RAID 1 uses data mirroring, whereby the entire disk contents are mirrored on an additional disk. RAID 4 and RAID 5 systems (RAID 5 is probably the most popular organization used in practice), employ the concept of a parity block to achieve fault tolerance. The multiple-disk system is viewed as a collection of stripes. A stripe consists of a block from each disk. One block of each stripe is designated as a parity block; it stores the bitwise exclusive-or of the corresponding bits of each of the other blocks in that stripe. In the event of a single-disk failure, the blocks on the failed disk can be reconstructed from the blocks in the same stripe on the working disks. The storage overhead for fault-tolerance is much less than the 100% redundancy of RAID 1 systems. The penalty, however, is the increased time for a write, since an update to a data block requires a read-modify-write operation on the parity block as well. A RAID 4 system uses a single designated disk to hold the parity blocks of all the stripes. In RAID 5 the use of a roving parity block, that associates different parity disks for different stripes, alleviates the potential parity-disk bottleneck of a RAID 4 design. Other RAID organizations have been since proposed. RAID 6 systems permit the failure of up to two disks without incurring any loss of data; these systems either use two parity blocks with differently computed parities, or employ a two-dimensional arrangement of disks with associated row and column parities. RAID 0 does not provide any fault tolerance, but allows data to be striped across multiple disks thereby allowing high-bandwidth transfers to and from the disks. Hybrid combination like RAID 10 and RAID 53 attempt to combine the advantages of different RAID levels in a hybrid architecture [51].

The interconnection between the disk system and the server is also undergoing changes to facilitate the increasingly parallel and distributed nature of storage systems. Traditional disk architectures use bus-based interconnects like the small computer system interconnect (SCSI) to connect a set of devices to the host [53]. A SCSI interconnect permits only a small number (7 or 15 depending on the SCSI protocol level) of devices to be connected to a single controller using the shared bus. The maximum transfer rate is small, starting at 5 MB/s for the original SCSI-1 protocol up to 40 MB/s for UltraSCSI.

More scalable I/O architectures are based on the use of switched interconnections. The high performance parallel interface (HIPPI) [29] defines a point-to-point interconnection, with high speed peak data transfer rates of 100 MB/s (HIPPI-800) to 800 MB/s (HIPPI-6400). Multiple devices are interconnected using a cross-point switch. Fiber channel refers to a set of standards [25] being developed by the American National Standards Institute (ANSI) that allows for an active intelligent interconnection scheme, called a fabric, to connect devices. It attempts to combine both network-oriented communication methods and dedicated hardware-based channel communication into a single I/O interface for both channel and network users. Different fiber channel topologies are supported including

point-to-point, cross-point switched, or an arbitrated loop (or ring topology) network. Fiber channel supports its own protocol, as well as higher level protocols such as the FDDI, SCSI, HIPPI, and IPI, enhancing its versatility, but increasing the potential compatibility problems as well. The fibre channel standard addresses the need for fast transfers, up to 1 Gbits/s, of large amounts of information. Other emerging interconnect standards include the switched InfiniBand architecture, a synthesis of formerly competing System I/O and NextGeneration I/O proposals, with projected peak bidirectional rates of up to 6 GB/s [30].

Another trend in I/O organizations is the decentralization of storage devices [24,45]. Storage area networks (SAN) and network-attached storage devices (NASD) are two such directions towards reducing the tight coupling between servers and devices in traditional I/O architectures. In a SAN, multiple servers and devices are connected together by a dedicated high-speed network different from, and in addition to, the local area network (LAN) connecting the servers and clients. Data transfer between a server and a device occurs over this dedicated back-end network. Networked storage architectures have several potential benefits. They facilitate sharing of disk-resident data between multiple servers by avoiding the three-step process (read I/O, network transfer, write I/O) required in transferring data on traditional server-hosted I/O architectures. Furthermore, they permit autonomous data transfer between devices simplifying backup and data replication for performance or reliability, and encourage the spatial distribution of devices on the network, while maintaining the capability for centralized management. A network-attached storage device [26] allows many of the server functions to be offloaded directly to the device. Once a request is authenticated by the server and forwarded to the device, data transfer to the network proceeds independently without further involvement of the server. In principle a NASD can be directly connected to the LAN or may serve as an independent module in a back-end SAN.

Highly parallel I/O organizations with high-bandwidth interconnections that have the capability of supporting hundreds of concurrent I/O transfers are a characteristic of current and evolving I/O architectures. The physical realization in terms of interconnection and communication protocols, redundancy and fault-tolerance, and balance between distribution and centralization of resources are a continuing topic of current research. Complex issues dealing with cost, performance, reliability, interoperability, security, and ease of configuration and management will need to be resolved, with perhaps different configurations suitable in different application domains.

Whatever the physical manifestation, managing hundreds of concurrent I/O devices in order to fully exploit their inherent parallelism and high interconnection bandwidth is a challenging problem. To study the issues at a high level, configuration-independent abstract models such as the parallel disk model (PDM) [58] have been proposed. Two extremes of logical I/O organizations based on the memory buffer can be identified: in a shared-buffer organization there is a centralized memory buffer shared by all the disks, and all accesses are routed through the buffer. In a distributed-buffer organization each disk has a private buffer used exclusively to buffer data from that disk. The shared configuration has the potential to make better use of the buffer space by dynamically changing the portion of the buffer devoted to any disk based on the load. In contrast, the performance of the distributed configuration can be limited by a few heavily loaded disks. Hybrid configurations are possible as in a logically shared but physically partitioned buffer. Such an architecture provides the scalability and modularity inherent in having distributed resources while providing increased resource utilization due to sharing.

## 17.3 Performance Model for Parallel I/O

Parallel I/O systems have the potential to improve I/O performance if one can exploit disk parallelism by performing multiple concurrent I/Os; however, it is a challenging problem to successfully exploit the available disk bandwidth to reduce application I/O latency. According to increasing evidence, traditional disk management strategies can severely under-utilize available bandwidth and therefore do not scale

well, leading to excessive I/O service time. As a consequence, several new algorithms for managing parallel I/O resources, with the explicit intention of exploiting I/O parallelism have been recently advanced [5,11,32–36,50,57].

The performance of a parallel I/O system is fundamentally determined by the pattern of disk accesses. The simplest form of data access, sequential reading of a file, represents the canonical application that can benefit from parallel I/O. Disk striping provides the natural solution for such an access pattern. The file is broken into blocks, and the blocks are placed in a round-robin fashion on the $D$ disks, so that every $D$th block is placed on the same disk. A main memory buffer of $D$ blocks is used. In each I/O an entire stripe of $D$ consecutive blocks, one block from each disk, is read into the memory buffer. The number of I/O steps is reduced by a factor of $D$ over sequentially accessing the file from a single disk. Despite its simplicity, disk striping is not the best solution for most other data access problems. For instance, generalizing the above problem to concurrently read $N$ sequential files, a disk-striping solution would read $D$ blocks of a single file in each I/O. The total buffer space required in this situation is ND blocks. A more resource-efficient solution is to perform concurrent, independent read I/Os on the different disks. In one parallel I/O, blocks from $D$ different files are fetched from the $D$ disks; this requires only $1/D$th the buffer of a disk striping solution if the blocks are consumed at the same rates. In fact, if the blocks are consumed at a rate comparable to the I/O time for a block, then by using independent I/Os only $\Theta(D)$ blocks of buffer suffice.

In contrast to the uniform access patterns implied by the previous examples, a skewed data access pattern results in hot spots, in which a single disk is repeatedly accessed in a short time period. The bandwidth of the multiple-disk system is severely underutilized in this situation, and the performance degrades to that of a single disk. Consider, for instance, the retrieval of constant data length (CDL) video data, in which the frames are packed into fixed-size data blocks; the blocks are then placed on the disks using either striped, random, or other disk allocation policy. If a number of such streams are read concurrently, the access pattern consists of an interleaving of the blocks that depends on the playback times of the blocks. For constant-bit rate (CBR) video streams the playback time of a block is fixed, and (assuming striped allocation) the accesses are spread uniformly across the disks as in the example on multiple file access. In the case of variable bit rate (VBR) video data streams, the accesses are no longer uniformly distributed across the disks, but depend on the relative playback times of each of the blocks. Consequently, both the load on a disk and the load across the disks varies as a function of time. In this case, simply reading the blocks in the time-ordered interleaving of blocks, may no longer maximize the disk parallelism, and more sophisticated scheduling strategies are necessary to maximize the number of streams that can be handled by the I/O system [22].

The abstract model of the I/O system that will be used to analyze the quality of different schedules is based on the PDM [58]: the I/O system consists of $D$ independent disks, which can be accessed in parallel, and has a buffer of capacity $M$, through which all disk accesses occur. The computation requests data in blocks—a block is the unit of disk access. The I/O trace of a computation is characterized by a reference string, which is an ordered sequence of I/O requests made by the computation. In serving a reference string the buffer manager determines which blocks to fetch and when to fetch them so that the computation can access the blocks in the order specified by the reference string. The computation waits for data from the I/O system only when the data are not available in the buffer. Additionally, when an I/O is initiated on one disk, blocks can be concurrently fetched from other disks. The number of parallel I/Os that are issued is a measure of performance in this model. Because the buffer is shared by all disks it is possible to allocate buffer space unevenly to different disks to meet the changing load on different disks. The PDM assumes unit time I/Os. In many applications like those dealing with streaming data, data logging or in several scientific computations, where large block sizes are natural, this is a viable and useful idealization. In these situations, the number of I/Os has a direct relationship to the I/O time. In other cases where requests are for small amounts of data and access times are dominated by the seek and rotational latency components, no analytical models are widely applicable. In these cases, empirical evaluations need to be employed in estimating performance [19,23].

## 17.4 Mechanisms for Improving I/O Performance

Prefetching and caching are two fundamental techniques that are employed for increasing I/O performance. Prefetching refers to the process of initiating a read from a disk before the computation demands the data. In a parallel I/O system, while a read progresses on one disk, reads can be started concurrently on other disks to prefetch data that are required later. These prefetched blocks are held in the I/O buffer till needed. In this way a temporary concentration of accesses to a small subset of the disks is tolerated by using the time to prefetch from the disks that are currently idle; when the locality shirts to the latter set of disks, the required data are already present in the buffer.

In contrast to prefetching that masks disk latencies by overlapping the access with that of I/Os to other disks, caching attempts to exploit temporal locality in the accesses. A selected subset of the recently accessed blocks are held in the I/O buffer in the expectation that they will be referenced again soon, thereby avoiding repeated disk accesses for the same block. Although both prefetching and caching are well-known techniques employed ubiquitously in computer systems and networking, deploying these mechanisms effectively in a parallel I/O system raises a unique set of challenges.

The I/O schedule determines the set of blocks that are fetched in each parallel I/O operation. The schedule is constructed dynamically so as to minimize the total number of parallel I/Os. This requires the scheduler to decide which blocks to prefetch, and, when the need for replacement arises, to decide which blocks in the buffer to cache and which to evict. Prefetching and caching in parallel I/O systems is fundamentally different from that in systems with a single disk, and requires the use of substantially different algorithms [11,32–36]. In a single-disk system, prefetching is used to overlap I/O operations with CPU computations. This is usually done using asynchronous I/O whereby a computation continues after making the I/O request without blocking. A stall model for analyzing the performance of overlapped I/O and computation was proposed in [17] for a single disk system; prefetching and caching algorithms to minimize stall time as a function of CPU and I/O speeds were presented in [5,17]. Disk scheduling algorithms that reorder I/O requests to minimize the disk seek times [59] can also be considered as a form of prefetching in single-disk systems.

In parallel I/O systems prefetching allows overlap between accesses on different disks thereby hiding the I/O latency behind the access latency on some other disk. The scheduler has to judiciously decide on questions like how much buffer to allocate for prefetching and how much for caching, which blocks to prefetch, and which blocks to cache. For instance, to utilize the available bandwidth, it may appear desirable to keep a large number of disks busy prefetching data during an I/O; however, excessive prefetching can fill up the buffer with blocks, which may not be used until much later in the computation. Such blocks have the adverse effects of choking the buffer and reducing the parallelism in fetching more immediate blocks. In fact, even when the problem does not involve the use of caching, the decisions of which blocks to prefetch and when to do so is not trivial.

Another issue needs to be addressed to employ prefetching and caching effectively. In order to prefetch accurately (rather than speculatively) some knowledge of future accesses is required. This is embodied in the notion of lookahead, which is a measure of the extent of knowledge about the future accesses that is available in making prefetching and caching decisions. Obtaining this lookahead has been the area of much active research [13,40,43,50]. In some applications like external sorting the lookahead can be obtained dynamically by using a sample of the data to accurately predict the sequence of block requests [10]. In video retrieval the sequence is determined by the playback times of blocks in the set of concurrently accessed streams; summary statistics of the streams are used to obtain the lookahead at run time [22]. Indexes in database systems can similarly be used to provide information about the actual sequence of data blocks that must be accessed. In broadcast servers the set of requests are prioritized by the system to maximize utilization of the broadcast channel [4]; the prioritized request sequence provides the lookahead for required I/O accesses. Access patterns can be revealed to the system either using programmer provided hints [50], or the system may attempt to uncover sequential or strided access patterns automatically at run time [40]. Speculative execution is another technique based on executing program code speculatively to determine the control path and the blocks accessed in the path [13].

## 17.5 Limitations of Simple Prefetching and Caching Strategies

In [11,32], the problem of scheduling read-once reference strings, in which each block is accessed exactly once, was considered. Such reference strings are characteristic of streaming applications like multimedia retrieval. Simple intuitive algorithms that work well in a single-disk scenario were analyzed and shown to have poor performance in the multiple-disk case. For instance, consider a natural scheduling algorithm that we refer to as *aggressive prefetching*. In each I/O, the next block required from each disk is fetched provided there is enough free buffer space; if not then only the block demanded immediately by the computation is read. Such an aggressive prefetching scheme, while intuitively attractive, can be shown to have poor worst-case as well as average-case performance. There exist worst-case reference strings for which aggressive prefetching can perform $\Theta(D)$ times as many I/Os as the optimal scheduling strategy [11]. In the average case, when the accesses are assumed to be randomly distributed across the disks with independent uniform probability, it has been shown that reading a reference string of length $N$ requires $\Theta(N/D)$ I/Os using a buffer of size $\omega(D^2)$ blocks [48].

The problem with aggressive prefetching is that it prefetches too deep on some disks, holding up buffer space that could better be used in fetching more immediately required blocks. A simple heuristic to correct for this is to place a bound on the depth of prefetching. One such attractive policy is to always give priority to a block that is required earlier in the reference string over one that is accessed later, whenever there is insufficient buffer space to hold both blocks. Intuitively this scheme tries to keep all disks busy by fetching greedily, but prevents blocks that are prefetched very much earlier than their time of usage from holding up buffer space that can be used by other more urgently needed blocks. This greedy algorithm is referred to as earliest required first (ERF) prefetching.

Consider the following example of an I/O system with three disks and an I/O buffer of capacity 6. Let the blocks labeled $a_i$ (respectively $b_i$, $c_i$) be placed on disk A (respectively B, C), and the reference string be

$$a_1\ a_2\ a_3\ a_4\ b_1\ c_1\ a_5\ b_2\ c_2\ a_6\ b_3\ c_3\ a_7\ b_4\ c_4\ c_5\ c_6\ c_7$$

Figure 17.1a shows the I/O schedule constructed by the ERF algorithm described above. In the first step blocks $a_1$, $b_1$, and $c_1$ are fetched concurrently in one I/O. When block $a_2$ is requested, blocks $a_2$, $b_2$, and $c_2$ are fetched in parallel in step 2. Subsequently the buffer contains five blocks: $a_2$, $b_1$, $b_2$, $c_1$, and $c_2$. Next when $a_3$ is requested, an I/O needs to be done to fetch it; however, there is buffer space for only one

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Disk A | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | | |
| Disk B | $b_1$ | $b_2$ | $b_3$ | | $b_4$ | | | | |
| Disk C | $c_1$ | $c_2$ | | | $c_3$ | $c_4$ | $c_5$ | $c_6$ | $c_7$ |

(a)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Disk A | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ |
| Disk B | $b_1$ | $b_2$ | | | $b_2$ | $b_3$ | $b_4$ |
| Disk C | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ | $c_7$ |

(b)

**FIGURE 17.1** (a) Greedy ERF schedule. (b) Optimal schedule.

additional block besides $a_3$, and the choice is between fetching $b_3$, $c_3$, or neither. Fetching greedily in the order of the reference string means that we fetch $b_3$. Continuing in this manner we obtain a schedule of length 9. Figure 17.1b presents an alternative schedule for the same reference string. The first two steps in the schedule are identical to the previous case. In step 3, $c_3$ that occurs after $b_3$ is prefetched; and in step 4, $c_4$ is fetched by evicting $b_2$ even though $c_4$ is referenced only after $b_4$; however, by doing so the overall length of the schedule is reduced to 7, better than the previous schedule.

The ERF algorithm was analyzed in [11]. It was shown that there exist reference strings for which ERF will perform $\Theta(\sqrt{D})$ times as many I/Os as the optimal schedule. For the average case, under the same assumptions as for aggressive prefetching, it can be shown that ERF can read an N block reference string in $\Theta(N/D)$ I/Os using a buffer of size $\omega(D \log D)$ blocks [10]. Hence, although ERF improves upon aggressive prefetching, it does not construct the optimal-length schedule.

In the previous discussion, all blocks were implicitly assumed to be distinct. Such reference strings are called read-once and are characteristic of streaming applications like multimedia retrieval. General reference strings where each block can be accessed repeatedly introduce additional issues related to caching. In particular, decisions need to be made regarding which blocks to evict from the buffer. In a single-disk system the optimal offline caching strategy is to use the MIN algorithm [12] that always evicts the block whose next reference is furthest in the future; however, it is easy to show that using this policy in a multiple-disk situation does not necessarily minimize the total number of parallel I/Os that are required. In fact, there exist reference strings for which the use of the MIN policy necessitates $\Theta(D)$ times as many I/Os as an optimal caching strategy [34].

## 17.6 Optimal Parallel-Disk Prefetching

In this section, we present an *online* prefetching algorithm L-OPT for read-once reference strings. L-OPT uses L-block lookahead; at any instant L-OPT knows the next $L$ references, and uses this lookahead to determine blocks to fetch in the next I/O. It uses a priority assignment scheme to determine the currently most useful blocks to fetch and to retain in the buffer. As the lookahead window advances and information about further requests are made available, the priorities of blocks are dynamically updated to incorporate the latest information. When considered as an offline algorithm for which the entire reference string is known in advance, it has been shown that L-OPT is the optimal prefetching algorithm that minimizes the number of parallel I/Os [32].

L-OPT is a priority-controlled greedy prefetching algorithm. A priority-controlled greedy prefetching scheme provides a general framework for describing different prefetching algorithms. Blocks in the lookahead are assigned priorities depending on the scheduling policy in effect. The scheduler fetches one block each from as many disks as possible in every I/O, while ensuring that the buffer never retains a lower-priority block in preference to fetching one with a higher priority, if necessary by evicting the lower-priority blocks. Algorithm priority-controlled greedy I/O describes the algorithm formally using the definitions below.

Different prefetching policies can be implemented using this framework merely by changing the priority function. For instance, to implement the ERF prefetching algorithm the priority of blocks should decrease with their position in the reference string. This is easily achieved if the priority function assigns the $i$th block in the reference string a priority equal to $-i$. Similarly, prefetching strategies akin to aggressive prefetching can be emulated by assigning the $i$th referenced block from each disk a priority of $+\infty$ if it is the demand block and $-i$ otherwise.

Definitions

1. Let $\Sigma = b_1, b_2, \ldots, b_n$ denote the reference string. If $b_i$ is a block in the lookahead, let *disk* ($b_i$) denote the disk from which it needs to be fetched and let *priority* ($b_i$) be the block's priority.
2. At the instant when $b_i$ is referenced, let $B_i$ denote the set of blocks in the lookahead that are present in the buffer.

3. When $b_i$ is referenced, let $H_i$ be the maximal set of (up to) $D$ blocks, such that if $b \in H_i$ then priority of b is the largest among all blocks from *disk*(b) in the lookahead but not present in the buffer.
4. Let $B_i^+$ be the maximal set of (up to) $M$ blocks with the highest priorities in $H_i \cup B_i$; in the case of ties the block occurring earlier in $\Sigma$ is preferred.

### 17.6.1 Algorithm Priority-Controlled Greedy I/O

On a request for a block, $b_i$, the algorithm takes the following actions.
If $b_i$ is present in the buffer then no I/O is necessary.
If $b_i$ is not present in the buffer then
update priorities of blocks using blocks revealed since the last I/O;
accommodate the blocks to be read in, evict the blocks in $B_i - B_i^+$; and
initiate an I/O to fetch the blocks in $H_i \cap B_i^+$.
Service the request for block $b_i$.

Implementing the priority-controlled greedy I/O algorithm can be done using a simple forecasting data structure similar to that in [10], to maintain the list of blocks with highest priority on each disk. On a hit in the buffer, the algorithm does not need to do any bookkeeping. When the requested block is not present in the buffer the algorithm needs to find the set of blocks to fetch and the corresponding set of blocks to evict from the buffer. If we have all the blocks in the buffer maintained and sorted in order of their priorities, then we can choose the $D$ blocks to fetch and evict in $O(M+D)$ time. With standard linked data structures, logarithmic update times are sufficient for these operations.

In contrast to the static priority assignments for ERF and aggressive prefetching, the priority function of the optimal algorithm L-OPT depends on the relative distribution of the load on different disks. Furthermore, as more lookahead is revealed, the previously assigned priorities of blocks may change as a result of the new information. At any time, the blocks in the lookahead are partitioned into two subsequences called the current and future window, respectively. At the start all blocks in the lookahead are in the current window and the future window is empty. As new blocks are revealed they are added to the future window. When the last block of the current window is referenced, the future window becomes the current window and a new (empty) future window begins. The priorities of blocks in the current window are fixed at the time the window became current, and do not change; however, the priorities of blocks in the future window are updated to reflect new additions. All blocks in the future window have priorities less than that of any block in the current window.

The priority assignment routine used by L-OPT to determine the priorities of blocks in a given piece of the reference string is described below. At any instant the priority of a block is a reflection of how urgently that block must be fetched. The lower the priority of a block, the later it can be fetched. The central idea is to set the priority of a block as low as possible, subject to two constraints. Blocks from the same disk are assigned priorities in order of their reference. Second, no block can have such a low priority that $M$ or more blocks referenced after it have a higher or same priority. In the routine below the variables *lowestPriorityOnDisk*[$d$] and *lowestPriority* track the smallest priority that can be assigned to a block without violating the two constraints. The former is incremented whenever a block is placed on disk $d$. The variable *lowestPriority* is incremented whenever $M$ blocks with priority *lowestPriority* or higher have been placed. A block is assigned the larger of these two priorities.

### 17.6.2 L-OPT: Priority Assignment

Assign priorities to blocks $\langle b_1, b_2, \ldots, b_n \rangle$ of the reference string.
Initialize

*lowestPriority* to 1
*numberOfBlocksPlaced* to 0
*lowestPriorityOnDisk*[$1 \ldots D$] to 0
blocksWithPriority[$1 \ldots n$] to 0

for $i$ from $n$ down to 1
  if ($lowestPriority > lowestPriorityOnDisk(disk(b_i))$) then assign
    $lowestPriorityOnDisk(disk(b_i)) \leftarrow lowestPriority$
  assign $priority(b_i) \leftarrow lowestPriorityOnDisk(disk(b_i))$
  increment $lowestPriorityOnDisk(disk(b_i))$
  increment $blocksWithPriority(priority(b_i))$
  increment *numberOfBlocksPlaced*
  if ($numberOfBlocksPlaced = M$) then
    decrement *numberOfBlocksPlaced* by $blocksWithPriority(lowestPriority)$
    increment *lowestPriority*

By using the priority assignment described here, it has been shown that L-OPT always creates a schedule that is within a factor $\Theta \sqrt{(MD/L)}$ times the length of the schedule created by the optimal *offline* algorithm, and that this is the best possible ratio. In addition, L-OPT's schedule is never more than twice the length of that created by any *online* algorithm (including algorithms that consistently make fortuitously correct guesses) that has the same amount of lookahead. Finally, note that if the entire reference string is known in advance, then L-OPT is the optimal offline algorithm [32].

## 17.7 Optimal Parallel-Disk Caching

For general reference strings where blocks may be repeatedly accessed, the buffer manager must decide which blocks to cache and which to evict. As noted earlier, the optimal single-disk caching policy embodied in the MIN algorithm can be decidedly suboptimal in the parallel I/O case. Prefetching and caching need to harmoniously cooperate in the multiple-disk situation. The caching problem has been studied by several researchers in the recent past for different I/O organizations. For a distributed-buffer configuration where each disk has its own private buffer, an algorithm P-MIN that generalizes MIN to multiple disks was shown to be optimal [57]. P-MIN uses the furthest forward reference policy on each disk independently to determine the eviction candidate for that disk. It initiates an I/O only on demand; in the ensuing I/O operation it prefetches aggressively from every disk unless the reference to the block to be prefetched is further than the references of all blocks currently in that buffer. For a shared-buffer configuration in the stall-model of computation, a sophisticated near-optimal algorithm called Reverse-Aggressive to minimize the stall time was proposed and analyzed in [36].

Recently, an optimal prefetching and caching algorithm, SUPERVISOR, for the parallel disk model was presented in [34]. Like the L-OPT algorithm for prefetching, SUPERVISOR uses the general framework of priority-controlled greedy I/O. The scheme for assigning priorities to references is, however, considerably more complex than that used by L-OPT for read-once reference strings. Just as a low priority with respect to prefetching indicates that an I/O for that block can be delayed, a low priority with respect to caching indicates that the block can be evicted from the buffer.

Intuitively, SUPERVISOR assigns priorities in accordance with two principles: issue prefetches for blocks close to their reference so that they do not wastefully occupy buffer space, and avoid caching a block if there is any later free I/O slot available, which can be used to fetch it. Among possible candidates for a block to cache, it is desirable to cache a block that will occupy the buffer for a smaller duration. Hence, the question to be answered is: Given that at some time we would like two previously referenced blocks in the buffer, which of these should have been cached and which should be fetched now? It is preferable to cache the block whose previous reference is closer to the current time, as this reduces the buffer pressure between the two previous accesses. SUPERVISOR uses this intuition to assign priorities to blocks for prefetching and caching.

The formal details of the priority assignment algorithm used by SUPERVISOR are presented in [34]. The routine examines subsets of the lookahead consisting of $M$ distinct references and then assigns priorities to one block from each disk. The idea behind the assignment can be understood by considering the largest subsequence of the lookahead including the last reference and having at most $M$ distinct

references. All blocks which are assigned the smallest priority should belong to this set. Otherwise there will be some reference such that $M$ or more blocks referenced after it have a higher, or same priority. Which among these blocks should have the lowest priority? The lowest priority can be assigned to, at most, one distinct reference from each disk. Additionally, among two blocks from the same disk, this priority is assigned to the block with the previous reference outside this subsequence is earlier, because we would rather not cache this block. It is shown in [34] that SUPERVISOR, which assigns priorities based on the above principle is the optimal offline algorithm for parallel prefetching and caching in the parallel disk model.

## 17.8 Randomized Data Placement

Randomizing the placement of blocks on the disks of a parallel I/O system is a method to reduce I/O serialization caused by hot spots [10,11,33,35,37,52,55]. If blocks are distributed on the disks randomly then the maximum number of accesses to a single disk in any sequence of requests can be bounded with high probability. There are two potential benefits of randomized placement: the amount of memory buffer required to smooth out the imbalance in disk accesses is greatly reduced, and good performance can be achieved using simpler prefetching and caching algorithms.

In a randomized data placement scheme each block is placed on any of the $D$ disks with a uniform probability $1/D$. The performance of two simple prefetching algorithms using randomized placement has been analyzed in [10,33]. Using the results of [46], aggressive prefetching was shown to read a reference string of $N$ blocks in an expected number $\Theta(N/D)$ I/Os using a smaller buffer, of size $\Theta(D^2)$ blocks [32]. Note that $\lceil N/D \rceil$ is the minimum number of I/Os needed to read $N$ blocks, so the scheme performs within a constant factor of the minimum possible number of I/Os. The performance of ERF that gives preference to blocks that occur earlier in the reference string was analyzed in [10] and shown to require an expected $\Theta(N/D)$ I/Os using a smaller buffer, of size $\Theta(D \log D)$ blocks. In an online situation the two prefetching algorithms require different lookahead information. The aggressive prefetching algorithm only needs to know the ordered sequence of accesses to be made from each disk independently. The greedy priority-based algorithm needs to know the global ordering of accesses across the disks. In some applications like external merging for instance, the global ordering can be inferred from the local ordering by using a small amount of preprocessing [10].

Recently, it was shown how randomized placement coupled with data replication can be used to improve I/O performance [37,50,55], particularly in [37] where two copies of each block are allocated randomly to the disks. A scheduling algorithm decides which of the copies should be read in an I/O. It was shown that $N$ blocks can be read in $\lceil N/D \rceil + 1$ I/Os with high probability, using only $\Theta(D)$ blocks of buffer storage [52].

For general reference strings a simple caching and prefetching algorithm that can be used in conjunction with randomized data placement was presented in [35]. The algorithm uses the ERF policy for prefetching and a variant of the least recently used buffer replacement policy to handle evictions. It was shown that the expected number of I/Os performed by this algorithm is within a factor $\Theta[\log D/\log (\log D)]$ of the number of I/Os performed by an optimal scheduling algorithm.

Randomized data placement can generally provide good expected performance using less buffer memory and simpler disk management algorithms than those required to deal with worst-case data placements.

## 17.9 Out-of-Core Computations

Out-of-core computation deals with the problems of solving computational problems that are too large for the entire data set to fit in primary memory. Although the virtual memory mechanisms of modern operating systems can handle the problem transparently by paging the required data in and out of main memory on demand, the performance of such a solution is usually poor. Improved performance is

achieved by optimizing the algorithm to be sensitive to the constraints of the I/O subsystem. The computation should be structured to provide spatial locality using data clustering, accesses should be organized to expose temporal locality, and declustering should be used to exploit the parallelism provided by the underlying I/O system. In many cases traditional in-core algorithms that deal with minimizing the number of computations without explicit consideration of the data access costs perform poorly when the data is disk-resident, necessitating the development of new algorithms or requiring radical restructuring of the known algorithms to achieve good I/O performance.

External or out-of-core algorithms using parallel data transfers can be traced to the work by Aggarwal and Vitter [3], generalizing earlier models, which dealt with sequential or nonblocked data transfers. The model used in that work was more powerful than the PDM that models multiple-disk systems. A number of out-of-core algorithms for external sorting, computational geometry, FFT data permutations, linear algebra computations, scientific codes, and data structures for indexing complex multidimensional data have since been developed [1–3,9,20,21,27,46,56,58]. The reader is referred to [1] and the references therein for a comprehensive bibliography and discussion of these works.

Run-time environments to increase efficiency and simplify the programming effort in applications requiring parallel I/O has been addressed by several research groups [6,8,14,16,18,28,31,38,39,42,44,54]. For a detailed discussion of the different proposals the reader is referred to [41,49].

## 17.10 Conclusion

Parallel I/O systems consisting of multiple concurrent devices are necessary to handle the storage and bandwidth requirements of modern applications. Parallel I/O hardware and interconnection technology will continue to evolve to meet the growing demands. New algorithms and system software are essential to effectively manage the hundreds of richly interconnected concurrent devices. Caching and prefetching are two fundamental techniques to improve data access performance by exploiting temporal locality and latency hiding. In a parallel I/O system using these mechanisms effectively involve challenging issues, which have been extensively studied over the past few years. These have resulted in the design of optimal algorithms for prefetching and caching, techniques to obtain lookahead of the I/O accesses, external algorithms for important problems, and file system and I/O primitives to support parallel I/O. As systems grow larger and more complex, challenging problems to control and manage the parallelism automatically and effectively will continue to be explored. Building on the fundamental understanding of what works and the algorithms required to control them, tools to automatically perform configuration, dynamic declustering, replication, prefetching, and caching will continue to be developed. Finally, although this chapter deals primarily with disk I/O, it can be readily seen that many of the issues transcend device specificity and apply in more general contexts dealing with managing and processing multiple concurrent I/O streams, using limited storage and bandwidth resources, as in embedded system environments.

### Acknowledgment

Supported in part by NSF grant CCR-9704562.

### References

1. J.M. Abello and J.S. Vitter (Eds.). *External Memory Algorithms*, Volume 50 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS, American Mathematical Society, Providence, RI, 1999.
2. P.K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proceedings ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 2000.
3. A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9): 1116–1127, Sep. 1988.

4. D. Aksoy and M. Franklin. RxW: A scheduling approach to large scale on-demand broadcast. *IEEE/ACM Transactions on Networking*, 7: 846–861, Dec. 1999.
5. S. Albers, N. Garg, and S. Leonardi. Minimizing stall times in single and parallel disk systems. In *Proceedings of the ACM Symposium on Theory of Computing*, pp. 454–462, 1998.
6. T.E. Anderson et al. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1): 41–79, Feb. 1996.
7. L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Windmayer (Eds.). *Algorithmic Foundations of GIS*, volume 1340, Lecture Notes in Computer Science, Springer-Verlag, 1997.
8. R.H. Arpaci-Dusseau et al. Cluster I/O with river: making the fast case common. In *Proceedings 6th ACM Workshop on I/O in Parallel and Distributed Systems,* pp. 68–77, Atlanta, GA, 1999.
9. L.M. Baptist and T.H. Cormen. Multidimensional, multiprocessor out-of-core FFTs with distributed memory and parallel disks. In *Proceedings 11th ACM Symposium on Parallel Algorithms and Architectures,* June 1999.
10. R.D. Barve, E.F. Grove, and J.S. Vitter. Simple randomized merge-sort on parallel disks. *Parallel Computing,* 23(4): 601–631, June 1996.
11. R.D. Barve, M. Kallahalla, P.J. Varman, and J.S. Vitter. Competitive parallel disk prefetching and buffer management. *J. of Algorithms,* 36(2): 152–181, Aug. 2000.
12. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal,* 5 (2): 78–101, 1996.
13. F. Chang and G.A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of Third Symposium on Operating Systems Design and Implementation,* pp. 1–14, Feb. 1999.
14. A. Choudhary et al. Data management for large-scale scientific computations in high performance distributed systems. In *Proceedings of the 8th IEEE Symposium on High Performance Distributed Systems,* Aug. 1999.
15. P.M. Chen et al. RAID: High performance and reliable secondary storage. *ACM Computing Surveys,* 26(2): 145–185, 1994.
16. Y.M. Chen et al. Automatic parallel I/O performance in Panda. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures,* pp. 108–118, June 1998.
17. P. Cao, E. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 1995.
18. P.F. Corbett and D.G. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems,* 14(3): 225–264, Aug. 1996.
19. T.H. Cormen and M. Hirschl. Early experiences in evaluating the parallel disk model with the ViC* implementation. *Parallel Computing,* 23(4–5): 571–600, June 1997.
20. T.H. Cormen and D.M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing,* 24(1): 5–20, Jan. 1998.
21. T.H. Cormen, T. Sundquist, and L.F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal of Computing,* 28(1): 105–136, 1999.
22. O. Ertug, M. Kallahalla, and P.J. Varman. Real-time parallel disk scheduling for VBR video servers. In *Proceedings 5th International Conference on Computer Science and Informatics,* Feb. 2000.
23. S. Evgenia and D.A. Reed. Workload characterization of input/output intensive parallel applications. In *Proceedings of Modeling Techniques and Tools for Computer Performance Evaluation,* Volume 1245, Lecture Notes in Computer Science, Springer-Verlag, pp. 169–280, June 1997.
24. M. Farley. *Building Storage Networks,* Osborne/McGraw-Hill, 2000.
25. Fibre Channel Industry Association. See www.fibrechannel.com.
26. G.A. Gibson et al. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems,* 1988.
27. M.T. Goodrich, J.-J. Tsay, D.E. Vengroff, and J.S. Vitter. External-memory computational geometry. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pp. 714–723, Nov. 1993.

28. R.L. Haskin. Tiger Shark—A scalable file system for multimedia. *IBM Systems Journal of Research and Development*, 42(2): 185–197, March 1998.
29. High Performance Networking Forum. See www.hnf.org.
30. InfniBand Trade Association. See www.infinibandta.org.
31. R. Jain, J. Werth, and J.C. Browne (Eds.). *Input/Output in Parallel and Distributed Computer Systems*. Kluwer Academic Publishers, Norwell, MA, 1996.
32. M. Kallahalla and P.J. Varman. Optimal read-once parallel disk scheduling. In *Proceedings 6th ACM Workshop on I/O in Parallel and Distributed Systems*, pp. 68–77, Atlanta, GA, 1999. (An expanded version is available at www.ece.rice.edu/∼pjv).
33. M. Kallahalla and P.J. Varman. Randomized prefetching and caching. In *Randomization in Parallel and Distributed Systems*, S. Rajasekaran and S. Pardalos (Eds.), Kluwer Academic Press, Dordrecht, the Netherlands, 1999.
34. M. Kallahalla and P.J. Varman. Optimal prefetching and caching for parallel I/O systems. *Proceedings 13th ACM Symposium on Parallel Algorithms and Architectures*, July 2001.
35. M. Kallahalla and P.J. Varman. Analysis of simple randomized buffer management for parallel I/O, Online version available at www/ece/rice.edu/∼pjv. (to be published in Information Processing Letters).
36. T. Kimbrel and A.R. Karlin. Near-optimal parallel prefetching and caching. *SIAM J. of Computing*, 5 (3): 79–119, March 1988.
37. J. Korst. Random duplicate assignment: an alternative to striping in video servers. In *Proceedings ACM Multimedia Conference*, pp. 219–226, 1997.
38. D. Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, 15(1): 41–74, Feb. 1997.
39. W.B. Ligon III and R.B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, pp. 471–480, Aug. 1996.
40. T. Madyastha and D.A. Reed. Input/output access pattern classification using hidden Markov models. In *Proceedings of 5th Workshop on I/O in Parallel and Distributed Systems*, Nov. 1997.
41. J.M. May. *Parallel I/O for High-Performance Computing*. Morgan Kaufmann Publishers, Academic Press, San Diego, CA, 2001.
42. E.L. Miller and R.H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4–5): 419–446, June 1997.
43. T.C. Mowry, A.K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings 2nd Symposium on Operating Systems Design and Implementation*, pp. 3–17, Oct. 1996.
44. S.A. Moyer and V.S. Sunderam. Scalable concurrency control for parallel file systems (in [1]).
45. National Storage Industry Consortium. See www.nsic.org.
46. M.H. Nodine and J.S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4): 919–933, July 1995.
47. G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4): 513–532, 1995.
48. V.S. Pai, A. Schaffer, and P.J. Varman. Markov analysis of multiple-disk prefetching strategies for external merging. *Theoretical Computer Science*, 128(1–2): 211–239, June 1994.
49. Parallel I/O Bibliography. See http://www.cs.dartmouth.edu/pario/bib/.
50. R.H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings 15th ACM Symposium on Operating System Principles*, pp. 79–95, Dec. 1995.
51. RAID Advisory Board. See www.raid-advisory.com.
52. P. Sanders, S. Egner, and J.H.M. Korst. Fast concurrent access to parallel disks. In *Proceedings of the SIAM Symposium on Discrete Algorithms*, pp. 849–858, Jan. 2000.
53. SCSI Trade Organization. See www.scsita.org.
54. R. Thakur et al. Passion: Optimized I/O for parallel applications. *Computer*, 29(6), June 1996.

55. R. Tewari, R. Mukherjee, D. Dias, and H. Vin. Design and performance tradeoffs in clustered video servers. In *Proceedings of the International Conference on Multimedia and Systems*, pp. 144–150, 1996.
56. S. Toledo. A survey of out-of-core algorithms in numerical linear algebra (in [1]).
57. P.J. Varman and R.M. Verma. Tight bounds for prefetching and buffer management algorithms for parallel I/O systems. *IEEE Transactions on Parallel and Distributed Systems*, 10: 1262–1275, Dec. 1999.
58. J.S. Vitter and E.A.M. Shriver. Optimal algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3): 110–147, 1994.
59. L. Zheng and Per-Ake Larson. Speeding up external mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2): 322–332, April 1996.

# 18

# A Read Channel for Magnetic Recording




Bane Vasić
*University of Arizona*

Miroslav Despotović
*University of Novi Sad*

Pervez M. Aziz
*Agere Systems*

Necip Sayiner
*Agere Systems*

Ara Patapoutian
*Maxtor*

Brian Marcus
*IBM Almaden Research Center*

Emina Šoljanin
*Lucent Technologies*

Vojin Šenk
*University of Novi Sad*

Mario Blaum
*IBM Almaden Research Center*


A steady increase in recording densities and data rates of magnetic hard drives during last 15 years are mostly due to advances in recording materials, read/write heads, and mechanical designs. The role of signal processing and coding has been to make the best use of the capacity and speed potentials offered by these advances. As the recording technology matures, the read channel is becoming more and more advanced, reaching the point where it uses equally or even more complicated signal processing, coding

and modulation algorithms than any other telecommunication channel and where, due to the speed, power consumption, and cost requirements, the challenges in implementing new architectures and designs have been pushed to today's integrated circuit manufacturing technology limits.

This chapter reviews advanced signal processing, modulation, coding techniques, and architectures for magnetic recording read channel. In the most general terms, the *read channel* controls the reading and writing the data to/from magnetic medium (unjustifiably the "write" part has disappeared from its name). The operations performed in the data channel are: *timing recovery, equalization, data detection, modulation coding/decoding,* and limited *error control.* Besides this, so called *data channel,* a read channel also has a *servo channel,* which role is to sense head position information, and, together with the head positioning servo system, to regulate a proper position of the head above the track. This chapter gives an in-depth treatment of all of these subsystems.

We begin with the review of the magnetic recording principles. We describe basic recording physics and explain how the interactions among neighboring magnetic domains cause intersymbol interference (ISI). Then we introduce a partial response signaling as a method of controlling the ISI. Section 18.1 also describes physical and logical organization of data on a disk and methods of increasing recording density.

Section 18.2 gives a block diagram of a state-of-the-art read channel and explain its subsystems. We explain organization of data on the disc tracks, servo sectors and data sectors, seeking and tracking operations, and phase and frequency acquisition. The section on servo information detection explains sensing radial information and read channel subsystem used to perform this operation.

The treatment of the data channel begins with an in-depth treatment of partial response signaling and adaptive equalization-standard techniques used in today's read channels. The novel equalization approaches and *generalized* partial response polynomials are also discussed in this section. We continue with a maximum likelihood sequence detection algorithm—Viterbi algorithm—and a noise predictive Viterbi algorithm, which enhances the performance by exploiting the fact that noise is highly colored and can be therefore predicted to some extent. The data detection also includes error event correction through post-processing, a new technique used in latest generation of read channels, as well as novel soft decoding and iterative decoding techniques.

The fourth part of the chapter discusses modulation and error control coding. Modulation coding in a read channel serves a variety of important roles. Generally speaking modulation coding eliminates those sequences from a recorded stream that would degrade the error performance, for example, long runs of consecutive like symbols that impact the timing recovery, or/and sequences that result in a signal on a small Euclidian distance. We complete the coding section with error control coding—both traditional algebraic techniques such as Reed Solomon codes, as well as with new trends such as iterative decoding. The error control coding is not part of present read channel chips, but will be integrated in the next generation of so-called "super chips."

We conclude this chapter by the review of read channel technology including novel read channel architectures such as postprocessor, super chip, etc., as well as the issues of digital design, chip testing, and manufacturing.

## 18.1 Recording Physics and Organization of Data on a Disk

*Bane Vasić and Miroslav Despotović*

### 18.1.1 Magnetic Recording Basics

The basic elements of a magnetic recording system are read/write head, which is an electromagnet with a carefully shaped ferrous core, and a rotating disk with a ferromagnetic surface. Since the core of the electromagnet is ferrous, the magnetic flux preferentially travels through the core. The core is deliberately broken at an air gap. In the air gap, the flux creates a fringing field that extends some distance from the core.

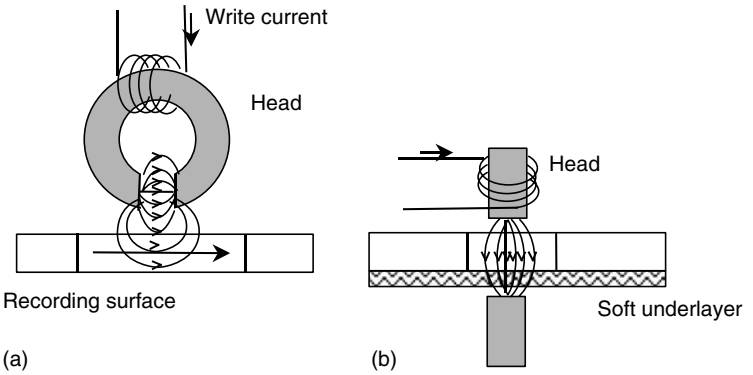



**FIGURE 18.1** (a) Longitudinal recording. (b) Perpendicular recording.

To record data on a surface of a disk, the modulated signal current, typically bipolar, is passed through the electromagnet coils thus generating a fringing magnetic field. The fringing magnetic field creates a remanent magnetization on the ferromagnetic surface, i.e., the ferromagnetic surface becomes permanently magnetic. The magnetic domains in the surface act like tiny magnets themselves and create their own fringing magnetic field above the ferromagnetic surface. The data are recorded in concentric tracks as a sequence of small magnetic domains with two senses of magnetization depending on a sign of writing current. In this, so-called *saturation recording*, the amplitude of two writing current signal levels are chosen sufficiently large so as to magnetize to saturation the magnetic medium in one of two directions. In this way, the nonlinear hysteresis effect does not affect domains recorded over previously recorded ones.

In a simple reading scenario the reading head flies over the disk-spinning surface (at head-to-medium velocity, $v$) and passes through the fringing magnetic fields above the magnetic domains. Depending on a head type, the output voltage induced in the electromagnet is proportional to the spatial derivative of the magnetic field created by the permanent magnetization in the material in the case of inductive heads, or is proportional to the fringing magnetic field in the case of magneto-resistive heads. Today's hard drives use magneto-resistive heads for reading, because of their higher sensitivity. Pulses sensed by a head in response to transition on the medium are amplified and then detected to retrieve back the recorded data. For both types of heads, it is arranged that the head readback signal responds primarily to transitions of the magnetization pattern. The simplest, single parameter model for an isolated magnetic *transition response* is the so-called Lorenzian pulse (Figs. 18.1 and 18.2).

$$g(t) = \frac{1}{1 + \left(\dfrac{2t}{PW_{50}}\right)^2}$$

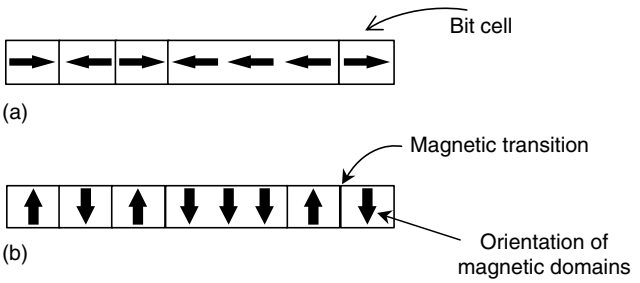



**FIGURE 18.2** Magnetic domains representing bits.

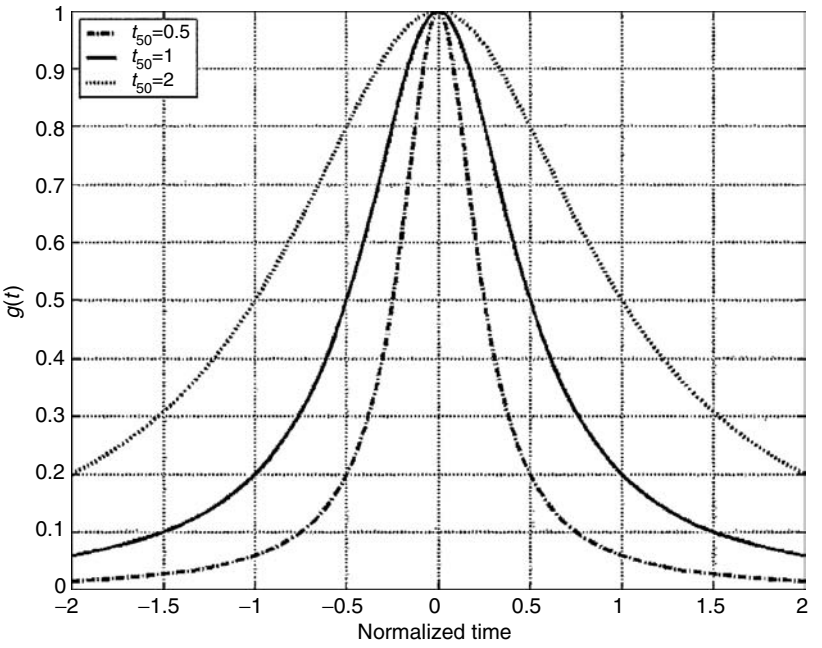



**FIGURE 18.3** Transition response $g(t)$—mathematical model.

where $t_{50}$ is a parameter representing the pulse width at 50% of the maximum amplitude. Simplicity and relatively good approximation of the channel response are the main reasons for attractiveness of this model. The family of $g(t)$ curves for different $t_{50}$ values is depicted in Fig. 18.3. The width at half amplitude defines the recording process *resolution,* i.e., $PW_{50}$,* as a spatial, while $t_{50}$, as a temporal measure, is alternatively in use ($PW_{50} = vt_{50}$).

Ideal conditions for readback process would be to have head that is sensing the medium in an infinitely narrow strip in front of the head; however, head resolution is limited, so that the head output depends on "past" and "future" bit cell magnetization patterns. Such dependence causes superposition of isolated transition responses partly canceling each other. This phenomenon is known as intersymbol interference (ISI). The interference is largest for transitions at minimum spacing, i.e., a spacing of a single bit cell $T$. The response to two physically adjacent transitions is designated *dibit response* or *symbol response,* i.e., $h(t) = g(t) - g(t - T)$. Typical readback waveform illustrating these types of responses is depicted in Fig. 18.4.

Mathematically, the noiseless input-output relationship can be expressed as

$$y(t) = \sum_{i=-\infty}^{\infty} x_i h(t - iT) = \sum_{i=-\infty}^{\infty} (x_i - x_{i-1}) g(t - iT)$$

where $y$ and $x \in \{-1, +1\}$ are readback and recorded sequences, respectively. Notice that every transition between adjacent bit cells yields a response $\pm 2g(t)$, while no transition in recorded sequence produces zero output.

*This is not so strict because, contrary to this, some authors use $PW_{50}$ designating temporal resolution.

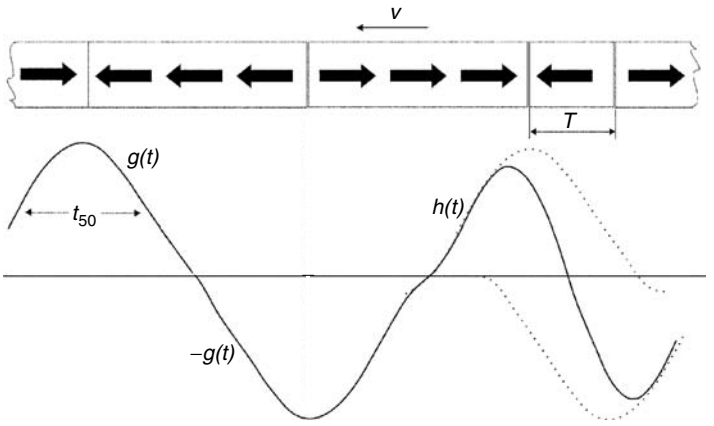



**FIGURE 18.4** Sketch of a typical readback waveform in magnetic recording.

Normalized measure of the *information density** is defined as the ratio $D = t_{50}/T$ showing how many channel bits are packed "under" the dispersed pulse of duration $t_{50}$. Case in which we are increasing density ($D > 2$) is accompanied by an increase of duration of $h(t)$ expressed in units of $T$, as well as rapid decrease of the amplitude of dibit response, which is equivalent to lowering of signal-to-noise ratio in the channel. As a consequence, any given bit will interfere with successively more preceding and subsequent bits producing more severe ISI. At low normalized information densities, peaks of the transition responses are clearly separated, so it is possible to read recorded data in simple manner by detecting these peaks, i.e., *peak detectors*. Contrary to this, high-density detectors have to implement more sophisticated detection methods in order to resolve combination of these effects. One of the most important techniques to combat ISI in magnetic recording channels is partial-response (PR) signaling with maximum-likelihood (ML) sequence detection, i.e., PRML detection, Section 18.5. The applicability of this scheme in magnetic recording channels was suggested over 30 years ago [4], but the advance in technology enabled first disk detectors of this type at the beginning of nineties [2].

The basic idea of a PR system is that certain controlled amount of ISI, at the channel equalizer output, is left for a detector to combat with. The nature of controlled ISI is defined by a PR. This method avoids full channel equalization and intolerable noise enhancement induced by it in a situation when amplitude distortions, as a result of increased density, are severe. In magnetic recording systems the PR detector reconstructs recorded sequence from samples of a suitable equalized readback signal at time instants $t = iT$, $i \geq 0$. The equalization result is designed in a manner that produces just a finite number of *nonzero* $h(t)$ samples $h_0 = h(0)$, $h_1 = h(T)$, $h_2 = h(2T), \ldots h_k = h(KT)$. This is usually represented in a compact *partial-response polynomial* notation $h(D) = h_0 + h_1 D + h_2 D^2 + \ldots + h_k D^k$, where the dummy variable $D^i$ signifies a delay of $i$ time units $T$. Then the "sampled" input–output relationship is of the form

$$y(jT) = \sum_{i=j-k}^{j} x_i h(jT - iT)$$

For channel bit densities around $D \approx 2$, the target PR channels is usually the class-4 partial response (PR4), described by $h(D) = 1 - D^2 = (1 - D)(1 + D)$. At higher recording densities Thapar and Patel [6] introduced a general class of PR models with PR polynomial in the form $h_n(D) = (1 - D)(1 + D)^n$, $n \geq 1$

*When channel coding is introduced, this density is greater than the user information density because of added redundancy in the recorded channel sequence.

that is a better match to the actual channel discrete-time symbol response. Notice that the PR4 model corresponds to the $n = 1$ case. The channel models with $n \geq 2$ are usually referred to as "extended class-4" models, and denoted by $E^{n-1}$ PR4 (EPR4, $E^2$PR4). Recently, the modified $E^2$PR4 ($ME^2$PR4) channel, $h(D) = (1 - D^2)(5 + 4D + 2D^2)$, was suggested due to its robustness in high-density recordings. Notice that as the degree of PR polynomials gets higher, the transition response, $g(t)$, becomes wider and wider in terms of channel bit intervals, $T$ (EPR4 response extends over 3-bit periods, $E^2$PR4 over 4), i.e., the remaining ISI is more severe.

The transfer characteristics of the Lorentzian model of the PR4 saturation recording channel (at densities $D \approx 2$), is close to transition response given by

$$g(t) = \frac{\sin\left(\pi \frac{t}{T}\right)}{\pi \frac{t}{T}} + \frac{\sin\left(\pi \frac{t-T}{T}\right)}{\pi \frac{t-T}{T}}$$

generating the output waveform described by

$$y(t) = \sum_{i=-\infty}^{\infty} (x_i - x_{i-1}) g(t - iT) = \sum_{i=-\infty}^{\infty} (x_i - x_{i-2}) \frac{\sin\left(\pi \frac{t-iT}{T}\right)}{\pi \frac{t-iT}{T}}$$

Note that $g(t) = 1$ at consecutive sample times $t = 0$ and $t = T$, while at all other discrete time instants, its value is 0. Such transition response results in known ISI at sample times, leading to output sample values that, in the absence of noise, take values from the ternary alphabet $\{0, \pm 2\}$. In order to decode the readback PR sequence it is useful to describe the channel using the *trellis state diagram*. This is a diagram similar to any other graph describing a finite-state machine, where states indicate the content of the channel memory and branches between states are labeled with output symbols as a response to the certain input (the usual convention is that for the upper branch leaving a state we associate input $-1$ and $+1$ for the lower). The EPR4 channel has memory length 3, its trellis has $2^3 = 8$ states, and any input sequence is tracing the path through adjacent trellis segments.

An example of the trellis diagram is given in Fig. 18.5 for the EPR4 channel. However, notice that in PR trellis there are also distinct states for which there exist mutually identical paths (output sequences) that start from those states, so that we can never distinguish between them (e.g., the all-zero paths emerging from the top and bottom states of the EPR4 trellis). Obviously, such a behavior can easily lead to great problems in detection in situations when noise can confuse us in resolving the current trellis

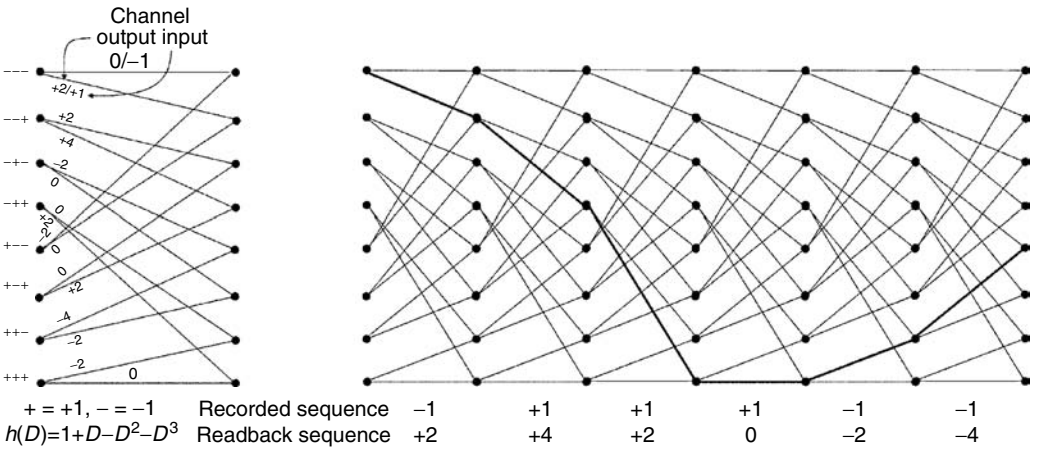



**FIGURE 18.5** Trellis representation of EPR4 channel outputs.

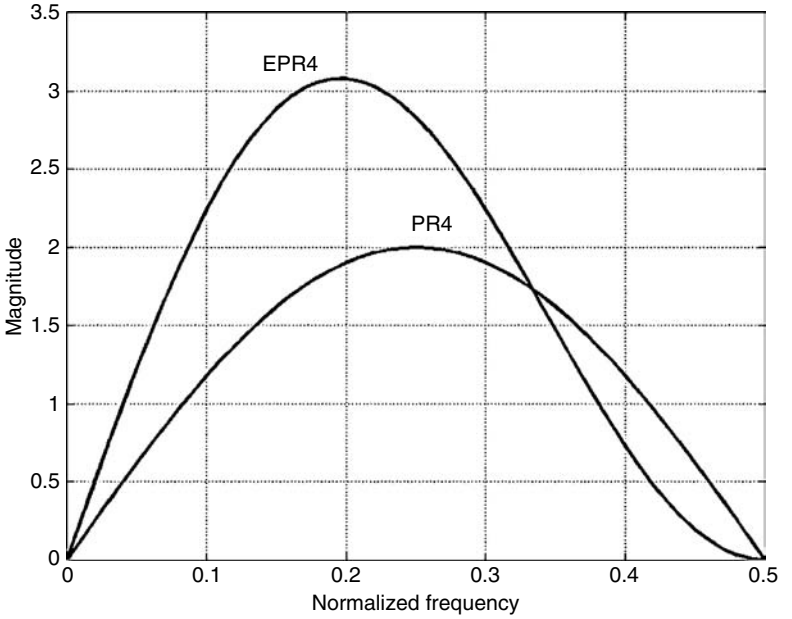



**FIGURE 18.6** Frequency response of PR4 and EPR4 channel.

state (e.g., the bottom one for the upper in the running example). Such a trellis is so-called *quasi-catastrophic* trellis and further details on this subject could be found in [3].

A common approach to studying the PR channel characteristics is to analyze its frequency spectra. Basically, when the recording density is low ($D \approx 0.5$) and readback pulses are narrow compared to the distance between transitions, such a signal contains a high-frequency component (highest frequency components correspond to the fastest edge of the signal). With the growth of density, the spectral energy distribution move towards lower frequency range. This means that for the system with $D = 2$, the signal spectrum is concentrated below half of the channel bit rate given by $1/T$. The power of the highest spectral components outside this half-bandwidth range is negligible. This means that for high-density recording we can limit the channel bandwidth to $1/2T$ without loss of information and filtering the high frequencies containing noise only.

Finding the Fourier transform of the dibit response we obtain the frequency response for the PR4 channel given by

$$H(w) = 1 - e^{i2wT}, \quad |H(w)| = 2\sin(wT), \quad 0 \leq w \leq \frac{\pi}{T}$$

For higher order PR channels we have different transition responses and accordingly different frequency responses calculated in a similar fashion as for the PR4 channel. The frequency response for these channels is shown in Fig. 18.6.

These lower frequency spectrum distributions of PR channels are closer to a typical frequency content of raw nonequalized pulses. Hence, equalization for extended PR channels can become less critical and requires less high frequency boost that may improve signal-to-noise ratio.

### 18.1.2 Physical Organization of Data on Disk

In most designs, the head is mounted on a slider, which is a small sled-like structure with rails. Sliders are designed to form an air bearing that gives the lift force to keep the slider-mounted head flying at the

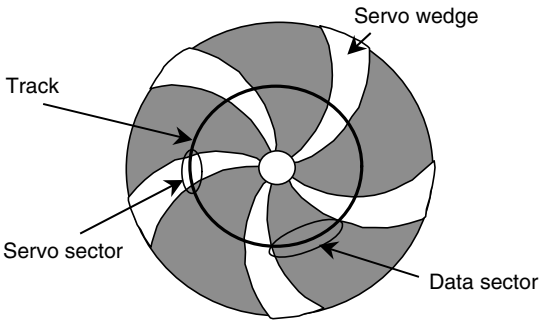



**FIGURE 18.7** Data and servo sectors.

small and closely controlled height (the so-called Winchester technology). A small flying height is desirable because it amplifies the readback amplitude and reduces the amount of field from neighboring magnetic domains picked by the head, thus enabling sharper transitions in the readback signal and recording more data on a disk; however, the surface imperfections and dust particles can cause the head to "crash." Controlling the head-medium spacing is of critical importance to ensure high readback signal, and stable signal range. It is also important during reading to keep the head center above the track being read to reduce magnetization picked up from neighboring tracks. The signal induced in the head as a result of magnetic transitions in a neighboring track is known as a cross talk or inter-track interference. In order to position the head, special, periodic, wedge-like areas, the so-called *servo wedges*, are reserved on a disk surface for radial position information. They typically consume 5–10% of the disk surface available. An arch of a track laying in a servo wedge is called a *servo sector*. The area between servo wedges is used to record data, and a portion of a track between two servo sectors is referred to as a *data sector* or *data field*. In other words, the data and servo fields are time multiplexed, or using disk drive terminology, the servo field is *embedded* in the data stream. To estimate radial position a periodic waveform in a servo sector is detected, and the radial position error signal is calculated based on the current estimated position of a head and the actual position of the track to be followed, and then used in a head positioning servo system (Fig. 18.7).

### 18.1.3 Logical Organization of Data on a Disk

On a disk, data are organized in *sectors*. For a computer system, the sector is a sequence of (8-bit) bytes within each addressable portion of data on a disk drive. The sector size is typically 512 bytes. For an error control system, the sector is a sequence of error control codewords or blocks. For interleaved error control systems, each sector contains as many blocks as there are interleave cycles. The block elements are symbols that are not necessarily eight bit long. In the most general terms, the symbols are used in the error control coding (ECC) system to calculate and describe error locations and values.

A read channel sees a sector as a sequence of modulation codewords together with synchronization bits. Synchronization field is partitioned into a sector address mark, or sync mark, typically of length around 20 bits, and phase lock loop (PLL) field, a periodic pattern whose length is about 100 bits used for PLL synchronization. In addition to this, a secondary sync mark is placed within a data field and used for increased reliability. A zero-phase start pattern of length 8–16 bits is used for initial estimation of phase in the PLL. Figure 18.8 illustrates the format of user data on a disk.

### 18.1.4 Increasing Recording Density

Increasing areal density of the data stored on a disk can be achieved by reducing lengths of magnetic domains along the track (increasing linear density) and by reducing a track width and track pitch (increasing radial density). Although the radial density is mostly limited by the mechanical design of the drive and ability to accurately follow the track, the linear density is a function of properties of magnetic materials and head designs, and ability to detect and demodulate recorded data.

As linear density increases, the magnetic domains on a surface become smaller and thus thermally unstable, which means that lower energy of an external field is sufficient to demagnetize them. This effect is known as a superparamagnetic effect [1]. Another physical effect is the finite sensitivity of the read head, i.e., at extremely high densities, since the domains are too small; the signal energy becomes so small as to be comparable with the ambient thermal noise energy.

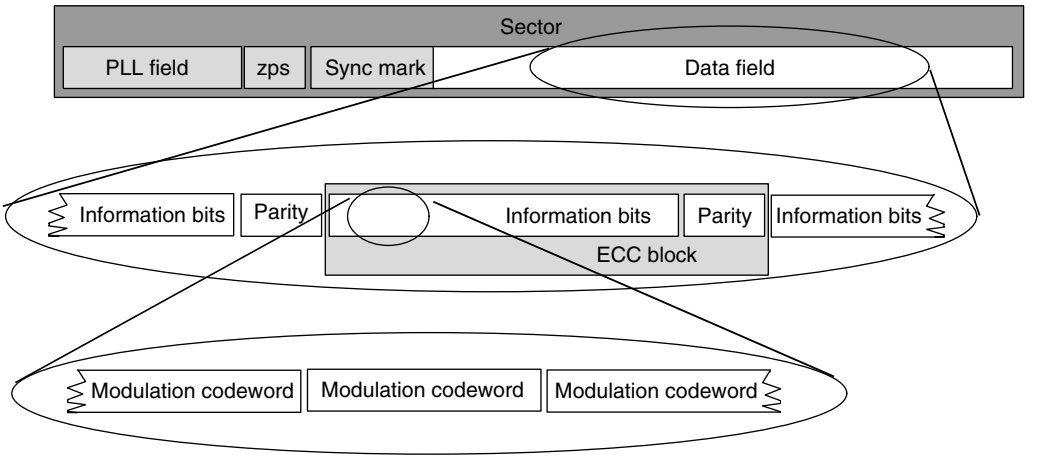



**FIGURE 18.8** Format of data on a disk.

The orientation of magnetization on a disk can be longitudinal, which is typical for today's systems, or perpendicular. The choice of the media influences the way the magnetization is recorded on the disk. Media with needle shaped particles oriented longitudinally tend to have a much higher remanent magnetization in the longitudinal direction, and favor longitudinal recording. The head design must support the favorable orientation of magnetization. Longitudinal orientation requires head shapes that promote longitudinal fields such as ring heads. Similarly, some media are composed of crystallites oriented perpendicularly to the field. Such media have a much higher remanent magnetization in the perpendicular direction, and favor perpendicular recording. If a head design promotes perpendicular fields, such as single pole heads, the result is perpendicularly recorded magnetization.

Some recent experiments have shown that media that favor perpendicular recording have better thermal stability. This is why, lately, perpendicular recording is attracting a considerable attention in magnetic recording community. Typically, in perpendicular recording a recording surface is made of a hard ferromagnetic material, i.e., material requiring large applied fields to permanently magnetize it. Once magnetized, the domains remain very stable, i.e., large fields are required to reverse the magnetization. The recording layer is made thick so that, since each magnetic domain contains a large number of magnetic particles, larger energy is required for demagnetization. The low remanence, low coercivity, materials (the so-called *soft materials*) are placed beneath hard ferromagnetic surface (soft underlayer) and used to conduct magnetic field back to another electromagnet pole. A pole-head geometry is used, so that the medium can effectively travel through the head gap, and be exposed to stronger magnetic field. A pole-head/soft-underlayer configuration can produce about twice the field that a ring head produces. In this way sharp transitions can be supported on relatively thick perpendicular media, and high frequencies (that get attenuated during readback) are written firmly. However, effects of demagnetizing fields are much more pronounced in perpendicular recording systems, because in longitudinal media the transitions are not that sharp.

### 18.1.5 Physical Limits on Recording Density

At extremely high areal densities each bit of information is written on a very small area. The track width is small and magnetic domains contain relatively small numbers of magnetic particles. Because the particles have random positions and sizes, large statistical fluctuations or noise on the recovered signal can occur. The signal-to-noise ratio is proportional to the track width, and is inversely proportional to the mean size of the particle and the standard deviation of the particle size. Therefore, increasing the track size, increasing the number of particles by increasing media thickness, and decreasing the particle

size will improve the signal-to-noise ratio. Uniaxial orientation of magnetic particles also gives higher signal-to-noise ratio; however, the requirement for thermal stability over periods of years dictates a lower limit to the size of magnetic particles in a magnetic domain because ambient thermal energy causes the magnetic signals to decay. Achieving both small particle size and thermal stability over time can be done by using magnetic materials with higher coercivity, but there is a strong practical upper limit to the coercivity that can be written, and it is determined by the saturation magnetization of the head material.

In addition to the basic physics, a number of practical engineering factors must be considered at extremely high densities. In particular, these factors include the ability to manufacture accurately the desired head geometries and control media thickness, the ability to closely follow the written tracks, to control head flying height, and the ability to maintain a very small, stable magnetic separation.

### 18.1.6 The Future

The hard drive areal densities have grown at an annual rate approaching 100%. Recently a 20 Gbit/in.$^2$ has been demonstrated [5], and some theoretical indications of feasibility of extremely high densities approaching 1 Tbit/in.$^2$ have been given [8,9]. Although the consideration related to user needs including higher capacity, speed, error performance, reliability, environment condition tolerances, etc. are important, the factors affecting cost tend to dominate read channel architecture and design considerations. Thus, achieving highest recording density with lowest component costs at high manufacturing yields is the ultimate goal.

With areal densities growing at an annual rate approaching 100%, real concern continues to be expressed that we may be approaching a limit to conventional magnetic recording technology; however, as long as the read channel is concerned, large opportunities are available to improve on the existing signal processing, both with detectors better matched to the channel and by applying more advanced detection, modulation, and coding schemes.

## References

1. S.H. Charrap, P.L. Lu, and Y. He, "Thermal stability of recorded information at high densities," *IEEE Trans. Magn.*, pt. 2, vol. 33, no. 1, pp. 978–983, Jan. 1997.
2. J.D. Coker, et. al., "Implementation of PRML in a rigid disk drive," in *Digest of Magnetic Recording Conf.* 1991, paper D3, June 1991.
3. G.D. Forney and A.R. Calderbank, "Coset codes for partial response channels; or, coset codes with spectral nulls," *IEEE Transactions on Information Theory*, vol. IT-35, no. 5, pp. 925–943, Sept. 1989.
4. H. Kobayashi and D.T. Tang, "Application of partial response channel coding to magnetic recording systems," *IBM J. Res. Dev.*, vol. 14, pp. 368–375, July 1970.
5. M. Madison, et al., "20 Gb/in.$^2$ Using a merged notched head on advanced low noise media," in *MMM Conference*, Nov. 1999.
6. H. Thapar and A. Patel, "A class of partial-response systems for increasing storage density in magnetic recording," *IEEE Trans. Magn.*, vol. MAG-23, pp. 3666–3668, Sept. 1987.
7. H. Osawa, Y. Kurihara, Y. Okamoto, H. Saito, H. Muraoka, and Y. Nakamura, "PRML systems for perpendicular magnetic recording," *J. Magn. Soc. Japan*, vol. 21, no. S2, 1997.
8. R. Wood, "Detection and capacity limits in magnetic media noise," *IEEE Trans Magn.*, vol. MAG-34, no. 4, pp. 1848–1850, July 1998.
9. R. Wood, "The feasibility of magnetic recording at 1 Tbit/in.$^2$," 36 *IEEE Trans. on Magnetics*, vol. 36, no. 1, pp. 36–42, Jan. 2000.

## 18.2 Read Channel Architecture

*Bane Vasić, Pervez M. Aziz, and Necip Sayiner*

The read channel is a device situated between the drive's controller and the recording head's preamplifier (Fig. 18.9). The read channel provides an interface between the controller and the analog recording head, so that digital data can be recorded and read back from the disk. Furthermore, it reads back the head positioning information from a disk and presents it to the head positioning servo system that resides in the controller. A typical read channel architecture is shown in Fig. 18.10. During a read operation, the head generates a pulse in response to magnetic transitions on the media. Pulses are then amplified by the preamplifier that resides in the arm electronics module, and fed to the read channel. In the read channel, the readback signal is additionally amplified and filtered to remove noise and to shape the waveform, and then the data sequence is detected (Fig. 18.10). The data to be written on a disk are sent from a read channel to a write driver that converts them into a bipolar current that is passed through the electromagnet coils. Prior to sending to read channel, user data coming from computer (or from a network in the network attached storage devices) are encoded by an error control system. Redundant bits are added in such a way to enable a recovery from random errors that may occur during reading data from a disk. The errors occur due to number of reasons including: demagnetization effects, magnetic field fluctuations, noise in electronic components, dust and other contaminants, thermal effects, etc. Traditionally, the read channel and drive controller have been separate chips. The latest architectures have integrated them into so called "super-chips."

### 18.2.1 Analog Front End

As a first step, the read signal is normalized with respect to gain and offset so that it falls into an expected signal range. Variation of gain and offset is a result of variations in the head media spacing, variations in magnetic and mechanical and electronic components in the drive, preamplifier and read channel. The front end also contains a thermal asperity (TA) circuit compensation. Thermal asperity occurs when head hits a dust particle or some other imperfection on a disk surface. At the moment of impact, the temperature of the head rises, and a large signal at the head's output is generated. During TA a useful readback signal appears as riding on the back of a low frequency signal of much higher energy.

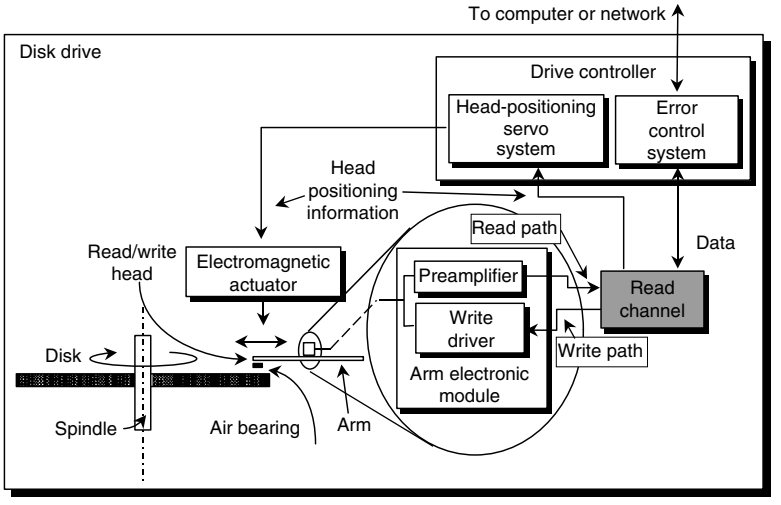



**FIGURE 18.9** The block diagram of a disk drive.

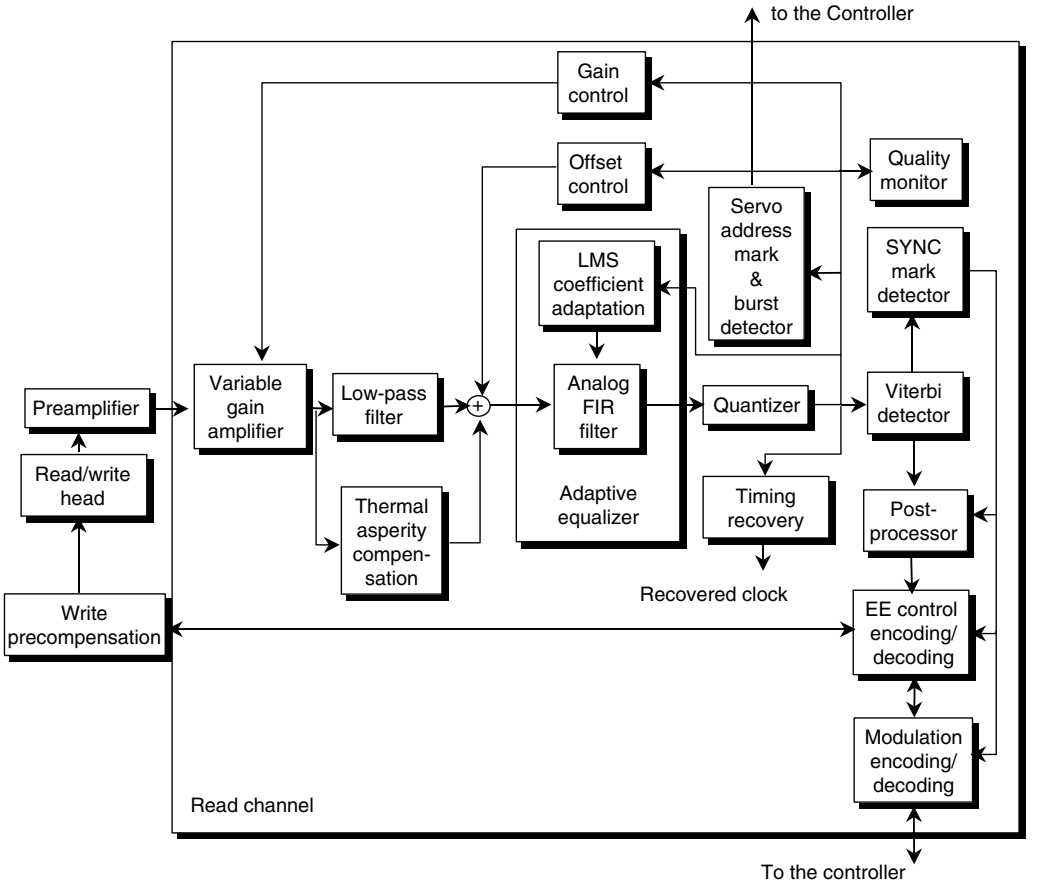



**FIGURE 18.10** A typical read channel architecture.

The beginning of this "background" signal can be easily predicted and the TA signal itself suppressed by a relatively simple filter.

High-frequency noise is then removed with a continuous-time low pass filter to permit a sampling of the signal without aliasing of high-frequency noise back into the signal spectrum. The filter frequently includes programable cut-off frequency, which can be used to shape the signal to optimize data detection. A programmable cut-off frequency is essential since the disk rotates with constant angular velocity, and data rate varies by approximately a factor of two from the inner to outer radius of the disc. It is also important for the analog filter bandwidth to be switched to allow for low cut-off frequencies when processing servo sector information.

## 18.2.2 Precompensation

Nonlinear bit shift in magnetic recording is the shift in position of a written transition due to the demagnetizing field from adjacent transitions. In a PRML channel, the readback waverofm is synchronously sampled at regular intervals, and the sample values depend on the position of written transitions. Therefore, nonlinear bit shift leads to error in sample values which, in turn, degrades the channel performance. The write precompensation is employed to counteract the nonlinear bit shift. However, determining the nonlinear bit shift is not simple and straightforward especially when one tries to fine tune each drive for its optimum precompensation. The precompensation circuit generates the write clock signal whose individual transition timing is delayed from its nominal position by the required

precompensation amount. The amount of precompensation and the bit patterns requiring precompensation can be found using the extracted pulse shape [10,18]. Another approach is a frequency-domain technique that offers simple measurement procedure and a possible hardware implementation using a band-pass filter [32] or using PRML sample values [33].

### 18.2.3 Partial-Response Signaling with Maximum Likelihood Sequence Estimation

After sampling with a rate $1/T$, the read signal is passed through an analog or digital front end filter and detected using a maximum likelihood sequence detector. The partial-response signaling with maximum likelihood (PRML) sequence estimation is proposed for use in magnetic recording by Kobayashi 30 years ago [15,16]. In 1990 IBM produced the first disk drives employing PRML detection. Today's all read channels are based on some version of the PRML. Cideciyan et al. [3] described a complete PRML systems including equalization, gain and timing control, and Viterbi detector. All basic functions of a PRML system have remained practically unchanged, until the introduction of a postprocessor that performs a special type of soft error correction after maximum likelihood sequence detection. Also, significant improvements in all the subsystems have been made during last 10 years. The term "partial response" comes from the fact that the sample of the equalized signal at, say, time $nT$ ($T$ is a signaling interval), contains information not only on data bits at time $nT$, but also on neighboring bits, i.e., magnetic transitions. The number of adjacent bits that determine the sample at $nT$ is referred to as *channel memory*. The channel memory is a parameter that can be selected in the process of defining a read channel architecture. The channel memory and the details of the partial response selection are made based on an attempt to have the partial response be as close a match to the channel as possible. Since the complexity of a maximum likelihood detector is an exponential function of a memory, it is desirable to keep the memory low, but, the equalization required to achieve this might boost the high-frequency noise, which result in decrease of a signal-to-noise ratio, called *equalization loss*. The typical value of channel memory in today's read channels is 4. The value of an equalized sample at time $nT$, $y_n$ can be written as

$$y_n = \sum_{k=0}^{L_h} h_k x_{n-k}$$

where $x_n$ is a user-data bit recorded at time $n$ ($x_n \in \{-1, +1\}$), and $L_h$ is a channel memory. The coefficients $h_k$ form, $h(D) = \sum_{k=0}^{L_h} h_k \cdot D^k$, a *partial response polynomial* or *partial response target* ($D$ is a formal, time-delay variable). The main idea in partial response equalization is to equalize the channel to a known and short target that is matched to the channel spectrum so that noise enhancement is minimum. Therefore, the deliberate inter-symbol interference is introduced, but since the target is known, the data can be recovered, as explained in the previous article.

### 18.2.4 Adaptive Equalization

To properly detect the user-data it is of essential importance to maintain the partial response target during the detection. This implies that channel gain, finite-impulse response (FIR) filter coefficients, and sampling phase must be adaptively controlled in real-time. Continuous automatic adaptations allow the read channel to compensate for signal variations and changes that occur when drive operation is affected by changes in temperature or when the input signals are altered by component aging. Comparing the equalizer output samples with the expected partial response samples generates an error signal, which is used to produce adaptive control signals for each of the adaptive loops. For filter coefficients control, a least-mean square (LMS) algorithm is used [4]. LMS operates in the time domain to find filter coefficients that minimize the mean-squared error between the samples and the desired response. Initial setting of the filter coefficients is accomplished by training the filter with an on-board training sequence,

and the adaptation is continued while chip is reading data. Adaptation can be in principle performed on all coefficients simultaneously at lower clock rate or on one coefficient at a time. Because disk channels variations are slow relative to the data rate, the time-shared coefficient adaptation achieves the same optimum filter response while consuming less power and taking up less chip area. Sometimes, to achieve better loop stability, not all filter coefficients are adapted during reading data. Also, before writing, data are scrambled to whiten the power spectral density and ensure proper filter adaptation.

The FIR filter also compensates for the large amount of group-delay variation that may be caused by a low-pass filter with a nonlinear phase characteristics. Filters with nonlinear characteristics, such as the Butter-worth filter, are preferred over, say, an equi-ripple design of the same circuit complexity, because they have much better roll-off characteristics. The number of FIR filter coefficients in practical read channels has been as low as 3 and as high as 10 with various trade-offs associated with the different choices, which can be made.

### 18.2.5 Viterbi Detection

In many communications systems, a symbol-by-symbol detector is used to convert individual received samples at the output of the channel to corresponding detected bits. In today's PRML channels, a Viterbi detector is a maximum likelihood detector that converts an entire *sequence* of received equalized samples to a corresponding sequence of detected bits. Let $y=(y_n)$ be the sequence of received equalized samples corresponding to transmitted bit sequence $x=(x_n)$. Maximum likelihood sequence estimation maximizes the probability density $p(y|x)$ across all choices of transmitted sequence $x$ [7]. In the absence of noise and mis-equalization, the relationship between the noiseless equalized samples $z_n$ and the corresponding transmitted bits is known by the Viterbi detector and is given by

$$z_n = \sum_{k=0}^{L} h_k x_{n-k} \tag{18.1}$$

In the presence of noise and mis-equalization the received samples will deviate from noiseless values. The Viterbi detector considers various bit sequences and efficiently compares the corresponding expected PR channel output values with those actually received. For Gaussian noise at the output of the equalizer and equally probable input bits, maximizing $p(y|x)$ is equivalent to choosing as the correct bit sequence the one closest in a (squared) Euclidean distance sense to the received samples. Therefore, the following expression needs to be to minimized

$$\min_{x_k} \left( \sum_{n=0}^{P-1} \left[ y_n - \sum_{k=0}^{L} h_k x_{n-k} \right]^2 \right) \tag{18.2}$$

The various components of Eq. 18.3 are also known as branch metrics. The Viterbi detector accomplishes the minimization in an efficient manner using a trellis-based search rather than an exhaustive search. The search is effectively performed over a finite window known as the decision delay or path memory length of the Viterbi detector. Increasing the window length beyond a certain value leads to only insignificant improvements of the bit detection reliability or bit error rate (BER).

Despite the efficient nature of the Viterbi algorithm the complexity of a Viterbi detector increases exponentially with the channel memory of the PR target. A target with channel memory of $L-1$ requires for example a $2^{L-1}$ state Viterbi detector trellis. For a fully parallel Viterbi implementation, each Viterbi state contains an add-compare-select (ACS) computational unit, which is used to sum up the branch metrics of Eq. 18.4 and keep the minimum metric paths for different bit sequences. Also required for the hardware is a $2^{L-1} \cdot P$ bit memory to keep a history of potential bit sequences considered across the finite decision delay window.

### 18.2.6 Timing Recovery

A phase-locked loop (PLL) is used to regenerate a synchronous clock from the data stream. The PRML detector use decision directed timing recovery typically with a digital loop filter. The digital loop filter parameters can be easily controlled using programmable registers and changed when read channel switches from acquisition to tracking mode. Because a significant pipelining is necessary in the loop logic to operate at high speeds, the digital loop filter architecture exhibits a relatively large amount of latency. It can affect considerably the acquisition time when the timing loop must acquire significant phase and frequency offsets. To ensure that only small frequency offsets are present, the synchronizer VCO is phase-locked to the synthesizer during nonread times. For fast initial adjustment of the sampling phase, a known preamble is recorded prior to user data. The time adjustment scheme is obtained by applying the stochastic gradient technique to minimize the mean squared difference between equalized samples and data signal estimates. To compensate for offset between the rate of the signal received and the frequency of the local timing source the loop filter design allows for a factor $\Delta T_n$ to be introduced, so that the sample at discrete time $n$ is taken $T + \Delta T_n$ seconds after the sample at discrete time $n - 1$. In acquisition mode, in order to quickly adjust the timing phase, large values for loop gains are chosen. In tracking mode, the loop gains are lowered to reduce loop bandwidth.

### 18.2.7 Read Channel Servo Detection

In an *embedded* servo system (introduced in the previous article), the radial position of the read head is estimated from two sequences recorded on servo wedges: *track addresses* and *servo-bursts*. The track address provides a unique number for every track on a disk, while a servo-burst pattern is repeated on each track or on a group of tracks. Determining the head position using only the track number is not sufficient because the head has to be centered exactly on a given track. Therefore, the servo-burst waveform is used in conjunction with the track address to determine the head position. Using the servo-burst pattern, it is possible to determine the off-track position of a head with respect to a given track with a high resolution. While positioning the head over a surface, the disk drive can be in either *seeking* or *tracking* operation mode. In a seeking mode, the head moves over multiple tracks, trying to reach the track with a desired address as quickly as possible, while in a tracking mode, the head tries to maintain its position over a track. The track addresses are therefore used mostly in the seeking mode, while servo-burst information is usually used in the tracking mode [25,30].

In read channels, periodic servo-burst waveforms are detected and used to estimate radial position. The radial position error signal is calculated based on the current estimated position and the position of the track to be followed, and then used in an external head positioning servo system. Generally, two types of position estimators are in use: maximum likelihood estimators based on a matched filtering and sub-optimal estimators based on averaging the area, or the peaks, of the incoming periodic servo-burst waveform. A variety of techniques have been used to demodulate servo bursts, including amplitude, phase, and null servo detectors. Today, most read channels use an amplitude modulation with either peak or area detection demodulators.

Older generation channels generally implemented the servo functions in analog circuitry. The analog circuitry of these servo channels partially duplicates functions present in the digital data channel. Now, several generations of read-channel chips have switched from analog to digital circuits and digital signal processing [8,34]. These channels reduce duplication of circuits used for servo and data and provide a greater degree of flexibility and programmability in the servo functions.

Typically, a single analog-to-digital converter (ADC) or quantizer is used for both data datection and servo position error signal estimation [8,20,27,34], but quantizer requirements are different in data and servo fields. Compared to position error signal estimators, data detectors require a quantizer with higher sampling clock rate. On the other hand position error signal estimators require a quantizer with finer resolution. A typical disk drive has a data resolution requirement of around 6 bits, and a servo resolution requirement of around 7 or 8 bits. Furthermore, servo bursts are periodic waveforms as opposed to data streams. In principle, both the lower sampling clock rate requirement in the servo field and the

periodicity property of servo-burst signals can be exploited to increase the detector quantization resolution for position error signal estimation. The servo field is oversampled asynchronously to increase the effective quantizer resolution.

Track densities in today's hard drives are higher than 25,000 tracks per inch, and the design of a tracking servo system is far from trivial. Some of the recent results include [2,24–26]. Increasing the drive level servo control loop bandwidth is extremely important. Typical bandwidth of a servo system is about 1.5 kHz, and is mainly limited by the parameters that are out of reach of a read channel designer, such as mechanical resonances of voice coil motor, arm holding a magnetic head, suspension, and other mechanical parameters.

Another type of disturbance with a mechanical origins, that has to be also detected and controlled in a read channel, is repeatable runout (RRO) in the position of the head with respect to the track center. These periodic disturbances are inherent in any rotating machine, and can be the result of an eccentricity of the track, offset of the track center with respect to the spindle center, bearing geometry, and wear and motor geometry. The frequencies of the periodic disturbances are integer multiplies of the frequency of rotation of the disk, and if not compensated they can be a considerable source of tracking error. In essence the control system possesses an adaptive method to learn online the true geometry of the track being followed, and a mechanism of continuous-time adaptive runout cancellation [31].

### 18.2.8 Postprocessor

Due to the channel memory and noise coloration, maximum likelihood sequence detector (Viterbi detector) produces some error patterns more often than others. They are referred to as *dominant error sequences*, or *error events*, and can be obtained analytically or through experiments and/or simulation. Relative frequencies of error events strongly depend on a recording density.

Parity check processors combine syndrome decoding and soft-decision decoding [35]. Error event likelihoods needed for soft decoding can be computed from a channel sequence by some kind of soft-output Viterbi algorithm. By using a syndrome calculated for a received codeword, a list is created of all possible *positions* where error events can occur, and then error event likelihoods are used to select the most likely position and most likely type of the error event. Decoding is completed by finding the error event position and type. The decoder can make two type of errors: it fails to correct if syndrome is zero or it makes a wrong correction if syndrome is nonzero, but the most likely error event or combination of error events do not produce right syndrome.

A code must be able to detect a single error from the list of dominant error events, and should minimize the probability of producing zero syndrome when more than one error event occur in a codeword.

Consider a linear code given by an $(n-k)\times n$ parity check matrix $H$, with $H$ capable of correcting or detecting dominant errors. If all errors from a list were contiguous and shorter than $m$, then a cyclic $n-k=m$ parity bit code could be used to correct a single error event; however, in reality, the error sequences are more complex, and occurrence probabilities of error events of lengths 6, 7, 8, or more are not negligible. Furthermore, practical reasons (such as decoding delay, thermal asperities, etc.) dictate using short codes, and consequently, in order to keep code rate high, only a relatively small number of parity bits is allowed, making the design of error event detection codes nontrivial.

The detection is based on the fact that we can calculate the likelihoods of each of dominant error sequences at each point in time. The parity bits serve to detect the errors, and to provide some localization in error type and time. The likelihoods are then used to choose the most likely error events (type and location) for corrections. The likelihoods are calculated as the difference in the squared Euclidean distances between the signal and the convolution of maximum likelihood sequence estimate and the channel partial response, versus that between the signal and the convolution of an alternative data pattern and the channel partial response. During each clock cycle, the lowest $M$ are chosen, and the syndromes for these error events are calculated. Throughout the processing of each block, a list is maintained of the $N$ most likely error events, along with their associated error types, positions, and

syndromes. At the end of the block, when the list of candidate error events is finalized, the likelihoods and syndromes are calculated for each of six combinations of two candidate error events which are possible. After disqualifying those pairs of candidates, which overlap in the time domain, and those candidates and pairs of candidates which produced a syndrome, which does not match the actual syndrome, the candidate or pair, which remains and which has the highest likelihood, is chosen for correction.

### 18.2.9 Modulation Coding

Modulation of constrained coding is used to translate an arbitrary sequence of input data to a channel sequence with special properties required by the physics of the medium [21]. Two large important classes of channel constraints are run-length and spectral constraints. The run-length constraints [12] bound the minimal and/or maximal lengths of certain types of channel subsequences, while the spectral constraints include dc-free [11] and higher order spectral-null constraints [6,23]. The spectral constraints also include codes that produce spectral zero at rational sub-multiples of symbol frequency as well as constraints that give rise to spectral lines. The most important class of runlength constraints is a $(d, k)$ constraint, where $d+1$ and $k+1$ represent minimum and maximum number of consecutive like symbols or space between the adjacent transitions. Bounding minimal length consecutive like symbols controls ISI in the excess bandwidth systems and reduces transition noise. Bounding the upper limits of the mark lengths improves timing recovery and automatic gain control. In order to keep code rate high, today's read channels employ only $k$ constrained codes. Typical code rates are: 16/17, 24/25, 32/34, 48/49. Modulation decodes can be either block-by-block or sliding-window. Block decoders determine data word by using a single codeword, while sliding-window decoders require so-called look-ahead, which means that the output data word is a function of several consecutive codewords. Due to inherent nonlinearity, a modulation decoder may produce multiple errors as a result of a single erroneous input bit. If a sliding-window decoding is used, an error can affect several consecutive data blocks. This effect is known as an *error propagation*. The block codes are favored because they do not propagate errors.

A mathematically rigorous code design approach based on symbolic dynamics was developed by Marcus and Siegel et al. [19,22]. The algorithm is known as the "state splitting algorithm" or Adler, Coppersmith, and Hassner (ACH) algorithm [1]. Another constrained coding approach, championed by Immink [14] emphasizes the low-complexity encoding and decoding algorithms [13]. Despite this nice mathematical theory, design of constrained codes remains too difficult to be fully automated, and in the art of designing efficient codes, human intervention and skill are still necessary.

### 18.2.10 Error Control Coding

In a conventional hard disk drives the error control coding (ECC) system does not reside in a read channel; however, the ECC performance is linked to the performance of a detection algorithm, error propagation in a modulation code, and it is natural to try to expand the read channel functionality to error control as well. A new trend in industry is aimed toward designing an integrated circuit, so called *super chip* with a such expanded functionality.

In the most general terms, the purpose of ECC is to protect user data, and this is achieved by including redundant, so-called *parity* bits along with the data bits. The codes used in hard drives belong to a large class of ECC schemes, called block codes. A block code is a set of codewords (or blocks) of a fixed length $n$. The length is expressed in number of symbols, and a symbol is a binary word of length $m$. Other parameters of a block code are $k$-number of data symbols in the block, and $t$-number of symbols correctable by the ECC system [17,36].

Reed–Solomon (RS) codes [28] have been the class of codes most often used in the magnetic disk storage for the last 15 years. The reason is their excellent performance in presence of error events that exhibit burstiness, which is typical for magnetic recording channels, and lend themselves to high-speed encoding/decoding algorithms required for high-speed disk drives [5,9,37]. Very often RS codes are interleaved to reduce effect of long error burst, and to reduce the implementation cost by eliminating

conversion of bytes to possibly longer code symbols used in encoding and decoding. The parameters of RS codes satisfy the following relations: $n \leq 2^m - 1$, number of parity symbols $n - k \geq 2t$, and code rate of the RS code $r = k/n$.

In today's hard drives typically, a part of ECC decoding is performed in hardware with a throughput equal to the data rate, and the other part is performed in firmware with much lower speed. In some cases, such as thermal asperities, no error control is sufficient to recover the data. In this case, it is necessary to retry reading the same sector. A choice between hardware or firmware correction depends on the application, the data transfer protocol, and the bus transfer rate. In applications such as single-user work-stations, short data transfers dominate, but streaming data transfers occasionally occurs (during boot, large file transfers, etc.). Additionally, data read from the disk drive can be transmitted to the host computer in a physically sequential or in any convenient order. If the bus transfer rate is higher than the ECC hardware throughput, and if sufficiently long ECC firmware buffer is available to store all the sectors, or if sectors are transmitted to the host computer in any convenient order all firmware error recovery can be performed in parallel with disk reads without interrupting streaming read operations. In the case of short packet transfers, it is better to perform read retry in parallel with firmware error correction. Retries in conjunction with hardware correction typically consume less time than firmware error correction. On the other hand, for long streaming transfers, correcting errors in firmware in parallel with reading the sector is better strategy, provided that the firmware ECC throughput is high enough to prevent buffer overflow. A detailed treatment of an error control system design considerations can be found in [29].

### 18.2.11 The Effect of Thermal Asperites

As explained earlier, if a head hits a dust particle, a long thermal asperity will occur, producing a severe transient noise burst, loss of timing synchronization, or even off-track perturbation. Error events caused by TAs are much less frequent than random error events, but they exist and must be taken into account during read channel design. If there were no TA protection in the read channel, a loss of lock in timing recovery system would occur, causing massive numbers of data errors well beyond the error correction capability of any reasonable ECC system. Despite TA protection, the residual error cannot be completely eliminated, and many bits will be detected incorrectly; however, the read channel should be designed to enable proper functioning of timing recovery in the presence of bogus samples. Typically, the read channel estimates the beginning and length of TA and sends this information to the ECC system, which may be able to improve its correction capability using so-called erasure information; however, since the TA starting location is not known precisely, and the probability of random error in the same sector is not negligible, the ECC system can misscorrect, which is more dangerous than not to detect the error.

### 18.2.12 Error Performance Measures

A commonly used measure of ECC performance is a BER, which is defined as a ratio of unrecoverable error events and total user data bits. An unrecoverable error event is a block that contains more erroneous symbols than the ECC system can correct, and it may contain as many as exist in a single data block protected by the ECC system. Various applications require different BERs, but they are typically in the range of $10^{-12}$–$10^{-15}$. Another ECC performance measure is undetected bit error rate (UBER), which is a number of undetected error events per total number of user bits. In some cases the ECC system detect that the sector contain errors, but is not able to correct them. Then a controller asks a read channel to retry reading the same sector. The *retry rate per bit* is a useful measure of a data throughput. The hard drive standards of a retry rate is $10^{-14}$. The performance measure used depends on the application. For example UBER is much more important for bank transactions than for multimedia applications ran on PC. All performance measures depend on a symbol length, number of correctable errors, and symbol error statistics. On the other hand symbol error statistics depend on the read channel error event distribution.

## References


1. R.L. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding block codes: An application of symbolic dynamics to information theory," *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 5–22, Jan. 1983.
2. D. Cahalan and K. Chopra, "Effects of MR head track profile characteristics on servo performance," *IEEE Trans. Magn.*, vol. 30, no. 6, Nov. 1994.
3. R.D. Cideciyan, F. Dolivo, R. Hermann, W. Hirt, and W. Schott, "A PRML system for digital magnetic recording," *IEEE J. Sel. Areas in Commun.*, vol. 10, no. 1, pp. 38–56, Jan. 1992.
4. J.M. Cioffi, W.L. Abbott, H.K. Thapar, C.M. Melas, and K.D. Fisher, "Adaptive equalization in magnetic-disk storage channels," *IEEE Comm. Magazine*, pp. 14–20, Feb. 1990.
5. E.T. Cohen, "On the implementation of Reed–Solomon decoders," Ph.D. dissertation, University of California, Berkeley, 1983.
6. E. Eleftheriou and R. Cideciyan, "On codes satisfying *M*th order running digital sum constraints," *IEEE Trans. Inform. Theory*, vol. 37, pp. 1294–1313, Sept. 1991.
7. G.D. Forney, "Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference," *IEEE Trans. Inform. Thoery*, vol. 18, no. 3, pp. 363–378, May 1972.
8. L. Fredrickson et al., "Digital servo processing in the Venus PRML read/write channel," *IEEE Trans. Magn.*, vol. 33, pp. 2616–2619, Sept. 1997.
9. M. Hassner, U. Schwiegelshohn, and S. Winograd, "On-the-fly error correction in data storage channels," *IEEE Trans. Magn.*, vol. 31, pp. 1149–1154, March 1995.
10. R. Hermann, "Volterra model of digital magnetic saturation recording channels," *IEEE Trans. Magn.*, vol. MAG-26, no. 5, 2125–2127, Sept. 1990.
11. K.A.S. Immink, "Spectral null codes," *IEEE Trans. Magn.*, vol. 26, pp. 1130–1135, March 1990.
12. K.A.S. Immink, "Runlength-limited sequences," *Proc. IEEE*, vol. 78, pp. 1745–1759, Nov. 1990.
13. K.A.S. Immink and L. Patrovics, "Performance assessment of DC-free multimode codes," *IEEE Trans. Commun.*, vol. 45, pp. 293–299, March 1997.
14. K.A.S. Immink, *Codes for Mass Data Storage Systems*, Essen, Germany: Shannon Foundation Publishers, 1999.
15. H. Kobayashi and D.T. Tang, "Application of partial-response channel coding to magnetic recording systems," *Bell J. Res. and Develop.*, July 1970.
16. H. Kobayashi, "Correlative level coding and maximum-likelihood decoding," *IEEE Trans. Inform. Theory*, vol. IT-17, pp. 586–594, Sept. 1971.
17. S. Lin, *An Introduction to Error-Correcting Codes*. Englewood Cliffs, NJ: Prentice-Hall, 1970.
18. Y. Lin and R. Wood, "An estimation technique for accurately modeling the magnetic recording channel including nonlinearities," *IEEE Trans. Magn.*, vol. MAG-25, no. 5, pp. 4058–4060, Sept. 1989.
19. R. Karabed and B.H. Marcus, "Sliding-block coding for input-restricted channels," *IEEE Trans. Inform. Theory*, vol. 34, pp. 2–26, Jan. 1988.
20. H. Kimura, T. Nishiya, T. Nara, and T. Komotsu, "A digital servo architecture with 8.8 bit resolution of position error signal for disk drives," in *IEEE Globecom 97*, Phoenix, AZ, 1997, pp. 1268–1271.
21. B. Marcus, P. Siegel, and J.K. Wolf, "Finite-state modulation codes for data storage," *IEEE J. Select. Areas Commun.*, vol. 10, no. 1, pp. 5–37, Jan. 1992.
22. B.H. Marcus, "Sofic systems and encoding data," *IEEE Trans. Inform. Theory*, vol. IT-31, pp. 366–377, May 1985.
23. C. Monti and G. Pierobon, "Codes with multiple spectral null at zero frequency," *IEEE Trans. Inform. Theory*, vol. 35, no. 2, pp. 463–472, March 1989.
24. A. Patapoutian, "Optimal burst frequency derivation for head positioning," *IEEE Trans. Magn.*, vol. 32, no. 5, pt. 1, pp. 3899–3901, Sept. 1996.
25. A. Patapoutian, "Signal space analysis of head positioning formats," *IEEE Trans. Magn.*, vol. 33, no. 3, pp. 2412–2418, May 1997.

26. A. Patapoutian, "Analog-to-digital converter algorithms for position error signal estimators," *IEEE Trans. Magn.*, vol. 36, no. 1, pt. 2, pp. 345–400, Jan. 2000.
27. D.E. Reed, W.G. Bliss, L. Du, and M. Karsanbhai, "Digital servo demodulation in a digital read channel," *IEEE Trans. Magn.*, vol. 34, pp. 13–16, Jan. 1998.
28. I.S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Indust. Appl. Math.*, vol. 8, pp. 300–304, 1960.
29. C.M. Riggle and S.G. McCarthy, "Design of error correction systems for disk drives," *IEEE Trans. Magn.*, vol. 34, 4 pt. 2, pp. 2362–2371, July 1998.
30. A.H. Sacks, "Position signal generation in magnetic disk drives," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1995.
31. A.H. Sacks, M. Bodson, and W. Messner, "Advanced methods for repeatable runout compensation [disc drives]," *IEEE Trans. Magn.*, vol. 31, no. 2, pp. 1031–1036, March 1995.
32. Y. Tang and C. Tsang, "A technique for measuring nonlinear bit shift," vol. 27, no. 6, pp. 5326–5318, Nov. 1991.
33. Y. Tang, R.L. Galbraith, J.D. Coker, P.C. Arnett, and R.W. Wood, "Precompesation value determination in a PRML channel," *IEEE Trans. Magn.*, vol. 32, no. 3, pp. 2013–1014, May 1996.
34. G.T. Tuttle et al., "A 130 Mb/s PRML read/write channel with digital-servo detection," in *Proc. IEEE Int. Solid State Circuits Conf. 96*, San Francisco, CA, Feb. 8–10, 1996, pp. 64–65.
35. J.L. Sonntag and B. Vasic, "Implementation and bench characterization of a read channel with parity check post processor," TMRC 2000, Santa Clara, CA, Aug. 2000.
36. S.B. Wicker, *Error Control Systems for Digital Communication and Storage*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
37. D.L. Whiting, "Bit-serial Reed-Solomon decoders in VLSI," Ph.D. dissertation, California Inst. Tech., Pasadena, 1984.

## 18.3 Adaptive Equalization and Timing Recovery

*Pervez M. Aziz*

Adaptive signal processing plays a crucial role in storage systems. Proper detection of the readback signal by a Viterbi detector assumes that the signal has the right gain, is equalized to the partial response, and is sampled at the proper sampling instances. In this section, the focus is mainly on equalization and timing recovery. Some of the basic algorithms employed in equalization and timing recovery are reviewed. Various architectures and algorithms are presented, which have been used in state-of-the-art read channels. Finally, comparative performance data for some of these architectures are presented.

### 18.3.1 Adaptive Equalization

What is equalization? It is the act of shaping the read back magnetic recording signal to look like a target signal specified by the partial response (PR). The equalized signal is made to look like the target signal in both the time and frequency domain. In this subsection, various equalization architectures and strategies are reviewed, which have been popular historically and still being used in present day read channels. A quick review of the well-known least mean square (LMS) algorithm used for adaptive equalizers is also provided. Finally, the performance implications of selecting several different equalizer architectures is explored. This performance is measured in terms of bit error rate (BER) at the output of the read channel's Viterbi detector.

#### 18.3.1.1 Equalization Architectures and Strategies

In PRML channels the read back signal will be sampled at some point in the data path for further digital signal processing. A continuous time filter (CTF) with a low-pass characteristic will be present as an antialiasing filter [1] prior to the sampling operation so that high-frequency noise is not aliased into the

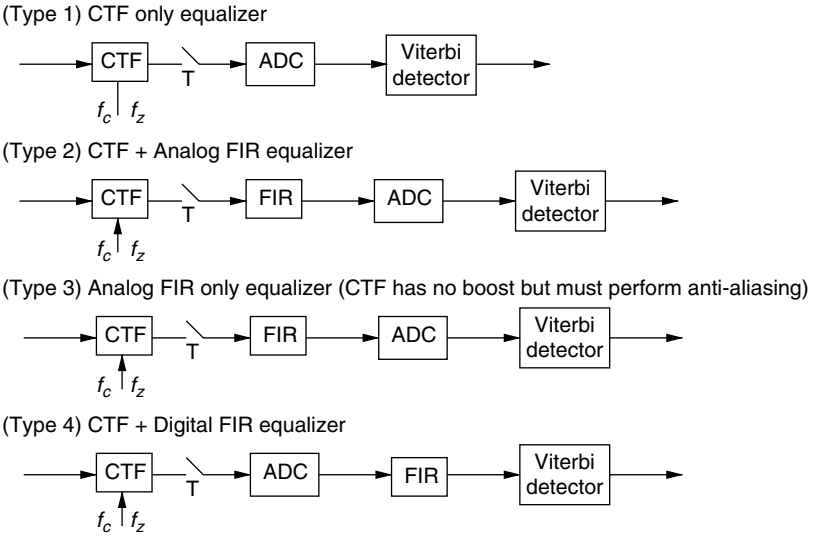



**FIGURE 18.11** Various equalizer architectures.

signal band. This same CTF may also play a role in equalizing the read back signal to the target partial response. Various architectures can be used to perform the required equalization. The equalizer architecture can consist of a CTF, a finite impulse response filter (FIR), or both. The CTF parameters may be fixed, programmable, or adaptive. The FIR filter coefficients may be fixed, programmable, or adaptive. In addition, the FIR operation may occur in the sampled data analog domain or digital domain. Following equalization, the data are detected using a Viterbi detector. Of course, quantization by an analog-to-digital converter (ADC) occurs at some point before the Viterbi detector.

Figure 18.11 shows some examples of various equalizer architecture configurations. The first architecture (Type 1) consists of a CTF-only equalizer. The CTF is comprised of an all-pole low-pass filter section whose purpose is to reject high-frequency noise for anti-aliasing. One key parameter in the CTF is its low-pass bandwidth determined by its cutoff or corner frequency, $f_c$. The type of CTF, $f_c$, and its order (or the number of poles it contains) will determine its low-pass rolloff characteristic. If the CTF is expected to take part in equalization, it must also be able to provide some boost and does so by typically having one or two real zeros at some frequency $f_z$ in its transfer function. These parameters are noted in the figure.

The second architecture (Type 2) is one where both the CTF and an analog FIR are involved in performing equalization. The third architecture (Type 3) is an analog FIR-only architecture in that the CTF design does not consist of any zeros, i.e., its main role is to perform anti-aliasing and not provide any boost for equalization. Finally, the last architecture (Type 4) is one where a CTF and FIR are both involved in equalization except that the FIR operation is done digitally.

In general, there is a clear trade-off between the degree of flexibility of the equalizer and implementation complexity. The read-back signal characteristics change across the disk surface as manifested by somewhat different channel bit densities (*cbds*) or $pw_{50}/T$. Consequently, programmability of the equalizer parameters is a mimimum requirement for optimum performance. The signal or some of the equalizer parameters themselve may change with chip ageing and temperature variations [2]. Therefore, it is often desirable for some of the equalizer parameters to be continually adaptive to be able to compensate for these effects.

### 18.3.1.2 CTF Configurations

Two common types of CTFs, which have been used in read channels are Butterworth filters and equiripple linear phase filters. Butterworth filters have a maximally flat magnitude response but

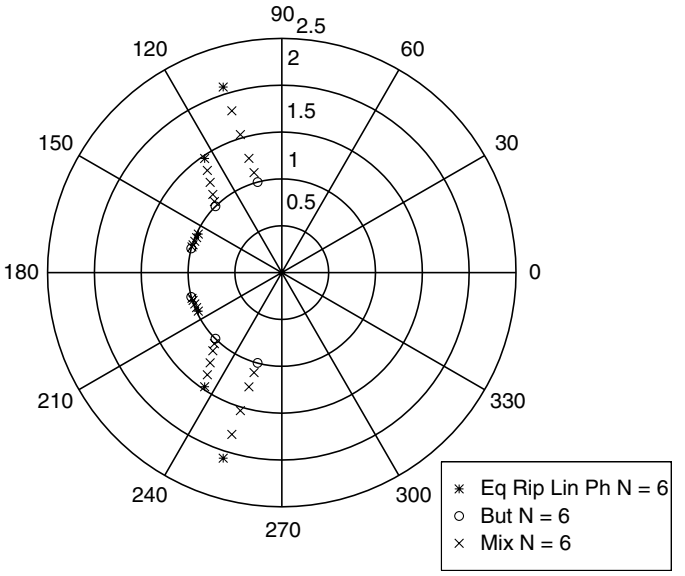



**FIGURE 18.12** Normalized pole locations for sixth-order Butterworth and equiripple linear phase filters as well as mixed filters.

nonlinear phase response. Equiripple linear phase filters, as their name implies, have linear phase and constant group delay over the passband [3,4]. For a given order, the Butterworth filters will have sharper rolloff characteristics. One could also consider *mixed* filters whose poles are chosen to lie some percentage of the distance in between the poles of a Butterworth and equiripple linear phase filter. Figure 18.12 shows the normalized pole location on the *s* plane for a sixth-order Butterworth, a sixth-order equiripple linear phase filter, as well as the poles for various sixth order mixed filters, which are 25%, 50%, 75%, and 90% away from the poles of the equiripple filter. Note that the Butterworth poles lie on the unit circle. Figure 18.13 shows the corresponding magnitude responses for the filters, while Fig. 18.14 shows the group delay responses. As can be observed, the Butterworth has the sharpest lowpass rolloff and the equiripple filter has the shallowest rolloff but constant group delay over the passband.

The CTF parameters can be programmable or adaptive [5,6]; however, most CTFs that have been used in actual read channels have had programmable bandwidth and boosts any adaptivity being left to the FIR. Adaptive CTF systems face some challenging issues as discussed in [7]. Also, some work has been done to analytically determine the optimum CTF transfer functions [8,9].

The performance of equalizers involving several CTF configurations will be compared: fourth-order Butterworth (*b4*), sixth-order Butterworth (*b6*), seventh-order equiripple linear phase (*e7*) all with single zeros. We also examine a seventh-order equiripple linear phase CTF with two zeros (*e7tz*). The linear phase of the all pole section is kept in the *e7tz* filter. Another filter considered is the fourth-order 50% mixed filter with one zero (*em4*).

### 18.3.1.3 FIR Filter and the LMS Algorithm

The FIR filter is important for equalization. Whether implemented as an analog sampled data filter or a digital filter the FIR filter produces output samples $y(n)$ in terms of input samples $x(n)$ as

$$y(n) = \sum_{k=0}^{L-1} h(k)x(n-k) \tag{18.3}$$

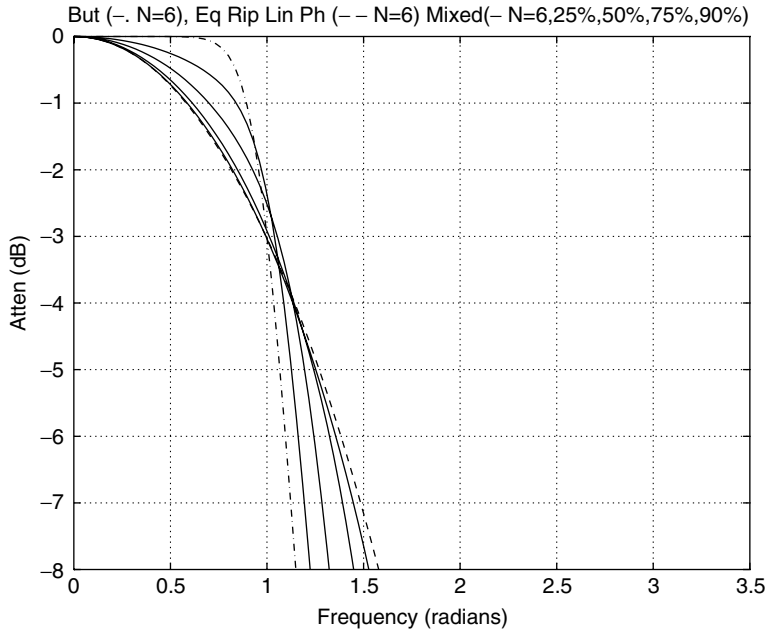



**FIGURE 18.13** Magnitude response for sixth-order Butterworth and equiripple linear phase filters as well as mixed filters.

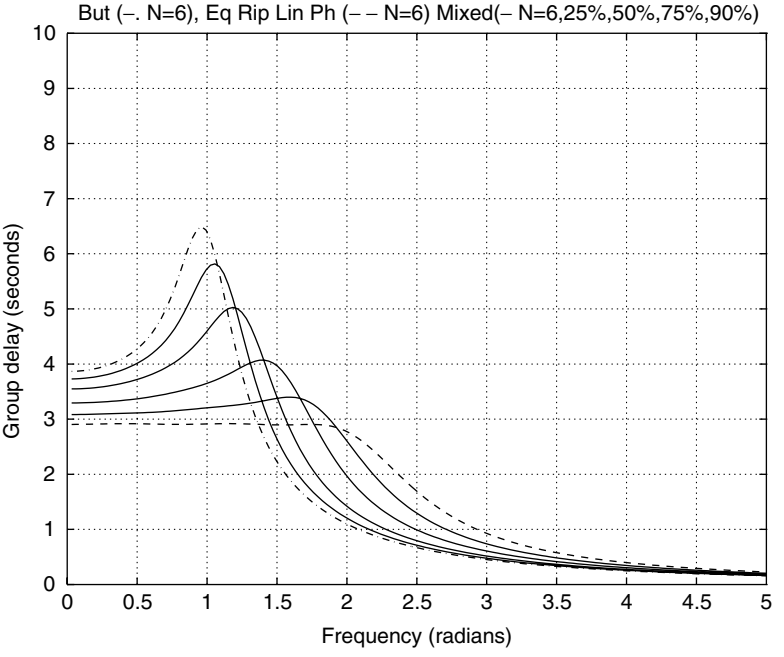



**FIGURE 18.14** Group delay response for sixth-order Butterworth and equiripple linear phase filters as well as mixed filters.

where $h(k)$ are the FIR filter tap weights. As noted, it is very desirable for the FIR to be adaptive. The FIR taps are adapted based on the well-known LMS algorithm [10,11]. Other adaptive algorithms can also be found in [12] and [13].

The basic idea is to minimize the mean squared error with respect to some desired or ideal signal. Let the desired or ideal signal be $\hat{y}(n)$ in which case the error is $e(n) = y(n) - \hat{y}(n)$. This minimization is achieved by adjusting the tap value in the direction opposite to the derivative (with respect to the tap values) of the expected value of the mean squared error. Dispensing with the expected value leads to the LMS or stochastic gradient algorithm. The stochastic gradient for the $k$th tap weight is

$$\begin{aligned}\Delta(k,n) &= -\frac{\partial}{\partial h(k)}\left(e^2(n)\right) = -2e(n)\frac{\partial \hat{e}(n)}{\partial h(k)} = -2e(n)\left[\frac{\partial y(n)}{\partial h(k)} - \hat{y}(k)\frac{\partial \hat{y}(n)}{\partial h(k)}\right] \\ &= -2e(n)\frac{\partial y(n)}{\partial h(k)} \end{aligned} \tag{18.4}$$

where the partial derivative of $\hat{y}(n)$ with respect to $h(k)$ is zero. We can now expand $y(n)$ as in Eq. 18.3 to further obtain

$$\Delta(k,n) = -2e(n)x(n-k) \tag{18.5}$$

The gradient would actually be scaled by some tap weight update gain $t_{ug}$ to give the following tap update equation:

$$h(k,n+1) = h(k,n) - 2t_{ug}e(n)x(n-k) \tag{18.6}$$

The choice of this update gain depends on several factors: (a) it should not be too large so as to cause the tap adaptation loop to become unstable, (b) it should be large enough that the taps converge within a reasonable amount of time, (c) it should be small enough that after convergence the adaptation noise is small and does not degrade the BER performance. In practice, during drive optimization in the factory the adaptation could take place in two steps, initially with higher update gain and then with lower update gain. During the factory optimization different converged taps will be obtained for different radii on the disk surface. Starting from factory optimized values means that the tap weights do not have to adapt extremely fast and so allows the use of lower update gains during drive operation. Also, this means that the tap weights need not all adapt every clock cycle—instead a round-robin approach can be taken, which allows for sharing of the adaptation hardware across the various taps. A simpler implementation can also be obtained by using the signed LMS algorithm whereby the tap update equation is based on using 2- or 3-level quantized version of $x(n-k)$. For read channel applications, this can be done without hardly any loss in performance.

A few other issues should be emphasized about the adaptive FIR. During a read event, the FIR filter is usually adapted after the initial gain and timing recovery operations are performed over a preamble field. Nevertheless, during the rest of the read event, the FIR filter equalizes the signal at the same time that the gain and timing loops are operating. The adaptive gain loop uses an automatic gain control (AGC) block to apply the correct gain to the signal to achieve the desired partial response target values. Likewise the adaptive timing recovery loop works to adjust the sampling phase to achieve the desired PR target values. It is necessary to minimize the interaction between these adaptive loops. The FIR filter will typically have one tap as a "main" tap, which is fixed to minimize its interaction with the gain loop. Another tap such as the one preceeding or following the main tap can be fixed (but allowed to be programmable) to minimize interaction with the timing loop [14]. In some situations it may be advantageous to have additional constraints to minimize the interaction with the timing loop [15].

#### 18.3.1.4 Performance Characterization

The performance of various equalizer architectures based on bit error rate simulations can now be characterized. The equalizer types (with reference to Fig. 18.11) actually simulated are of Types 2 (CTF + analog FIR) and 3 (anti-aliasing CTF + analog FIR). One can consider the case where there are very few taps as an approximation of the Type 1 (CTF only) equalizer. Although many actual read channel architectures do use digital FIRs (Type 4), we do not consider this type for simulations here. Although a digital FIR filter may be cost effective for implementation given a particular technology, it does have two disadvantages compared with the analog FIR. With the analog FIR, quantization noise is added *after* equalization and so is not enhanced through the equalizer whereas for the digital FIR the quantization noise does pass through the equalizer and could be enhanced. Consequently, fewer quantization levels can be used and this results in reduced chip area and power dissipation with the analog FIR. Also, the digital FIR is likely to have more latency in producing its final output and this extra latency may not hurt significantly but is nonetheless not beneficial for the timing loop.

##### *18.3.1.4.1 Simulation Environment and Optimization Procedure*

The simulation environment, including two system simulation models, and the optimization methodology by which the optimum performance is obtained for each equalizer architecture can now be described. Finally, BER results quantifying the performance of the various architectures are presented.

To obtain a simulation bound for the performance of the best possible equalizer we use the system of Fig. 18.15. The signal + noise is fractionally sampled at a rate of $T/5$ and filtered with a fractionally spaced FIR filter equalizer, which equalizes the signal to an EPR4 target. The channel bit period is $T$. The output of the equalizer is then sampled at the channel bit rate of $T$ and these samples are presented to the EPR4 Viterbi. The FIR has 125 taps (spanning $25T$). The FIR tap weights are adapted from zero starting values using the LMS algorithm. There is no quantization, AGC, or timing recovery. Therefore, the performance is solely determined by the noise.

Pseudo random data are 16/17 modulation encoded to generate a signal at various *cbds* based on a Lorentzian pulse shape. For each *cbd*, the SNR needed by the "ideal" $T/5$ oversampled system of Fig. 18.15 to produce a BER of $10^{-5}$ is determined. SNR is defined as the ratio of the isolated pulse peak to rms noise in the Nyquist band. The (*cbd*, SNR) pairs are used for performing simulations with the practical system of Fig. 18.16, which accurately models an actual read channel chip and a version of the $T/5$ system where the equalized samples are quantized before being detected with the Viterbi detector. Signals at several *cbds* or $PW50/T$ values 1.9, 2.4, and 2.8 are examined. The SNRs needed for 1e-5 BER for these densities are 21.66, 22.90, and 24.70 dB, respectively.

Let us now describe the simulation model for the actual read channel system. A block diagram of this system is shown in Fig. 18.16. The system consists of AGC, CTF, $T$ rate sampled analog FIR equalizer, ADC quantizing the equalizer output, and an EPR4 Viterbi detector. Three decision directed adaptive loops are used: LMS tap update loop, AGC loop, and digital PLL (DPLL) timing recovery loop. Note that

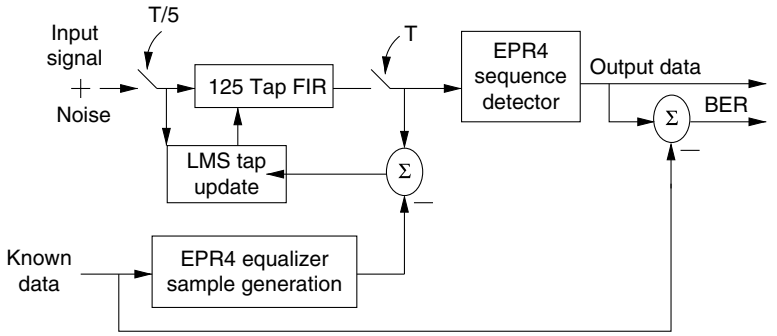



**FIGURE 18.15** Block diagram of system with five times oversampled equalizer.

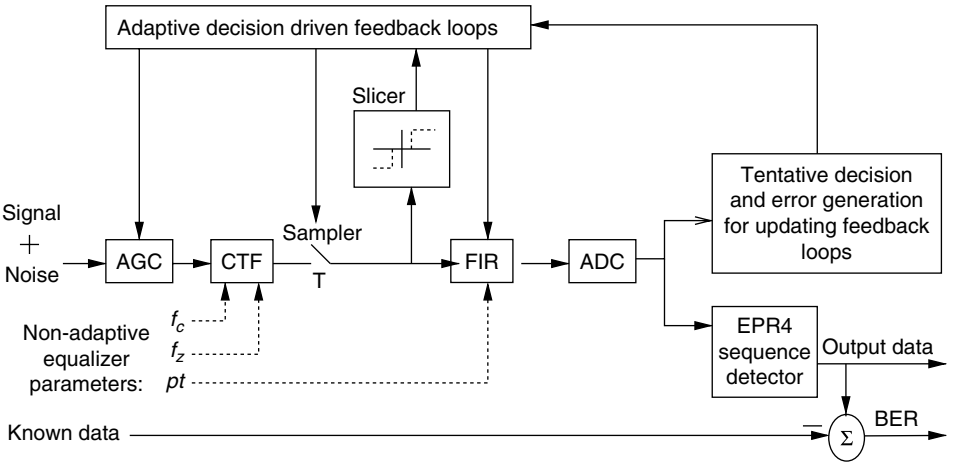



**FIGURE 18.16** Block diagram and simulation model of practical symbol rate sampled read channel system.

in this practical system the adaptive feedback loops are updated not based on known data but on tentative or preliminary decisions made by the read channel. The algorithm used for the tap update is the signed LMS algorithm as implied by the 3-level slicer shown in the figure.

Using the practical system model, BER simulations are performed for the various CTFs mentioned earlier and FIRs of various number of taps. The simulations are performed with this realistic system using the SNRs mentioned earlier. This allows the calculation of the BER degradation of the realistic system with respect to the $T/5$ system for a given *cbd* and SNR.

For each CTF type, *cbd*, and FIR length we simulate the BER of the system across a space of equalizer parameters $f_c$, $f_z$ (which determines CTF boost), and the fixed programmable tap of the FIR, which is labeled as *pt* in Fig. 18.16. The parameters are varied across the following ranges: $f_c$ is varied between 20% and 38% of the channel bit rate, $f_z$ is varied to provide boosts between 2.6 and 8.6 dB, while the programmable tap is varied between 40% and 60% of the main tap value. For CTFs with two zeros the zeros are adjusted such that the total boost is in the above range. For the 10-tap FIR the fourth tap is chosen to be the fixed main tap while for the 6- and 3-tap filters the second tap is chosen as the main tap. For the other taps, the analog tap range was kept to be relatively large at $\pm 80\%$ of the main tap value. In a real system, one would choose smaller ranges based on tap settings fitting into the smaller range, which produced good results. For the equalizer configuration involving the FIR only equalizer, FIRs with 4–20 taps are examined. The programmable tap *pt* is re-optimized for each *cbd* and FIR filter length.

#### *18.3.1.4.2 Results*

Before comparing BER results across different equalizers, some results from the equalizer optimization procedure are illustrated. Figure 18.17 shows a contour plot of the BER obtained with the *b*4 CTF with a 10-tap FIR at a *cbd* of 2.4 and a SNR, which gives close to $10^{-5}$ BER. The horizontal axis is CTF corner frequency ( $f_c$) and the vertical axis is CTF boost in decibel. The plot is for one particular value of the programmable FIR tap *pt*. The numbers on the contour plot are $10 \times \log 10(\text{BER})$ so that $10^{-5}$ BER would correspond to 100. We observe that good BERs result for a large range of boosts and range of $f_c$'s centered in the plot. Upon examining contour plots for all the CTFs, we concluded that the *b*4 CTF achieves good BERs for $f_c$'s typically in the center of the range explored, while the *b*6 CTF the performance is better at somewhat higher $f_c$'s; however, the linear phase CTFs achieve good BERs at very low $f_c$'s. This is because CTFs with worse rolloff characteristics require a smaller $f_c$ to provide enough attenuation at the Nyquist frequency for anti-aliasing. We observe that the BER performance is mostly insensitive to the boost. This is because the adaptive FIR is able to provide any remaining

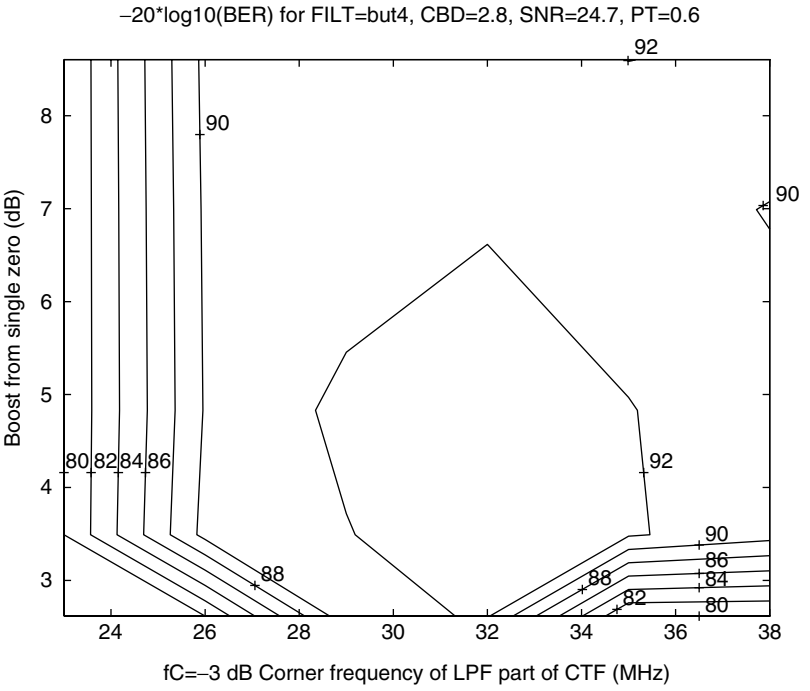



**FIGURE 18.17** Boost bandwidth optimization.

equalization needed. In practice, there will be a trade-off in how much boost the CTF is able to provide and the corresponding analog tap ranges required by the FIR to provide any necessary remaining equalization—the more equalization the FIR has to provide, the larger tap ranges it will require.

Now compare the BER performance of various Type 2 (CTF + analog FIR) equalizers. Figure 18.18 shows the BER performance of different CTFs with a 10-tap FIR. The horizontal axis is *cbd* and the vertical axis is the BER degradation (in dB) of the optimum BER with respect to the BER of $10^{-5}$ achieved by the ideal oversampled system using the $T/5$ equalizer. The performance of the CTFs is similar across all *cbds*—they perform within 0.15 dB of one another. All perform within 0.25–0.4 dB of the $10^{-5}$ BER achieved by the $T/5$ system. The linear phase of the seventh-order CTFs does not necessarily yield superior performance. A final comment is needed about the plot—one should not expect a fixed or monotonic relationship between *cbd* and the practical system BER in this plot. This is due to the finite resolution of the equalizer optimization search and the fact that BERs are based on observing 100 (vs. even larger number) bit errors.

As noted, the previous results were with a 10-tap FIR. Further simulations of the various CTFs with a 6-tap or even 3-tap show that the *optimum* BER performance is not very different than that with a 10-tap FIR. These results are presented in Fig. 18.19 where the BER degradation (again with respect to the $10^{-5}$ achieved by the ideal system) of the optimum BER obtained for the various CTFs is plotted versus the number of FIR taps for $cbd = 2.4$. This initially appears to be a surprising result, but this is not unreasonable when one observes that with fewer number of taps, a large percentage of the CTF programmings result in poor BER performance. This effect is shown in Fig. 18.20, which plots the percentage of CTF programmings (with respect to the total number of programmings producing convergent tap weights) producing BER worse than $4 \times 10^{-5}$. With more taps the percentage of poor programmings decreases. Thus, FIR filters with a few taps, with appropriately optimized CTF settings, can perform as well as a FIR with 10 taps; however, the difficulty in keeping the nonadaptive CTF parameters correct in the presence of realistic device conditions makes such a FIR with few taps impractical to use.

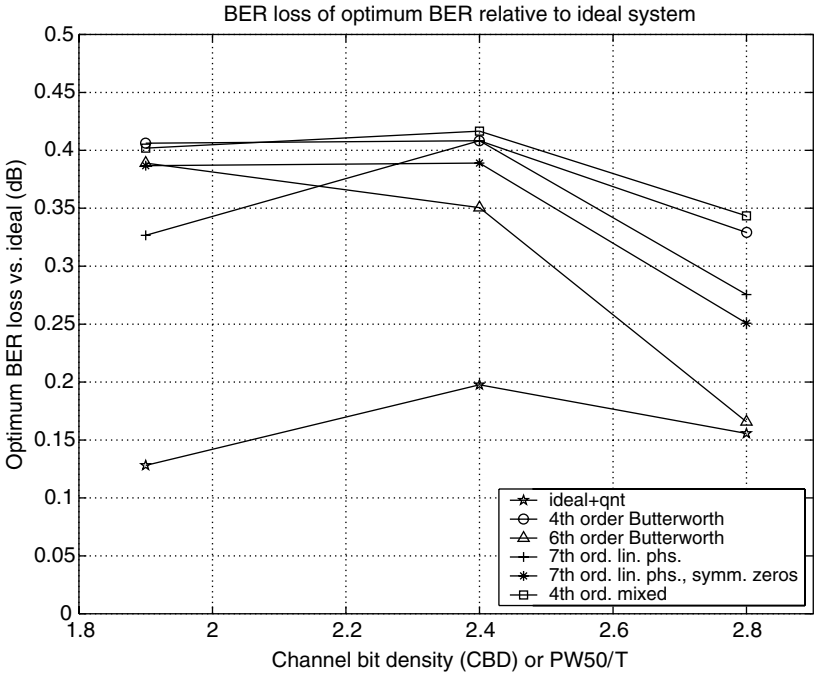



**FIGURE 18.18** CTF BER performance degradation with respect to oversampled ideal system vs. *cbd* with 10-tap FIR.

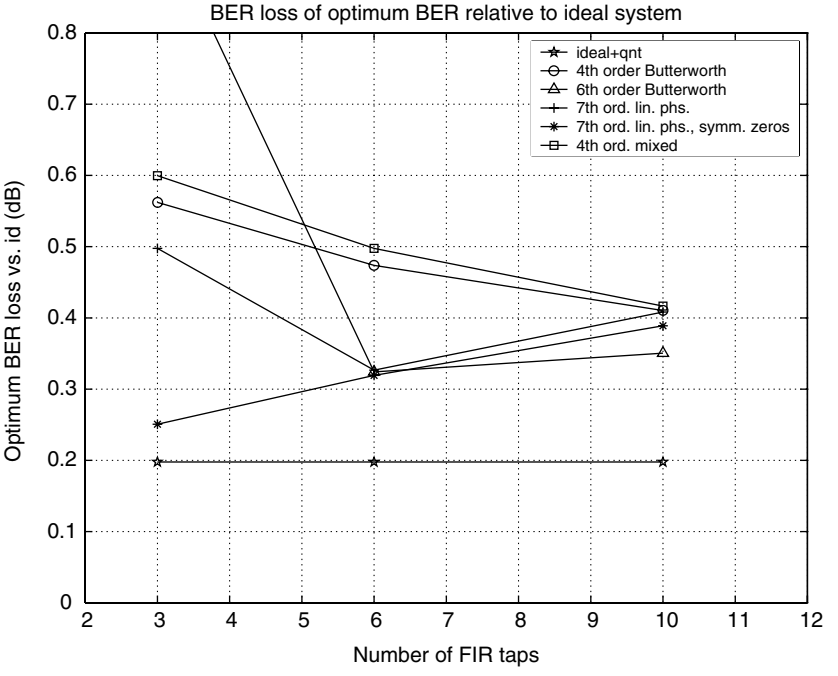



**FIGURE 18.19** CTF BER degradation with respect to oversampled ideal system vs. number of taps ($cbd = 2.4$).

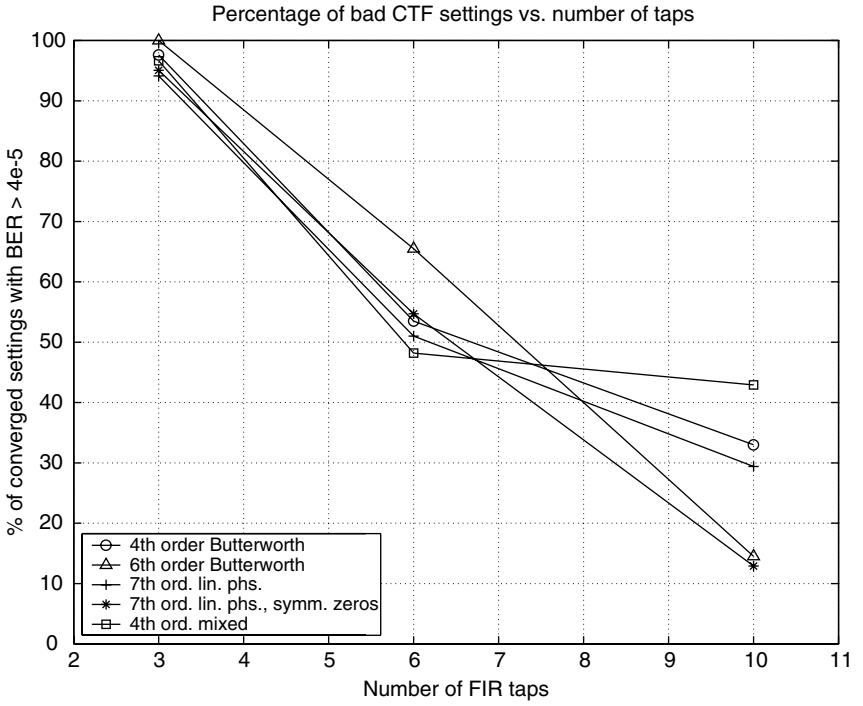



**FIGURE 18.20** Percentage of bad CTF settings vs. number of taps ($cbd = 2.4$).

Finally, examine the performance of Type 3 equalizers. Here, the anti-aliasing CTF is a seventh-order linear phase filter. The performance of this equalizer is a function of the number of FIR taps. The BER performance are shown in Fig. 18.21. The vertical axis is again the degradation with respect to the ideal

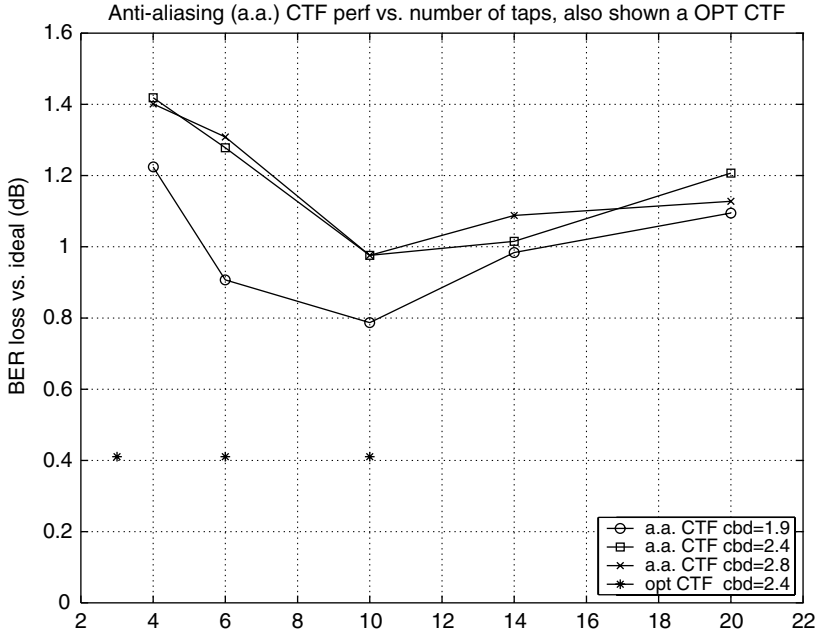



**FIGURE 18.21** BER degradation with respect to oversampled ideal system vs. number of taps with anti-aliasing CTF (no boost).

system BER of $10^{-5}$. The programmable tap of the FIR is optimized to yield the best performance in each case. The main tap is placed roughly in the center. There is benefit in increasing the number of taps from 4 to 6 to 10. Beyond 10 taps, however, there is more latency in the timing loop as the main tap position is more delayed. This causes increased phase errors to enter the timing loop and outweighs the benefit of enhanced equalization obtained with more taps. Although one could increase the number of taps while keeping the main tap location mostly fixed, the FIR will then not be able to cancel the precursor ISI as well with a CTF, which is not involved in equalization. Also shown (dashed plot) is the performance of a Type 2 equalizer (CTF, with its corner frequency optimized and with an optimized zero included to provide boost). Clearly the Type 2 equalizer outperforms the Type 3 equalizer.

#### 18.3.1.5 Actual Equalizer Architectures

Various equalization architectures and examined their performance have been considered. Let us now examine what actual architectures read channel vendors are using. Table 18.1 summarizes some of the most commonly used architectures. For example, Agere Systems (*Note:* storage products unit of AT&T was spun off to Lucent Technologies in 1996 and again spun off to Agere Systems in 2001) has been using a Type 2 architecture with a fourth-order Butterworth CTF and 10-tap analog FIR. The CTF has a programmable corner frequency and zero for providing boost. This architecture is still in place now. Most other vendors have used Type 4 architectures (digital FIR) but with seventh-order equiripple linear phase filters. The linear phase filters typically have two programmable zeros to provide boost. In the examples of the Cirrus equalizers, the digital FIR does not appear to be adaptive. Some vendors such as Cirrus and Marvell seem to have increased the number of FIR taps or the number of adaptive (vs. only programmable) taps as the years have gone by. The Datapath equalizer cited is one of the few examples of an all CTF equalizer.

#### 18.3.1.6 Conclusions

The performance of various CTF + adaptive analog FIR (Type 2) equalizers in equalizing a signal to an EPR4 target has been quantified. It is shown that regardless of the number of taps in the FIR and CTF type, the BER performance of the CTF + FIR equalizers is approximately the same if the optimum fixed equalizer parameters (CTF corner frequency, boost, FIR fixed tap) are chosen.

Therefore, the choice of CTF type should be based on other constraints such as area, power, speed (data rate), as well the benefit of having one less analog block. It has also been shown that as the number of taps is increased, the space of CTF parameter programmings producing BERs close to the optimum increases significantly. Therefore, one can trade-off the cost of the FIR filter versus required accuracy in the CTF setting and the sensitivity of the resulting performance.

The performance of Type 3 equalizers consisting of a $T$ spaced FIR filter with only a Nyquist antialiasing CTF was also examined. Also, the Type 3 equalizer cannot approach the performance of a system whose CTF is involved in equalization and is optimized. Therefore, to make a valid comparison between FIR and CTF equalizers, one must include a reasonably optimum CTF prior to the FIR.

It has been demonstrated that a wide variety of optimized CTF + FIR equalizers can perform within 0.25 dB of the quantized system using the oversampled $T/5$ equalizer. As this 0.25 dB includes

**TABLE 18.1** Examples of Equalizers Implemented on Read Channel Chips

| | CTF | | | FIR | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Company | Type | Order | Zeros | Taps | Adaptive? | Analog/ Digital | Type (Fig. 18.11) | Ref/Yr | Comments |
| Agere | But | 4th | 2 | 10 | yes | analog | 2 | [16], 1995 | 8 adaptive taps |
| Cirrus | EqRip | 7th | 2 | 3 | no | digital | 2 | [17], 1995 | — |
| Cirrus | EqRip | 7th | 2 | 5 | no | digital | 2 | [18], 1996 | — |
| Datapath | EqRip | 7th | 2 | N/A | N/A | N/A | 1 | [19], 1997 | No FIR |
| Marvell | EqRip | 7th | ? | 7 | yes | digital | 2 | [20], 1997 | — |

performance losses due to AGC and timing recovery, there is very little space left for improved equalization with any other equalizer architecture.

## 18.3.2 Adaptive Timing Recovery

In storage systems such as PRML magnetic recording channels a clock is used to sample the analog waveform to provide discrete samples to symbol-by-symbol (s/s) and sequence (Viterbi) detectors. Improper synchronization of these discrete samples with respect to those expected by the detectors for a given partial response will degrade the eventual BER of the system. The goal of adaptive timing recovery is to produce samples for the s/s or sequence detector, which are at the desired sampling instances for the partial response being used. In this subsection, the basics of timing recovery as well as commonly used algorithms for timing recovery in magnetic recording channels are reviewed. Two classes of timing recovery algorithms are presented: symbol rate VCO-based and interpolation-based algorithms. After a discussion of the trade-offs between these two types of algorithms, the focus will be on the traditional symbol rate VCO algorithms for the rest of the discussion. One of these timing recovery algorithms from first principles will be derived. An analytical framework for comparing the performance of such algorithms using timing loop noise induced output jitter as the performance criterion is provided. Finally, quantitative comparative performance data for some of these algorithms based on the jitter analysis as well as simulations, which measure timing loop jitter and BER, is provided.

### 18.3.2.1 Timing Recovery Basics

#### *18.3.2.1.1 Symbol Rate VCO versus Interpolative Timing Recovery*

Timing recovery schemes, which have been considered for magnetic recording channels, can be broadly classified into two groups: traditional symbol rate VCO-based schemes and interpolative schemes, [21,22], which sample slightly above the symbol rate. The key difference between the schemes is that the symbol rate VCO scheme adapts or adjusts the phase and frequency of the sampling clock to produce the desired samples whereas interpolative timing recovery samples the analog waveform using a uniformly sampled clock to produce samples from which the desired samples are interpolated.

Figure 18.22 shows high-level block diagrams of both approaches. Let us describe the VCO-based approach first. For the sake of the discussion the VCO approach is shown with an analog FIR equalizer. Consequently the sampling occurs at the input of the FIR equalizer. The noisy equalized output $y(k)$ must be used to detect the timing error present in these samples. This is done using a phase detector. The phase detector transforms an amplitude error in the samples to $\Delta(k)$, which is related to the desired change in the sampling phase. The phase detector output is also called the timing gradient.

The phase detector may require the use of the noisy equalized samples $y(k)$ or other signals derived from it. The other signals may be the preliminary or tentative decisions $\hat{d}(k)$s, decision directed estimates of $y(k)$, which are $\hat{y}(k)$ or other signals. These auxiliary signals are generated by the block labeled "Signal Generation for Phase Detector." The $y(k)$s are used to generate preliminary (tentative) decisions $\hat{d}(k)$ and an error signal $e(k)$, and a decision directed estimate of the ideal equalized sample value, $\hat{y}(k)$.

The phase detector output is filtered by a loop filter $T(z)$. The loop filter $T(z)$ is usually a second order DPLL with an additional delay term $z^{-L}$, which models any latency through the timing loop. Such latency arises from the group delay of the FIR, computations in the DPLL, calculating the signals needed by the phase detector, etc. The filtered phase detector output is the input to a VCO, which causes the actual sampling phase $\tau$ to change. The VCO is an analog circuit under digital control. The analog part can be a phase mixer, which is capable of adjusting the timing phase by small fractions of $T$ where $T$ is the channel bit period. In such a case, the VCO acts as an amplitude to time converter and so modeled as a simple gain. To give physical meaning to the system, the units of the signals are noted: the equalized output after quantization by the ADC is in amplitude units of LSBs, the timing gradient $\Delta(k)$ or phase detector output is proportional to an amplitude error and so is also in LSBs. The loop filter provides a

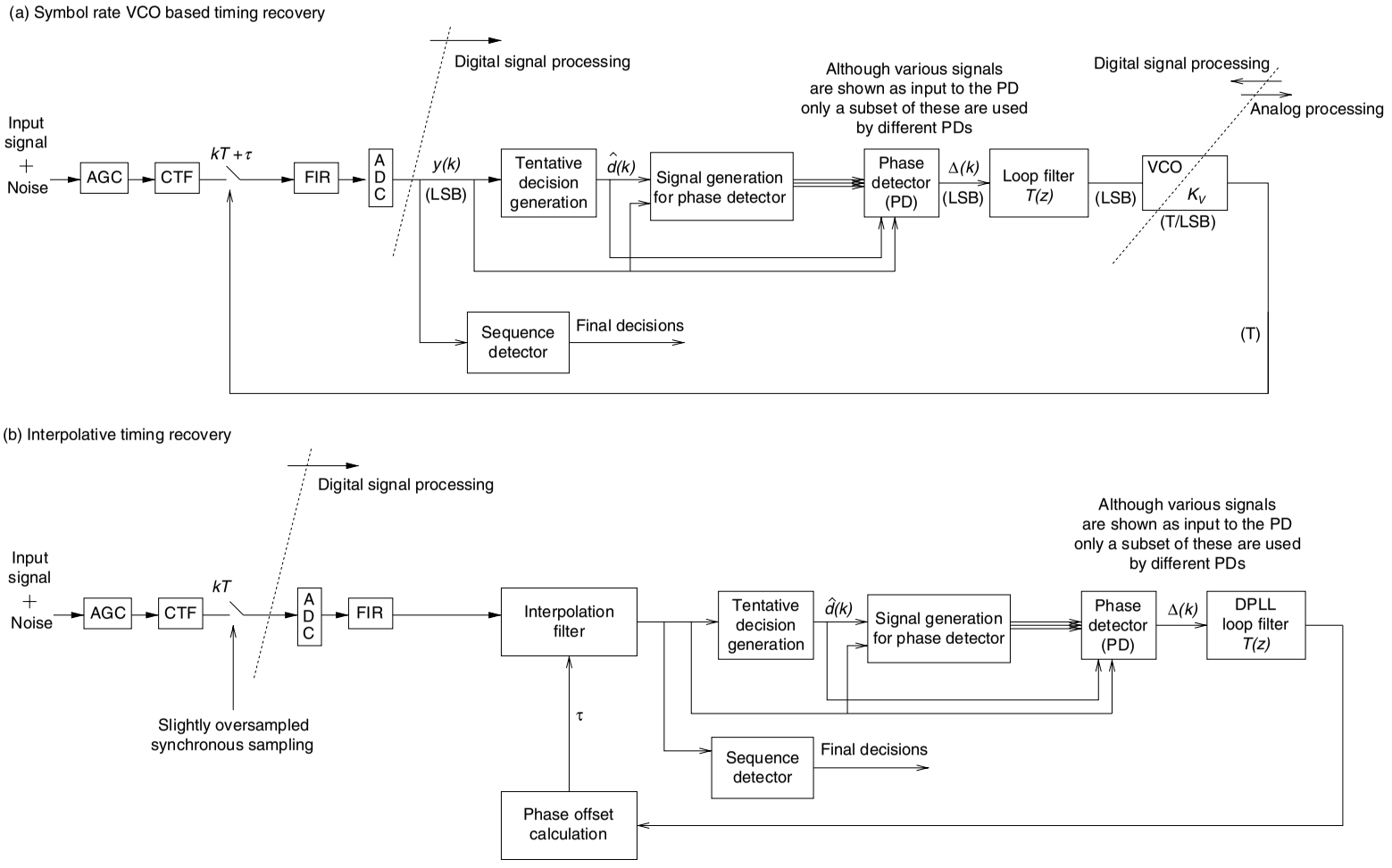



**FIGURE 18.22** (a) Symbol rate VCO-based timing recovery loop. (b) Interpolative timing recovery loop.

frequency dependent gain, so the input of the VCO is LSBs. The VCO has a gain of $K_v$ in units of T/LSB, so the output of the VCO has units of time, $T$. The VCO gain can also be thought of as a clock update gain. For the specific system we will consider later, the phase mixer can make changes in the sampling phase in steps of 0, ±1, ±2 $T/64$ or more. The choice of this factor of 64 is such that the quantization of timing phase adjustment is well below the ADC quantization noise floor.

Let us now describe the interpolative timing recovery loop of Fig. 18.22. As noted, with this scheme, an asynchronous clock is used to sample the input to the ADC after which a FIR filter performs the necessary equalization. The asynchronous equalized samples are now used to interpolate samples at the correct sampling instances dictated by the partial response. This is done with the interpolation filter, which can be thought of as a filter which delays its input by an amount $\tau$, which is a fraction of the channel bit period $T$ [21]. Such an interpolation filter's transfer function is $z^{-\tau}$. The samples $y(k)$ at the output of the interpolation filter drive the phase detector and loop filter as in the VCO-based timing loop. The loop filter output after being processed by the phase offset calculator produces the required sampling phase change. For good operation, the loop must be able to produce a large number of fractional delays (such as 32 or 64) and correspondingly would require as many such filters for each of these delays. Figure 18.22 noted that the asynchronous sampling was performed at slightly above the Nyquist rate. The reasons for this is to accomodate a frequency offset between the written signal and the clock used to perform the asynchronous sampling. The magnitude of this frequency offset is usually limited in practical systems to 1% or less and so very little oversampling is required; however, oversampling ratios of up to 5% produce some improvement in performance by reducing the sensitivity of the aliasing with respect to the phase of the asynchronous sampling clock.

The advantages of the ITR-based timing loops are that they are all digital timing loops, which are more amenable for design, verification, and less susceptible to process variations. Also, for the ITR timing loop, the delays in the equalization filter and ADC do not contribute to the timing loop latency; however, the interpolation filter is not an extremely small piece of hardware and could make the ITR timing loop consume more chip area and power than a VCO-based loop. Practical design issues with the ITR-based system such as adaptation of the equalizer based on asynchronous samples [22] and design of the interpolation filter, have not been discussed. From a performance point of view, there is no significant difference between the ITR- or VCO-based approaches as indicated by simulation results in [21]. This also seems reasonable based on our observation in the subsection on adaptive equalization where it was noted that a read channel system with practical equalization and timing recovery performed within a few tenths of a decibel of the corresponding "ideal" system. Therefore, the choice between all digital ITR-based system or a conventional VCO-based system needs to be based on the relative merits of both systems from an ease of design and area/power standpoint.

#### *18.3.2.1.2 Timing Loop Modes*

Let us now further describe the operation of the entire timing loop. The entire timing recovery process occurs in several steps: zero phase start (ZPS), acquistion mode (ACQ), and tracking mode (TRK). During the ZPS and ACQ modes the disk controller must guarantee that the read channel is reading a preamble signal known to the timing loop. The preamble signal for almost all magnetic recording channels is a $2T$ pattern, which is the periodic data sequence "...11001100..." The purpose of the ZPS is to obtain a good estimate of the initial phase error between the readback signal and the desired samples for the $2T$ pattern. Once this estimate is obtained the sampling clock's phase is changed by the calculated amount to approximate the desired sampling phase. The next step is the ACQ process where the sampling phase error is further reduced and the frequency offset between the input signal and the sampling clock is compensated for to produce even more accurately sampled preamble samples. Because the preamble is a known signal pattern, timing recovery is facilitated in that the preliminary decisions can be obtained more reliably with less loop latency. Consequently, high loop filter update gains can be used. Once this initial acquisition is complete, the timing loop transitions into a TRK, which is intended for tracking slow variations in timing. In this mode the signal may contain any excess preamble as well as random data, but no a priori assumption about the signal is made. The tentative

| Mode name | ZPS | Acquisition (ACQ) | Tracking (TRK) |
|---|---|---|---|
| Assumed data | Preamble | Preamble | Excess preamble/ regular random data |
| Loop filter gains | Zero | High/ medium | Medium/ low |

**FIGURE 18.23** Timing loop operational modes: zero phase start, acquisition, and tracking.

decisions in the TRK mode are obtained with more loop latency and are not as reliable. The loop filter update gains are correspondingly lower. A summary of the operation described is provided in Fig. 18.23. More fine gradations of the loop filter gains (beyond the high/medium/low gains shown in Fig. 18.23 can be made across ACQ and TRK to produce improved performance [23]. Of course, there is a trade-off between improved performance and somewhat enhanced circuit complexity so that one would choose to increase the complexity only, until diminishing returns in performance is reached.

### 18.3.2.2 Symbol Rate Timing Recovery Schemes

Now consider in more detail the traditional symbol rate VCO-based schemes. A decision directed baud or symbol rate timing recovery algorithm was first proposed by Mueller and Muller [24]. Their technique relied on the concept of a "timing function." $f(\tau)$, which generates the proper amount of timing phase adjustment for a given phase shift, $\tau$, in the signal. The function should be monotonic, and have a zero output for zero sampling phase error. The Mueller and Muller (MM) technique provides a means to derive a timing function from a linear combination of samples of the channel's impulse response. In practice, one can design timing gradients where the expected value equals the suitably defined timing function. The timing gradients can be used to obtain the corresponding phase adjustment signal. In some magnetic recording systems using a PR4 target, a MM timing gradient with a second order DPLL was used to produce the necessary timing phase updates [8,25].

One can also derive timing recovery schemes based on other criteria such as the minimum mean square error (MMSE) criterion. MMSE methods seek to minimize the expectation of the square of an error signal $e(k,\tau)$ with respect to the timing phase. The error signal is obtained by subtracting the received equalized samples $y(k,\tau)$ from the corresponding "ideal" samples $\hat{y}(k)$. The minimization is done by adjusting the timing phase in the direction opposite to the derivative of the expected value of the squared error. In practice, one ignores the expected value and minimizes the squared error resulting in a stochastic gradient algorithm. MMSE timing recovery has been proposed in [26] and examined to some degree in [27] for PR magnetic recording channels. Another criterion, the maximum likelihood (ML) criterion, has also been used to derive a phase detector [28].

The derivation of the MMSE gradient is reviewed, and note that the MMSE gradient yields suitable timing functions. Also formulated is the MMSE timing recovery in the framework of a slope lookup table (SLT) instead of a discrete time filtered version of symbol rate spaced equalized samples $y(k,\tau)$. The SLT approach leads to an efficient implementation with slopes expressed directly in terms of a discrete time filtered version of the data bits $d(k)$ instead of the equalized signal samples.

A methodology for an analytical performance evaluation of the timing loop where the timing loop output noise jitter is the performance criterion. The analysis is described in detail for the SLT-based MMSE timing loop and also applied to the MM timing loop. The quantitative results from this technique are used to compare the SLT and MM timing loops. The ML loop is not considered further here as it has somewhat adverse jitter properties compared with the other two timing loops [29]. Finally, simulations results comparing the SLT and MM timing loops in terms of output noise jitter as well as BER performance are presented.

#### *18.3.2.2.1 MMSE Slope Lookup Table Timing Recovery*

Let us review MMSE timing recovery from first principles. The discussion is along the lines of [26] and [27]. The expectation of the square of the error, $e(k, \tau) = y(k, \tau) - \hat{y}(k)$, is minimized with respect to the timing or sampling phase. Here,

$$\hat{y}(k) = \sum_{p=0}^{p-1} h(p)\hat{d}(k-p) \tag{18.7}$$

and in the absence of any channel impairments we would have $\hat{d}(k) = d(k)$ and $\hat{y}(k) = y(k)$. The derivative of the expectation needs to be obtained with respect to $\tau$. Ignoring the expectation operator we obtain a stochastic gradient algorithm [26]:

$$\begin{aligned}\frac{\partial}{\partial\tau}\left(e^2(k,\tau)\right) &= 2y(k,\tau)\frac{\partial y(k,\tau)}{\partial\tau} - 2\hat{y}(k)\frac{\partial y(k,\tau)}{\partial\tau} = -2e(k,\tau)\frac{\partial y(k,\tau)}{\partial\tau} \\ &= -2e(k,\tau)\left[\frac{dy(t)}{dt}\right]_{t=kT+\tau} = -2e(k,\tau)s(k,\tau)\end{aligned} \tag{18.8}$$

Note that the MM approach was to generate a timing gradient from a suitably defined timing function $f(\tau)$. Here, a timing gradient has been derived from the MMSE criterion; however, the resulting timing gradient should be a valid timing function, i.e., be monotonic, and have a zero-crossing for zero sampling phase error. This has been shown in [27]. An expression for the timing function in terms of the PR channel coefficients is [29]

$$f(\tau) = \sum_{l=-\infty}^{\infty}\left[\underbrace{\sum_{p=0}^{P-1}}_{l\neq p} h(p)\frac{(-1)^{l-p}}{(l-p)^T}\right]^2 \tag{18.9}$$

The result of plotting $f(\tau)$ in Eq. 18.9 for EPR4 is shown in Fig. 18.24. Let us now consider a MMSE-based timing gradient or phase detector formulated in terms of a SLT. The signal slope is modeled in terms of a slope generating filter, which when used to filter the data $d(k)$ produces the slopes:

$$s(k) = d(k) \times \psi(k) = \sum_{c=-C_1}^{C_2} \psi(c)d(k-c) \tag{18.10}$$

where the negative coefficient index indicates that the slope at time $k$ depends on future bits (accomodated by delaying the slope and adding the delay into the model as additional latency). $C_1 + C_2 + 1$ is the number of nonzero coefficients of the slope filter's impulse response, $\psi$. The SLT output $\hat{s}(k)$ approximates $s(k)$, which depends on the data pattern. Such a SLT can be derived for any PR by correlating the data with the actual signal slopes. In practice, it is enough to use fewer terms from the filter. Therefore, the simplified SLT output can be represented as

$$\hat{s}(k) = \sum_{b=-B_1}^{B_2} \psi(b)d(k-b) \tag{18.11}$$

where $B = B_1 + B_2 + 1$ is the size of the slope table input i.e., the number of data bits used in calculating the slope. The SLT-based gradient is then,

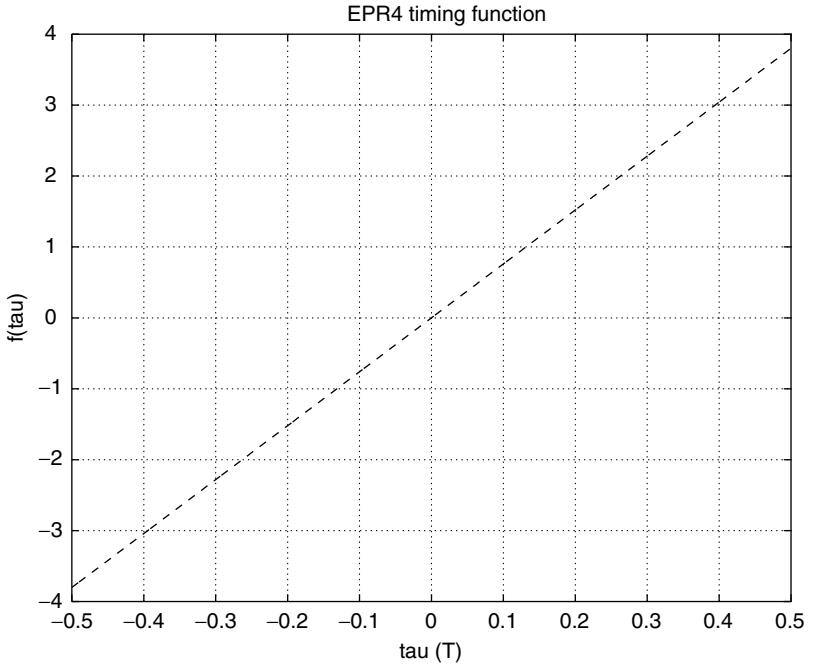



**FIGURE 18.24** Timing function for an EPR4 partial-response channel.

$$\Delta(k) = e(k)\hat{s}(k) \tag{18.12}$$

where the factor of $-2$ in Eq. 18.8 can be absorbed in the lookup table. In our analysis we need the slope generating filter coefficients $\psi(c)$. These coefficients $\psi(c)s$ are obtained in the process of numerically generating the signal slopes, which are correlated with the data.

*Phase Detector Properties*—Before computing the output noise jitter of the entire timing loop, the properties of the phase detector must be analyzed. Quantities important for the performance of the phase detector are its *KPD* and output noise standard deviation ratio $KPD/\sigma_{n_o}$. The *KPD* is the ratio of the mean phase detector output to a constant sampling phase error, $\tau$. The *KPD* can thus be thought of as the signal gain of the timing loop where the signal is the sampling phase error. The output noise $n_o(k)$ is the equivalent noise at the output of the phase detector for a given input noise $n(k)$ at the phase detector input. The error, $e(k)$, at the equalizer output is a combination of contributions from the sampling phase error, $\tau(k)$ and noise. Let $n(k)$ represent the noise at the equalizer output (intersymbol interference + filtered equalized noise). We then have,

$$e(k) = \tau(k)s(k) + n(k) \tag{18.13}$$

The phase detector output, $\Delta(k)$, is then

$$\Delta(k) = [\tau(k)s(k) + n(k)]\hat{s}(k) = \tau(k)s(k)\hat{s}(k) + n_o(k) \tag{18.14}$$

Figure 18.25 shows in detail the timing loop of Fig. 18.22 with the details of the SLT phase detector and the composition of the error signal from the sampling phase and noise per Eq. 18.13.

Now find the statistical properties of *KPD* and $n_o$ using $\boldsymbol{\epsilon}$ as the expectation operator. For a tractable analysis we assume $n(k)$ is AWG. To easily relate $\sigma_n$ to the error event rate (EER) at the output of the Viterbi detector, we assume that channel errors are dominated by a minimum distance error event (with distance $d_{\min}$).

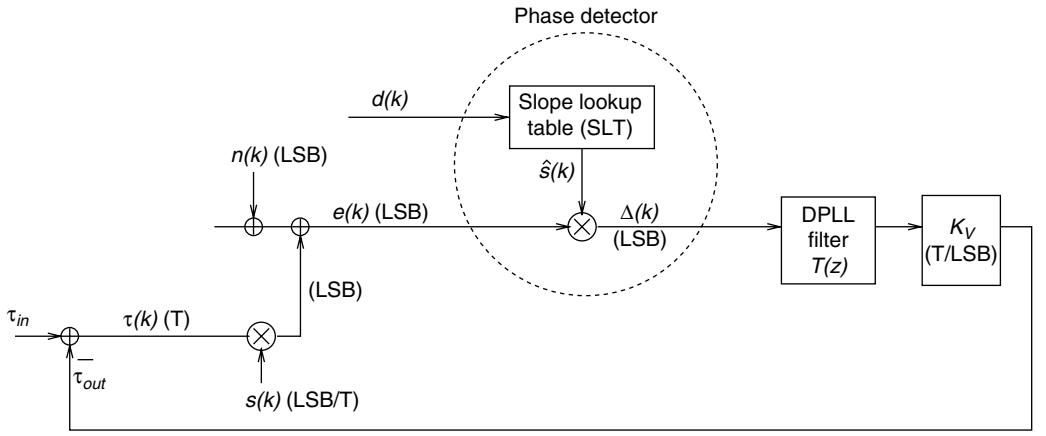



**FIGURE 18.25** Timing loop with SLT phase detector.

$$\sigma_n = \frac{d_{\min}/2}{Q^{-1}(\text{EER})} \tag{18.15}$$

The EER is the BER divided by the number of bit errors in the dominant error event. In Eq. 18.15 $Q$ refers to the well-known $Q$ function defined by

$$Q(x) = \frac{1}{2\pi}\int_x^\infty \exp\left(\frac{y^2}{2}\right)dy \tag{18.16}$$

*Signal Gain (KPD) of the Phase Detector*—Using the definition of *KPD*, for a constant sampling phase error $\tau$,

$$KPD = \frac{\epsilon\{\tau\hat{s}(k)s(k) + n(k)\hat{s}(k)\}}{\tau} = \epsilon\{\hat{s}(k)s(k)\} + \frac{\epsilon\{n(k)\hat{s}(k)\}}{\tau} \tag{18.17}$$

Consider $\epsilon\{n(k)\hat{s}(k)\}$, where $\hat{s}(k)$ is a linear function of the data bits, which can be realistically assumed to be uncorrelated with the noise $n(k)$. Therefore, this term is zero and as we should expect, the noise does not contribute to the mean phase detector output. Thus,

$$KPD = \epsilon\{\hat{s}(k)s(k)\} = \sum_{b=-B_1}^{B_1}\sum_{c=-C_1}^{C_2-1}\psi(b)\psi(c)\epsilon\{d(k-b)d(k-c)\} \tag{18.18}$$

If $d$ is uncoded, hence white, with zero mean, $\epsilon\{d(k-b)\ d(k-c)\} = \sigma_d^2$ if $b = c$ and is 0 if $b \neq c$. Consequently, the *KPD* is

$$KPD = \sigma_d^2 \sum_{b=-B_1}^{B_2}\Psi^2(b) \tag{18.19}$$

where it is assumed that slope table ouput is based on fewer than $C_1 + C_2 + 1$ terms to reduce the summation to be from $b = -B_1$ to $B_2$. We note that the *KPD* values obtained here are equivalent to the slopes of the $f(\tau)$ versus $\tau$ curve plotted in Fig. 18.24.

*Output Noise of the Phase Detector*—Computing the autocorrelation,

$$\epsilon\{n_0(k)n_0(k+l)\} = \epsilon\{n(k)\hat{s}(k)n(k+1)\hat{s}(k+l)\}$$

Because $\hat{s}(k)$ is a filtered version of $d(k)$, which is uncorrelated with $n$, $n$ and $\hat{s}$ are uncorrelated. Therefore,

$$\begin{aligned}\epsilon\{n_0(k)n_0(k+l)\} &= \epsilon\{n(k)n(k+l)\}\epsilon\{\hat{s}(k)\hat{s}(k+l)\} \\ &= R_n(l)\epsilon\{\hat{s}(k)\hat{s}(k+l)\}\end{aligned}$$

$$\epsilon\{\hat{s}(k)\hat{s}(k+l)\} = \sum_{b=-B_1}^{B_2}\sum_{b'=-B_1}^{B_2}\psi(b)\psi(b')\epsilon\{d(k-d)d(k+l-b')\} \tag{18.20}$$

With $d$ being uncoded (hence white) and zero mean, $\epsilon\{d(k-b)d(k+l-b')\} = \sigma_d^2$ if $b' = b+l$ and 0 if $b' \neq b+l$. Also assuming, $R_n(l) = \sigma_n{}^2\,\delta[l]$, i.e., $n$ to be white, we need to consider only $l=0$ in which case we have $b=b'$. In that case,

$$\epsilon\{n_0(k)n_0(k+l)\} = \sigma_n^2\sigma_d^2\delta[l]\sum_{b=-B_1}^{B_2}\psi^2(b) \tag{18.21}$$

Observe that the noise at the phase detector output is indeed white with standard deviation,

$$\sigma_{n_0} = \sigma_n\sigma_d\sqrt{\sum_{b=-B_1}^{B_2}\psi^2(b)} \tag{18.22}$$

#### *18.3.2.2.2 Mueller and Muller Timing Loop*

Now examine the properties of the MM timing gradient. This gradient is obtained as

$$\Delta(k) = y(k)\hat{y}(k-1) - y(k-1)\hat{y}(k) \tag{18.23}$$

in terms of the equalized signal $y(k)$ and its delayed version as well as the corresponding estimates of the "ideal" values $\hat{y}$ for these signals. A block diagram of a MM timing loop using this gradient is shown in Fig. 18.26. It is possible to evaluate this phase detector's *KPD* and noise performance. This is accomplished by writing $y(k)$ as $\hat{y}(k) + e(k)$, expanding $e(k)$ as in Eq. 18.13 from which $s(k)$ is further expressed in terms of the slope generating filter based on Eq. 18.10. Likewise, $\hat{y}(k)$ is expressed in terms of the PR coefficients as per Eq. 18.7. The analysis makes the usual assumptions about the data and noise $n(k)$ being white. The details of the analysis can be found in [29] which yields,

$$KPD = \sigma_d^2\left(\underbrace{\sum_c\sum_m}_{m-c=-1}\psi(c)h(m) - \underbrace{\sum_c\sum_m}_{m-c=-1}\psi(c)h(m)\right) \tag{18.24}$$

where the sum over $m$ is from 0 to $P-1$ and that over $c$ is from $-C_1$ to $C_2+1$.

The autocorrelation, $R_{n_o}(l)$ for the noise at the output of the phase detector, assuming the data to be white, is also computed in [29]. It is shown that even with AWG noise at the phase detector input, i.e., noise with autocorelation $R_n(l) = \sigma_n{}^2\,\delta[l]$, noise at the phase detector output is not white; however, it is shown that if $R_n(l) = \sigma_d^2\,\delta[l]$ the autocorrelation of $R_{n_o}(l)$ will be limited to only the first delay terms, i. e., $l=1$ and $-1$ so we have,

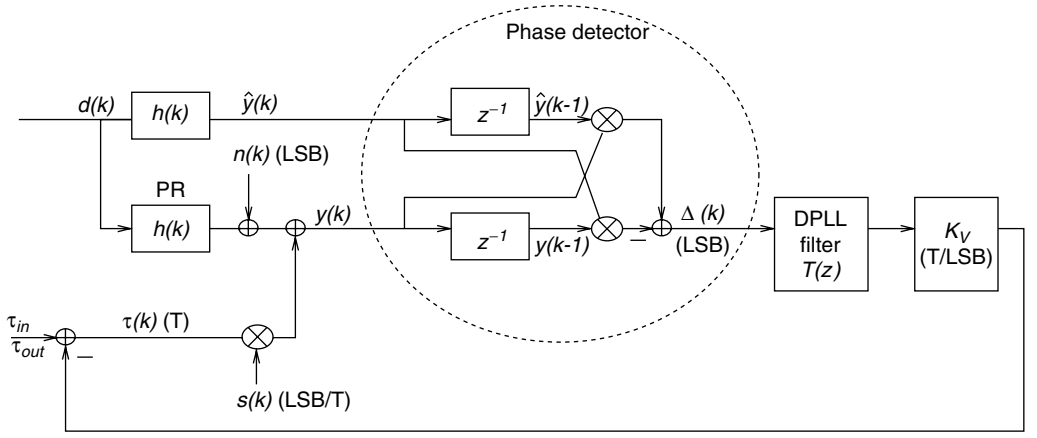



**FIGURE 18.26** Timing loop with Mueller-Muller phase detector.

$$R_{n_0}(0) = \sigma_{n_0}^2 = 2\sigma_d^2\sigma_n^2 \sum_{p=0}^{P-1} h(p)^2 \tag{18.25}$$

and

$$R_{n_o}(l) = R_{n_o}(-l) = -\sigma_d^2\sigma_n^2 \underbrace{\sum_{p=0}^{P-1}\sum_{m=0}^{P-1}}_{p-m=2} h(m)h(p) \tag{18.26}$$

### 18.3.2.3 Performance Comparison of Symbol Rate Timing Loops

So far, the properties of the SLT and MM timing gradients or phase detectors have been examined. If the noise at the phase detector output for both systems were white we could directly compare their performance by comparing their respective *KPD* to $\sigma_{n_o}$ ratio as a kind of signal-to-noise ratio (SNR) of the phase detector. The ratio would measure a signal gain (experienced by sampling phase errors) to noise gain across the entire bandwidth. If the noise had been white for both systems this ratio would scale similarly for both systems when measured over the effective noise bandwidth determined by the loop filter; however, for the MM loop we observed that the noise at the phase detector output was not white. Therefore, we must examine the timing loop performance at the output of the loop filter not just at the output of the phase detector. Before continuing our analysis let us make some qualitative comments about the loop filter.

#### *18.3.2.3.1 Qualitative Loop Filter Description*

A timing loop is a feedback control loop. Therefore, the stability/loop dynamics are determined by the "gain" (in converting observed amplitude error to a timing update) of the phase detector and the details of the loop filter. If the timing loop were needed to remove the effect of a sampling phase error, a first order DPLL would be sufficient; however, the timing loop must also recover the proper frequency with which to sample the signal. Therefore, the use of a second order DPLL loop filter is needed. This allows the timing loop to continually generate small phase updates to produce a clock, which not only has the correct sampling phase within a symbol interval $T$ but which also has the correct value for the symbol interval i.e., the correct clock frequency. DPLL here refers to the portion of the overall loop filter transfer function $T(z)$ without the fixed delay term $z^{-L}$. In addition, important to the performance of the loop is its noise performance, i.e., for a given level of input noise, the effect on

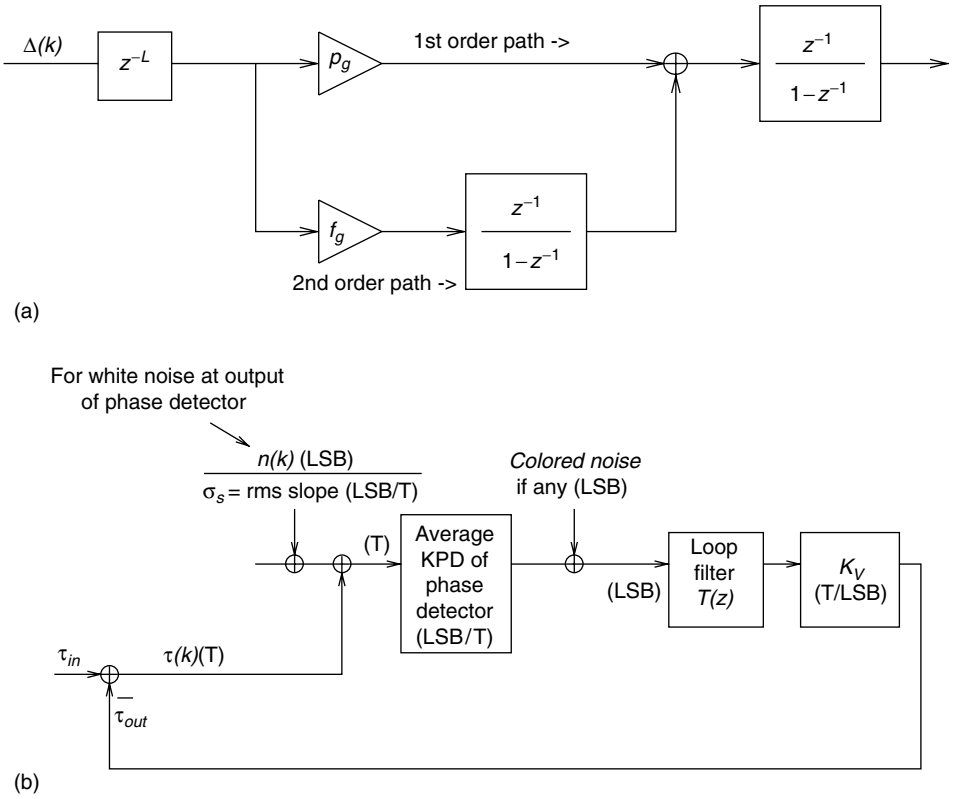



**FIGURE 18.27** Linearized model: (a) second-order DPLL loop filter, (b) timing loop with phase detector modeled by its average signal gain.

the jitter in sampling phase updates. The jitter properties are determined by the noise gain of the phase detector as well as the loop filter properties. The loop filters out noise beyond the bandwidth of interest, this bandwidth being determined by how rapidly the loop is designed to react to timing changes. As mentioned earlier, the DPLL loop filter is a second order filter with an additional latency term. Its transfer function is given by

$$T(z) = z^{-L}\left(\frac{f_g z^{-1}}{1-z^{-1}} + p_g\right)\left(\frac{z^{-1}}{1-z^{-1}}\right) \tag{18.27}$$

where $f_g$ and $p_g$ are frequency and phase update gains for the second order and first order sections, respectively, while $L$ is the loop latency. A block diagram of $T(z)$ is also shown in Fig. 18.27a.

#### *18.3.2.3.2 Noise Jitter Analysis of Timing Loop*

Linearized $Z$ domain analysis of the DPLL is now performed by replacing the phase detector with its *KPD* (denoted by $K_p$ in the equations for readability). In evaluating the SLT and MM DPLLs three sets or combinations of $p_g$ and $f_g$ will be used: "LOW," "MED," and "HGH" where the LOW gains, are relatively low update gains, which would be used in tracking mode, MED gains are moderate gains, and HGH gains, are high gains, which might be used during acquisition. For the SLT and MM DPLLs the $p_g$ and $f_g$ are scaled so that the same settings result in the about same transient response for a given sized phase or frequency disturbance.

The open loop DPLL transfer function, $G(z)$, incorporating the loop filter $L(z)$ and clock update gain is

$$G(z) = K_v\left(\frac{f_g z^{-1}}{1 - z^{-1}} + p_g\right) z^{-L}\left(\frac{z^{-1}}{1 - z^{-1}}\right)$$

Referring to the timing loop model of Fig. 18.27b, the *closed* loop transfer function $(T_{\text{out}}/T_{\text{in}}) = H(z)$ is

$$H(z) = \frac{K_p G(z)}{1 + K_p G(z)} \tag{18.28}$$

Note that $K_p$ has dimensions of LSB/T, $K_v$ and $G(z)$ have dimensions of T/LSB and $H(z)$ is a transfer function with respect to two time quantities. The effective noise bandwidth is then,

$$ENB = 2\int_0^{0.5} |H(f)|^2 df$$

An example of a closed loop transfer function for the SLT DPLL is shown in Fig. 18.28 for LOW update gains. To find the effect of AWG noise, $n(k)$, first convert the $\sigma_n$ to an effective timing noise by dividing by the rms slope, $\sigma_s$, of the signal that is obtained during the numerical generation of the signal slopes and calculation of the slope generating filter coefficients. Now it can be multiplied by

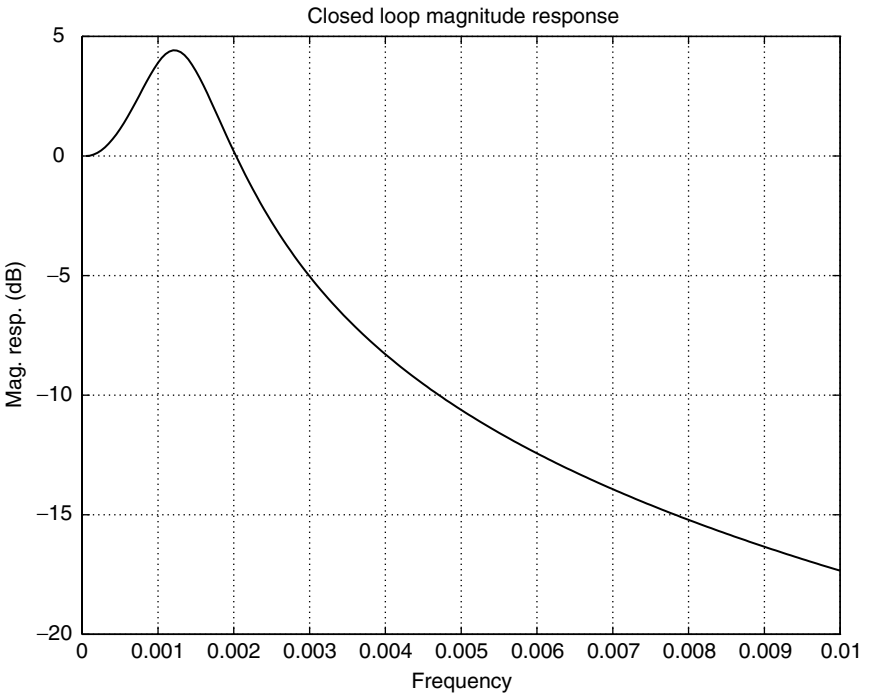



**FIGURE 18.28** Closed loop frequency response of SLT DPLL for low $p_g$ and $f_g$ update gains.

the square root of the *ENB* to determine the corresponding noise induced timing jitter $\sigma_j$ (units of T). Therefore,

$$\sigma_j = \frac{\sigma_n}{\sigma_s}\sqrt{ENB} \tag{18.29}$$

The equivalent model for the above method of analysis is shown in Fig. 18.27b.

For the SLT-based DPLL, the total jitter is simply the above $\sigma_j$. For the MM DPLL the phase detector output noise is colored; however, we know its properties here and can examine its effect from this point onwards. The only difference is that the closed loop transfer function seen by the MM phase detector output noise is

$$F(z) = \frac{G(z)}{1 + K_p G(z)} \tag{18.30}$$

The noise jitter is then obtained as

$$\sigma_j = \sqrt{2\int_0^{0.5} P_n(f)|F(f)|^2 df} \tag{18.31}$$

where $P_n(f)$ is the noise p.s.d. at the phase detector output.

Figure 18.29 plots the jitter performance of the SLT- and MM-based DPLLs for three sets of $(p_g, f_g)$: LOW, MED, HGH. Shown are the output, noise-induced timing jitter of the loop for four channel error event rates. Observe that the MM timing loop's output noise jitter is almost the same but slightly better than that of the SLT-based timing loop.

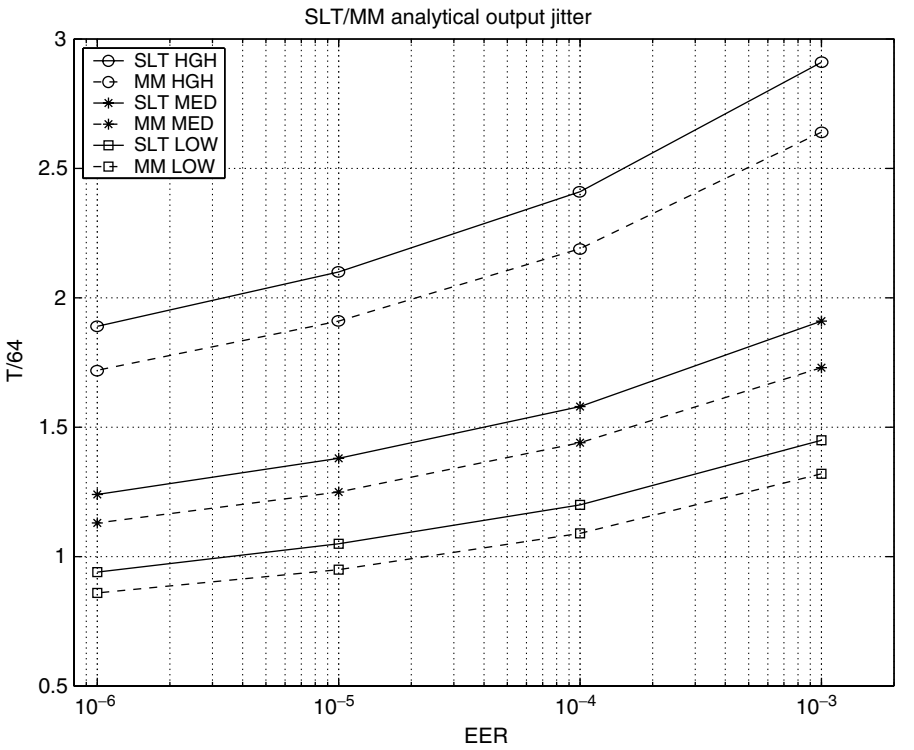



**FIGURE 18.29** Analytically calculated output jitter for SLT and MM timing loops.

#### 18.3.2.4 Jitter and BER Simulation Results

Simulations on the SLT-based timing loop and the MM loop are run within the simulator framework described in Fig. 18.22. The same DPLL loop filter structure is used for both systems. Simulations are run at a channel bit density bit of 2.8 without noise and SNRs, which correspond with channel EERs of $10^{-4}$ and $10^{-2}$. The steady-state jitter is examined in the DPLL output phase and the response of the timing loop to a phase step in the data field. Figure 18.30 shows the transient phase response plots of the SLT and MM DPLLs responding to a $0.1875T$ $(12T/64)$ phase step in data field for the same LOW $p_g$ and $f_g$ settings. Note that they have very similar responses. Table 18.2 shows the steady-state output jitter of the two timing loops for various combinations of gains and noise levels corresponding to EERs of $10^{-4}$ and $10^{-2}$. The settled DPLL phases show some nonzero jitter without additive noise from quantization effects. Timing jitter at the DPLL output is measured by measuring the standard deviation of the DPLL phase. Again, observe that the two timing loops have very similar jitter numbers although the MM timing loop jitter is slightly lower.

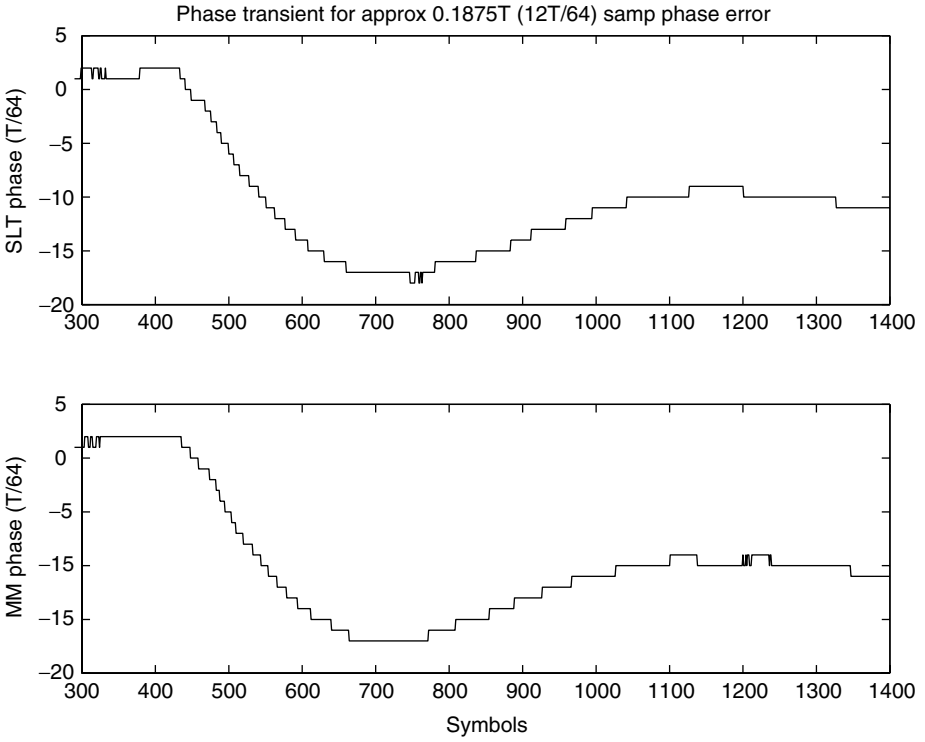



**FIGURE 18.30** SLT and MM DPLL reaction to $0.1875T$ $(12T/64)$ phase step. Low $p_g$, $f_g$ gains. No noise in this simulation.

**TABLE 18.2** Simulation-Based Timing Loop Output Jitter $\sigma_{jt}$ (Units of T/64) Performance of SLT and MM Timing Loops for Final EERs of Zero (Noiseless), $10^{-4}$, and $10^{-2}$

| | SLT | | | MMPD | | |
|---|---|---|---|---|---|---|
| $p_g$, $f_g$ GAINS | EER 0 | EER $10^{-4}$ | EER $10^{-2}$ | EER 0 | EER $10^{-4}$ | EER $10^{-2}$ |
| LOW | 0.49 | 1.30 | 2.18 | 0.45 | 1.16 | 1.86 |
| MED | 0.49 | 1.69 | 2.99 | 0.46 | 1.56 | 2.51 |
| HGH | 0.67 | 2.67 | 4.86 | 0.70 | 2.67 | 4.38 |

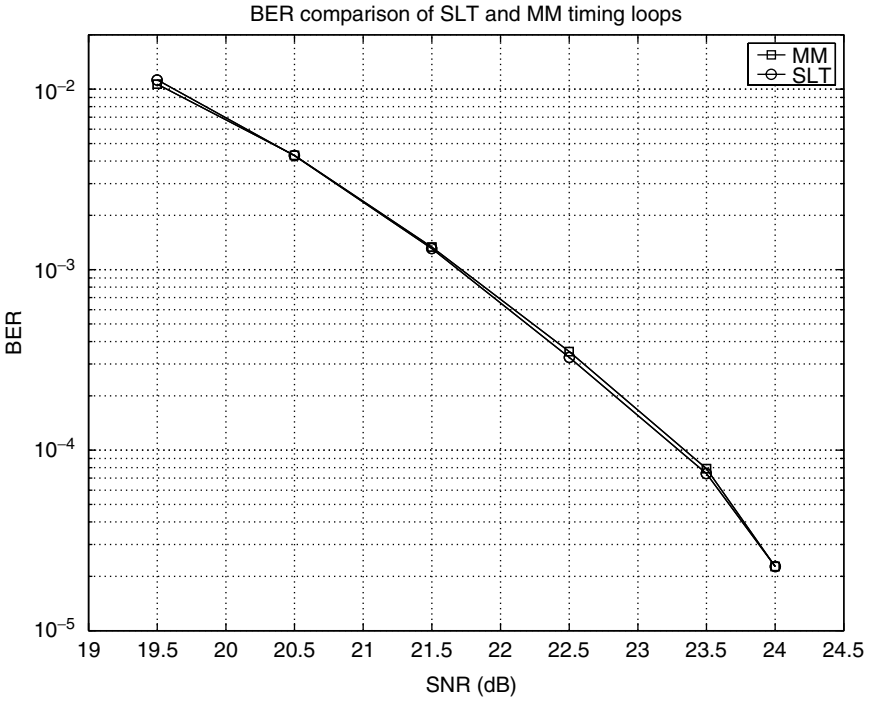



**FIGURE 18.31** Simulated BERs of practical read channel using SLT and MM timing loops.

Finally, the Viterbi detector BER performance is examined instead of the timing loop jitter performance for the read channel architecture of Fig. 18.31 employing the MM and SLT timing loops. Observe that the BERs of the two systems are practically indistinguishable.

### 18.3.2.5 Conclusions

An overview of timing recovery methods for PRML magnetic recording channels, including interpolative and traditional symbol rate VCO-based timing recovery methods, was provided. Also reviewed was the MMSE timing recovery from first principles and its formulation in the framework of a SLT-based timing gradient. A framework for analyzing the performance of the timing loops in terms of output noise jitter was provided. The jitter calculation is based on obtaining linearized $Z$ domain closed loop transfer functions of the timing loop. Also compared was the output timing jitter, due to input noise, of the SLT and MM timing loops—two commonly used timing loops. The jitter performance of the MM loop is almost the same but very slightly better than that obtained with the SLT-based timing loop; however, the Viterbi BER performance of read channel systems employing the two timing loops are practically indistinguishable.

## References

1. A. Oppenheim and R. Schafer, *Discrete Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
2. K. Fisher, W. Abbott, J. Sonntag, and R. Nesin, "PRML detection boosts hard-disk drive capacity" *IEEE Spectrum,* pp. 70–76, November, 1996.
3. M.E. Van Valkenburg, *Analog Filter Design,* Holt Rinehart Winston, 1982.
4. R. Schaumann, M. Ghausi, and K. Laker, *Design of Analog Filters,* Prentice-Hall, Englewood Cliffs, NJ, 1990.

5. J. Park and L.R. Carley, "Analog complex graphic equalizer for EPR4 channels," *IEEE Transactions on Magnetics,* pp. 2785–2787, September, 1997.
6. A. Bishop, et al., "A 300 Mb/s BiCMOS disk drive channel with adaptive analog equalizer," *Digests, Int. Solid State Circuits Conf.*, pp. 46–47, 1999.
7. P. Pai, A. Brewster, and A. Abidi, "Analog front-end architectures for high speed PRML magnetic recording channels," *IEEE Transactions on Magnetics,* pp. 1103–1108, March 1995.
8. R. Cideciyan and F. Dolivo, et al., "A PRML system for digital magnetic recording," *IEEE Journal on Selected Areas in Communications,* pp. 38–56, January, 1992.
9. G. Mathew, et al., "Design of analog equalizers for partial response detection in magnetic recording," *IEEE Transactions on Magnetics,* pp. 2098–2107.
10. S. Qureshi, "Adaptive equalization," in *Proceedings of the IEEE*, September, 1973, pp. 1349–1387.
11. S. Haykin, *Communication Systems,* John Wiley & Sons, New York, 1992, pp. 487–497.
12. S. Haykin, *Adaptive Filter Theory,* Prentice-Hall, 1996.
13. J. Bergmans, *Digital Baseband Transmission and Recording,* Kluwer Academic Publishers, Dordrecht, the Netherlands, 1996.
14. P. Aziz and J. Sonntag, "Equalizer architecture tradeoffs for magnetic recording channels," *IEEE Transactions on Magnetics*, pp. 2728–2730, September, 1997.
15. L. Du, M. Spurbeck, and R. Behrens, "A linearly constrained adaptive FIR filter for hard disk drive read channels," in *Proceedings, IEEE Int. Conf. on Communications,* pp. 1613–1617.
16. J. Sonntag, et al., "A high speed low power PRML read channel device," *IEEE Transactions on Magnetics,* pp. 1186–1189, March, 1995.
17. D. Welland, et al., "Implementation of a digital read/write channel with EEPR4 detection," *IEEE Transactions on Magnetics,* pp. 1180–1185, March 1995.
18. G. Tuttle, et al., "A 130 Mb/s PRML read/write channel with digital-servo detection," *Digests, IEEE Int. Solid-State Circuits Conf.,* 1996.
19. J. Chern, et al., "An EPRML digital read/write channel IC," *Digests, IEEE Int. Solid State Circuits Conf.,* 1997.
20. N. Nazari, "A 500 Mb/s disk drive read channel in 0.25 $\mu$m CMOS incorporating programmable noise predictive Viterbi detection and trellis coding," *Digests, Intl. Solid-State Circuits Conf.,* pp. 78–79, 2000.
21. M. Spurbeck and R. Behrens, "Interpolated timing recovery for hard disk drive read channels," in *Proceedings, IEEE Int. Conf. on Communications,* 1997, pp. 1618–1624.
22. Z. Wu and J. Cioffi, "A MMSE interpolated timing recovery scheme for the magnetic recording channel", in *Proceedings, IEEE Int. Conf. on Communications,* 1997, pp. 1625–1629.
23. A. Patapoutian "On phase-locked loops and Kalman filters," *IEEE Transactions on Communications,* pp. 670–672, May, 1999.
24. K. Mueller and M. Muller, "Timing recovery in digital synchronous data receivers," *IEEE Transactions on Communications,* pp. 516–531, May, 1976.
25. F. Dolivo, W. Schott, and G. Ungerbock, "Fast timing recovery for partial response signaling systems," *IEEE Conf. on Communications,* pp. 18.5.1–18.5.4, 1989.
26. S. Qureshi, "Timing recovery for equalized partial-response systems," *IEEE Transactions on Communications,* pp. 1326–1331, December, 1976.
27. H. Shafiee, "Timing recovery for sampling detectors in digital magnetic recording," *IEEE Conf. on Communications,* pp. 577–581, 1996.
28. J. Bergmans, "Digital baseband transmission and recording," Kluwer Academic Publishers, Dordrecht, the Netherlands, pp. 500–513, 1996.
29. P. Aziz and S. Surendran "Symbol rate timing recovery for higher order partial response channels," *IEEE Journal on Selected Areas in Communications,* April, 2001 (to appear).

# 18.4 Head Position Sensing in Disk Drives

*Ara Patapoutian*

## 18.4.1 Introduction

Data in a disk drive is stored in concentric tracks on one or more disk platters. As the disks spin, a magnetic transducer known as a read/write head, transfers information between the disks and a user [1]. When the user wants to access a given track, the head assembly moves the read/write head to the appropriate location. This positioning of the head is achieved by use of a feedback servo system as shown in Fig. 18.32. First, a position sensor generates a noisy estimate of the head location. Then by comparing the difference between this estimate and the desired location, a controller is able to generate a signal to adjust the *actuator* accordingly.

Two known approaches are used in sensing the head position. An external optical device can be used to estimate the head position by emitting a laser light and then by measuring the reflected beam. This approach is relatively expensive, may need frequent calibrations, and at present is limited to servo writers, which are discussed later. In the second approach, a read head, which is designed primarily to detect the recorded user data pattern, will itself sense position specific magnetic marks recorded on a disk surface. Using statistical signal-processing techniques, the read waveform is decoded into a head position estimate. At present this second approach is preferred for disk drives and is the topic of this article.

In an *embedded servo* scheme, as shown in Fig. 18.33, a portion of each platter, which is divided into multiple wedges, is reserved to provide radial and sometimes angular position information for

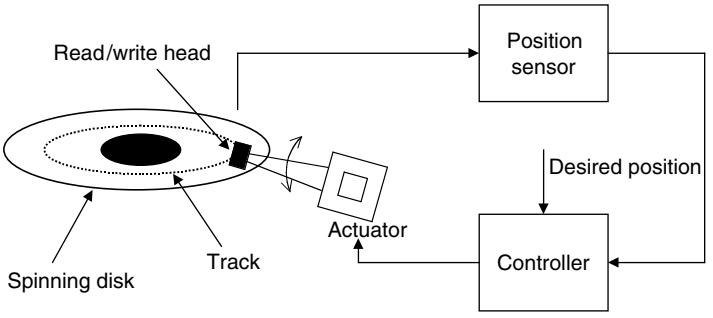



**FIGURE 18.32** Position control loop for a disk drive.

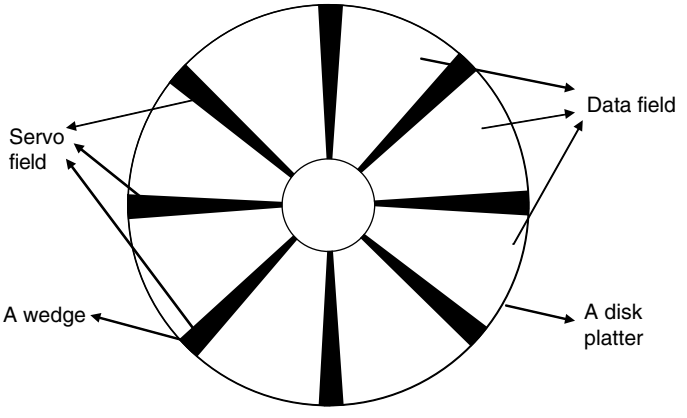



**FIGURE 18.33** Data and servo fields on a disk drive.

the read head. These reserved wedges are known as *servo fields*, and the number of wedges per surface varies significantly amongst different products. A generic servo wedge provides radial estimates in two steps. On a disk surface, each track is assigned a number known as the *track address*. These addresses are included in a servo field, providing complete radial position information with accuracy of up to a track. In other words, the information provided by a track address is complete but coarse. The positional error signal (PES) complements the track address by providing a more accurate estimate within a track. By combining these two estimates, a complete and accurate head position estimate can be obtained.

A wedge may also contain coarse information regarding angular position if a specific address is assigned to each wedge. The user data field, with its own address mark and timing capability, can complement the wedge address by providing finer angular position estimates.

A typical wedge will have multiple sub-fields, as shown in Fig. 18.34. A periodic waveform, known as a *preamble*, provides ample information to calibrate the amplitude of the waveform and, if necessary, to acquire the timing of the recorded pattern. Frame synchronization, or the start of a wedge, is recognized by a special sequence known as the *address mark*. This is followed by the track and wedge addresses, and finally by the servo burst that provides information regarding the PES. These multiple sub-fields can be divided into two distinct areas. Since the address mark, track address, and wedge address are all encoded as binary strings, they are referred to as the *digital field,* as shown in Fig. 18.34. By contrast, ignoring quantization effects of the *read channel,* the periodic servo burst field is decoded to a real number representing the analog radial position. Thus, the format designs as well as the demodulation techniques for the digital and burst fields are fundamentally different. The digital field demodulator is known as the *detector* while the servo burst field demodulator is known as the *estimator*.

Despite their differences, the two fields are not typically designed independently of each other. For example, having common sample rates and common front-end hardware simplifies the receiver architecture significantly. Furthermore, it makes sense to use coherent or synchronous detection algorithms with coherent estimation algorithms and vice versa.

Having a reserved servo field comes at the expense of user data capacity. A major optimization goal is to minimize the servo field overhead for a given cost and reliability target. Both the servo format design as well as that of the detectors/estimators in the read channel chip of a disk drive are optimized to minimize this overhead.

This section reviews position sensing formats and demodulators. Because estimation and detection are well-known subjects, presented in multiple textbooks [2,3], issues that are particular to disk drive position sensors are emphasized. Furthermore, rather than the servo control loop design, the statistical signal processing aspects of position sensing are presented. For a general introduction to disk drive servo control design, the reader is referred to [4], where the design of a disk drive servo is presented as a case study of a control design problem. In general, because of the proprietary nature of this technology, the literature regarding head position sensing is limited to a relatively few published articles, with the exception of patents.

When a disk drive is first assembled in a factory, the servo fields have to somehow be recorded on the disk platters. Once a drive leaves the factory, these fields will only be read and never rewritten.

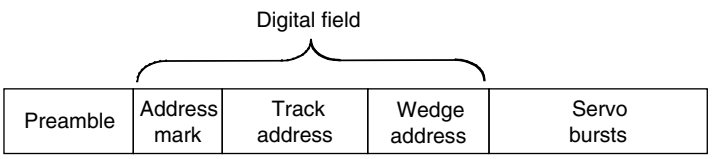



**FIGURE 18.34** A generic composition of a servo field.

Traditionally, an expensive external device, known as the *servo writer,* introduced in Section 18.4.2, records the servo fields. In general, the servo writing process constrains and affects the servo field format choices as well as the demodulator performance. In the next section, the digital field format and detection approaches are addressed, while in Section 18.4.4, the servo burst format and PES estimation approaches are introduced.

## 18.4.2 Servo Writers

After a disk drive is assembled, the function of a servo writer is to record the servo wedges on a drive. While the disks are spinning, an *external* servo writer senses the radial position usually through the head assembly using the reflection of a laser beam. An external mechanical device moves the head assembly. Finally, an external read head locks on a clocking track on a disk to locate the angular position. By knowing both the radial and angular position, as well as controlling the radial position, the servo writer records the wedges, track by track, using the native head of the drive.

Servo writing has been an expensive process. The servo writing time per disk is an important interval that disk manufacturers try to minimize and is proportional to the number of tracks per disk surface, to the spin of the disk drive, and to the number of passes needed to record a track. Since the number of tracks per disk is increasing faster than the spin speed, the servo writer time becomes a parameter that needs to be contained. To this end, the disk drive industry has attempted to minimize both servo writer time and the servo writer cost.

Self-servo writing is a procedure where the wedges are written by the disk drive itself without using any external device [5,6]. Here, the servo writing time is increased but the process is less costly. Many hybrid proposals also use a combination of an external servo writer to record some initial marks and then complete the wedges by using the drive itself. An example of such a process is the printed media approach [7,8], where a master disk "stamps" each disk, and afterward the drive completes writing the wedges.

In general, the servo writer cannot record an arbitrary wedge format. For example, it is very difficult for a servo writer that records wedges track-by-track to record a smooth angled transition across the radius. Furthermore, the wedges generated by a servo writer are not ideal. For example, servo writers that record wedges track-by-track create an *erase band* between tracks [9], where due to head and disk media characteristics, no transition is recorded in a narrow band between two adjacent tracks. Similarly, because of uncertainties in the angular position, two written tracks may not be aligned properly causing *radial incoherence* between adjacent tracks. These two impairments are illustrated in Fig. 18.35. In summary, the servo writer architecture affects both the wedge format design as well as the demodulator performance of a disk drive sensor.

## 18.4.3 The Digital Field

The digital servo field has many similarities to the disk drive user data field [10] and to a digital communications system [2]. Each track in a digital field is encoded and recorded as a binary string

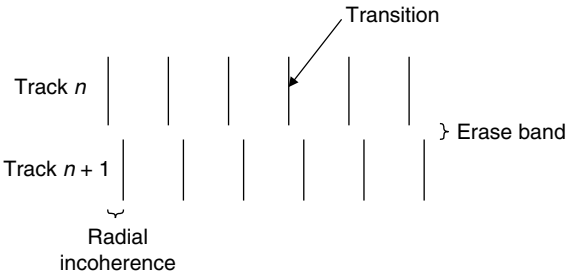



**FIGURE 18.35** Servo writer impairments: erase bands and radial incoherence.

similar to a data field. What differentiates a digital servo field from others is its short block length, and more importantly its off-track detection requirement.

Before discussing these differences, let us start by saying that a magnetic recording channel, for both the data and servo digital fields, is an intersymbol interference (ISI) channel. When read back, the information recorded in a given location modifies the waveform not only at that given location but also in the neighboring locations. Finite length ISI channels can be optimally detected using sequence detectors [11], where at least theoretically, all the field samples are observed before detecting them as a string of ones and zeros. For about a decade now, such sequence detectors have been employed in disk drives to detect the user data field.

The digital servo field length is very short relative to a data field. The present data sector length is around 512 bytes long, whereas the servo digital information string is only a few bytes long. So, whereas the percentage of overhead attributable to a preamble, address marks, and error correcting codes (ECC) is relatively small compared to the user data field, the overhead associated with a digital servo field can easily exceed one hundred percent. For example, it is well known that ECC coding efficiency increases with block length, i.e. codes with very short block lengths have weak error correction capability.

One strategy in minimizing the preamble field length is to use asynchronous detection, which usually trades performance for format, since it does not require exact timing information.

A simple strategy in minimizing the digital field is to write only partial information per wedge [12]. For example, with a smart controller, the absolute track or wedge address may not be needed, since it may be predicted using a state machine; however, such strategies improve format efficiency at the expense of performance robustness and complexity.

#### 18.4.3.1 Offtrack Detection

A primary factor that differentiates digital servo fields from other types of detection channels is the requirement to detect data reliably at any radial position, even when the read head is between two adjacent tracks. In contrast, a user data field is expected to be read reliably only if the read head is directly above that specific track. As will be discussed shortly, such a constraint influences the ECC as well as sequence detection strategies.

A related concern is the presence of radial incoherence, and the erase field introduced during servo writing that are present when the read head straddles two tracks. The detector performance will suffer from such impairments. Formats that tolerate such impairments are desired.

Because the recorded address mark and wedge address does not vary from one track to the next, the emphasis is on track addresses. When the read head is in the middle of two adjacent tracks, with track addresses X and Y, the read waveform is the superposition of the waveforms generated from each of the respective addresses. In general, the resulting waveform cannot be decoded reliably to any one of the two track addresses. A common solution is the use of a Gray code to encode track addresses, as shown in Fig. 18.36, where any two adjacent tracks differ in their binary address representation in only one symbol value. Hence, for the moment ignoring ISI, when the head is midway between adjacent tracks, the detector will decode the address bits correctly except for the bit location where the two adjacent tracks differ, that is, for the two track addresses labeled as X and Y, the decoder will decode the waveform to either track address X or Y, introducing an error of at most one track. By designing a radially periodic servo burst field, with period of at least two track widths, track number ambiguity generated by track addresses is resolved; however, as will be discussed next, Gray codes complicate the use of ECC codes and sequence detectors.

| | | | | | | |
|---|---|---|---|---|---|---|
| X | + | − | + | + | + | − |
| Y | + | − | − | + | + | − |

**FIGURE 18.36** An example of two Gray-coded track addresses. The two addresses are different only in the third location.

A Gray code restricts two adjacent tracks to differ in only a single position, or equivalently forcing the *Hamming distance* between two adjacent track addresses to be one. Adding an

ECC field to the digital fields is desirable since reliable detection of track addresses is needed in the presence of miscellaneous impairments such as electronic and disk media noise, radial incoherence, erase bands, etc.; however, any ECC has a minimum Hamming distance larger than one. That is, it is not possible to have two adjacent track-addresses be Gray and ECC encoded simultaneously. If an ECC field is appended to each track address, it can be used only when the head is directly above a track. A possible alternative is to write the track addresses multiple times with varying radial shifts so that, at any position, the head is mostly directly above a track address [13]. Such a solution improves reliability at the expense of significant format efficiency loss.

Another complication of introducing Gray codes results from the interaction of these codes with the ISI channel. Consider an ISI free channel where the magnetic transitions are written ideally and where the read head is allowed to be anywhere between two adjacent Gray coded track addresses X and Y. As was discussed earlier, the track address reliability, or the probability that the decoded address is neither X nor Y, is independent of the read head position. Next, it is shown that for an ISI channel the detector performance depends on the radial position. In particular, consider the simple ISI channel with pulse response $1 - D$, which approximates a magnetic recording channel. For such a channel, events of length two are almost as probable as errors of length one (same distance but different number of neighbors). Now, as the head moves from track X to Y, the waveform modification introduced by addresses X and Y, at that one location where the two tracks differ, can trigger an error of length two. The detector may decode the received waveform to a different track address Z, which may lie far from addresses X or Y. In other words, in an ISI channel, whenever the head is between two tracks X and Y, the probability that the received waveform is decoded to some other address Z increases.

For readers familiar with signal space representation of codewords, the ISI free and $1 + D$ channels are shown for the three addresses X, Y, and Z with their decision regions in Fig. 18.37. Let $d_h$ denote to the shortest distance from codeword X to the decision boundaries of codeword Z. As shown in Fig. 18.37, when the head is midway between tracks X and Y, the shortest distance to cross the decision boundaries of codeword Z is reduced by a factor of $\sqrt{3}$ (or 4.77 dB). Therefore, when the head is in the middle of two tracks, represented by addresses X and Y, the probability that the decoded codeword is Z increases significantly. For an arbitrary ISI channel this reduction factor in shortest distance varies, and it can be shown to be at most $\sqrt{3}$.

A trivial solution to address both the ECC and ISI complications introduced by the Gray code is not to use any coding and to write address bits far enough from each other to be able to ignore ISI effects. Then a simple symbol-by-symbol detector is sufficient to detect the address without the need for a sequence detector. Actually this is a common approach taken in many disk drive designs; however, dropping ECC

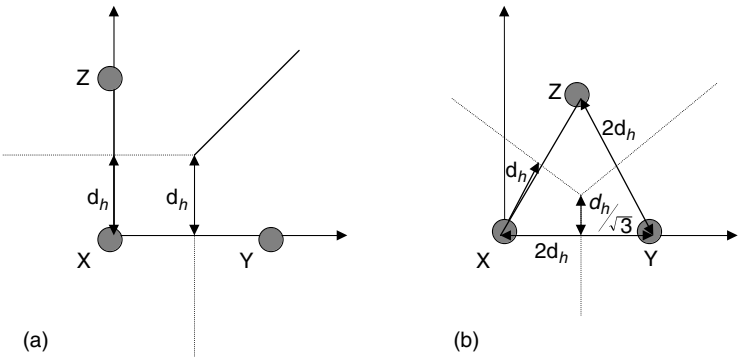



**FIGURE 18.37** Signal space representation of three codewords. Configuration (a) ISI free (b) with ISI.

capability affects reliability and forcing the magnetic recording channel to behave as ISI free requires additional format.

Another approach is to use symbol based codes, such as a bi-phase code, rather than sequence-based codes, that is, rather than maximizing the minimum distance between any two codewords, the distance between two symbols is maximized. For example, in a magnetic recording channel, a bi-phase code produces a positive pulse at the middle of the symbol for a symbol "1" and a negative pulse for a symbol "0," increasing symbol reliability [13,14]. In this example, it can be shown that the ISI related degradations are minimized and the detector performance is improved.

A fundamentally different approach would not make use of a Gray code at all. Instead, codes would be designed from scratch in such a way that for any two addresses X and Y the distance between X and Y would increase, as they are radially located further away from each other.

### 18.4.4 The Burst Field

In the previous subsection, track addresses were introduced, which provide head position information to about single-track accuracy. To be able to read the data field reliably, it is essential to position the read head directly upon the desired track within a small fraction of a track. To this end, the track number addresses are complemented with the servo burst field, where the analog radial position is encoded in a periodic waveform such as a sinusoid. Three ways to encode a parameter in a sinusoidal waveform are used: amplitude, phase, and frequency [3]. Servo burst fields are also periodic radially. Because the track address already provides an absolute position, such a periodicity does not create any ambiguity.

In a disk platter, information is recorded in one of two stable domains. Hence, a servo burst is recorded as a periodic binary pattern. The read back waveform, at the head output, is periodic and will contain both the fundamental and higher harmonics. The sinusoidal waveform is obtained by retaining only the fundamental harmonic. For a given format budget, it is possible to maximize the power of the read back waveform by optimizing the fundamental period [15]. If recorded transitions get too close, ISI destroys most of the signal power. On the other hand, if transitions are far from each other, then the read back pulses are isolated and contain little power.

In this subsection, first the impairment sources are identified. Afterward, various servo burst formats and their performances are discussed [16,17]. Finally, various estimator characteristics and options are reviewed.

#### 18.4.4.1 Impairments

Here, impairments in a servo burst field are classified into three categories: servo-writer induced, read head induced, and read channel induced. Not all impairments are present in all servo burst formats.

As was discussed in Section 18.4.2, when the servo-writer records wedges track-by-track, erase band as well as radial incoherence may be present between tracks, degrading the performance of some of the servo burst formats. Also, the duty cycle of the recorded periods may be different than the intended 50%. Finally, write process limitations result in nonideal recorded transitions.

The read head element as well as the preamplifier, which magnifies the incoming signal, generate electronic noise, modeled by additive white Gaussian noise (AWGN). Also, in many situations the width of the read head element ends up, being shorter than the servo burst radial width as shown in Fig. 18.38a. As will be discussed shortly, for some formats, this creates saturated radial regions where the radial estimates are not reliable [9]. Finally, the rectangular approximation of the read head element shown in Fig. 18.38a is not accurate. More specifically, different regions of the read head may respond differently to a magnetic flux. Hence, the read head profile may be significantly different than a rectangle [18,19].

The read channel, while processing the read waveform, induces a third class of errors. Most present estimators are digitally implemented and have to cope with *quantization error*. If only the first harmonic

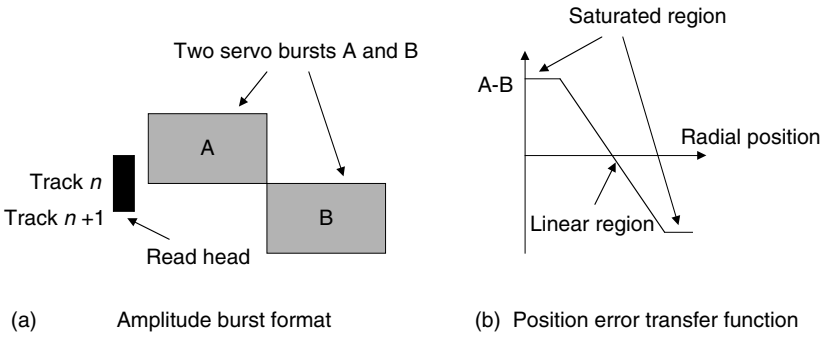



**FIGURE 18.38** The amplitude burst format and its position error transfer function as the head moves from center-track $n$ to center-track $n + 1$.

of the received waveform is desired then suppressing higher harmonics may leave residues that may interact with the first harmonic inside the estimator. Furthermore, sampling a waveform with higher harmonic residues creates *aliasing effects,* where higher harmonics fold into the first harmonic. Many read channel estimators require that the phase, frequency, or both phase and frequency of the incoming waveform are known. Any discrepancy results in estimator degradation. Finally, estimatorcomplexity constraints result in suboptimal estimators, further degrading the accuracy of the position estimate.

### 18.4.4.2 Formatting Strategies

At present, the amplitude servo burst format, shown in Fig. 18.38a, is the most common format used in the disk drive industry. Depending on the radial position of the read head, the overlap between the head and the bursts A and B varies. Through this overlap, or amplitude variation, it is possible to estimate the radial position. First, the waveforms resulting from the overlap of the read head and the burst fields A and B are transformed into amplitude estimates. These amplitude estimates are then subtracted from each other and scaled to get a positional estimate. As the head moves from track center $n$ to track center $(n+1)$, the noiseless positional estimate, known as *position error transfer function,* is plotted in Fig. 18.38b. Here, since the radial width of the servo burst is larger than the read element, any radial position falls into either the *linear* region, where radial estimate is accurate, or in the *saturated* region, where the radial estimate is not accurate [9]. One solution to withstand saturated regions is to include multiple burst pairs, such that any radial position would fall in the linear region of at least one pair of bursts. The obvious drawback of such a strategy is the additional format loss. The amplitude format just presented does not suffer from radial incoherence since two bursts are not recorded radially adjacent to each other.

Because nonrecorded areas do not generate any signal, in Fig. 18.38a only 50% of the servo burst format is recorded with transitions or utilized. In an effort to improve the position estimate performance, the whole allocated servo area can be recorded. As a result, at least two alternative formats have emerged, both illustrated by Fig. 18.39.

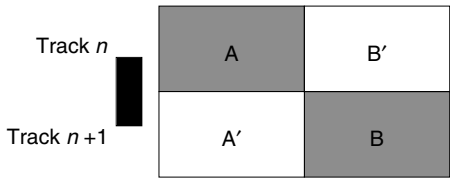



**FIGURE 18.39** Alternative burst formats where A′ and B′ are either orthogonal to or of opposite polarity of A and B, respectively.

In the first improved format, burst A is radially surrounded by an *antipodal* or "opposite polarity" burst A′. For example, if burst A is recorded as $++--++--\cdots$ then burst A′ is recorded as $--++--++\cdots$. For readers familiar with digital communications, the difference between the amplitude and antipodal servo burst formats can be compared to the difference

between on-off and antipodal signaling. In on-off signaling, a symbol "0" or "1" is transmitted while in antipodal signaling 1 or −1 is transmitted. Antipodal signaling is 6 dB more efficient than on-off signaling. Similarly, it can be shown that the antipodal servo burst format gives a 6-dB advantage with respect to amplitude servo burst format under the AWGN assumption [17].

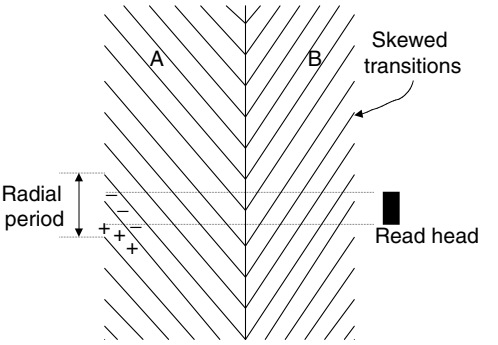



**FIGURE 18.40** The phase format.

Instead of recording A′ to be the opposite polarity of A, another alternative is to record a pattern A′ that is orthogonal to A. For example, it is possible to pick up two sinusoids with different frequencies such that the two waveforms are orthogonal over a finite burst length interval. The resulting format is known as the *dual frequency* format [20]. Inside the read channel, two independent estimates of the head position can be obtained from two estimators, each tuned to one of the two frequencies. The final radial estimate is the average of the two estimates, resulting in a 3-dB improvement with respect to the amplitude format, again under AWGN assumption.

Unlike the amplitude format, these more sophisticated formats are in general more sensitive to other impairments such as erase band and radial incoherence.

A fundamentally different format is presented in Fig. 18.40. Here, the transitions are skewed and the periodic pattern gradually shifts in the angular direction as the radius changes. The radial information is stored in the phase of the period, so it is called the phase format. In Fig. 18.40 two burst fields A and B are presented where the transition slopes have the same magnitude but opposite polarities. An estimator makes two phase estimates, one from the sinusoid in field A and another one from the sinusoid in field B. By subtracting the second phase estimate from the first, and then by scaling the result, the radial position estimate can be obtained. Similar to the antipodal format, it can be shown that the phase pattern is 6 dB superior to the amplitude pattern [17] under AWGN. A major challenge for the phase format is successfully recording the skewed transitions on a disk platter without significant presence of radial incoherence and erase band.

#### 18.4.4.3 Position Estimators

Estimating various parameters of a sinusoid is well documented in textbooks [3]. A decade ago position estimators were mostly implemented by analog circuitry, whereas at present, digital implementation is the norm and the one considered in this article [21–25]. One way of classifying estimators is to determine whether the phase and/or the frequency of the incoming waveform are known.

Assume that the amplitude of a noisy sinusoid needs to be determined. If the phase of this waveform is known, a matched filter can be used to generate the amplitude estimate. This is known as coherent estimation. Under certain assumptions and performance criteria such a filter becomes optimal. When the phase of the waveform is not known, but the frequency is known, then two matched filters can be used, one tuned to a sine waveform while the other filter is tuned to a cosine waveform. The outputs of the two filters are squared and added to give the energy estimate of the waveform. This is known as noncoherent estimation and is equivalent to computing the Fourier transform at the first harmonic. Other ad hoc estimators include the peak estimator and digital area estimators [26], which respectively estimate the averaged peak and the mean value of the unsigned waveform. Neither of these estimators requires the phase or the frequency of the waveform.

For the amplitude format, all the estimators mentioned here can be used. For the antipodal format, the phase of the waveform is needed and therefore a single matched filter is the required estimator. For

dual frequency format, we need two estimators, each tuned to a different frequency. Since the two waveforms are orthogonal to each other, an estimator tuned to one of the waveforms will not observe the other waveform. Each estimator can utilize a single matched filter for coherent estimation or two matched filters for noncoherent estimation. Finally, for phase estimation, two matched filters are utilized, similar to noncoherent estimation; however, rather than squaring and adding the filter outputs, the inverse tangent function is performed on the ratio of the filter outputs.

## References

1. Comstock, R.L. and Workman, M.L., Data storage in rigid disks, in *Magnetic Storage Handbook*, 2nd ed., Mee, C.D. and Daniel, E.D., Eds., McGraw-Hill, New York, 1996, chap. 2.
2. Proakis, J.G., *Digital Communications*, 4th ed., McGraw-Hill, New York, 2000.
3. Kay, S.M., *Fundamentals of Statistical Signal Processing: Estimation Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1993.
4. Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital control of dynamic systems*, 3rd ed., Addison-Wesley, Reading, MA, 1997, chap. 14.
5. Brown, D.H., et al., Self-servo writing file, *US patent 06,040,955*, 2000.
6. Liu, B., Hu, S.B., and Chen, Q.S., A novel method for reduction of the cross track profile asymmetry of MR head during self servo-writing, *IEEE Trans. on Mag.*, 34, 1901, 1998.
7. Bernard, W.R. and Buslik, W.S., Magnetic pattern recording, *U.S. patent 03,869,711,* 1975.
8. Tanaka, S., et al., Characterization of magnetizing process for pre-embossed servo pattern of plastic hard disks, *IEEE Trans. on Mag.,* 30, 4209, 1994.
9. Ho, H.T. and Doan, T., Distortion effects on servo position error transfer function, *IEEE Trans. on Mag.,* 33, 2569, 1997.
10. Bergmans, J.W.M., *Digital Baseband Transmission and Recording,* Kluwar Academic Publishers, Dordrecht, 1996.
11. Forney, G.D., Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference, *IEEE Trans. on Info. Thy.*, 18, 363, 1972.
12. Chevalier, D., Servo pattern for location and positioning of information in a disk drive, *U.S. patent 05,253,131,* 1993.
13. Leis, M.D., et al., Synchronous detection of wide bi-phase coded servo information for disk drive, *U.S. patent 05,862,005,* 1999.
14. Patapoutian, A., Vea, M.P., and Hung, N.C., Wide biphase digital servo information detection, and estimation for disk drive using servo Viterbi detector, *U.S. patent 05,661,760,* 1997.
15. Patapoutian, A., Optimal burst frequency derivation for head positioning, *IEEE Trans. on Mag.*, 32, 3899, 1996.
16. Sacks, A.H., Position signal generation in magnetic disk drives, Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, 1995.
17. Patapoutian, A., Signal space analysis of head positioning formats, *IEEE Trans. on Mag.,* 33, 2412, 1997.
18. Cahalan, D. and Chopra, K., Effects of MR head track profile characteristics on servo performance, *IEEE Trans. on Mag.,* 30, 4203, 1994.
19. Sacks, A.H. and Messner, W.C., MR head effects on PES generation: simulation and experiment, *IEEE Trans. on Mag.,* 32, 1773, 1996.
20. Cheung, W.L., Digital demodulation of a complementary two-frequency servo PES pattern, *U.S. patent 06,025,970,* 2000.
21. Tuttle, G.T., et al., A 130 Mb/s PRML read/write channel with digital-servo detection, *Proc. IEEE ISSCC'96,* 64, 1996.

22. Fredrickson, L., et al., Digital servo processing in the Venus PRML read/write channel, *IEEE Trans. on Mag.,* 33, 2616, 1997.
23. Yada, H. and Takeda, T., A coherent maximum-likelihood, head position estimator for PERM disk drives, *IEEE Trans. on Mag.,* 32, 1867, 1996.
24. Kimura, H., et al., A digital servo architecture with 8.8 bit resolution of position error signal for disk drives, *IEEE Globecom'97,* 1268, 1997.
25. Patapoutian, A., Analog-to-digital converter algorithms for position error signal estimators, *IEEE Trans. on Mag.,* 36, 395, 2000.
26. Reed, D.E., et al., Digital servo demodulation in a digital read channel, *IEEE Trans. on Mag.,* 34, 13, 1998.

## 18.5 Modulation Codes for Storage Systems

*Brian Marcus and Emina Šoljanin*

### 18.5.1 Introduction

Modulation codes are used to constrain the individual sequences that are recorded in data storage channels, such as magnetic or optical disk or tape drives. The constraints are imposed in order to improve the detection capabilities of the system. Perhaps the most widely known constraints are the runlength limited (RLL($d$,$k$)) constraints, in which ones are required to be separated by at least $d$ and no more than $k$ zeros. Such constraints are useful in data recording channels that employ peak detection: waveform peaks, corresponding to data ones, are detected independently of one another. The $d$-constraint helps increase linear density while mitigating intersymbol interference, and the $k$-constraint helps provide feedback for timing and gain control.

Peak detection was widely used until the early 1990s. Although it is still used today in some magnetic tape drives and some optical recording devices, most high density magnetic disk drives now use a form of maximum likelihood (Viterbi) sequence detection. The data recording channel is modeled as a linear, discrete-time, communications channel with inter-symbol interference (ISI), described by its transfer function and white Gaussian noise. The transfer function is often given by $h(D) = (1 - D)(1 + D)^{\mathrm{N}}$, where $N$ depends on and increases with the linear recording density.

Broadly speaking, two classes of constraints are of interest in today's high density recording channels: (1) constraints for improving timing and gain control and simplifying the design of the Viterbi detector for the channel and (2) constraints for improving noise immunity. Some constraints serve both purposes.

Constraints in the first class usually take the form of a PRML ($G$, $I$) constraint: the maximum run of zeros is $G$ and the maximum run of zeros, within each of the two substrings defined by the even indices and odd indices, is $I$. The $G$-constraint plays the same role as the $k$-constraint in peak detection, while the $I$-constraint enables the Viterbi detector to work well within practical limits of memory.

Constraints in the second class eliminate some of the possible recorded sequences in order to increase the minimum distance between those that remain or eliminate the possibility of certain dominant error events. This general goal does not specify how the constraints should be defined, but many such constraints have been constructed; see [20] and the references therein for a variety of examples. Bounds on the capacities of constraints that avoid a given set of error events have been given in [26].

Until recently, the only known constraints of this type were the matched-spectral-null (MSN) constraints. They describe sequences whose spectral nulls match those of the channel and therefore increase its minimum distance. For example, a set of DC-balanced sequences (i.e., sequences of $\pm 1$

whose accumulated digital sums are bounded) is an MSN constraint for the channel with transfer function $h(D) = 1 - D$, which doubles its minimum distance [18].

During the past few years, significant progress has been made in defining high capacity distance enhancing constraints for high density magnetic recording channels. One of the earliest examples of such a constraint is the maximum transition run (MTR) constraint [28], which constrains the maximum run of ones. We explain the main idea behind this type of distance-enhancing codes in Section 18.5.3.

Another approach to eliminating problematic error events is that of parity coding. Here, a few bits of parity are appended to (or inserted in) each block of some large size, typically 100 bits. For some of the most common error events, any single occurrence in each block can be eliminated. In this way, a more limited immunity against noise can be achieved with less coding overhead [5].

Coding for more realistic recording channel models that include colored noise and intertrack interference are discussed in Section 18.5.4. The authors point out that different constraints, which avoid the same prescribed set of differences, may have different performance on more realistic channels. This makes some of them more attractive for implementation.

For a more complete introduction to this subject, the reader is referred to any one of the many expository treatments, such as [16,17,24].

## 18.5.2 Constrained Systems and Codes

Modulation codes used in almost all contemporary storage products belong to the class of constrained codes. These codes encode random input sequences to sequences that obey the constraint of a labeled directed graph with a finite number of states and edges. The set of corresponding constrained sequences is obtained by reading the labels of paths through the graph. Sets of such sequences are called constrained systems or constraints. Figures 18.41 and 18.42 depict graph representations of an RLL constraint and a DC-balanced constraint.

Of special interest are those constraints that do not contain (globally or at certain positions) a finite number of finite length strings. These systems are called systems of finite type (FT). An FT system $X$ over alphabet $\mathcal{A}$ can always be characterized by a finite list of forbidden strings $\mathcal{F} = \{w_1, \ldots, w_N\}$ of symbols in $\mathcal{A}$. Defined this way, FT systems will be denoted by $X_{\mathcal{F}}^{\mathcal{A}}$. The RLL constraints form a prominent class of FT constraints, while DC-balanced constraints are typically not FT.

Design of constrained codes begins with identifying constraints, such as those described in the Introduction, that achieve certain objectives. Once the system of constrained sequences is specified, information bits are translated into sequences that obey the constraints via an *encoder*, which usually has the form of a finite-state machine. The actual set of sequences produced by the encoder is called a constrained code and is often denoted $C$. A *decoder* recovers user sequences from constrained sequences. While the decoder is also implemented as a finite-state machine, it is usually required to have a stronger property, called sliding-block decodablility, which controls error propagation [24].

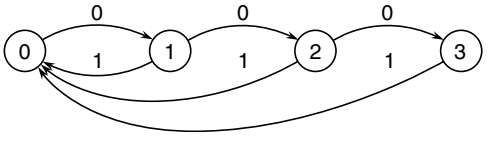



**FIGURE 18.41** RLL (1,3) constraint.

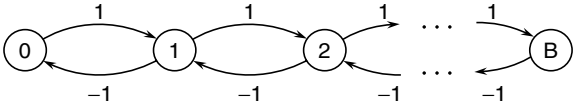



**FIGURE 18.42** DC-balanced constraint.

The maximum rate of a constrained code is determined by *Shannon capacity*. The Shannon capacity or simply *capacity* of a constrained system, denoted by $C$, is defined as

$$C = \lim_{n \to \infty} \frac{\log_2 N(n)}{n}$$

where $N(n)$ is the number of sequences of length $n$. The capacity of a constrained

system represented by a graph $G$ can be easily computed from the *adjacency matrix* (or *state transition matrix*) of $G$ (provided that the labeling of $G$ satisfies some mildly innocent properties). The adjacency matrix of $G$ with $r$ states and $a_{ij}$ edges from state $i$ to state $j$, $1 \leq i, j \leq r$, is the $r \times r$ matrix $A = A(G) = \{a_{ij}\}_{r \times r}$. The Shannon capacity of the constraint is given by

$$C = \log_2 \lambda(A)$$

where $\lambda(A)$ is the largest real eigenvalue of $A$.

The *state-splitting algorithm* [1] (see also [24]) gives a general procedure for constructing constrained codes at any rate up to capacity. In this algorithm, one starts with a graph representation of the desired constraint and then transforms it into an encoder via various graph-theoretic operations including splitting and merging of states. Given a desired constraint and a desired rate $p/q \leq C$, one or more rounds of state splitting are performed; the determination of which states to split and how to split them is governed by an approximate eigenvector, i.e., a vector $\mathbf{x}$ satisfying $A^q \mathbf{x} \geq 2^p \mathbf{x}$.

Many other very important and interesting approaches are used to constrained code construction—far too many to mention here. One approach combines state-splitting with look-ahead encoding to obtain a very powerful technique which yields superb codes [14]. Another approach involves variable-length and time-varying variations of these techniques [2,13]. Many other effective coding constructions are described in the monograph [17].

For high capacity constraints, graph transforming techniques, such as the state-splitting algorithm, may result in encoder/decoder architectures with formidable complexity. Fortunately, a block encoder/decoder architecture with acceptable implementation complexity for many constraints can be designed by well-known enumerative [6], and other combinatorial [32] as well as heuristic techniques [25].

Translation of constrained sequences into the channel sequences depends on the modulation method. Saturation recording of binary information on a magnetic medium is accomplished by converting an input stream of data into a spatial stream of bit cells along a track where each cell is fully magnetized in one of two possible directions, denoted by 0 and 1. Two important modulation methods are commonly used on magnetic recording channels: non-return-to-zero (NRZ) and modified non-return-to-zero (NRZI). In NRZ modulation, the binary digits 0 and 1 in the input data stream correspond to 0 and 1 directions of cell magnetizations, respectively. In NRZI modulation, the binary digit 1 corresponds to a magnetic transition between two bit cells, and the binary digit 0 corresponds to no transition. For example, the channel constraint which forbids transitions in two neighboring bit cells, can be accomplished by either $\mathcal{F} = \{11\}$ NRZI constraint or $\mathcal{F} = \{101, 010\}$ NRZ constraint. The graph representation of these two constraints is shown in Fig. 18.43. The NRZI representation is, in this case, simpler.

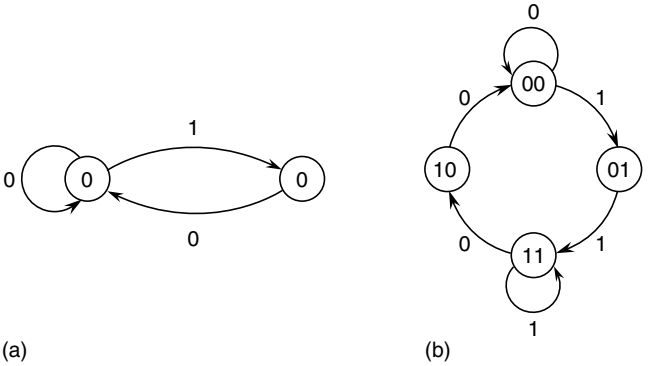



**FIGURE 18.43** Two equivalent constraints: (a) $\mathcal{F} = \{11\}$ NRZI and (b) $\mathcal{F} = \{101,010\}$ NRZ.

## 18.5.3 Constraints for ISI Channels

This subsection discusses a class of codes known as codes, which avoid specified differences. This is the only class of distance enhancing codes used in commercial magnetic recording systems. Two main reasons for this are: these codes simplify the channel detectors relative to the uncoded channel and even high rate codes in this class can be realized by low complexity encoders and decoders.

### 18.5.3.1 Requirements

A number of papers have proposed using constrained codes to provide coding gain on channels with high ISI (see, for example, [4,10,20,28]). The main idea of this approach can be described as follows [20]. Consider a discrete-time model for the magnetic recording channel with possibly constrained input $a = \{a_n\}$ $C \in \subseteq \{0,1\}^{\infty}$, impulse response $\{h_n\}$, and output $y = \{y_n\}$ given by

$$y_n = \sum_m a_m h_{n-m} + \eta_n \tag{18.32}$$

where $h(D) = \Sigma_n h_n D^n = (1-D)(1+D)^3$ ($E^2$PR4) or $h(D) = \Sigma_n h_n D^n = (1-D)(1+D)^4$ ($E^3$PR4), $\eta_n$ are independent Gaussian random variables with zero mean and variance $\sigma^2$. The quantity $1/\sigma^2$ is referred to as the signal-to-noise ratio (SNR). The minimum distance of the uncoded channel given by Eq. 18.32 is

$$d^2_{\min} = \min_{\epsilon(D)\neq 0} \|h(D)\epsilon(D)\|^2$$

where $\epsilon(D) = \sum_{i=0}^{i-l} \epsilon_i D^i$, ($\epsilon_i \in \{-1,0,1\}$, $\epsilon_0 = 1$, $\epsilon_{l-1} \neq 0$) is the polynomial corresponding to a normalized input error sequence $\epsilon = \{\epsilon_i\}_{i=0}^{l-1}$ of length $l$, and the squared norm of a polynomial is defined as the sum of its squared coefficients. The minimum distance is bounded from above by $||h(D)||^2$, denoted by

$$d^2_{\text{MFB}} = \|h(D)\|^2 \tag{18.33}$$

This bound is known as the matched-filter bound (MFB) and is achieved when the error sequence of length $l = 1$, i.e., $\epsilon(D) = 1$, is in the set

$$\arg \min_{\epsilon(D)\neq 0} \|h(D)\epsilon(D)\|^2 \tag{18.34}$$

For channels that fail to achieve the MFB, i.e., for which $d^2_{\min} < ||h(D)||^2$, any error sequences $\epsilon(D)$ for which

$$d^2_{\min} < \|h(D)\epsilon(D)\|^2 < \|h(D)\|^2 \tag{18.35}$$

are of length $l \geq 2$ and may belong to a constrained system $X_{\mathcal{L}}{}^{\{-1,0,1\}}$, where $\mathcal{L}$ is an appropriately chosen finite list of forbidden strings.

For code $\mathcal{C}$, the set of all admissible nonzero error sequences is written as

$$\mathcal{E}(\mathcal{C}) = \{\epsilon \in \{-1,0,1\}^{\infty} | \epsilon \neq 0,\ \epsilon = (\mathbf{a} - \mathbf{b}), \mathbf{a},\mathbf{b} \in \mathcal{C}\}$$

Given the condition $\mathcal{E}(\mathcal{C}) \subseteq X_{\mathcal{L}}^{\{-1,0,1\}}$, the least restrictive finite collection $\mathcal{F}$ of blocks over the alphabet {0,1} can be identified so that

$$\mathcal{C} \subseteq X_{\mathcal{F}}^{\{0,1\}} \Rightarrow \mathcal{E}(\mathcal{C}) \subseteq X_{\mathcal{L}}^{\{-1,0,1\}} \tag{18.36}$$

### 18.5.3.2 Definitions

A constrained code is defined by specifying $\mathcal{F}$, the list of forbidden strings for code sequences. Prior to that one needs to first characterize error sequences that satisfy Eq. 18.35 and then specify $\mathcal{L}$, the list of forbidden strings for error sequences. Error event characterization can be done by using any of the methods described by Karabed, Siegel, and Soljanin in [20]. Specification of $\mathcal{L}$ is usually straightforward.

A natural way to construct a collection $\mathcal{F}$ of blocks forbidden in code sequences based on the collection $\mathcal{L}$ of blocks forbidden in error sequences is the following. From the above definition of error sequences $\boldsymbol{\epsilon} = \{\epsilon_i\}$ we see that $\epsilon_i = 1$ requires $a_i = 1$ and $\epsilon_i = -1$ requires $a_i = 0$, i.e., $a_i = (1 + \epsilon_i)/2$. For each block $\mathbf{w}_{\mathcal{E}} \in \mathcal{L}$, construct a list $\mathcal{F}_{w_{\mathcal{E}}}$ of blocks of the same length $l$ according to the rule:

$$\mathcal{F}_{\mathbf{w}_{\mathcal{E}}} = \{\mathbf{w}_c \in \{0,1\}^l | w_{\mathcal{C}}^i = (1 + w_{\mathcal{E}}^i)/2 \text{ for all } i \text{ for which } w_{\mathcal{E}}^i \neq 0\}.$$

Then the collection $\mathcal{F}$ obtained as $\mathcal{F} = \cup_{\mathbf{w}_{\mathcal{E}} \in \mathcal{L}} \mathcal{F}_{\mathbf{w}_{\mathcal{E}}}$ satisfies requirement (Eq. 18.36); however, the constrained system $X_{\mathcal{F}}^{\{0,1\}}$ obtained this way may not be the most efficient. (Bounds on the achievable rates of codes which avoid specified differences were found recently in [26].)

The previous ideas are illustrated in the example of the $E^2PR4$ channel. Its transfer function is $h(D) = (1 - D)(1 + D)^3$, and its MFB is $||(1 - D)(1 + D)^3 \cdot 1||^2 = 10$. The error polynomial $\boldsymbol{\epsilon}(D) = 1 - D + D^2$ is the unique error polynomial for which $||(1 - D)(1 + D)^3\, \boldsymbol{\epsilon}(D)||^2 = 6$, and the error polynomials $\boldsymbol{\epsilon}(D) = 1 - D + D^2 + D^5 - D^6 + D^7$ and $\boldsymbol{\epsilon}(D) = \sum_{i=0}^{l-1} (-1)^i D^i$ for $l \geq 4$ are the only polynomials for which $||(1 - D)(1 + D)^3\, \boldsymbol{\epsilon}(D)||^2 = 8$ (see, for example, [20]).

It is easy to show that these error events are not in the constrained error set defined by the list of forbidden error strings $\mathcal{L} = \{+ - + 00, + - + -\}$, where $+$ denotes 1 and $-$ denotes $-1$. To see this, note that an error sequence that does not contain the string $+-+00$ cannot have error polynomials $\boldsymbol{\epsilon}(D) = 1 - D + D^2$ or $\mathcal{E}(D) = 1 - D + D^2 + D^5 - D^6 + D^7$, while an error sequence that does not contain string $+ - + -$ cannot have an error polynomial of the form $\boldsymbol{\epsilon}(D) = \sum_{i=0}^{l-1} (-1)^i D^i$ for $l \geq 4$. Therefore, by the above procedure of defining the list of forbidden ode strings, we obtain the $\mathcal{F} = \{+ - +\}$ NRZ constraint. Its capacity is about 0.81, and a rate 4/5 c code into the constraint was first given in [19].

In [20], the following approach was used to obtain several higher rate constraints. For each of the error strings in $\mathcal{L}$, we write all pairs of channel strings whose difference is the error string. To define $\mathcal{F}$, look for the longest string(s) appearing in at least one of the strings in each channel pair. For the example above and the $+-+00$ error string, a case-by-case analysis of channel pairs is depicted in Fig. 18.44. We can distinguish two types (denoted by *A* and *B* in the figure) of pairs of code sequences involved in forming an error event. In a pair of type *A*, at least one of the sequences has a transition run of length 4. In a pair of type B, both sequences have transition runs of length 3, but for one of them the run starts at an even position and for the other at an odd position. This implies that an NRZI

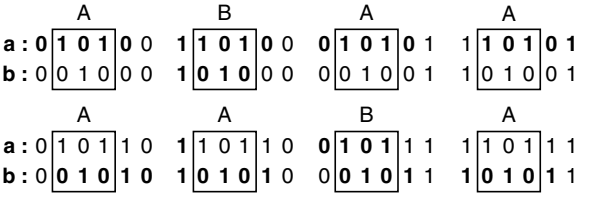



**FIGURE 18.44** Possible pairs of sequences for which error event $+-+00$ may occur.

constrained system that limits the run of 1s to 3 when it starts at an odd position, and to 2 when it starts at an even position, eliminates all possibilities shown bold-faced in Fig. 18.44. In addition, this constraint eliminates all error sequences containing the string $+-+-$. The capacity of the constraint is about .916, and rate 8/9 block codes with this constraint have been implemented in several commercial read channel chips. More about the constraint and the codes can be found in [4,10,20,28].

### 18.5.4 Channels with Colored Noise and Intertrack Interference

Magnetic recording systems always operate in the presence of colored noise intertrack interference, and data dependent noise. Codes for these more realistic channel models are studied in [27]. The following is a brief outline of the problem.

The data recording and retrieval process is usually modeled as a linear, continuous-time, communications channel described by its Lorentzian step response and additive white Gaussian noise. The most common discrete-time channel model is given by Eq. 18.32. Magnetic recording systems employ channel equalization to the most closely matching transfer function $h(D) = \sum_n h_n D^n$ of the form $h(D) = (1 - D)(1 + D)^N$. This equalization alters the spectral density of the noise, and a better channel model assumes that the $\eta_n$ in Eq. 18.32 are identically distributed, Gaussian random variables with zero mean, variance $\sigma^2$, and normalized cross-correlation $E\{\eta_n \eta_k\}/\sigma^2 = \rho_{n-k}$.

In practice, there is always intertrack interference (ITI), i.e., the read head picks up magnetization from an adjacent track. Therefore, the channel output is given by

$$y_n = \sum_m a_m h_{n-m} + \sum_m x_m g_{n-m} + \eta_n \tag{18.37}$$

where $\{g_n\}$ is the discrete-time impulse response of the head to the adjacent track, and $\mathbf{x} = \{x_n\} \in \mathcal{C}$ is the sequence recorded on that track. Assuming that the noise is white.

In the ideal case (Eq. 18.32), the probability of detecting **b** given that **a** was recorded is equal to $Q(d(\boldsymbol{\epsilon})/\sigma)$, where $d(\boldsymbol{\epsilon})$ is the distance between **a** and **b** given by

$$d^2(\boldsymbol{\epsilon}) = \sum_n \left( \sum_m \epsilon_m h_{n-m} \right)^2 \tag{18.38}$$

Therefore, a lower bound, and a close approximation for small $\sigma$, to the minimum probability of an error-event in the system is given by $Q(d_{\min,\mathcal{C}}/\sigma)$, where

$$d_{\min,\mathcal{C}} = \min_{\boldsymbol{\epsilon} \in \boldsymbol{\epsilon}_e} d(\boldsymbol{\epsilon})$$

is the channel minimum distance of code $\mathcal{C}$. We refer to

$$d_{\min} = \min_{\boldsymbol{\epsilon} \in \{-1,0,1\}^\infty} d(\boldsymbol{\epsilon}) \tag{18.39}$$

as the minimum distance of the uncoded channel, and to the ratio $d_{\min,\mathcal{C}}/d_{\min}$ as the gain in distance of code $\mathcal{C}$ over the uncoded channel.

In the case of colored noise, the probability of detecting **b** given that **a** was recorded equals to $Q(\Delta(\boldsymbol{\epsilon})/\sigma)$, where $\Delta(\boldsymbol{\epsilon})$ is the distance between **a** and **b** given by

$$\Delta^2(\boldsymbol{\epsilon}) = \frac{\left[ \sum_n \left( \sum_m \epsilon_m h_{n-m} \right)^2 \right]^2}{\sum_n \sum_k \left( \sum_m \epsilon_m h_{n-m} \right) \rho_{n-k} \left( \sum_m \epsilon_m h_{k-m} \right)}$$

Therefore, a lower bound to the minimum probability of an error-event in the system is given by $Q(\Delta_{\min,\mathcal{C}}/\sigma)$, where

$$\Delta_{\min,\mathcal{C}} = \min_{\boldsymbol{\epsilon}\in\mathcal{E}_{\mathcal{C}}} \Delta(\boldsymbol{\epsilon})$$

In the case of ITI (Eq. 18.37), an important factor is the probability of detecting sequence **b** given that sequence **a** was recorded on the track being read and sequence **x** was recorded on an adjacent track. This probability is

$$Q(\delta(\boldsymbol{\epsilon},\mathbf{x})/\sigma)$$

where $\delta(\boldsymbol{\epsilon},\mathbf{x})$ is the distance between **a** and **b** in the *presence* of **x** given by [30]

$$\delta^2(\boldsymbol{\epsilon},\mathbf{x}) = \frac{1}{\left[\sum_n \left(\sum_m \epsilon_m h_{n-m}\right)^2\right]} \left[\sum_n \left(\sum_m \epsilon_m h_{n-m}\right)^2 + \sum_n \left(\sum_m x_m g_{n-m}\right)\left(\sum_m \epsilon_m h_{n-m}\right)\right]^2$$

Therefore, a lower bound to the minimum probability of an error-event in the system is proportional to $Q(\delta_{\min,\mathcal{C}}/\sigma)$, where

$$\delta_{\min,\mathcal{C}} = \min_{\boldsymbol{\epsilon}\neq 0,\mathbf{x}\in\mathcal{C}} \delta(\boldsymbol{\epsilon},\mathbf{x})$$

Distance $\delta_{\min,\mathcal{C}}$ can be bounded as follows [30]:

$$\delta_{\min,\mathcal{C}} \geq (1-\mathcal{M}) d_{\min,\mathcal{C}} \tag{18.40}$$

where $\mathcal{M} = \max_{n,\mathbf{x}\in\mathcal{C}} \Sigma_m x_m g_{n-m}$, i.e., $\mathcal{M}$ is the maximum absolute value of the interference. Note that $\mathcal{M} = \Sigma_n |g_n|$. We will assume that $\mathcal{M} < 1$. The bound is achieved if and only if there exists an $\boldsymbol{\epsilon}$, $d(\boldsymbol{\epsilon}) = d_{\min,\mathcal{C}}$, for which $\Sigma_m \epsilon_m h_{n-m} \in \{-1, 0, 1\}$ for all $n$, and there exists an $x \in \mathcal{C}$ such that $\Sigma_m x_m g_{n-m} = \mp \mathcal{M}$ whenever $\Sigma_m \epsilon_m h_{n-m} = \pm 1$.

### 18.5.5 An Example

Certain codes provide gain in minimum distance on channels with ITI and colored noise, but not on the AWGN channel with the same transfer function. This is best illustrated using the example of the partial response channel with the transfer function $h(D) = (1-D)(1+D)^2$ known as EPR4. It is well known that for the EPR4 channel $d^2_{\min} = 4$. Moreover, as discussed in Section 18.5.3, the following result holds:

**Proposition 1.** *Error events $\boldsymbol{\epsilon}(D)$ such that*

$$d^2(\boldsymbol{\epsilon}) = d^2_{\min} = 4$$

*take one of the following two forms:*

$$\boldsymbol{\epsilon}(D) = \sum_{j=0}^{k-1} D^{2j},\ k \geq 1$$

*or*

$$\boldsymbol{\epsilon}(D) = \sum_{i=0}^{l-1} (-1)^i D^i,\ l \geq 3$$

Therefore, an improvement of error-probability performance can be accomplished by codes which eliminate the error sequences $\boldsymbol{\epsilon}$ containing the strings $-1\ +1\ -1$ and $+1\ -1\ +1$. Such codes were extensively studied in [20].

In the case of ITI (Eq. 18.37), it is assumed that the impulse response to the reading head from an adjacent track is described by $g(D) = \alpha H(D)$, where the parameter $\alpha$ depends on the track to head distance. Under this assumption, the bound Eq. 18.40 gives $\delta^2_{\min} \geq \delta^2_{\min}\ (1 - 4\alpha)^2$. The following result was shown in [30]:

**Proposition 2.** *Error events* $\boldsymbol{\epsilon}(D)$ *such that*

$$\min_{\mathbf{x}\in\mathcal{C}} \delta^2(\boldsymbol{\epsilon},\mathbf{x}) = \delta^2_{\min} = d^2_{\min}(1 - 4\alpha)^2 = 4(1 - 4\alpha)^2$$

*take the following form:*

$$\boldsymbol{\epsilon}(D) = \sum_{i=0}^{l-1} (-1)^i D^i,\ l \geq 5$$

*For all other error sequences for which* $d^2(\boldsymbol{\epsilon}) = 4$, *we have* $\min_{\mathbf{x}\in\mathcal{C}} \delta^2\ (\boldsymbol{\epsilon}, \mathrm{x}) = 4(1 - 3\alpha)^2$.

Therefore, an improvement in error-probability performance of this channel can be accomplished by limiting the length of strings of alternating symbols in code sequences to four. For the NRZI type of recording, this can be achieved by a code that limits the runs of successive ones to three. Note that the set of minimum distance error events is smaller than in the case with no ITI. Thus, performance improvement can be accomplished by higher rate codes that would not provide any gain on the ideal channel.

Channel equalization to the EPR4 target introduces cross-correlation among noise samples for a range of current linear recording densities (see [27] and references therein). The following result was obtained in [27]:

**Proposition 3.** *Error events* $\boldsymbol{\epsilon}(D)$ *such that*

$$\Delta^2(\boldsymbol{\epsilon}) = \Delta^2_{\min}$$

*take the following form:*

$$\boldsymbol{\epsilon}(D) = \sum_{i=0}^{l-1} (-1)^i D^i, l \geq 3,\ l\ odd$$

Again, the set of minimum distance error events is smaller than in the ideal case (white noise), and performance improvement can be provided by codes which would not give any gain on the ideal channel. For example, since all minimum distance error events have odd parity, a single parity check code can be used.

## 18.5.6 Future Directions

### 18.5.6.1 Soft-Output Decoding of Modulation Codes

Detection and decoding in magnetic recording systems is organized as a concatenation of a channel detector, an inner decoder, and an outer decoder, and as such should benefit from techniques known as erasure and list decoding. To declare erasures or generate lists, the inner decoder (or channel detector) needs to assess symbol/sequence reliabilities. Although the information required for this is the same one necessary for producing a single estimate, some additional complexity is usually required. So far, the predicted gains for erasure and list decoding of magnetic recording channels with additive white Gaussian noise were not sufficient to justify increasing the complexity of the channel detector and

inner and outer decoder; however, this is not the case for systems employing new magneto-resistive reading heads, for which an important noise source, thermal asperities, is to be handled by passing erasure flags from the inner to the outer decoder.

In recent years, one more reason for developing simple soft-output channel detectors has surfaced. The success of turbo-like coding schemes on memoryless channels has sparked the interest in using them as modulation codes for ISI channels. Several recent results show that the improvements in performance turbo codes offer when applied to magnetic recording channels at moderate linear densities are even more dramatic than in the memoryless case [12,29]. The decoders for turbo and low-density parity check codes (LDPC) either require or perform much better with soft input information which has to be supplied by the channel detector as its soft output. The decoders provide soft outputs which can then be utilized by the outer Reed-Solomon (RS) decoder [22]. A general soft-output sequence detection was introduced in [11], and it is possible to get information on symbol reliabilities by extending those techniques [21,31].

### 18.5.6.2 Reversed Concatenation

Typically, the modulation encoder is the inner encoder, i.e., it is placed downstream of an error-correction encoder (ECC) such as an RS encoder; this configuration is known as standard concatenation (Fig. 18.45). This is natural since otherwise the ECC encoder might well destroy the modulation properties before passing across the channel; however, this scheme has the disadvantage that the modulation decoder, which must come before the ECC decoder, may propagate channel errors before they can be corrected. This is particularly problematic for modulation encoders of very high rate, based on very long block size. For this reason, a good deal of attention has recently focused on a reversed concatenation scheme, where the encoders are concatenated in the reversed order (Fig. 18.46). Special arrangements must be made to ensure that the output of the ECC encoder satisfies the modulation constraints. Typically, this is done by insisting that this encoder be systematic and then re-encoding the parity information using a second modulation encoder (the "parity modulation encoder"), whose corresponding decoder is designed to limit error propagation; the encoded parity is then appended to the modulation-encoded data stream (typically a few merging bits may need to be inserted in between the two streams in order to ensure that the entire stream satisfies the constraint). In this scheme, after passing through the channel the modulation-encoded data stream is split from the modulation-encoded parity stream, and the latter is then decoded via the parity modulation decoder before being passed on to the ECC decoder. In this way, many channel errors can be corrected before the data modulation decoder, thereby mitigating the problem of error propagation. Moreover, if the data


**FIGURE 18.45** Standard concatenation.

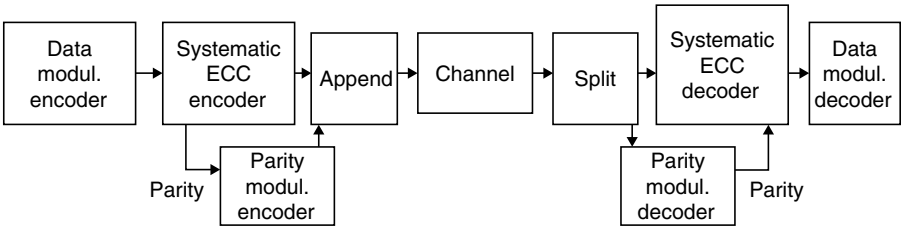



**FIGURE 18.46** Reversed concatenation.

modulation encoder has high rate, then the overall scheme will also have high rate because the parity stream is relatively small.

Reversed concatenation was introduced in [3] and later in [23]. Recent interest in the subject has been spurred on by the introduction of a lossless compression scheme, which improves the efficiency of reversed concatenation [15], and an analysis demonstrating the benefits in terms of reduced levels of interleaving [8]; see also [9]. Research on fitting soft decision detection into reversed concatenation can be found in [7,33].

## References


1. R. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding-block codes," *IEEE Trans. Inform. Theory,* vol. 29, no. 1, pp. 5–22, Jan. 1983.
2. J. Ashley and B. Marcus, "Time-varying encoders for constrained systems: an approach to limiting error propagation," *IEEE Trans. Inform. Theory,* 46 (2000), 1038–1043.
3. W.G. Bliss, "Circuitry for performing error correction calculations on baseband encoded data to eliminate error propagation," *IBM Tech. Discl. Bull.*, 23 (1981), 4633–4634.
4. W.G. Bliss, "An 8/9 rate time-varying trellis code for high density magnetic recording," *IEEE Trans. Magn.*, vol. 33, no. 5, pp. 2746–2748, Sept. 1997.
5. T. Conway, "A new target response with parity coding for high density magnetic recording," *IEEE Trans. Magn.*, vol. 34, pp. 2382–2386, 1998.
6. T. Cover, "Enumerative source encoding," *IEEE Trans. Inform. Theory,* pp. 73–77, Jan. 1973.
7. J. Fan. "Constrained coding and soft iterative decoding for storage," PhD Dissertation, Stanford University, 1999.
8. J. Fan and R. Calderbank, "A modified concatenated coding scheme, with applications to magnetic data storage," *IEEE Trans. Inform. Theory,* 44 (1998), 1565–1574.
9. J. Fan, B. Marcus, and R. Roth, "Lossless sliding-block compression of constrained systems," *IEEE Trans. Inform. Theory,* 46 (2000), 624–633.
10. K. Knudson Fitzpatrick, and C.S. Modlin, "Time-varying MTR codes for high density magnetic recording," in *Proc. 1997 IEEE Global Telecommun. Conf. (GLOBECOM '97),* Phoenix, AZ, Nov. 1997, pp. 1250–1253.
11. J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft–decision outputs and its applications," in *Proc. 1989 IEEE Global Telecommun. Conf. (GLOBECOM '89),* Dallas, TX, Nov. 1989, pp. 1680–1687.
12. C. Heegard, "Turbo coding for magnetic recording," in *Proc. 1998 Information Theory Workshop,* San Diego, CA, Feb. 8–11, 1998, pp. 18–19.
13. C.D. Heegard, B.H. Marcus, and P.H. Siegel, "Variable-length state splitting with applications to average runlength-constrained (ARC) codes," *IEEE Trans. Inform. Theory,* 37 (1991), 759–777.
14. H.D.L. Hollmann, "On the construction of bounded-delay encodable codes for constrained systems," *IEEE Trans. Inform. Theory,* 41 (1995), 1354–1378.
15. K.A. Schouhamer Immink, "A practical method for approaching the channel capacity of constrained channels," *IEEE Trans. Inform. Theory,* 43 (1997), 1389–1399.
16. K.A. Schouhamer Immink, P.H. Siegel, and J.K. Wolf, "Codes for Digital Recorders," *IEEE Trans. Infor. Theory,* vol. 44, pp. 2260–2299, Oct. 1998.
17. K.A. Schouhamer Immink, *Codes for Mass Data Storage,* Shannon Foundation Publishers, The Netherlands, 1999.
18. R. Karabed and P.H. Siegel, "Matched spectral null codes for partial response channels," *IEEE Trans. Inform. Theory,* 37 (1991), 818–855.
19. R. Karabed and P.H. Siegel, "Coding for higher order partial response channels," in *Proc. 1995 SPIE Int. Symp. on Voice, Video, and Data Communications,* Philadelphia, PA, Oct. 1995, vol. 2605, pp. 115–126.
20. R. Karabed, P.H. Siegel, and E. Soljanin, "Constrained coding for binary channels with high intersymbol interference," *IEEE Trans. Inform. Theory,* vol. 45, pp. 1777–1797, Sept. 1999.

21. K.J. Knudson, J.K. Wolf, and L.B. Milstein, "Producing soft-decision information on the output of a class IV partial response Viterbi detector," in *Proc. 1991 IEEE Int. Conf. Commun. (ICC '91),* Denver, CO, June 1991, pp. 26.5.1.–26.5.5.
22. R. Koetter and A. Vardy, preprint 2000.
23. M. Mansuripur, "Enumerative modulation coding with arbitrary constraints and post-modulation error correction coding and data storage systems," *Proc. SPIE,* 1499 (1991), 72–86.
24. B. Marcus, R. Roth, and P. Siegel, "Constrained systems and coding for recording channels," Chapter 20 of *Handbook of Coding Theory,* edited by V. Pless, C. Huffman, 1998, Elsevier.
25. D. Modha and B. Marcus, "Art of constructing low complexity encoders/decoders for constrained block codes," *IEEE J. Sel. Areas in Comm.*, (2001), to appear.
26. B.E. Moision, A. Orlitsky, and P.H. Siegel, "On codes that avoid specified differences," *IEEE Trans. Inform. Theory,* vol. 47, pp. 433–441, Jan. 2001.
27. B.E. Moision, P.H. Siegel, and E. Soljanin, "Distance Enhancing Codes for High-Density Magnetic Recording Channel," *IEEE Trans. Magn.*, submitted, Jan. 2001.
28. J. Moon and B. Brickner, "Maximum transition run codes for data storage systems," *IEEE Trans. Magn.*, vol. 32, pp. 3992–3994, Sept. 1996.
29. W. Ryan, L. McPheters, and S.W. McLaughlin, "Combined turbo coding and turbo equalization for PR4-equalized Lorentzian channels," in *Proc. 22nd Annual Conf. Inform. Sciences and Systems,* Princeton, NJ, March 1998.
30. E. Soljanin, "On-track and off-track distance properties of Class 4 partial response channels," in *Proc. 1995 SPIE Int. Symp. on Voice, Video, and Data Communications,* Philadelphia, PA, vol. 2605, pp. 92–102, Oct. 1995.
31. E. Soljanin, "Simple soft-output detection for magnetic recording channels," in *1998 IEEE Int. Symp. Inform. Theory (ISIT'00),* Sorrento, Italy, June 2000.
32. A.J. van Wijngaarden and K.A. Schouhamer Immink "Combinatorial construction of high rate runlength-limited codes," *Proc. 1996 IEEE Global Telecommun. Conf. (GLOBECOM '96),* London, U.K., pp. 343–347, Nov. 1996.
33. A.J. van Wijngaarden and K.A. Schouhamer Immink, "Maximum run-length limited codes with error control properties," *IEEE J. Select. Areas Commun.*, vol. 19, April 2001.
34. A.J. van Wijngaarden and E. Soljanin, "A combinatorial technique for constructing high rate MTR-RLL codes," *IEEE J. Select. Areas Commun.*, vol. 19, April 2001.

## 18.6 Data Detection

*Miroslav Despotović and Vojin Šenk*

### 18.6.1 Introduction

Digital magnetic recording systems transport information from one time to another. In communication society jargon, it is said that recording and reading information back from a (magnetic) medium is equivalent to sending it through a time channel. There are differences between such channels. Namely, in communication systems, the goal is a user error rate of $10^{-5}$ or $10^{-6}$. Storage systems, however, often require error rates of $10^{-12}$ or better. On the other hand, the common goal is to send the greatest possible amount of information through the channel used. For storage systems, this is tantamount to increasing recording density, keeping the amount redundancy as low as possible, i.e., keeping the bit rate per recorded pulse as high as possible. The perpetual push for higher bit rates and higher storage densities spurs a steady increment of the amplitude distortion of many types of transmission and storage channels.

When recording density is low, each transition written on the magnetic medium results in a relatively isolated peak of voltage, and peak detection method is used to recover written information; however,

when PW50 (pulse width at half maximum response) becomes comparable with the channel bit period, the peak detection channel cannot provide reliable data detection, due to intersymbol interference (ISI). This interference arises because the effects of one readback pulse are not allowed to die away completely before the transmission of the next. This is an example of a so-called baseband transmission system, i.e., no carrier modulation is used to send data. Impulse dispersion and different types of induced noise at the receiver end of the system introduce combination of several techniques (equalization, detection, and timing recovery) to restore data. This section gives a survey of most important detection techniques in use today assuming ideal synchronization.

Increasing recording density in new magnetic recording products necessarily demands enhanced detection techniques. First detectors operated at densities at which pulses were clearly separated, so that very simple, symbol-by-symbol detection technique was applied, the so-called *peak detector* [30]. With increased density, the overlap of neighboring dispersed pulses becomes so severe (i.e., large intersymbol interference—ISI) that peak detector could not combat with such heavy pulse shape degradation. To accomplish this task, it was necessary to master signal processing technology to be able to implement more powerful *sequence detection* techniques. This section will both focus on this type of detection already applied in commercial products and give advanced procedures for searching the detection trellis to serve as a tutorial material for research on next generation products.

## 18.6.2 Partial-Response Equalization

In the classical peak detection scheme, an equalizer is inserted whose task is just to remove all the ISI so that an isolated pulse is acquired, but the equalization will also enhance and colorize the noise (from readback process) due to spectral mismatch. The noise enhancement obtained in this manner will increase with recording density and eventually become intolerable. Namely, since such a full equalization is aimed at slimming the individual pulse, so that it does not overlap with adjacent pulses, it is usually too aggressive and ends up with huge noise power.

Let us now review the question of recording density, also known as packing density. It is often used to specify how close two adjacent pulses stay to each other and is defined as PW50/T (see Section 18.1 for definition). Whatever tricks are made with peak detection systems, they barely help at PW50/T ratios above 1.

Section 18.6 discusses two receiver types that run much less rapidly out of steam. These are the partial-response equalizer (PRE) and the decision-feedback equalizer (DFE). Both are rooted in old telegraph tricks and, just as is the case with peak detector, they take instantaneous decisions with respect to the incoming data. Section 18.6 will focus mainly on these issues, together with sequence detection algorithms that accompany partial-response (PR) equalization.

What is PR equalization? It is the act of shaping the readback magnetic recording signal to look like the target signal specified by the PR. After equalization the data are detected using a sequence detector. Of course, quantization by an analog-to-digital converter (ADC) occurs at some point before the sequence detector.

The common readback structure consists of a linear filter, called a whitened matched filter, a symbol-rate sampler (ADC), a PRE, and a sequence detector, Fig. 18.47. The PRE in this scheme can also be put before the sampler, meaning that it is an analog, not a digital equalizer. Sometimes part of the equalizer is implemented in the analog, the other part in the digital domain. In all cases, analog signal, coming

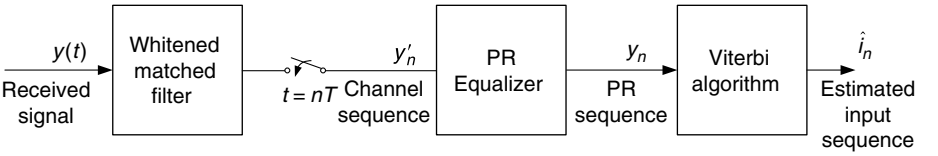



**FIGURE 18.47** Maximum-likelihood sequence detector.

from the magnetic head, should have a certain and constant level of amplification. This is done in a variable gain amplifier (VGA). To keep a signal level, VGA gets a control signal from a clock and gain recovery system. In the sequel, we will assume that VGA is already (optimally) performed. In the design of equalizers and detectors, low power dissipation and high speed are both required. The error performances need to be maintained as well. So far, most systems seek for the implementations in the digital domain, as is the case in Fig. 18.47, but it has been shown that ADC may contribute to the high-frequency noise during the PR target equalization, causing a long settling time in clock recovery loop, as well as degrading performance [33]. In addition, the ADC is also usually the bottleneck for the low-power high-speed applications. On the other hand, the biggest problem for an analog system is the imperfection of circuit elements. The problems encountered with analog systems include nonideal gain, mismatch, nonlinear hold step, offset, etc.

Let us now turn to the blocks shown in Fig. 18.47. The first of them, the whitened matched filter, has the following properties [7]:

*Simplicity*: a single filter producing single sample at the output is all that is needed. The response of the filter is either chosen to be causal and hence realizable, or noncausal, meaning some delay has to be introduced, yielding better performance.

*Sufficiency*: the filter is information lossless, in the sense that its sampled outputs are a set of sufficient statistics for estimation of the input sequence.

*Whiteness*: the noise components of the sampled outputs are independent identically distributed Gaussian random variables.

The whiteness and sufficiency property follow from the fact that the set of waveforms at the output of the matched filter is an orthonormal basis for the signal space.

The next block is PRE. What is PR? Essential to PR techniques is that the PR sequence is obtained from the channel sequence via a simple linear filter. More specifically, the impulse response of this filter is such that the overall response is modeled as having only a few small integer-valued coefficients, the condition actually considered crucial for the system to be called PR. This condition subsequently yields relatively simple sequence detectors. The correlative level coding [3], also known as PR [31] is adopted in digital communication applications for long time. Kobayashi [9] suggested in 1971 that this type of channels can be treated as a linear finite state machine, and thus can be represented by the state diagram and its time instant labeled counterpart, trellis diagram. Consequently, its input is best inferred using some trellis search technique, the best of them (if we neglect complexity issues) being the Viterbi algorithm [2] (if one is interested in maximizing the likelihood of the whole sequence; otherwise, a symbol-by-symbol detector is needed). Kobayashi also indicated that the magnetic recording channel could be regarded as the PR channel due to the inherent differentiation property in the readback process [8]. This is both present in inductive heads and in magnetoresistive (MR) heads, though the latter are directly sensitive to magnetization and not to its change (this is due to the fact that the head has to be shielded). In other words, the pulse will be read only when the transition of opposite magnet polarities is sensed.

Basic to the PR equalization is the fact that a controlled amount of ISI is not suppressed by the equalizer, but rather left for a sequence detector to handle. The nature of the controlled ISI is defined by a PR. A proper match of this response to the channel permits noise enhancement to remain small even when amplitude distortion is severe. In other words, PR equalization can provide both well-controlled ISI and spectral match.

PR equalization is based on two assumptions:

- The shape of readback signal from an isolated transition is exactly known and determined.
- The superposition of signals from adjacent transitions is linear.

Furthermore, it is assumed that the channel characteristics are fixed and known, so that equalization need not be adaptive. The resulting PR channel can be characterized using $D$-transform of the sequences that occur, $X(D) = I(D)H(D)$ [7] where $H(D) = \sum_{i=0}^{M-1} h_i D^i$, $D$ represents the delay factor in

$D$-transform and $M$ denotes the order of the PR signals. When modeling differentiation, $H(D) = 1 - D$. The finite state machine (FSM) of this PR channel is known as the dicode system since there are only two states in the transition diagram.

The most unclear signal transformation in Fig. 18.47 is equalization. What does it mean that the pulse of voltage should look like the target signal specified by the PR (the so-called PR target)? To answer this question let us consider the popular Class IV PR, or PRIV system.

For magnetic recording systems with PW50/T approximately equal to 2, comparatively little equalization is required to force the equalized channel to match a class-4 PR (PR4) channel where $H(D) = (1 - D)(1 + D) = 1 - D^2$. Comparing to the Lorentzian model of Section 18.1, PR4 channel shows more emphasis in the high frequency domain. The equalizer with the PR4 as the equalization target thus suppresses the low frequency components and enhances the high frequency ones, degrading the performance of all-digital detectors since the quantization noise, that is mainly placed at higher frequencies, is boosted up.

The isolated pulse shape in a PR4 system is shown in Fig. 18.48. The transition is written at time instant $t = 0$, where $T$ is the channel bit period. The shape is oscillating and the pulse values at integer number of bit periods before the transition are exactly zeros. Obviously, it is this latter feature that should give us future advantage; however, at $t = 0$ and at $t = T$, i.e., one bit period later, the values of the pulse are equal to "1". The pulse of voltage reaches its peak amplitude of 1.273 at one half of the bit period. Assume that an isolated transition is written on the medium and the pulse of voltage shown in Fig. 18.48 comes to the PRML system. The PR4 system requires that the samples of this pulse should correspond to the bit periods. Therefore, samples of the isolated PR4 pulse will be $00\ldots011000\ldots$ (of course, "1" is used for convenience, and in reality it corresponds to some ADC level).

Because the isolated transition has two nonzero samples, when the next transition is written, the pulses will interfere. Thus, writing two pulses adjacent to each other will introduce superposition between them, usually called a dipulse response, as shown in Fig. 18.49. Here, the samples are $[\ldots,0,0,1,0,-1,0,0,\ldots]$, resulting from

$$
\begin{array}{rl}
0\ 0\ 0\ 1\quad 1\quad 0\ 0\ 0 & \text{from the first transition} \\
+\ 0\ 0\ 0\ 0 - 1\ -1\ 0\ 0 & \text{from the second transition} \\
\hline
0\ 0\ 0\ 1\quad 0\ -1\ 0\ 0 &
\end{array}
$$

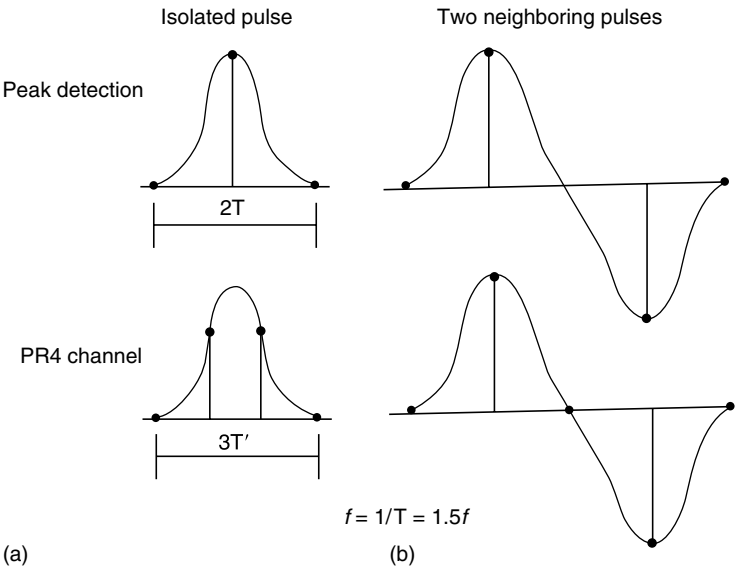



**FIGURE 18.48** Capacity of PR4 channel.

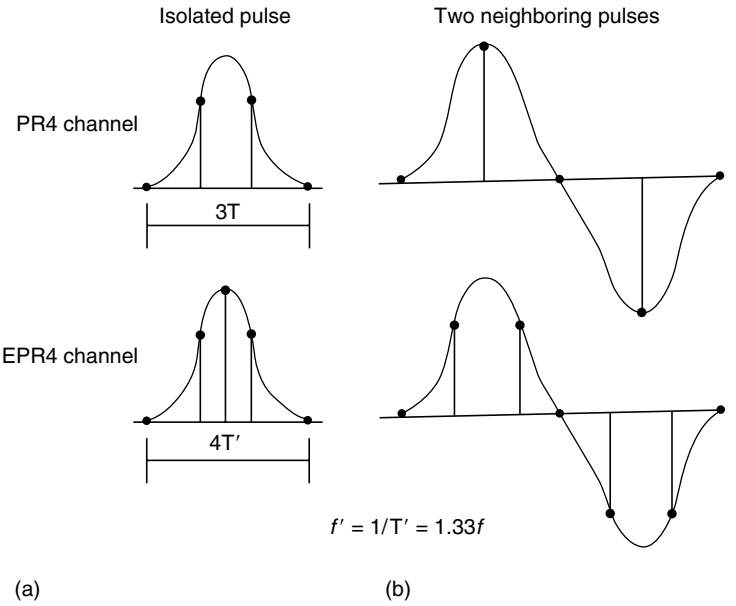



**FIGURE 18.49** Capacity of EPR4 channel.

Now, there is no concern about linear ISI; once the pulses can be reduced to the predetermined simple shape, the data pattern is easily recovered because superposition of signals from adjacent transitions is known. In the above example, we see that sample "1" is suppressed by "−1" from the next transition. It is a simple matter to check that all possible linear combinations of the samples result in only three possible values $\{-1, 0, +1\}$ (naturally, it is that all parts of the system are working properly, i.e., equalization, gain, and timing recovery, and that the signal is noise free). A positive pulse of voltage is always followed by a negative pulse and vice versa, so that the system can be regarded as an alternative mark inversion (AMI) code.

The higher bit capacity of the PR4 channel can best be understood from Fig. 18.48. It is observed that PR4 channel provides a 50% enhancement in the recording density as compared with the peak detection (fully equalized) one, since the latter requires isolation of single bits from each other. In the next figure, we see that the EPR4 channel (explained later) adds another 33% to this packing density. PR4 has another advantage over all the other PR systems; since $H(D) = 1 - D^2$, the current symbol is correlated to the second previous one, allowing the system to be modeled as two interleaved dicode channels, implying the use of simple dicode detectors for even and odd readback samples. RLL coding is necessary in this case, since nonideal tracking and timing errors result in a residual intermediate term (linear in $D$) that induces correlation between two interleaved sequences, and thus degrades systems that rely on decoupled detection of each of them.

RLL codes are widely used in conjunction with PR equalization in order to eliminate certain data strings that would render tracking and synchronization difficult. If PR4 target is used, a special type of RLL coding is used, characterized by $(0, G/I)$. Here, $G$ and $I$ denote the maximum number of consecutive zeros in the overall data string, and in the odd/even substrings, respectively. The latter parameter ensures proper functioning of the clock recovery mechanism if deinterleaving of the PR4 channel into two independent dicode channels is performed. The most popular is the (0,4/4) code, whose data rate is 7/8, i.e., whose data loss is limited to 12.5%.

Other PR targets are used besides PR4. The criterion of how to select the appropriate PR target is based on spectral matching, to avoid introducing too much equalization noise. For instance, for $\mathrm{PW50/T} \approx 2.25$, it is better to model ISI pattern as the so-called EPR4 (i.e. extended class-4 partial

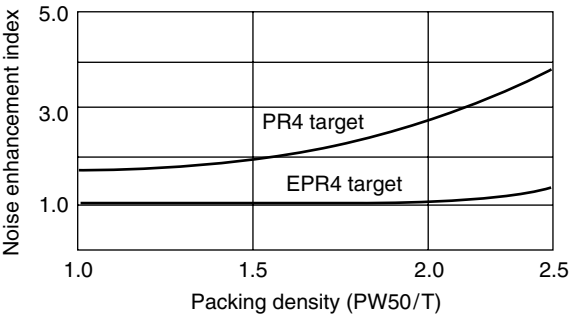



**FIGURE 18.50** Equalization noise enhancement in PR channels.

response) channel with $H(D) = (1 + D)^2\ (1 - D) = 1 + D - D^2 - D^3$. As the packing density goes up, more low frequency components are being introduced (low compared to $1/T$, that also increases as $T$ is shortened, in reality those frequencies are higher than those met for lower recording densities, respectively greater $T$). This is the consequence of the fact that intersymbol interference blurs the boundary between individual pulses, flattening the overall response (in time domain). The additional $1 + D$ term in the target PR effectively suppresses the unwanted high frequencies. EPR4 enables even higher capacities of the magnetic recording systems than PRIV, observing the difference of 33% in the recording density displayed in Fig. 18.49; however, a practical implementation of EPR4 is much more complex than is the case with PR4. First, the deinterleaving idea used for PR4 cannot be implemented. Second, the corresponding state diagram (and consequently trellis) now has eight states instead of four (two if deinterleaving is used). Furthermore, its output is five-leveled, instead of ternary for the PR4 and the dicode channel, so that a 4.4 dB degradation is to be expected with a threshold detector. Naturally, if sequence detector is used, such as Viterbi algorithm (VA), this loss does not exist, but its elimination is obtained at the expense of a significantly increased complexity of the detector. Furthermore, if such a detector can be used, EPR4 has a performance advantage over PR4 due to less equalization noise enhancement, cf. Fig. 18.50.

Let us reconsider the PR equalizer shown in Fig. 18.47. Following the approach from Reference 44, its aim is to transform the input spectrum $Y'(e^{j2\pi\Omega})$ into a spectrum $Y(e^{j2\pi\Omega}) = Y'(e^{j2\pi\Omega})|C(e^{j2\pi\Omega})|^2$, where $C(e^{j2\pi\Omega})$ is the transfer function of the equalizer. The spectrum $Y(e^{j2\pi\Omega}) = I(e^{j2\pi\Omega})|H(e^{j2\pi\Omega})|^2 + N(e^{j2\pi\Omega})$ where $H(D)$ is the PR target. For instance, duobinary PR target $(H(D) = 1 + D)$ enhances low frequencies and suppresses those near the Nyquist frequency $\Omega = 0.5$, whereas dicode $H(D) = (1 - D)$ does the opposite: it suppresses low frequencies and enhances those near $\Omega = 0.5$.

In principle, the spectral zeros of $H(e^{j2\pi\Omega})$ can be undone via a linear (recursive) filter, but this would excessively enhance any noise components added. The schemes for tracking the input sequence to the system based on the PR target equalized one will be reviewed later in this section. For instance, for a PR4 system, a second-order recursive filter can in principle be used to transform its input into an estimate of the information sequence, Fig. 18.51.

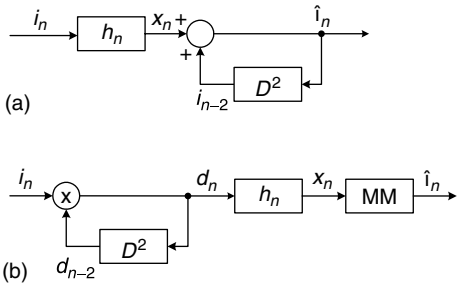



**FIGURE 18.51** (a) PR4 recursive restoration of information sequence and (b) precoder derived from it.

Unfortunately, if an erroneous estimate is produced at any moment, all subsequent estimates will be in error (in fact, they will no longer be in the alphabet $\{-1, 0, 1\}$, enabling error monitoring and simple forms of error correction [31]). To avoid this catastrophic error propagation, resort can be taken to a precoder.

Let us analyze the functioning of this precoder in the case of the PR4 channel $(1 - D^2)$

(generalization to other PR channels is trivial). Its function is to transform $i_n$ into a binary sequence $d_n = i_n d_{n-2}$ to which the PR transformation is applied, Fig. 18.51b. This produces a ternary sequence

$$x_n = (d * h)_n = d_n - d_{n-2} = i_n d_{n-2} - d_{n-2} = (i_n - 1)d_{n-2}$$

Because $d_{n-2}$ cannot be zero, $x_n$ is zero iff $i_n - 1 = 0$, i.e., $i_n = 1$. Thus, the estimate $\hat{i}_n$ of $i_n$ can be formed by means of the memoryless mapping (MM)

$$\hat{i}_n = \begin{cases} 1, & x_n = 0 \\ 0, & \text{else} \end{cases}$$

This decoding rule does not rely on past data estimates and thereby avoids error propagation altogether. In practice, the sequences $i_n$ and $d_n$ are in the alphabet $\{0, 1\}$ rather than $\{-1, 1\}$, and the multiplication in Fig. 18.51b becomes a modulo-2 addition (where 0 corresponds to 1, and 1 to $-1$).

The precoder does not affect the spectral characteristics of an uncorrelated data sequence. For correlated data, however, precoding need not be spectrally neutral. It is instructive to think of the precoder as a first-order recursive filter with a pole placed so as to cancel the zero of the partial response. The filter uses a modulo-2 addition instead of a normal addition and as a result the cascade of filter and PR, while memoryless, has a nonbinary output. The MM serves to repair this "deficiency."

Catastrophic error propagation can be avoided without precoder by forcing the output of the recursive filter of Fig. 18.52 to be binary (Fig. 18.52a). An erroneous estimate $\hat{i}_{n-2} = -i_{n-2}$ leads to a digit

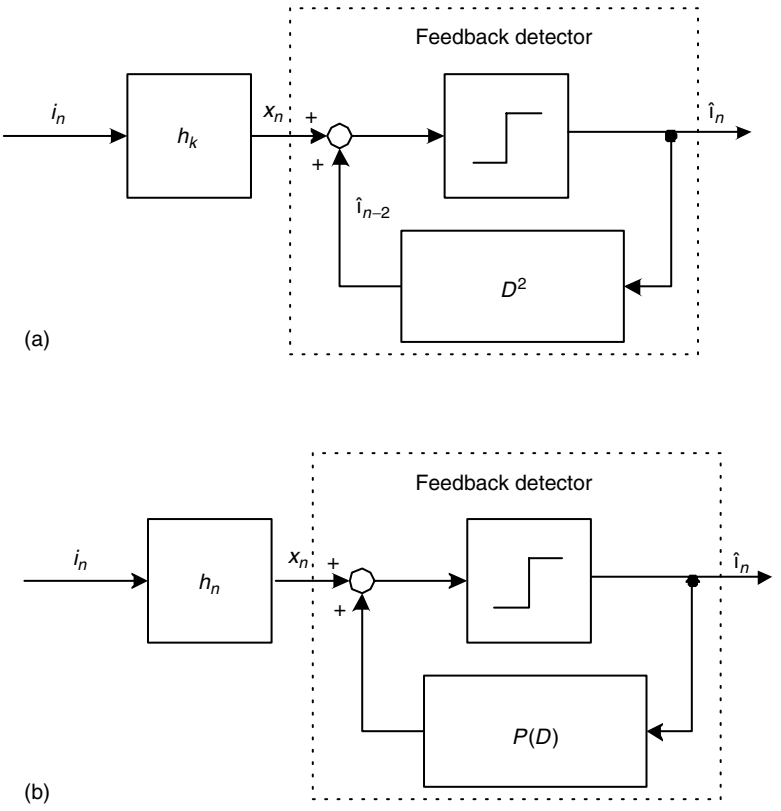



**FIGURE 18.52** Feedback detector.

$$\hat{i}_n = x_n + \hat{i}_{n-2} = i_n - i_{n-2} + \hat{i}_{n-2} = i_n - 2i_{n-2}$$

whose polarity is obviously determined by $i_{n-2}$. Thus, the decision $\hat{i}_n$ that is taken by the slicer in Fig. 18.52a will be correct if $i_n$ happens to be the opposite of $i_{n-2}$. If data is uncorrelated, this will happen with probability 0.5, and error propagation will soon cease, since the average number of errors in a burst is $1 + 0.5 + (0.5)^2 + \cdots = 2$. Error propagation is thus not a serious problem.

The feedback detector of Fig. 18.52 is easily generalized to arbitrary partial response $H(D)$. For purposes of normalization, $H(D)$ is assumed to be causal and monic (i.e., $h_n = 0$ for $n < 0$ and $h_0 = 1$). The nontrivial taps $h_1, h_2, \ldots$ together form the "tail" of $H(D)$. This tail can be collected in $P(D)$, with $p_n = 0$ for $n \leq 0$ and $p_n = h_n$ for $n \geq 1$. Hence, $h_n = \delta_n + p_n$, where the Kronecker delta function $\delta_n$ represents the component $h_0 = 1$. Hence

$$x_n = (i * h)_n = (i * (\delta + p))_n = i_n + (i * p)_n.$$

The term $(i * p)_n$ depends exclusively on past digits $i_{n-1}, i_{n-2}, \ldots$ that can be replaced by decisions $\hat{i}_{n-1}, \hat{i}_{n-2}, \ldots$. Therefore, an estimate $\hat{i}_n$ of the current digit $\hat{i}_n$ can be formed according to $\hat{i}_n = x_k\,(\hat{i} * p)_n$ as in Fig. 18.52b. As before, a slicer quantizes $\hat{i}_n$ into binary decisions $\hat{i}_n$ so as to avoid catastrophic error propagation. The average length of bursts of errors, unfortunately, increases with the memory order of $H(D)$. Even so, error propagation is not normally a serious problem [21]. In essence, the feedback detector avoids noise enhancement by exploiting past decisions. This viewpoint is also central to decision-feedback equalization, to be explained later.

Naturally, all this can be generalized to nonbinary data; but in magnetic recording, so far, only binary data are used (the so-called saturation recording). The reasons for this are elimination of hysteresis and the stability of the recorded sequence in time.

Let us consider now the way the PR equalizer from Fig. 18.47 is constructed. In Fig. 18.53, a discrete-time channel with transfer function $F(e^{j2\pi\Omega})$ transforms $i_n$ into a sequence $y_n = (i * f)_n + u_n$, where $u_n$ is the additive noise with power spectral density $U(e^{j2\pi\Omega})$, and $y_n$ represents the sampled output of a whitened matched filter. We might interpret $F(e^{j2\pi\Omega})$ as comprising two parts: a transfer function $H(e^{j2\pi\Omega})$ that captures most of the amplitude distortion of the channel (the PR target) and a function $F_r(e^{j2\pi\Omega}) = F(e^{j2\pi\Omega})/H(e^{j2\pi\Omega})$ that accounts for the remaining distortion. The latter distortion has only a small amplitude component and can thus be undone without much noise enhancement by a linear equalizer with transfer function

$$C(e^{j2\pi\Omega}) = \frac{1}{F_r(e^{j2\pi\Omega})} = \frac{H(e^{j2\pi\Omega})}{F(e^{j2\pi\Omega})}$$

This is precisely the PR equalizer we sought for. It should be stressed that the subdivision in Fig. 18.53 is only conceptual. The equalizer output is a noisy version of the "output" of the first filter in Fig. 18.53 and is applied to the feedback detector of Fig. 18.52, to obtain decision variables $\hat{i}'_n$ and $\hat{i}_n$. The precoder and MM of Fig. 18.51 are, of course, also applicable and yield essentially the same performance.

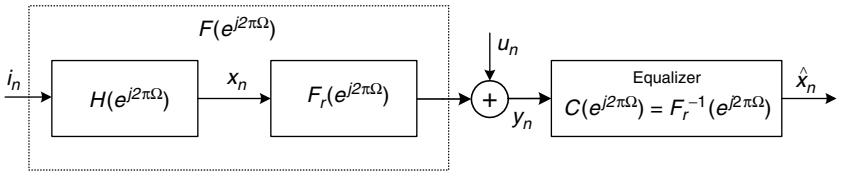



**FIGURE 18.53** Interpretation of PR equalization.

The choice of the coefficients of the PRE in Fig. 18.47 is the same as for full-response equalization and is explained in Section 18.3. Interestingly, zero-forcing here is not as bad as is the case with full-response signaling and yields approximately the same result as minimum mean-square equalization. To evaluate the performance of the PRE, let us assume that all past decisions that affect $\hat{i}_n$ are correct and that the equalizer is zero forcing (see Section 18.3 for details). The only difference between $\hat{i}_n'$ and $\hat{i}_n$ is now the filtered noise component $(u * c)_n$ with variance

$$\sigma^2_{\text{ZFPRE}} = \int_{-0.5}^{0.5} U(e^{j2\pi\Omega})|C(e^{j2\pi\Omega})|^2 d\Omega = \int_{-0.5}^{0.5} \frac{U(e^{j2\pi\Omega})|H(e^{j2\pi\Omega})|^2}{|F(e^{j2\pi\Omega})|^2} d\Omega$$

Because $|H(e^{j2\pi\Omega})|$ was selected to be small wherever $|F(e^{j2\pi\Omega})|$ is small, the integrand never becomes very large, and the variance will be small. This is in marked contrast with full-response equalization. Here, $H(e^{j2\pi\Omega}) = 1$ for all $\Omega$, and the integrand in the above formula can become large at frequencies where $|F(e^{j2\pi\Omega})|$ is small. Obviously, the smallest possible noise enhancement occurs if $H(e^{j2\pi\Omega})$ is selected so that the integrand is independent of frequency, implying that the noise at the output of the PRE is white. This is, in general, not possible if $H(e^{j2\pi\Omega})$ is restricted to be PR (i.e., small memory-order, integer-valued). The generalized feedback detector of Fig. 18.52, on the other hand, allows a wide variety of causal responses to be used, and here $|H(e^{j2\pi\Omega})|$ can be chosen at liberty. Exploitation of this freedom leads to decision feedback equalization (DFE).

### 18.6.3 Decision Feedback Equalization

This subsection reviews the basics of decision feedback detection. It is again assumed that the channel characteristics are fixed and known, so that the structure of this detector need not be adaptive. Generalizing to variable channel characteristics and adaptive detector structure is tedious, but straightforward.

A DFE detector shown in Fig. 18.54, utilizes the noiseless decision to help remove the ISI. There are two types of ISI: *precursor* ISI (ahead of the detection time) and *postcursor* (behind detection time). Feedforward equalization (FFE) is needed to eliminate the precursor ISI, pushing its energy into the postcursor domain. Supposing all the decisions made in the past are correct, DFE reproduces exactly the modified postcursor ISI (with extra postcursor ISI produced by the FFE during the elimination of precursor ISI), thus eliminating it completely, Fig. 18.55. If the length of the FFE can be made infinitely long, it should be able to completely suppress the precursor ISI, redistributing its energy into the postcursor region, where it is finally cancelled by feedback decision part. No spectrum inverse is needed for this process, so noise boosting is much less than is the case with linear equalizers.

The final decision of the detector is made by the memoryless slicer, Fig. 18.54. The reason why a slicer can perform efficient sequence detection can be explained with the fact that memory of the DFE system is located in two equalizers, so that only symbol-by-symbol detection can suffice. In terms of performance, the DFE is typically much closer to the maximum likelihood sequence detector than to the LE. If the equalization target is not the main cursor, but a PR system, a sequence detection algorithm can be used afterwards. A feasible way to implement this with minimum additional effort is the tree search algorithm used instead of VA [6]. The simple detection circuitry of a DFE, consisting of two equalizers and one slicer, makes implementation possible. The DFE may be regarded as a generalization of the PRE. In the DFE, the trailing portion of the ISI is not suppressed by a forward equalizer but rather canceled by a feedback filter that is excited by past decisions. Fortunately,

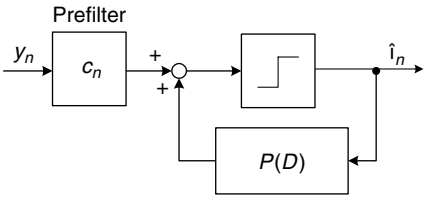



**FIGURE 18.54** Decision feedback equalizer.

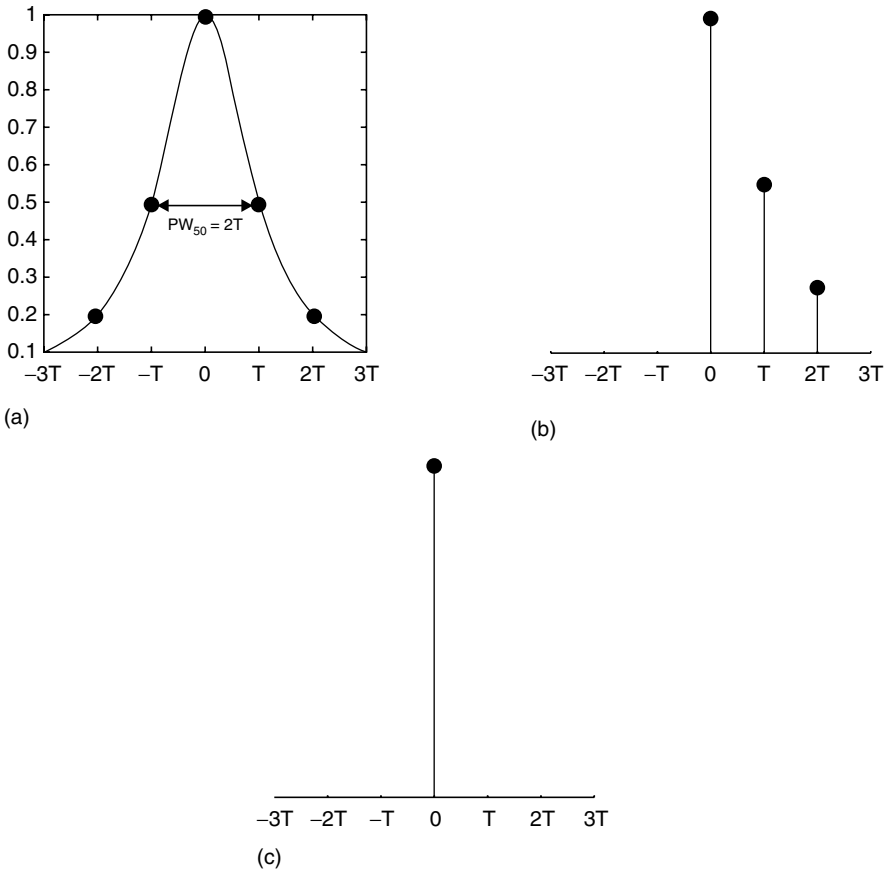



**FIGURE 18.55** Precursor and postcursor ISI elimination with DFE (a) sampled channel response, (b) after feed-forward filter and (c) slicer output.

error propagation is typically only a minor problem and it can, in fact, be altogether avoided through a technique that is called Tomlinson/Harashima precoding.

Performance differences between zero-forcing and minimum mean-square equalizers tend to be considerably smaller in the DFE case than for the LE, and as a result it becomes more dificult to reap SNR benefits from the modulation code. It can be proved that DFE is the optimum receiver with no detection delay. If delay is allowed, it is better to use trellis-based detection algorithms.

### 18.6.3.1 RAM-Based DFE Detection

Decision feedback equalization or RAM-based DFE (Fig. 18.56) is the most frequent alternative to PRML detection. Increase of bit density leads to significant nonlinear ISI in the magnetic recording channel. Both the linear DFE [12,26] and PRML detectors do not compensate for the nonlinear ISI. Furthermore, the implementation complexity of a Viterbi detector matched to the PR channel grows exponentially with the degree of channel polynomial. Actually, in order to meet requirements for a high data transfer rate, high-speed ADC is also needed. In the RAM-based DFE [19,24], the linear feedback section of the linear DFE is replaced with a lookup table. In this way, detector decisions make up a RAM address pointing to the memory location that contains an estimate of the post cursor ISI for the particular symbol sequence. This estimate is subtracted from the output of the forward filter forming the equalizer output. Lookup table size is manageable and typically is less than 256 locations. The major disadvantage of this approach is that it requires complicated architecture and control to recursively update ISI estimates based on equalizer error.

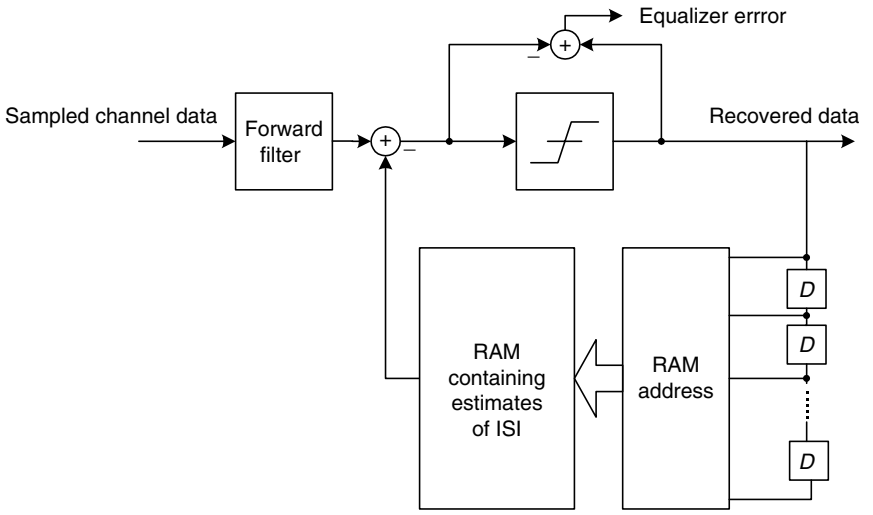



**FIGURE 18.56** Block diagram of a RAM-based DFE.

## 18.6.4 Detection in a Trellis

A trellis-based system can be simply described as a finite state machine (FSM) whose structure may be displayed with the aid of a graph, tree, or trellis diagram. A FSM maps input sequences (vectors) into output sequences (vectors), not necessarily of the same length. Although the system is generally nonlinear and time-varying, linear fixed trellis based systems are usually met. For them,

$$F(a \cdot \mathbf{i}_{[0,\infty]}) = a \cdot F(\mathbf{i}_{[0,\infty]})$$
$$F\left(\mathbf{i}'_{[0,\infty]} + \mathbf{i}''_{[0,\infty]}\right) = F\left(\mathbf{i}'_{[0,\infty]}\right) + F\left(\mathbf{i}''_{[0,\infty)}\right)$$

where $a$ is a constant, $\mathbf{i}_{[0,\infty]}$ is any input sequence and $F(\mathbf{i}_{[0,\infty]})$ is the corresponding output sequence. It is assumed that input and output symbols belong to a subset of a field. Also, for any $d > 0$, if $\mathbf{x}_{[0,\infty]} = F(\mathbf{i}_{[0,\infty]})$ and $\mathbf{i}'_l = \mathbf{i}_{l-d}$, $\mathbf{i}'_{[0,d]} = \mathbf{0}_{[0,d]}$ then $F(\mathbf{i}'_{[0,\infty]}) = \mathbf{x}'_{[0,\infty]}$, where $\mathbf{x}'_l = \mathbf{x}_{l-d}$, $\mathbf{x}'_{[0,d]}, = \mathbf{0}_{[0,d]}$. It is easily verified that $F(\cdot)$ can be represented by the convolution, so that $\mathbf{x} = \mathbf{i} * \mathbf{h}$, where $\mathbf{h}$ is the system impulse response (this is also valid for different lengths of $\mathbf{x}$ and $\mathbf{i}$ with a suitable definition of $\mathbf{h}$). If $\mathbf{h}$ is of finite duration, $M$ denotes the system memory length.

Let us now consider a feedforward FSM with memory length $M$. At any time instant (depth or level) $l$, the FSM output $\mathbf{x}_l$ depends on the current input $\mathbf{i}_l$ and $M$ previous inputs $\mathbf{i}_{l-1}, \ldots, \mathbf{i}_{l-M}$. The overall functioning of the system can be mapped on a trellis diagram, whereon a node represents one of $q^M$ encoder states ($q$ is the cardinality of the input alphabet including the case when the input symbol is actually a subsequence), while a branch connecting two nodes represents the FSM output associated to the transition between the corresponding system states.

A trellis, which is a visualization of the state transition diagram with a time element incorporated, is characterized by $q$ branches stemming from and entering each state, except in the first and last $M$ branches (respectively called head and tail of the trellis). The branches at the $l$th time instant are labeled by sequences $\mathbf{x}_l \in X$. A sequence of $l$ information symbols, $\mathbf{i}_{[0,l]}$ specifies a path from the root node to a node at the $l$th level and, in turn, this path specifies the output sequence $\mathbf{x}_{[0,l]} = \mathbf{x}_0 \bullet \mathbf{x}_1 \bullet \ldots \bullet \mathbf{x}_{l-1}$, where $\bullet$ denotes concatenation of two sequences.

The input can, but need not, be separated in frames of some length. For framed data, where the length of each input frame equals $L$ branches (thus $L$ $q$-ary symbols) the length of the output frame

is $L+M$ branches ($L+M$ output symbols), where the $M$ known symbols (usually all zeros) are added at the end of the sequence to force the system into the desired terminal state. It is said that such systems suffer a fractional rate loss by $L/(L+M)$. Clearly, this rate loss has no asymptotic significance.

In the sequel, the detection of the input sequence, $\mathbf{i}_{(0,\ \infty)}$, will be analyzed based on the corrupted output sequence $\mathbf{y}_{[0,\infty]} = \mathbf{x}_{[0,\infty]} + \mathbf{u}_{[0,\infty]}$. Suppose there is no feedback from the output to the input, so that

$$P[y_n|x_0,\ldots,x_{n-1},x_n,y_0,\ldots,y_{n-1}] = P[y_n|x_n]$$

and

$$P[y_1,\ldots,y_N|x_1,\ldots,x_N] = \prod_{n=1}^{N} P[y_n|x_n]$$

Usually, $\mathbf{u}_{(0,\infty)}$ is a sequence that represents additive white Gaussian noise sampled and quantized to enable digital processing.

The task of the detector that minimizes the sequence error probability is to find a sequence which maximizes the joint probability of input and output channel sequences

$$P\left(\mathbf{y}_{[0,L+M]},\mathbf{x}_{[0,L+M]}\right) = P\left(\mathbf{y}_{[0,L+M]}|\mathbf{x}_{[0,L+M]}\right)P(\mathbf{x}_{[0,L+M]})$$

Since usually the set of all probabilities $P[\mathbf{x}_{[0,\ L+M]}]$ is equal, it is sufficient to find a procedure that maximizes $P[\mathbf{y}_{[0,\ L+M]}|\mathbf{x}_{[0,\ L+M]}]$, and a decoder that always chooses as its estimate one of the sequences that maximize it or

$$\mu\left(\mathbf{y}_{[0,L+M]}|\mathbf{x}_{[0,L+M]}\right) = A\log_2 P\left(\mathbf{y}_{[0,L+M]}|\mathbf{x}_{[0,L+M]}\right)$$
$$-f\left(\mathbf{y}_{[0,L+M]}\right) = A\sum_{l=0}^{L+M}\log_2\left(P[\mathbf{y}_l|\mathbf{x}_l] - f(\mathbf{y}_l)\right)$$

(where $A \geq 0$ is a suitably chosen constant, and $f(\cdot)$ is any function) is called a maximum-likelihood decoder (MLD). This quantity is called a metric, $\mu$. This type of metric suffers one significant disadvantage because it is suited only for comparison between paths of the same length. Some algorithms, however, employ a strategy of comparing paths of different length or assessing likelihood of such paths with the aid of some thresholds. The metric that enables comparison for this type of algorithms is called the Fano metric. It is defined as

$$\mu_F\left(\mathbf{y}_{[0,l]}|\mathbf{x}_{[0,l]}\right) = A\log_2\frac{P\left(\mathbf{y}_{[0,l]},\mathbf{x}_{[0,l]}\right)}{P(y_{[0,l]})}$$
$$= A\sum_{n=0}^{l}\left(\log_2\frac{P[\mathbf{y}_n|\mathbf{x}_n]}{P[\mathbf{y}_n]} - R\right)$$

If the noise is additive, white, and Gaussian (an assumption that is not entirely true, but that usually yields systems of good performances), the probability distribution of its sample is

$$p[\mathbf{y}_n|\mathbf{x}_n] = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(\mathbf{y}_n-\mathbf{x}_n)^2}{2\sigma^2}\right)$$

The ML metric to be used in conjunction with such a noise is the logarithm of this density, and thus proportional to $-(\mathbf{y}_n - \mathbf{x}_n)^2$, i.e., to the negative squared Euclidean distance of the readback and supposed written signal. Thus, maximizing likelihood amounts to minimizing the squared Euclidean distance of the two sequences, leading to minimizing the squared Euclidean distance between two sampled sequences given by $\Sigma_n(y_n - x_n)^2$.

The performance of a trellis-based system, as is the case with PR systems, depends on the detection algorithm employed and on the properties of the system itself. The distance spectrum is the property of the system that constitutes the main factor of the event error probability of a ML (optimum) detector, if the distance is appropriately chosen for the coding channel used [45]. For PR channels with additive white Gaussian noise, it is the squared Euclidean distance that has to be dealt with. Naturally, since the noise encountered is neither white, nor entirely Gaussisan, this is but an approximation to the properly chosen distance measure.

As stated previously, the aim of the search procedure is to find a path with the highest possible likelihood, i.e., metric. There are several possible classifications of detecting procedures. This classification is in-line with systematization made in coding theory, due to fact that algorithms developed for decoding in a trellis are general so that it could be applied to problem of detection in any trellis-based system as well. According to detector's strategies in extending the most promising path candidates we classify them into breadth-first, metric-first, and depth-first, bidirectional algorithms, and into sorting and nonsorting depending on whether the procedure performs any kind of path comparison (sifting or sorting) or not. Moreover, detecting algorithms can be classified into searches that *minimize* the *sequence* or *symbol* error rate.

The usual measure of algorithm efficiency is its complexity (arithmetic and storage) for a given probability of error. In the strict sense, arithmetic or computational complexity is the number of arithmetic operations per detected symbol, branch, or frame; however, it is a usual practice to track only the number of node computations, which makes sense because all such computations require approximately the same number of basic machine instructions. A node computation (or simply computation) is defined as the total number of nodes extended (sometimes it is the number of metrics computed) per detected branch or information frame $\mathbf{i}_{[0,L+M]}$. One single computation consists of determining the state in which the node is computing the metrics of all its successors. For most practical applications with finite frame length, it is usually sufficient to observe node computations since a good prediction of search duration can be precisely predicted. Nevertheless, for asymptotic behavior it is necessary to track the sorting requirements too. Another important aspect of complexity is storage (memory or space), which is the amount of auxiliary storage that is required for detecting memory, processors working in parallel, etc. Thus, space complexity of an algorithm is the size (or number) of resources that must be reserved for its use, while the computational, or more precisely time complexity, reflects the number of accesses to this resources taking into account that any two operations done in parallel by the spatially separated processors should be counted as one. The product of these two, the time-space complexity, is possibly the best measure of the algorithm cost for it is insensitive to time-space tradeoff such as parallelization or the use of precomputed tables, although it also makes sense to keep the separate track of these two. Finally, for selecting which algorithm to use, one must consider additional details that we omit here, but which can sometimes cause unexpected overall performance or complicate the design of a real-time detector. They include complexity of the required data structure, buffering needs, and applicability of available hardware components.

### 18.6.4.1 Basic Breadth-First Algorithms

#### *18.6.4.1.1 The Viterbi Algorithm (VA)*

The VA was introduced in 1967 as a method of decoding convolutional codes. Forney showed in 1972 [7] that the VA solves the maximum-likelihood sequence detection (MLSD) problem in the presence of ISI and additive white noise. Kobayashi and Tang [8] recognized that this algorithm is

possible to apply in magnetic recording systems for detection purposes. Strategy to combine Viterbi detector with PR equalization in magnetic recording channel resulted with many commercial products.

The VA is an optimal decoding algorithm in the sense that it always finds the nearest path to the noisy modification of the FSM output sequence $\mathbf{x}_{[0,L+M)}$, and it is quite useful when FSM has a short memory. The key to Viterbi (maximum-likelihood, ML) decoding lies in the *Principle of Nonoptimality* [17]. If the paths $\mathbf{i}'_{[0,l]}$ and $\mathbf{i}''_{[0,l]}$ terminate at the same state of the trellis and

$$\mu(\mathbf{y}_{[0,l]}, \mathbf{x}'_{[0,l]}) > \mu(\mathbf{y}_{[0,l]}, \mathbf{x}''_{[0,l]})$$

then $\mathbf{i}''_{[0,l]}$ cannot be the first $l$ branches of one of the paths $\mathbf{i}_{[0,L+M)}$ that maximize the overall sequence metric. This principle which some authors call the *Principle of Optimality* literally specifies the most efficient MLD procedure for decoding/detecting in the trellis.

To apply VA as an ML sequence detector for a PR channel, we need to define the channel trellis describing the amount of controlled ISI. Once we define the PR channel polynomial, it is an easy task. An example of such trellis for PR4 channel with $P(D) = 1 - D^2$ is depicted in Fig. 18.57. The trellis for this channel consists of four states according to the fact that channel input is binary and channel memory is 2, so that there are four possible state values (00, 10, 01, 11). Generally, if the channel input sequence can take $q$ values, and the PR channel forms the ISI from the past $M$ input symbols, then the PR channel can be described by a trellis with $q^M$ states. Branches joining adjacent states are labeled with the pair of expected noiseless symbols in the form channel_output/channel_input. Equalization to $P(D) = 1 - D^2$ results in ternary channel output, taking values $\{0, \pm 1\}$. Each noiseless output channel sequence is obtained by reading the sequence of labels along some path through the trellis.

Now the task of detecting $\mathbf{i}_{[0,\infty]}$ is to find $\mathbf{x}_{[0,\infty]}$ that is closest to $\mathbf{y}_{[0,\infty]}$ in the Euclidean sense. Recall that we stated as an assumption that channel noise is AWGN, while in magnetic recording systems after equalization the noise is colored so that the minimum-distance detector is not an optimal one, and additional post-processing is necessary, which will be addressed later in this chapter.

The Viterbi algorithm is a classical application of dynamic programming. Structurally, the algorithm contains $q^M$ lists, one for each state, where the paths whose states correspond to the label indices are stored, compared, and the best one of them retained. The algorithm can be described recursively as follows:

1. *Initial condition*: Initialize the starting list with the root node (the known initial state) and set its metric to zero, $l = 0$.
2. *Path extension*: Extend all the paths (nodes) by one branch to yield new candidates, $l = l + 1$, and find the sum of the metric of the predecessor node and the branch metric of the connecting branch (ADD). Classify these candidates into corresponding $q^M$ lists (or less for $l < M$). Each list (except in the head of the trellis) contains $q$ paths.

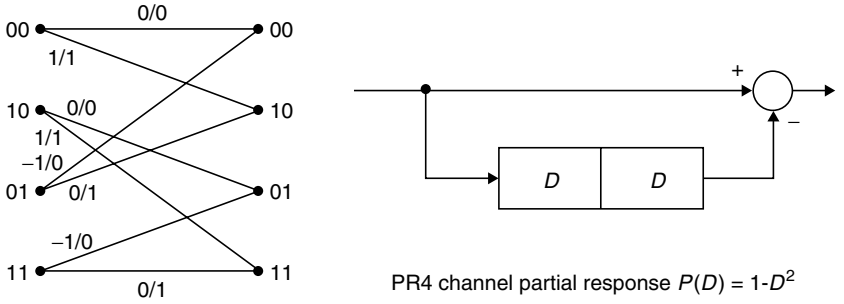



**FIGURE 18.57** PR4 channel trellis.

3. *Path selection*: For each end-node of extended paths determine the maximum/minimum* of these sums (COMPARE) and assign it to the node. Label the node with the best path metric to it, selecting (SELECT) that path for the next step of the algorithm (discard others). If two or more paths have the same metric, i.e., if they are equally likely, choose the best one at random. Find the best of all the survivor paths, $\mathbf{x}'_{[0,l]}$, and its corresponding information sequence $\mathbf{i}'_{[0,l]}$ and release the bit $\mathbf{i}'_{[l-\delta]}$. Go to step 2.

In the description of the algorithm we emphasized three Viterbi-characteristic operations—add, compare, select (ADC)—that are performed in every recursion of the algorithm. So today's specialized signal processors have this operation embedded optimizing its execution time. Consider now the amount of "processing" done at each depth $l$, where all of the $q^M$ states of the trellis code are present. For each state it is necessary to compare $q$ paths that merge in that state, discard all but the best path, and then compute and send the metrics of $q$ of its successors to the depth $l+1$.

Consequently, the computational complexity of the VA exponentially increases with $M$. These operations can be easily parallelized, but then the number of parallel processors rises as the number of node computations decreases. The total time-space complexity of the algorithm is fixed and increases exponentially with the memory length.

The sliding window VA decodes infinite sequences with delay of $\delta$ branches from the last received one. In order to minimize its memory requirements ($\delta+1$ trellis levels), and achieve bit error rate only insignificantly higher than with finite sequence VA, $\delta$ is chosen as $\delta \approx 4M$. In this way, the Viterbi detector introduces a fixed decision delay.

### 18.6.4.2 Example

Assume that a recorded channel input sequence $\mathbf{x}$, consisting of $L$ equally likely binary symbols from the alphabet {0, 1}, is "transmitted" over PR4 channel. The channel is characterized by the trellis of Fig. 18.57, i.e., all admissible symbol sequences correspond to the paths traversing the trellis from $l=0$ to $l=L$, with one symbol labeling each branch, Fig. 18.58. Suppose that the noisy sequence of

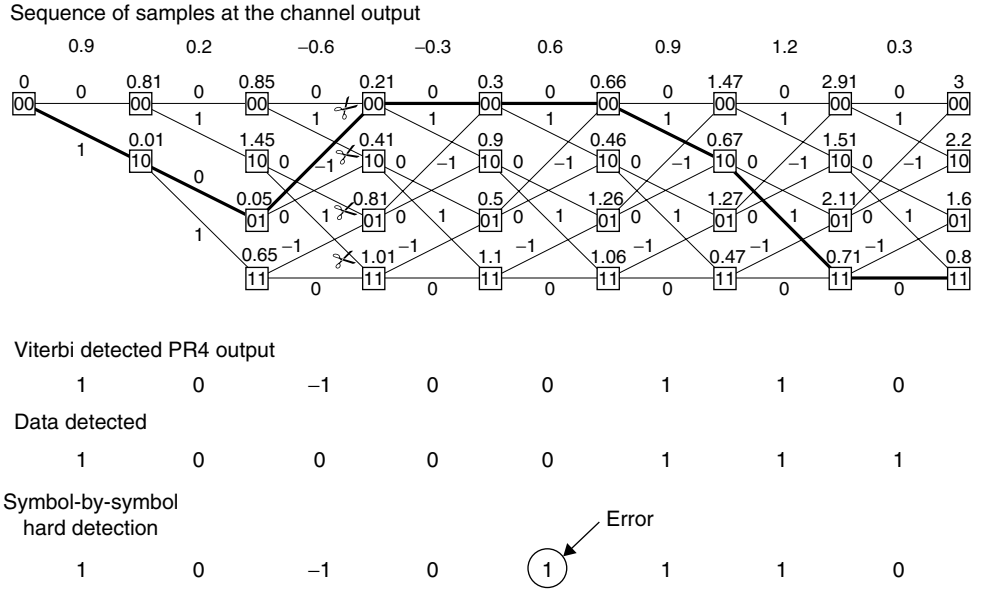



**FIGURE 18.58** Viterbi algorithm detection on the PR4 trellis.

*It depends on whether the metric or the distance is accumulated.

samples at the channel output is $\mathbf{y} = 0.9,\ 0.2,\ -0.6,\ -0.3,\ 0.6,\ 0.9,\ 1.2,\ 0.3, \ldots$ If we apply a simple symbol-by-symbol detector to this sequence, the fifth symbol will be erroneous due to the hard quantization rule for noiseless channel output estimate

$$\hat{y}_k = \begin{cases} -1 & y_k < -0.5 \\ 1 & y_k > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

The Viterbi detector will start to search the trellis accumulating branch distance from sequence **y**. In the first recursion of the algorithm, there are two paths of length 1 at the distance

$$d(\mathbf{y},0) = (0.9 - 0)^2 = 0.81$$
$$d(\mathbf{y},1) = (0.9 - 1)^2 = 0.01$$

from **y**. Next, each of the two paths of length 1 are extended in two ways forming four paths of length 2 at squared Euclidean distance from the sequence **y**

$$d(\mathbf{y},(0,0)) = 0.81 + (0.2 - 0)^2 = 0.85$$
$$d(\mathbf{y},(0,1)) = 0.81 + (0.2 - 1)^2 = 1.45$$
$$d(\mathbf{y},(1,0)) = 0.01 + (0.2 - 0)^2 = 0.05$$
$$d(\mathbf{y},(1,1)) = 0.01 + (0.2 - 1)^2 = 0.65$$

and this accumulated distance of four paths labels the four trellis states. In the next loop of the algorithm each of the paths are again extended in two ways to form eight paths of length 3, two paths to each node at level (depth) 3.

Node 00

$$d(\mathbf{y},(0,0,0)) = 0.85 + (-0.6 - 0)^2 = 1.21$$
$$d(\mathbf{y},(1,0,-1)) = 0.05 + (-0.6 + 1)^2 = 0.21 \quad \textit{surviving path}$$

Node 10

$$d(\mathbf{y},(0,0,1)) = 0.85 + (-0.6 - 1)^2 = 3.41$$
$$d(\mathbf{y},(1,0,0)) = 0.05 + (-0.6 - 0)^2 = 0.41 \quad \textit{surviving path}$$

Node 01

$$d(\mathbf{y},(0,1,0)) = 1.45 + (-0.6 - 0)^2 = 1.81$$
$$d(\mathbf{y},(1,1,-1)) = 0.65 + (-0.6 + 1)^2 = 0.81 \quad \textit{surviving path}$$

Node 11

$$d(\mathbf{y},(0,1,1)) = 1.45 + (-0.6 - 1)^2 = 4.01$$
$$d(\mathbf{y},(1,1,0)) = 0.65 + (-0.6 - 0)^2 = 1.01 \quad \textit{surviving path}$$

Four paths of length 3 are selected as the surviving most likely paths to the four trellis nodes. The procedure is repeated and the detected sequence is produced after a delay of $4M = 8$ trellis sections.

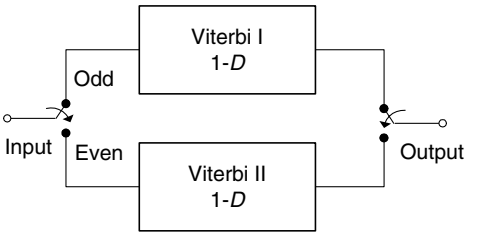



**FIGURE 18.59** Implementation of 1-$D^2$ Viterbi detector with two half-rate, 1-$D$ detectors.

Note, Fig. 18.58, that the symbol-by-symbol detector error is now corrected. Contrary to this example, a 4-state PR4ML detector is implemented with two interleaved 2-state dicode, $(1 - D)$, detectors each operating at one-half the symbol rate of one full-rate PR4 detector [35]. The sequence is interleaved, such that the even samples go to the first and the odd to the second dicode detector, Fig. 18.59, so the delay $D$ in the interleaved detectors is actually twice the delay of the PR4 detector. A switch at the output resamples the data to get them out in the correct order.

For other PR channels this type of decomposition is not possible, so that their complexity can become great for real-time processing. In order to suppress some of the states in the corresponding trellis diagram of those PR systems, thus simplifying the sequence detection process, some data loss has to be introduced. For instance, in conjunction with precoding (1,7) code prohibits two states in EPR4 trellis: [101] and [010]. This can be used to reduce the 8-state EPR4 trellis to 6-state trellis depicted in Fig. 18.60 and the number of add-compare-select units in the VA detector to 4. The data rate loss is 33% in this case. Using the (2,7) code eliminates two more states, paying the complexity gain by a 50% data rate loss.

Because VA involves addition, multiplication, compare and select functions, which require complex circuitry at the read side, simplifications of the receiver for certain PRs were sought. One of them is the dynamic threshold technique [22]. This technique implies generating a series of thresholds. The read-back samples are compared with them, just as for the threshold detector, and are subsequently included in their modification. While preserving the full function of the ML detector, this technique saves a substantial fraction of the necessary hardware. Examples of dynamic threshold detectors are given in [30] and [6].

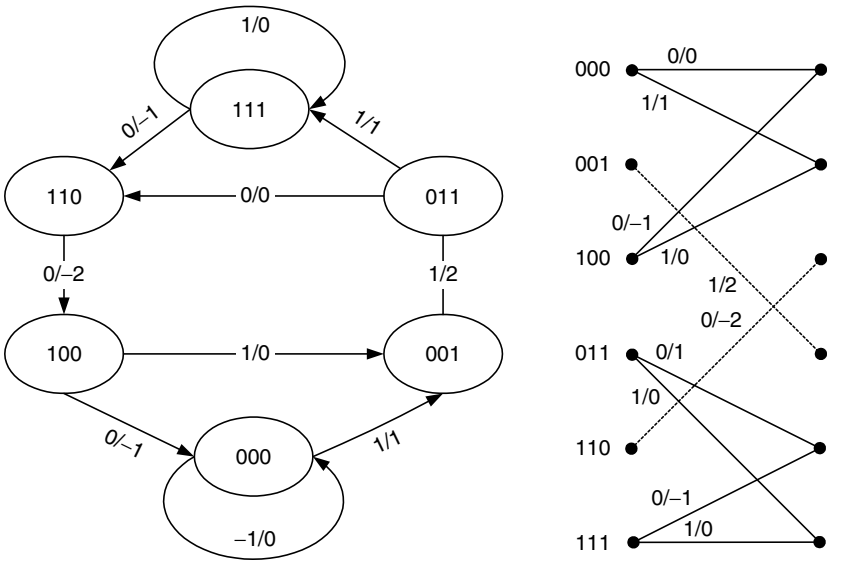



**FIGURE 18.60** (1,7) coded EPR4 channel.

#### *18.6.4.2.1 Noise-Predictive Maximum Likelihood Detectors*

Maximum likelihood detection combined with PR equalization is a dominant type of detection electronics in today's digital magnetic recording devices. As described earlier, in order to simplify hardware realization of the receiver, the degree of the target PR polynomial is chosen to be small with integer coefficients to restrict complexity of Viterbi detection trellis. On the other hand, if the recording density is increased, to produce longer ISI, equalization to the same PR target will result in substantial noise enhancement and detector performance degradation. Straightforward solution is to increase the duration of the target PR polynomial decreasing the mismatch between channel and equalization target. Note that this approach leads to undesirable increase in detector complexity fixing the detector structure in a sense that its target polynomial cannot be adapted to changing channel density.

The noise-predictive maximum likelihood (NPML) detector [20,32] is an alternative data detection method that improves reliability of the PRML detector. This is achieved by embedding a noise prediction/whitening process into the branch metric computation of a Viterbi detector. Using reduced-state sequence-estimation [43] (see also the description of the generalized VA in this chapter), which limits the number of states in the detector trellis, compensates for added detector complexity.

A block diagram of a NPML system is shown in Fig. 18.61. The input to the channel is binary sequence, **i**, which is written on the disk at a rate of $1/T$. In the readback process data are recovered via a lowpass filter as an analog signal $y(t)$, which can be expressed as $y(t) = \sum_n i_n h(t - nT) + u(t)$ where $h(t)$ denotes the pulse response and $u(t)$ is the additive white Gaussian noise. The signal $y(t)$ is sampled periodically at times $t = nT$ and shaped into the PR target response by the digital equalizer. The NPML detector then performs sequence detection on the PR equalized sequence **y** and provides an estimate of the binary information sequence **i**. Digital equalization is performed to fit the overall system transfer function to some PR target, e.g., the PR4 channel.

The output of the equalizer $y_n + i_n + \sum_{i=1}^{M} f_i x_{n-i} + w_n$ consists of the desired response and an additive total distortion component $w_n$, i.e., the colored noise and residual interference. In conventional PRML detector, an estimate of the recorded sequence is done by the minimum-distance criteria as described for the Viterbi detector. If the mismatch between channel and PR target is significant, the power of distortion component $w_n$ can degrade the detector performance. The only additional component compared to the Viterbi detector, NPML noise-predictor, reduces the power of the total distortion by whitening the noise prior to the Viterbi detector. The whitened total distortion component (Fig. 18.62) of the PR equalized output $y_n$ is

$$w_n - \hat{w}_n = w_n - \sum_{i=1}^{N} w_{n-i} p_i$$

where the $N$-coefficient MMSE predictor transfer polynomial is $P(D) = p_1 D^1 + p_2 D^2 + \cdots + p_N D^N$. Note that an estimate of the current noise sample $\hat{w}_n$ is formed based on estimates of previous $N$ noise

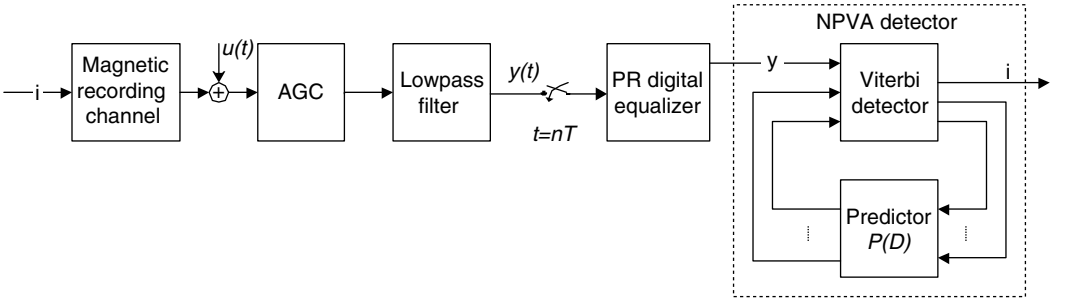



**FIGURE 18.61** Block diagram of NPVA detector.

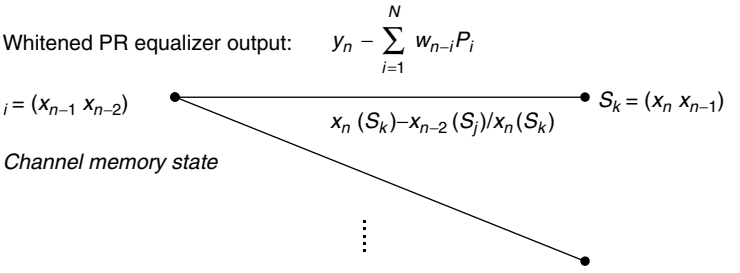



**FIGURE 18.62** NPML metric computation for PR4 trellis.

samples. Assuming the PR4 equalization of sequence **y**, the metric of the Viterbi detector can be modified in order to compensate for distortion component. In this case, the equalizer output is $y_n = x_n - x_{n-2} + w_n$ and the NPML distance is

$$\left[\left(y_n - \sum_{i=l}^{N} w_{n-i}p_i\right) - (x_n(S_k) - x_{n-2}(S_j))\right]^2$$
$$= \left[\left(y_n - \sum_{i=l}^{N} (y_{n-i} - \hat{x}_{n-i}(S_j) - \hat{x}_{n-i-2}(S_j))p_i\right) - (x_n(S_k) - x_{n-2}(S_j))\right]^2$$

where $\hat{x}_{n-i}(S_j)$, $\hat{x}_{n-i-2}(S_j)$ represent past decisions taken from the Vitrebi survivor path memory associated with state $S_j$. The last expression gives the flavor of this technique, but it is not suitable for implementation so that the interested reader can find details in [20] how to modify this equation for RAM lookup realization. Furthermore, in the same paper, a description of the general procedure to compute the predictor coefficients based on the autocorrelation of the total distortion $w_n$ at the output of a finite-length PR equalizer is given.

#### *18.6.4.2.2 Postprocessor*

As explained earlier, Viterbi detector improves the performance of a read channel by tracing the correct path through the channel trellis [8]. Further performance improvement can be achieved by using soft output Viterbi algorithm (SOVA) [14]. Along with the bit decisions, SOVA produces the likelihood of these decisions, that combined create *soft information*. In principle, soft information can be passed to hard drive controller and used in RS decoder that resides there, but at the present time soft decoding of RS codes is still too complex to be implemented at 1 Gb/s speeds. Alternatively, much shorter inner code is used. Because of the nonlinear operations on bits performed by the modulation decoder logic, the inner code is used in inverse concatenation with modulation encoder in order to simplify calculation of bit likelihood. Due to the channel memory and noise coloration, Viterbi detector produces some error patterns more often than others [5], and the inner code is designed to correct these so-called *dominant error sequences* or *error events*. The major obstacle for using soft information is the speed limitations and hardware complexity required to implement SOVA. Viterbi detector is already a bottleneck and the most complex block in a read channel chip, occupying most of the chip area, and the architectural challenges in implementing even more complex SOVA would be prohibitive. Therefore, a postprocessor architecture is used [18]. The postprocessor is a block that resides after Viterbi detector and comprises the block for calculating error event likelihood and an inner-soft error event correcting decoder.

The postprocessor is designed by using the knowledge on the set of dominant error sequences $E = \{e_i\}_{l \le i}$ and their occurrence probabilities $P = (p_i)_{1 \le i}$. The index $i$ is referred to as an *error type*, while the position of the *error event end* within a codeword is referred as an *error position*. The relative frequencies of error events will strongly depend on recording density [36]. The detection is based on the fact that we can calculate the likelihoods of each of dominant error sequences at each point in time. The

parity bits detect the errors, and provide localization in error type and time. The likelihoods are then used to choose the most likely error events for corrections.

The error event likelihoods are calculated as the difference in the squared Euclidean distances between the signal and the convolution of maximum likelihood sequence estimate and the channel PR, versus that between the signal and the convolution of an alternative data pattern and the channel PR. During each clock cycle, the best $M$ of them are chosen, and the syndromes for these error events are calculated. Throughout the processing of each block, a list is maintained of the $N$ most likely error events, along with their associated error types, positions and syndromes. At the end of the block, when the list of candidate error events is finalized, the likelihoods and syndromes are calculated for each of $\binom{N}{L}$ combinations of $L$-set candidate error events that are possible. After disqualifying those $L$-sets of candidates, which overlap in the time domain, and those candidates and $L$-sets of candidates, which produce a syndrome which does not match the actual syndrome, the candidate or $L$-set of candidates, which remains and which has the highest likelihood is chosen for correction. Finding the error event position and type completes decoding.

The decoder can make two types of errors: it fails to correct if the syndrome is zero, or it makes a wrong correction if the syndrome is nonzero, but the most likely error event or combination of error events does not produce the right syndrome. A code must be able to detect a single error from the list of dominant error events and should minimize the probability of producing zero syndrome when more than one error event occurs in a codeword. Consider a linear code given by an $(n-k)\times n$ parity check matrix $H$. We are interested in capable of correcting or detecting dominant errors. If all errors from a list were contiguous and shorter than $m$, a cyclic $n-k=m$ parity bit code could be used to correct a single error event [16]; however, in reality, the error sequences are more complex, and occurrence probabilities of error events of lengths 6, 7, 8 or more are not negligible. Furthermore, practical reasons (such as decoding delay, thermal asperities, etc.) dictate using short codes, and consequently, in order to keep the code rate high, only a relatively small number of parity bits is allowed, making the design of error event detection codes nontrivial. The code redundancy must be used carefully so that the code is optimal for a given $E$.

The parity check matrix of a code can be created by a recursive algorithm that adds one column of $H$ at a time using the criterion that after adding each new column, the code error-event-detection capabilities are still satisfied. The algorithm can be described as a process of building a directed graph whose vertices are labeled by the portions of parity check matrix long enough to capture the longest error event, and whose edges are labeled by column vectors that can be appended to the parity check matrix without violating the error event detection capability [4]. To formalize code construction requirements, for each error event from $E$, denote by $s_{i,l}$ a syndrome of error vector $\sigma_l(e_i)$ $(s_{i,l}=\sigma_l(e_i)\cdot H^T)$, where $\sigma_l(e_i)$ is an $l$-time shifted version of error event $e_i$. The code should be designed in such a way that any shift of any dominant error sequence produces a nonzero syndrome, i.e., that $s_{i,l}\neq 0$ for any $1\leq i\leq I$ and $1\leq l\leq n$. In this way, a single error event can be detected (relying on error event likelihoods to localize the error event). The correctable shifts must include negative shifts as well as shifts larger than $n$ in order to cover those error events that straddle adjacent codewords, because the failure to correct straddling events significantly affects the performance. A stronger code could have a parity check matrix that guaranties that syndromes of any two-error event-error position pairs $((i_1, l_1), (i_2, l_2))$ are different, i.e., $s_{i_1,l_1}\neq s_{i_2,l_2}$. This condition would result in a single error event correction capability. The codes capable of correcting multiple error events can be defined analogously. We can even strengthen this property and require that for any two shifts and any two dominant error events, the Hamming distance between any pair of syndromes is larger than $\delta$; however, by strengthening any of these requirements the code rate decreases.

If $L_i$ is a length of the $i$th error event, and if $L$ is the length of the longest error event from $E$, $(L=\max_{1\leq i\leq I}\{L_i\})$, then it is easy to see that for a code capable of detecting an error event from $E$ that ends at position $j$, the linear combination of error events and the columns of $H$ from $j-L+1$ to $j$ has to be nonzero. More precisely, for any $i$ and any $j$ (ignoring the codeword boundary effects)

$$\sum_{1 \le m \le L_i} e_{i,m} \cdot h^T_{j-L_i+m} \neq 0$$

where $e_{i,m}$ is the $m$th element of the error event $e_i$, and $h_j$ is the $j$th column of $H$.

## 18.6.5 Advanced Algorithms and Algorithms under Investigation

This subsection gives a brief overview of less complex procedures for searching the trellis. It is intended to give background information that can be used in future development if it shows up that NPVA detectors and postprocessing are not capable of coping with ever-increasing storage densities and longer PRs needed for them. In such cases, a resort has to be made to some sort of reduced complexity suboptimal algorithms, whose performance is close to optimal. Explained algorithms are not yet implemented in commercial products, but all of them are a natural extension of already described procedures for searching the trellis.

### 18.6.5.1 Other Breadth-First Algorithms

#### *18.6.5.1.1 The M-Algorithm*

Since most survivors in the VA usually possess much smaller metrics than does the best one, all the states or nodes kept are not equally important. It is intuitively reasonable to assume that unpromising survivors can be omitted with a negligible probability of discarding the best one. The $M$-algorithm [10] is one such modification of the VA; all candidates are stored in a single list and the best $M \le q^M$ survivors are selected from the list in each cycle. The steps of the $M$-algorithm are:

1. *Initial condition*: Initialize the list with the root node and set its metric to zero.
2. *Path extension*: Extend all the paths of length $l$ by one branch and classify all contenders (paths of length $l+1$) into the list. If two or more paths enter the same state keep the best one.
3. *Path selection*: From the remaining paths find the best $M$ candidates and delete the others. If $l = L + M$, take the only survivor and transfer its corresponding information sequence to the output (terminated case, otherwise use the sliding window variation). Otherwise, go to step 2.

Defined in this way, the $M$-algorithm performs trellis search, while, when the state comparison in step 2 is omitted, it searches the tree, saving much time on comparisons but with slightly increased error probability. When applied to decoding/detecting infinitely long sequences, it is usual that comparisons performed in step 2 are substituted with the so-called ambiguity check [10] and a release of one decoded branch. In each step this algorithm performs $M$ node computations, and employing any sifting procedure (since the paths need not be sorted) perform $\sim Mq$ metric comparisons. If performed, the Viterbi-type discarding of step 2 requests $\sim M^2 q$ state and metric comparisons. This type of discarding can be performed with $\sim M \log_2 M$ comparisons (or even linearly) but than additional storage must be provided. The space complexity grows linearly with the information frame length $L$ and parameter $M$.

#### *18.6.5.1.2 The Generalized Viterbi Algorithm*

In contrast to the VA, which is a multiple-list single survivor algorithm, the $M$-algorithm is a single-list multiple-survivor algorithm. The natural generalization to a multiple-list multiple-survivor algorithm was first suggested by Hashimoto [39]. Since all the lists are not equally important, this algorithm, originally called the generalized Viterbi algorithm (GVA), utilizes only $q^{M_1}$ lists (labels), where $M_1 \le M$. In each list from all $q^{M-M_1+1}$ paths, it retains the best $M_1$ candidates. The algorithm can be described as follows:

1. *Initial condition*: Initialize the starting label with the root node and set its metric to zero.
2. *Path extension*: Extend all the paths from each label by one branch and classify all successors into the appropriate label. If two or more paths enter the same state keep the best one.
3. *Path selection*: From the remaining paths of each label find the best $M_1$ and delete the others. If $l = L + M$, take the only survivor and transfer its information sequence to the output (for the terminated case, otherwise use the sliding window variant). Go to step 2.

When $M_1 = M$, and $M_1 = 1$, the GVA reduces to the VA, and for $M_1 = 0$, $M_1 = M$ it reduces to the $M$-algorithm. Like the $M$-algorithm, GVA in each step performs $M_1$ node computations per label, and employing any sifting procedure $\sim M_1 q$ metric comparisons. If performed, the Viterbi-type discarding of step 2 requests $\sim M_1^2 q$ or less state and metric comparisons per label.

### 18.6.5.2 Metric-First Algorithms

Metric-first and depth-first sequential detection is a name for a class of algorithms that compare paths according to their Fano metric (one against another or with some thresholds) and on that basis decide which node to extend next, which to delete in metric first procedures or whether to proceed with current branch or go back. These algorithms generally extend fewer nodes for the same performance, but have increased sorting requirements.

Sequential detecting algorithms have a variable computation characteristic that results in large buffering requirements, and occasionally large detecting delays and/or incomplete detecting of the received sequence. Sometimes, when almost error-free communication is required or when retransmission is possible, this variable detecting effort can be an advantage. For example, when a detector encounters an excessive number of computations, it indicates that a frame is possibly very corrupted meaning that the communication is insufficiently reliable and can ultimately cause error patterns in detected sequence. In such situations the detector gives up detecting and simply requests retransmission. These situations are commonly called erasures, and detecting incomplete. A complete decoder such as the Viterbi detector/decoder would be forced to make an estimate, which may be wrong. The probability of buffer overflow is several orders of magnitude larger than the probability of incorrect decision when the decoder operates close to the so-called (computational) cutoff rate.

The performance of sequential detecting has traditionally been evaluated in terms of three characteristics: the probability of sequence error, the probability of failure (erasure), and the Pareto exponent associated with detecting effort.

#### *18.6.5.2.1 The Stack Algorithm*

The stack (or ZJ) algorithm was for the first time suggested by Zigangirov [1] and later independently by Jelinek [1]. As its name indicates, the algorithm contains a stack (in fact, a list) of already searched paths of varying lengths, ordered according to their metric values. At each step, the path at the top of the stack (the best one) is replaced by its $q$ successors extended by one branch, with correspondingly augmented metrics. The check whether two or more paths are in the same state is not performed. This algorithm has its numerous variations and we first consider the basic version that is closest to Zigangirov's:

1. *Initial condition*: Initialize the stack with the root node and set its Fano metric to zero (or some large positive number to avoid arithmetic with negative numbers, but low enough to avoid overflow).
2. *Path extension*: Extend the best path from the stack by one branch, delete it, sort all successors, and then merge them with the stack so that it is ordered according to the path metrics.
3. *Path selection*: Retain the best $Z$ paths according to the Fano metric. If the top path has the length $l = L + M$ branches, transfer its information sequence to the output (terminated case; otherwise, a sliding window version has to be used); otherwise, go to step 2.

It is obvious that this algorithm does not consider path merging since the probability that the paths of the same depth and the same state can be stored in the stack simultaneously is rather small. Nonetheless, some authors [1] propose that a following action should be added to the step 2: If any of the $2^K$ new paths merges with a path already in the stack, keep the one with the higher metric.

The stack algorithm is based on the nonselection principle [17]. If the paths $i'_{[0,L+M)}$ and $i''_{[0,L+M)}$ through the tree diverge at depth $j$ and

$$\min\{u(\mathbf{x}'_{[0,l)}, \mathbf{y}_{[0,l)})\}_{l\in[j+1,L+M)} > \min\{u(\mathbf{x}''_{[0,l)}, \mathbf{y}_{[0,l)})\}_{l\in[j+1,L+M)}$$

then $i''_{[0,L+M)}$ cannot be the path at the top of the stack when the stack algorithm stops.

The computational complexity of the stack algorithm is almost unaffected by the code memory length, but well depends on the channel performance. Its computational complexity is a random variable and so is its stack size if not otherwise limited. The upper bound on the computational complexity is given by

$$P[\mathcal{C} \geq \eta] < A\eta^{-\rho} \quad 0 < \rho \leq 1$$

where $A$ is a constant and $\rho$ is a power that goes to unity as $R \to R_0 < R_C$ and to zero as $R \to R_C$, where $R_C$ is the channel capacity and $R_0$ is the cutoff rate [17]. The distribution described previously is called a Pareto distribution, and $\rho$ a Pareto exponent.

Omit depth-first algorithms, such as the Fano algorithm, from consideration here, because they are not interesting for PR detection.

### 18.6.5.3 Bidirectional Algorithms

Another class of algorithms are those that exploit bidirectional decoding/detection which is designed for framed data. Almost all unidirectional procedures have their bidirectional supplements since Forney showed that detecting could start from the end of the sequence provided that the trellis contains a tail. All bidirectional algorithms employ two searches from both sides. The forward search is performed using the original trellis code while the backward one employs the reverse code. The reverse trellis code is obtained from the original code by time reversing.

#### *18.6.5.3.1 The Bidirectional Stack Algorithm*

This algorithm was independently proposed by Šenk and Radivojac [40–42], and Kallel and Li [38]. It uses two stacks F (forward) and B (backward, that uses the reverse code). It is based on notions of tunnel, tentative decision, and discarding criteria. The tunnel is the unique sequence $T\,(0 \leq T \leq M)$ branches long that connect two states in the trellis. The tentative decision is the sequence $L + M$ branches long that connects the known initial and terminal trellis states (direction does not matter here) that has the highest accumulated metric of all the sequences of that length analyzed so far. A set of discarding criteria is a means to tell beforehand whether a partly explored path is likely to be a part of the finally detected sequence or not (in the latter case, the path may be eliminated from the subsequent search). Because the version [30] of the algorithm is a special case of [41] (when $T = 0$), the steps of the BSA are:

1. Place the root node into F stack, and the unique terminal node into B stack, associating them the zero metric. Make one of these stacks active (e.g., the F one).
2. Choose the node with the largest metric (of length, say, $l$) from the active stack and eliminate it from the stack. Link it via a tunnel (if a tunnel is possible, i.e., if the states match) to each of the existing paths in the other stack whose lengths are $L - l + M - T$ (if a tunnel is $M$ branches long, then the best path from the active stack can be linked to all the paths from the other stack whose lengths are $L - l$). The total length of the paths obtained in this way is $l + T + (L - l + M - T) = L + M$ branches. Store the best one into the tentative decision register. If there is already a path in the register, keep the better. Prune the paths remaining in both stacks according to any of discarding criteria used. If both stacks are emptied in this way, output the tentative decision as the decoder's final decision and terminate the algorithm. Otherwise, evaluate the metrics of all the successors of the processed path, and eliminate all of them that do not conform to the discarding criteria established.
3. Sort the remaining successors into the active stack according to their metrics applying any tie-breaking rule. Change the active stack and return to step 2.

After each tentative decision, several discarding criteria can be applied. In [41] Šenk and Radivojac applied the nonselection principle and the maximum-likelihood criterion described. The algorithm can be easily performed by two processors, although one node computation lasts longer than in the original stack algorithm. Simulations showed [41] that the Pareto exponent of the BSA in the moment when the final decision is obtained is approximately doubled, but the discarding criteria used did not provide the termination at the same time. However, the algorithm may be stopped after the assigned time for its execution has elapsed, and in such cases the erasure probability is substantially decreased.

Two additional bidirectional algorithms are worth mentioning. Belzile and Haccoun [11] investigated the bidirectional *M*-algorithm. Since the *M*-algorithm inherently avoids erasures by its breadth-first nature it still suffers from the correct path loss in its unidirectional version. Another interesting algorithm is the bidirectional multiple stack algorithm [23]. It additionally decreases the erasure probability of the MSA without compromising the error performance.

### 18.6.5.4 Algorithms That Minimize Symbol Error Rate

#### *18.6.5.4.1 The BCJR Algorithm*

So far, the algorithms that minimize the error probability of information sequence $\mathbf{i}_{[0,L+M)}$ have been considered. They accomplish it by searching for the "closest" sequence $\mathbf{x}_{[0,L+M)}$ according to the metric chosen; however, these algorithms do not necessarily minimize the symbol or bit error rate. The BCJR algorithm was independently proposed by Bahl et al. [25] and McAdam et al. [27], but a more detailed description can be found in [25]. The algorithm is a special case of a more general problem of estimating the *a posteriori* probabilities of the states and transitions of a Markov source observed through a DMC, i.e., the probabilities

$$P[s_l = i,\ s_{l+1} = j|\mathbf{y}_{[0,L+M)}] \tag{18.41}$$

or equivalently

$$\sigma_l(i,j) = P[s_l = i,\ s_{l+1} = j,\ \mathbf{y}_{[0,L+M)}] \tag{18.42}$$

where $s_l$ is the state of the trellis during $l$th branch. Introducing

$$\begin{aligned} \alpha_l(i) &= P[s_l = i,\ \mathbf{y}_{[0,l)}] \\ \beta_l(i) &= P[\mathbf{y}_{[l,L+M)}|s_l = i] \\ \gamma_l(i,j) &= P[s_{l+1} = j,\ y_l|s_l = i] \end{aligned} \tag{18.43}$$

it is not hard [25] to show that

$$\begin{aligned} \alpha_{l+1}(j) &= \sum_{i=0}^{2^{KM}-1} \alpha_l(i)\gamma_l(i,j) \\ \beta_l(j) &= \sum_{i=0}^{2^{KM}-1} \beta_{l+1}(i)\gamma_l(i,j) \\ \gamma_l(i,j) &= \sum_{x_l} P[x_l|s_l = i,\ s_{l+1} = j]P[S_{l+1} = j|s_l = i]P[y_l|x_l] \\ \sigma_l(i,j) &= \alpha_l(i)\gamma_l(i,j)\beta_l(j) \end{aligned} \tag{18.44}$$

The known initial conditions are $\alpha_0(i=0) = 1, \alpha_0(i\neq 0) = 0, \beta_{L+M}(i=0) = 1, \beta_{L+M}(i\neq 0) = 0$. Assuming that the initial and terminating state in the trellis is the all zero state, the steps of the algorithm are

1. Initialize $\alpha_0(i)$, and $\beta_{L+M}(i)$, for $i = 0, 1, \ldots, q^M - 1$ according to Eq. 18.44.
2. As soon as $\mathbf{y}_l$ is received, compute $\alpha_l(i)$ and $\gamma_l(i,j)$. Store $\alpha_l(i,j)$ for all $l$ and $i$.
3. When the complete sequence $\mathbf{y}_{[0,L+M)}$ is received, compute $\beta_l(i)$ using Eq. 18.44, and immediately the probabilities $\sigma_l(i,j)$. Group those $\sigma_l(i,j)$ that have the same information sequence $\mathbf{i}_l$, and choose the largest as the decoder estimate.

The basic problem with the algorithm is that it requires both large storage and great number of computations. All the values of $\alpha_l(i)$ must be stored, which requires almost $(L+M)q^{KM}$ memory locations. The number of multiplications required for determining the $\alpha_l(i)$ and $\beta_l(i)$ for each $l$ is $q^{M+1}$, and there are $q^M$ additions of $q^K$ numbers as well. The computation of $\gamma_l(i,j)$ is not costly and can

be accomplished by a table lookup. Finally, the computation of all $\sigma_l(i,j)$ requires $q^{(M+1)+1}$ multiplications for each $l$, and $q-1$ comparisons in choosing the largest $\mathbf{i}_l$. Consequently, this is an algorithm with exponential complexity and in practice can be applied only when $M$ and $L$ are short. Nevertheless, it is used for iterative decoding where such requirements can be fulfilled, such as for turbo codes. The main advantage of the algorithm in such cases is its ability to estimate $P[s_{l+1}=j|s_l=i]$, which for the possible transitions equals $q^{-1}$ only in the first iteration.

#### *18.6.5.4.2 The SOVA Algorithm*

The soft-output Viterbi algorithm (SOVA) [15] is a modification of the VA that was designed with the aim of estimating the reliability of every detected bit by the VA. It is applicable only when $q=2$. The VA is used here in its sliding window form, which detects infinite sequence with delay of $\delta$ branches from the last received one.

The reliability (or soft value) of a bit $i$, $L(i)$, is defined as $L(i)=\ln(P[i=0]/P[i=1])$. The SOVA further extends the third step in order to obtain this value, in the following way:

*Path selection* (extension): Let $i^{(j)}_{[0,i-j)}$, $j\in\{0,1,\ldots,\delta-1\}$ be the information sequences which merge with $i'_{[0,l)}$ at depths $l-j$. Their paths have earlier been discarded due to their lower metrics. Let the corresponding metric differences in the merging states be denoted $\Delta_j$, and let$J=\{J: i^{(j)}_{l-\delta}\neq i'_{l-\delta}\}$. Then $L(i'_{l-\delta})\approx(1-2i'_{l-\delta})\min_{j\in J}\Delta_j$.

Because VA detecting metric can be modified in a way to take into account a priori knowledge of input bit probabilities, the SOVA can be used as soft input-soft output (SISO) block in turbo decoding schemes.

## References


1. A.J. Viterbi and J. Omura, *Principles of Digital Communication and Coding,* McGraw-Hill, Tokyo, 1979.
2. A.J. Viterbi, "Error bounds for convolutional codes and asymptotically optimum decoding algorithm," *IEEE Trans. Inform. Theory,* vol. IT-13, pp. 260–269, 1967.
3. A. Lender, "Correlative level coding for binary-data transmission," *IEEE Trans. Commun. Technol.* (Concise paper), vol. COM-14, pp. 67–70, 1966.
4. B. Vasic, "A graph based construction of high-rate soft decodable codes for partial response channels," to be presented at ICC2001, Helsinki, Finland, June 2001, 10–15.
5. C.L. Barbosa, "Maximum likelihood sequence estimators, a geometric view," *IEEE Trans. Inform. Theory,* vol. IT-35, pp. 419–427, March 1989.
6. C.D. Wei, "An analog magnetic storage read channel based on a decision feedback equalizer," *PhD final report,* University of California, Electrical Eng. Department, July 1996.
7. G.D. Forney, "Maximum likelihood sequence estimation of digital sequences in the presence of intersymbol interference," *IEEE Trans. Inform. Theory,* vol. IT-18, pp. 363–378, May 1972.
8. H. Kobayashi and D.T. Tang, "Application of partial response channel coding to magnetic recording systems," *IBM J. Res. and Dev.,* vol. 14, pp. 368–375, July 1979.
9. H. Kobayashi, "Correlative level coding and maximum-likelihood decoding," *IEEE Trans. Inform. Theory,* vol. IT-17(5), pp. 586–594, 1971.
10. J.B. Anderson and S. Mohan, "Sequential coding algorithms: a survey and cost analysis," *IEEE Trans. Comm.,* vol. COM-32(2), pp. 169–176, Feb. 1984.
11. J. Belzile and D. Haccoun, "Bidirectional breadth-first algorithms for the decoding of convolutional codes," *IEEE Trans. Comm.,* vol. COM-41, pp. 370–380, Feb. 1993.
12. J. Bergmans, "Density improvements in digital magnetic recording by decision feedback equalization," *IEEE Trans. Magn.,* vol. 22, pp. 157–162, May 1986.
13. J. Hagenauer, "Applications of error-control coding," *IEEE Trans. Inform. Theory,* vol. IT-44(6), pp. 2531–2560, Oct. 1998.
14. J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft decision outputs and its applications," in *Proc. GLOBECOM 89,* Dallas, Texas, pp. 47.1.1–47.1.7, Nov. 1989.
15. J. Hagenauer, "Source-controlled channel decoding," *IEEE Trans. Comm.,* vol. COM-41, pp. 370–380, Feb. 1995.

16. J.K. Wolf and D. Chun, "The single burst error detection performance of binary cyclic codes," *IEEE Trans. Commun.,* vol. 42(1), pp. 11–13, Jan. 1994.
17. J.L. Massey, *Coding and Complexity,* CISM courses and lectures No. 216, Springer-Verlag, Wien, 1976.
18. J.L. Sonntag and B. Vasic, "Implementation and bench characterization of a read channel with parity check post processor," *Digest of TMRC 2000,* Santa Clara, CA, August 2000.
19. J.M. Cioffi, W.L. Abbott, H.K. Thapar, C.M. Melas, and K.D. Fisher, "Adaptive equalization in magnetic disk storage channels," *IEEE Communications Magazine,* pp. 14–29, Feb. 1990.
20. J.D. Coker, E. Eleftheriou, R. Galbraith, and W. Hirt, "Noise-predictive maximum likelihood (NPML) detection," *IEEE Trans. Magn.*, vol. 34(1), pp. 110–117, Jan. 1998.
21. J.J. O'Reilly and A.M. de Oliveira Duarte, "Error propagation in decision feedback receivers," *IEEE Proc.*, Pt. F, vol. 132(7), pp. 561–566, Dec. 1985.
22. K. Knudson et al., "Dynamic threshold implementation of the maximum likelihood detector for the EPR4 channel," *Conf. Rec. Globecom '91,* pp. 2135–2139, 1991.
23. K. Li and S. Kallel, "A bidirectional multiple stack algorithm," *IEEE Trans. Comm.,* vol. COM-47(1), pp. 6–9, Jan. 1999.
24. K.D. Fisher, J. Cioffi, W. Abbott, P. Bednarz, and C.M. Melas, "An adaptive RAM-DFE for storage channels," *IEEE Trans. Comm.,* vol. 39, no. 11, pp. 1559–1568, Nov. 1991.
25. L.R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory,* pp. 284–287, March 1974.
26. M. Fossorier, "Performance evaluation of decision feedback equalization for the Lorentzian channel," *IEEE Trans. Magn.,* vol. 32(2), March 1996.
27. P.L. McAdam, L.R. Welch, and C.L. Weber, "M.A.P. bit decoding of convolutional codes," in *Proc. of ISIT 1972,* Asilomar, USA.
28. P.R. Chevillat and D.J. Costello Jr., "A multiple stack algorithm for erasure free decoding of convolutional codes," *IEEE Trans. Comm.*, vol. COM-25, pp. 1460–1470, Dec. 1977.
29. P. Radivojac and V. Šenk, "The Generalized Viterbi-T algorithm," in *Proc. of XXXIX Conference on ETRAN,* vol. 2, pp. 13–16, Zlatibor, Yugoslavia, June 1995.
30. P.H. Siegel and J.K. Wolf, "Modulation and coding for information storage," *IEEE Comm. Magazine,* Dec. 1991, pp. 68–86.
31. P. Kabal and S. Pasupathy, "Partial response signaling" *IEEE Trans. Comm.*, vol. COM-23(9), pp. 921–934, Sept. 1975.
32. P.R. Chevillat, E. Eleftheriou, and D. Maiwald, "Noise-predictive partial-response equalizers and applications," *IEEE Conf. Records ICC'92,* pp. 942–947, June 1992.
33. P.K. Pai, A.D. Brewster, and A.A. Abidi, "Analog front-end architectures for high-speed PRML magnetic read channels," *IEEE Trans. Magn.*, vol. 31, pp. 1103–1108, 1995.
34. R.M. Fano, "A heuristic discussion of probabilistic decoding," *IEEE Trans. Inform. Theory,* vol. IT-9, pp. 64–74, April 1963.
35. R.D. Cideciyan et al, "A PRML system for digital magnetic recording," *IEEE J. Sel. Areas Communications,* vol. 10, pp. 38–56, Jan. 1992.
36. S.A. Altekar, M. Berggren, B.E. Moision, P.H. Siegel, and J.K. Wolf, "Error-event characterization on partial-response channels," *IEEE Trans. Inform. Theory,* vol. 45, no. 1, pp. 241–247, Jan. 1999.
37. S.J. Simmons, "Breadth-first trellis decoding with adaptive effort," *IEEE Trans. Comm.,* vol. COM-38(1), pp. 3–12, Jan. 1990.
38. S. Kallel and K. Li, "Bidirectional sequential decoding," *IEEE Trans. Inform. Theory,* vol. IT-43(4) pp. 1319–1326, July 1997.
39. T. Hashimoto, "A list-type reduced-constraint generalization of the Viterbi algorithm," *IEEE Trans. Inform. Theory,* vol. IT-33(6), pp. 866–876, Nov. 1987.
40. V. Šenk and P. Radivojac, "The bidirectional stack algorithm—simulation results," in *Proc. of TELSIKS'95*, pp. 349–352, Niš, Yugoslavia, Oct. 1995.
41. V. Šenk and P. Radivojac, "The bidirectional stack algorithm," in *Proc. of ISIT'97*, p. 500, Ulm, Germany, July 1997.

42. V. Šenk, "Bistack—a bidirectional stack algorithm for decoding trellis codes," in *Proc. of XXXVI Conference on ETAN,* pp. 153–160, Kopaonik, Yugoslavia, 1992.
43. V.M. Eyuboglu and S.U. Quereshi, "Reduced-state sequence estimation with decision feedback and set partitioning," *IEEE Trans. Comm.,* vol. 36(1), pp. 13–20, Jan. 1988.
44. J. Bergmans, *Digital Baseband Transmission and Recording,* Kluwer Academic, Dordrecht, the Netherlands, 1996.
45. M. Despotović and V. Šenk, "Distance spectrum of channel trellis codes on precoded partial response 1-D channel," J. Facta Universitatis (Niš), vol. 1., pp. 57–72, 1995, http://factaee.elfak.ni.ac.yu.

## 18.7 An Introduction to Error-Correcting Codes

*Mario Blaum*

### 18.7.1 Introduction

When digital data are transmitted over a noisy channel, it is important to have a mechanism allowing recovery against a limited number of errors. Normally, a user string of 0s and 1s, called bits, is encoded by adding a number of redundant bits to it. When the receiver attempts to reconstruct the original message sent, it starts by examining a possibly corrupted version of the encoded message, and then makes a decision. This process is called the decoding.

The set of all possible encoded messages is called an error-correcting code. The field was started in the late 1940s by the work of Shannon and Hamming, and since then thousands of papers on the subject have been published. Several very good books are available to touch different aspects of error-correcting codes, for instance, [1,3–5,7,8], to mention just a few.

The purpose of Section 18.7 is to give an introduction to the theory and practice of error-correcting codes. In particular, it will be shown how to encode and decode the most widely used codes, Reed–Solomon (RS) codes.

In principle, it will assumed that the information symbols are bits, i.e., 0s and 1s. The set {0, 1} has a field structure under the exclusive-OR ($\oplus$) and product operations. This field is denoted as $GF(2)$, which means Galois field of order 2.

Roughly, two types of error-correcting codes are used—codes of block type and codes of convolutional type. Codes of block type encode a fixed number of bits, say $k$ bits, into a vector of length $n$. So, the information string is divided into blocks of $k$ bits each. Convolutional codes take the string of information bits globally and slide a window over the data in order to encode. A certain amount of memory is needed by the encoder; however, this section concentrates on block codes only. For more on convolutional codes, see [3,8].

As stated previously, $k$ information bits are encoded into $n$ bits. So, we have a 1-1 function $f$,

$$f: GF(2)^k \rightarrow GF(2)^n$$

The function $f$ defines the encoding procedure. The set of $2^k$ encoded vectors of length $n$ is called a code of length $n$ and dimension $k$, and we denote it as an $[n, k]$ code. Codewords are called the elements of the code while words are called the vectors of length $n$ in general. The ratio $k/n$ is called the *rate* of the code.

The error-correcting power of a code is characterized by a parameter called the minimum (Hamming) distance of the code. Formally:

**Definition 1** Given two vectors of length $n$, say $\underline{a}$ and $\underline{b}$, we call the Hamming distance between $\underline{a}$ and $\underline{b}$ the number of coordinates in which they differ (notation, $d_H(\underline{a}, \underline{b})$).

Given a code $\mathcal{C}$ of length $n$ and dimension $k$, let

$$d = \min\{d_H(\underline{a}, \underline{b}): \underline{a} \neq \underline{b}, \underline{a}, \underline{b} \in \mathcal{C}\}$$

$d$ is the minimum (Hamming) distance of the code $\mathcal{C}$, and $\mathcal{C}$ is an $[n, k, d]$ code.

It is easy to verify that $d_H(\underline{a}, \underline{b})$ satisfies the axioms of distance, i.e.,

1. $d_H(\underline{a}, \underline{b}) = d_H(\underline{b}, \underline{a})$,
2. $d_H(\underline{a}, \underline{b}) = 0$ if and only if $\underline{a} = \underline{b}$,
3. $d_H(\underline{a}, \underline{c}) \leq d_H(\underline{a}, \underline{b}) + d_H(\underline{b}, \underline{c})$.

A sphere of radius $r$ and center $\underline{a}$ are called the set of vectors that are at distance at most $r$ from $\underline{a}$. The relation between $d$ and the maximum number of errors that code $\mathcal{C}$ can correct is given by the following lemma:

**Lemma 1** The maximum number of errors that an $[n, k, d]$ code can correct is $\lfloor (d-1)/2 \rfloor$, where $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to $x$.

*Proof:* Assume that vector $\underline{a}$ was transmitted but a possibly corrupted version of $\underline{a}$, for instance $\underline{r}$, was received. Moreover, assume that no more than $\lfloor (d-1)/2 \rfloor$ errors have occurred.

Consider the set of $2^k$ spheres of radius $\lfloor (d-1)/2 \rfloor$ whose centers are the codewords in $\mathcal{C}$. By the definition of $d$, all these spheres are disjoint. Hence, $\underline{r}$, belongs to one and only one sphere: the one whose center is codeword $\underline{a}$. So, the decoder looks for the sphere in which $\underline{r}$ belongs, and outputs the center of that sphere as the decoded vector. Subsequently, whenever the number of errors is at most $\lfloor (d-1)/2 \rfloor$, this procedure will give the correct answer.

Moreover, $\lfloor (d-1)/2 \rfloor$ is the maximum number of errors that the code can correct. For let $\underline{a}, \underline{b} \in \mathcal{C}$ such that $d_H(\underline{a}, \underline{b}) = d$. Let $\underline{u}$ be a vector such that $d_H(\underline{a}, \underline{u}) = 1 + \lfloor (d-1)/2 \rfloor$ and $d_H(\underline{b}, \underline{u}) = d - 1 - \lfloor (d-1)/2 \rfloor$. We easily verify that $d_H(\underline{b}, \underline{u}) \leq d_H(\underline{a}, \underline{u})$, so, if $\underline{a}$ is transmitted and $\underline{u}$ is received (i.e., $1 + \lfloor (d-1)/2 \rfloor$ errors have occurred), the decoder cannot decide that the transmitted codeword was $\underline{a}$, since codeword $\underline{b}$ is at least as close to $\underline{u}$ as $\underline{a}$.

**Example 1**

Consider the following 1-1 relationship between $GF(2)^2$ and $GF(2)^5$ defining the encoding:

$$
\begin{aligned}
00 &\leftrightarrow 00000 \\
10 &\leftrightarrow 00111 \\
01 &\leftrightarrow 11100 \\
11 &\leftrightarrow 11011
\end{aligned}
$$

The four vectors in $GF(2)^5$ constitute a [2,3,5] code $\mathcal{C}$. From Lemma 1, $\mathcal{C}$ can correct one error.

For instance, assume that we receive the vector $\underline{r} = 10100$. The decoder looks into the four spheres of radius 1 (each sphere has six elements) around each codeword, finding that $\underline{r}$ belongs in the sphere with center 11100. If we look at the table above, the final output of the decoder is the information block 01.

Example 1 shows that the decoder has to make at most 24 checks before arriving to the correct decision. When large codes are involved, as is the case in applications, this decoding procedure is not practical, since it amounts to an exhaustive search over a huge set of vectors.

One of the goals in the theory of error-correcting codes is finding codes with rate and minimum distance as large as possible. The possibility of finding codes with the right properties is often limited by bounds that constrain the choice of parameters $n$, $k$, and $d$. Some of these bounds are given in the next subsection.

Let us point out that error-correcting codes can be used for detection instead of correction of errors. The simplest example of an error-detecting code is given by a parity code: a parity is added to a string of bits in such a way that the total number of bits is even (a more sophisticated way of saying this is that the sum modulo 2 of the bits has to be 0). For example, 0100 is encoded as 01001. If an error occurs, or, more generally, an odd number of errors, these errors will be detected since the sum modulo 2 of the received bits will be 1. Notice that two errors will be undetected. In general, if an $[n, k, d]$ code is used for detection only, the decoder checks whether the received vector is in the code or not. If it is not, then

errors are detected. It is easy to see that an $[n, k, d]$ code can detect up to $d-1$ errors. Also, one can choose to correct less than $\lfloor (d-1)/2 \rfloor$ errors, say $s$ errors, by taking disjoint spheres of radius $s$ around codewords, and using the remaining capacity to detect errors. In other words, correct up to $s$ errors or detect up to $s+t$ errors when more than $s$ errors occur.

Another application of error-correcting codes is in erasure correction. An erased bit is a bit that cannot be read, so the decoder has to decide if it was a 0 or a 1. An erasure is normally denoted with the symbol "?". For instance, 01?0 means that we cannot read the third symbol. Obviously, it is easier to correct erasures than to correct errors, since in the case of erasures we already know the location, we simply have to find what the erased bit was. It is not hard to prove that an $[n, k, d]$ code can correct upto $d-1$ erasures. One may also want to simultaneously correct errors and erasures. In fact, a code $\mathcal{C}$ with minimum distance $d$ can correct $s$ errors together with $t$ erasures whenever $2s+t \leq d-1$.

### 18.7.2 Linear Codes

The previous subsection showed that a binary code of length $n$ is a subset of $GF(2)^n$. Notice that, being $GF(2)$ a field, $GF(2)^n$ has a structure of vector space over $GF(2)$. A code $\mathcal{C}$ is linear if it is a subspace of $GF(2)^n$, i.e.

1. $\underline{0} \in \mathcal{C}$
2. $\forall \underline{a}, \underline{b} \in \mathcal{C}, \underline{a} \oplus \underline{b} \in \mathcal{C}$

The symbol $\underline{0}$ denotes the all-zero vector. In general, vectors will be denoted with underlined letters, otherwise letters denote scalars.

In the first subsection, it was assumed that a code had $2^k$ elements, $k$ being the dimension; however, a code of length $n$ can be defined as any subset of $GF(2)^n$.

Many interesting combinatorial questions can be asked regarding nonlinear codes. Probably, the most important question is the following: Given the length $n$ and the minimum distance $d$, what is the maximum number of codewords that a code can have? For more about nonlinear codes, the reader is referred to [4]. From now on, we assume that all codes are linear. Linear codes are in general easier to encode and decode than their nonlinear counterparts; hence they are more suitable for implementation in applications.

In order to find the minimum distance of a linear code, it is enough to find its minimum *weight*. The (Hamming) weight of a vector $\underline{u}$ is the distance between $\underline{u}$ and the zero vector. In other words, the weight of $\underline{u}$, denoted $w_H(\underline{u})$, is the number of nonzero coordinates of the vector $\underline{u}$. The minimum weight of a code is the minimum between all the weights of the nonzero codewords. The proof of the following lemma is left as an exercise.

**Lemma 2** Let $\mathcal{C}$ be a linear $[n, k, d]$ code. Then, the minimum distance and the minimum weight of $\mathcal{C}$ are the same.

Next, two important matrices are introduced that define a linear error-correcting code. A code $\mathcal{C}$ is now a subspace, so the dimension $k$ of $\mathcal{C}$ is the cardinality of a basis of $\mathcal{C}$. Consider then an $[n, k, d]$ code $\mathcal{C}$. A $k \times n$ matrix $G$ is a *generator* matrix of a code $\mathcal{C}$ if the rows of $G$ are a basis of $\mathcal{C}$. Given a generator matrix, the encoding process is simple. Explicitly, let $\underline{u}$ be an information vector of length $k$ and $G$ a $k \times n$ generator matrix, then $\underline{u}$ is encoded into the $n$-vector $\underline{v}$ given by

$$\underline{v} = \underline{u}G \tag{18.45}$$

**Example 2**

Let $G$ be the $2 \times 5$ matrix

$$G = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

It is easy to see that $G$ is a generator matrix of the [2,3,5] code described in Example 1.

Notice that, although a code may have many generator matrices, the encoding depends on the particular matrix chosen, according to Eq. 18.45. We say that $G$ is a *systematic* generator matrix if $G$ can be written as

$$G = (I_k | V) \tag{18.46}$$

where $I_k$ is the $k \times k$ identity matrix and $V$ is a $k \times (n-k)$ matrix. A systematic generator matrix has the following advantage: given an information vector $\underline{u}$ of length $k$, the encoding given by Eq. 18.45 outputs a codeword $(\underline{u}, \underline{w})$, where $\underline{w}$ has length $n-k$. In other words, a systematic encoder adds $n-k$ redundant bits to the $k$ information bits, so information and redundancy are clearly separated. This also simplifies the decoding process, since, after decoding, the redundant bits are simply discarded. For that reason, most encoders used in applications are systematic.

A permutation of the columns of a generator matrix gives a new generator matrix defining a new code. The codewords of the new code are permutations of the coordinates of the codewords of the original code, therefore, the two codes are *equivalent*. Notice that equivalent codes have the same distance properties, so their error correcting capabilities are exactly the same.

By permuting the columns of the generator matrix in Example 2, the following generator matrix $G'$ is obtained:

$$G' = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{18.47}$$

The matrix $G'$ defines a systematic encoder for a code that is equivalent to the one given in Example 1. For instance, the information vector 11 is encoded into 11 101.

The second important matrix related to a code is the so-called *parity check* matrix. An $(n-k) \times n$ matrix $H$ is a parity check matrix of an $[n, k]$ code $\mathcal{C}$ if and only if, for any $\underline{c} \in \mathcal{C}$,

$$\underline{c} H^T = \underline{0} \tag{18.48}$$

where $H^T$ denotes the transpose of matrix $H$ and $\underline{0}$ is a zero vector of length $n-k$. The parity check matrix $H$ is in systematic form if

$$H = (W | I_{n-k}) \tag{18.49}$$

where $I_{n-k}$ is the $(n-k) \times (n-k)$ identity matrix and $W$ is an $(n-k) \times k$ matrix.

Given a systematic generator matrix $G$ of a code $\mathcal{C}$, it is easy to find the systematic parity check matrix $H$ (and conversely). Explicitly, if $G$ is given by Eq. 18.46, $H$ is given by

$$H = (V^T | I_{n-k}) \tag{18.50}$$

The proof of this fact is left to the reader.

For example, the systematic parity check matrix of the code, whose systematic generator matrix is given by Eq. 18.47, is

$$H = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{18.51}$$

Next is an important property of parity check matrices.

**Lemma 3** Let $\mathcal{C}$ be a linear $[n, k, d]$ code and $H$ a parity check matrix. Then, any $d-1$ columns of $H$ are linearly independent.

*Proof:* Numerate the columns of $H$ from 0 to $n-1$. Assume that columns $0 \leq i_1 < i_2 < \cdots < i_m \leq n-1$ are linearly dependent, where $m \leq d-1$. Without loss of generality, assume that the sum of these columns is equal to the column vector zero. Let $\underline{v}$ be a vector of length $n$ whose nonzero coordinates are in locations $i_1, i_2, \ldots, i_m$. Then,

$$\underline{v}H^T = \underline{0}$$

hence $\underline{v}$ is in $\mathcal{C}$. But $\underline{v}$ has weight $m \leq d-1$, contradicting the fact that $\mathcal{C}$ has minimum distance $d$.

**Corollary 1** For any linear $[n, k, d]$ code, the minimum distance $d$ is the smallest number $m$ such that there is a subset of $m$ linearly dependent columns.

*Proof:* It follows immediately from Lemma 3.

**Corollary 2 (Singleton Bound)** For any linear $[n, k, d]$ code,

$$d \leq n - k + 1$$

*Proof:* Notice that, because $H$ is an $(n-k) \times n$ matrix, any $n-k+1$ columns are going to be linearly dependent, so if $d > n-k+1$ we would contradict Corollary 1.

Codes meeting the Singleton bound are called maximum distance separable (MDS). In fact, except for trivial cases, binary codes are not MDS. In order to obtain MDS codes, we will define codes over larger fields, like the so-called Reed Solomon codes, to be described later in the chapter.

A second bound is also given relating the redundancy and the minimum distance of an $[n, k, d]$ code the so-called Hamming or volume bound. Let us denote by $V(r)$ the number of elements in a sphere of radius $r$ whose center is an element in $GF(2)^n$. It is easy to verify that

$$V(r) = \sum_{i=0}^{r} \binom{n}{i} \tag{18.52}$$

We then have:

**Lemma 4 (Hamming bound)** Let $\mathcal{C}$ be a linear $[n, k, d]$ code, then

$$n - k \geq \log_2 V(\lfloor (d-1)/2 \rfloor) \tag{18.53}$$

*Proof:* Notice that the $2^k$ spheres with the $2^k$ codewords as centers and radius $\lfloor (d-1)/2 \rfloor$ are disjoint. The total number of vectors contained in these spheres is $2^k\ V(\lfloor (d-1)/2 \rfloor)$. This number has to be smaller than or equal to the total number of vectors in the space, i.e.,

$$2^n \geq 2^k V(\lfloor (d-1)/2 \rfloor) \tag{18.54}$$

Inequality (Eq. 18.53) follows immediately from Eq. 18.54.

A *perfect* code is a code for which inequality Eq. 18.53 is in effect equality. Geometrically, a perfect code is a code for which the $2^k$ spheres of radius $\lfloor (d-1)/2 \rfloor$ and the codewords as centers cover the whole space.

Not many perfect codes exist. In the binary case, the only nontrivial linear perfect codes are the Hamming codes (to be presented in the Section 18.7.3) and the [7,12,23] Golay code. For details, the reader is referred to [4].

### 18.7.3 Syndrome Decoding, Hamming Codes, and Capacity of the Channel

This subsection studies the first important family of codes, the so-called Hamming codes. As will be shown, Hamming codes can correct up to one error.

Let $\mathcal{C}$ be an $[n, k, d]$ code with parity check matrix $H$. Let $\underline{u}$ be a transmitted vector and $\underline{r}$ a possibly corrupted received version of $\underline{u}$. We say that the syndrome of $\underline{r}$ is the vector $\underline{s}$ of length $n - k$ given by

$$\underline{s} = \underline{r}H^T \tag{18.55}$$

Notice that, if no errors occurred, the syndrome of $\underline{r}$ is the zero vector. The syndrome, however, tells us more than a vector being in the code or not. For instance, as before, that $\underline{u}$ was transmitted and $\underline{r}$ was received, where $\underline{r} = \underline{u} \oplus \underline{e}$, $\underline{e}$ an error vector. Notice that,

$$\underline{s} = \underline{r}H^T = (\underline{u} \oplus \underline{e})H^T = \underline{u}H^T \oplus \underline{e}H^T = \underline{e}H^T$$

because $\underline{u}$ is in $\mathcal{C}$. Hence, the syndrome does not depend on the received vector but on the error vector. In the next lemma, we show that to every error vector of weight $\leq (d-1)/2$ corresponds a unique syndrome.

**Lemma 5** Let $\mathcal{C}$ be a linear $[n, k, d]$ code with parity check matrix $H$. Then, there is a 1-1 correspondence between errors of weight $\leq (d-1)/2$ and syndromes.

*Proof:* Let $\underline{e}_1$ and $\underline{e}_2$ be two distinct error vectors of weight $\leq (d-1)/2$ with syndromes $\underline{s}_1 = \underline{e}_1 H^T$ and $\underline{s}_2 = \underline{e}_2 H^T$. If $\underline{s}_1 = \underline{s}_2$, then $\underline{s} = (\underline{e}_1 \oplus \underline{e}_2)H^T = \underline{s}_1 \oplus \underline{s}_2 = \underline{0}$, hence $\underline{e}_1 \oplus \underline{e}_2 \in \mathcal{C}$. But $\underline{e}_1 \oplus \underline{e}_2$ has weight $\leq d-1$, a contradiction.

Lemma 5 gives the key for a decoding method that is more efficient than exhaustive search. We can construct a table with the 1-1 correspondence between syndromes and error patterns of weight $\leq (d-1)/2$ and decode by lookup table. In other words, given a received vector, we first find its syndrome and then we look in the table to which error pattern it corresponds. Once we obtain the error pattern, we add it to the received vector, retrieving the original information. This procedure may be efficient for small codes, but it is still too complex for large codes.

**Example 3**

Consider the code whose parity matrix $H$ is given by Eq. 18.51. We have seen that this is a [2,3,5] code. We have six error patterns of weight $\leq 1$. The 1-1 correspondence between these error patterns and the syndromes can be immediately verified to be

$$\begin{aligned}
00000 &\leftrightarrow 000 \\
10000 &\leftrightarrow 011 \\
01000 &\leftrightarrow 110 \\
00100 &\leftrightarrow 100 \\
00010 &\leftrightarrow 010 \\
00001 &\leftrightarrow 001
\end{aligned}$$

For instance, assume that we receive the vector $\underline{r} = 10111$. We obtain the syndrome $\underline{s} = \underline{r}H^T = 100$. Looking at the table above, we see that this syndrome corresponds to the error pattern $\underline{e} = 00100$. Adding this error pattern to the received vector, we conclude that the transmitted vector was $\underline{r} \oplus \underline{e} = 10011$.

Given a number $r$ of redundant bits, we say that a $[2^r - 1, 2^r - r - 1, 3]$ Hamming code is a code having an $r \times (2^r - 1)$ parity check matrix $H$ such that its columns are all the different nonzero vectors of length $r$.

A Hamming code has minimum distance 3. This follows from its definition and Corollary 1. Notice that any two columns in $H$, being different, are linearly independent. Also, if we take any two different columns and their sum, these three columns are linearly dependent, proving our assertion.

A natural way of writing the columns of $H$ in a Hamming code, is by considering them as binary numbers on base 2 in increasing order. This means, the first column is 1 on base 2, the second column is 2, and so on. The last column is $2^r - 1$ on base 2, i.e., $(1, 1, \ldots, 1)^T$. This parity check matrix, although nonsystematic, makes the decoding very simple.

In effect, let $\underline{r}$ be a received vector such that $\underline{r} = \underline{v} \oplus \underline{e}$, where $\underline{v}$ was the transmitted codeword and $\underline{e}$ is an error vector of weight 1. Then, the syndrome is $\underline{s} = \underline{e}H^T$, which gives the column corresponding to the location in error. This column, as a number on base 2, tells us exactly where the error has occurred, so the received vector can be corrected.

**Example 4**

Consider the [3,4,7] Hamming code $\mathcal{C}$ with parity check matrix

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{18.56}$$

Assume that vector $\underline{r} = 1100101$ is received. The syndrome is $\underline{s} = \underline{r}H^T = 001$, which is the binary representation of the number 1. Hence, the first location is in error, so the decoder estimates that the transmitted vector was $\underline{v} = 0100101$.

We can obtain 1-error correcting codes of any length simply by shortening a Hamming code. This procedure works as follows: assume that we want to encode $k$ information bits into a 1-error correcting code. Let $r$ be the smallest number such that $k \leq 2^r - r - 1$. Let $H$ be the parity check matrix of a $[2^r - 1, 2^r - r - 1, 3]$ Hamming code. Then construct a matrix $H'$ by eliminating some $2^r - r - 1 - k$ columns from $H$. The code whose parity check matrix is $H'$ is a $[k + r, k, d]$ code with $d \geq 3$, hence it can correct one error. We call it a shortened Hamming code. For instance, the [2,3,5] code whose parity check matrix is given by Eq. 18.51 is a shortened Hamming code.

In general, if $H$ is the parity check matrix of a code $\mathcal{C}$, $H'$ is a matrix obtained by eliminating a certain number of columns from $H$ and $\mathcal{C}'$ is the code with parity check matrix $H'$ we say that $\mathcal{C}'$ is obtained by shortening $\mathcal{C}$.

A $[2^r - 1, 2^r - r - 1, 3]$ Hamming code can be extended to a $[2^r, 2^r - r - 1, 4]$ Hamming code by adding to each codeword a parity bit, that is, the exclusive-OR of the first $2^r - 1$ bits. The new code is called an extended Hamming code.

So far, we have not talked about probabilities of errors. Assume that we have a binary symmetric channel (BSC), i.e., the probability of a 1 becoming a 0 or of a 0 becoming a 1 is $p < .5$. Let $P_{\text{err}}$ be the probability of error after decoding using a code, i.e., the output of the decoder does not correspond to the originally transmitted information vector. A fundamental question is the following: given a BSC with bit error probability $p$, does it exist a code of high rate that can arbitrarily lower $P_{\text{err}}$? The answer, due to Shannon, is yes, provided that the code has rate below a parameter called the capacity of the channel, as defined next.

**Definition 2** Given a BSC with probability of bit error $p$, we say that the capacity of the channel is

$$C(p) = 1 + p \log_2 p + (1 - p) \log_2 (1 - p) \tag{18.57}$$

**Theorem 1 (Shannon)** For any $\epsilon > 0$ and $R < C(p)$, there is an $[n, k]$ binary code of rate $k/n \geq R$ with $P_{\text{err}} < \epsilon$.

For a proof of Theorem 1 and some of its generalizations, the reader is referred to [5], or even to Shannon's original paper [6].

Theorem 1 has enormous theoretical importance. It shows that reliable communication is not limited in the presence of noise, only the rate of communication is. For instance, if $p = .01$, the capacity of the

channel is $C(.01) = .9192$. Hence, there are codes of rate $\geq .9$ with $P_{\text{err}}$ arbitrarily small. It also tells us not to look for codes with rate .92 making $P_{\text{err}}$ arbitrarily small.

The proof of Theorem 1, though, is based on probabilistic methods and the assumption of arbitrarily large values of $n$. In practical applications, $n$ cannot be too large. The theorem does not tell us how to construct efficient codes, it just asserts their existence. Moreover, when we construct codes, we want them to have efficient encoding and decoding algorithms. In the last few years, coding methods approaching the Shannon limit have been developed, the so-called *turbo codes*. Although great progress has been made towards practical implementations of turbo codes, in applications like magnetic recording their complexity is still a problem. A description of turbo codes is beyond the scope of this introduction. The reader is referred to [2].

### 18.7.4 Codes over Bytes and Finite Fields

So far, we have considered linear codes over bits. Next we want to introduce codes over larger symbols, mainly over bytes. A byte of size $v$ is a vector of $v$ bits. Mathematically, bytes are vectors in $GF(2)^v$. Typical cases in magnetic and optical recording involve 8-bit bytes. Most of the general results in the previous sections for codes over bits easily extend to codes over bytes. It is trivial to multiply bits, but we need a method to multiply bytes. To this end, the theory of finite fields has been developed. Next we give a brief introduction to the theory of finite fields. For a more complete treatment, the reader is referred to Chapter 4 of [4].

We know how to add two binary vectors, we simply exclusive-OR them componentwise. What we need now is a rule that allows us to multiply bytes while preserving associative, distributive, and multiplicative inverse properties, i.e., a product that gives to the set of bytes of length $v$ the structure of a field. To this end, we will define a multiplication between vectors that satisfies the associated and commutative properties, it has a 1 element, each nonzero element is invertible and it is distributive with respect to the sum operation.

Recall the definition of the ring $Z_m$ of integers modulo $m$: $Z_m$ is the set $\{0, 1, 2, \ldots, m-1\}$, with a sum and product of any two elements defined as the residue of dividing by $m$ the usual sum or product. It is not difficult to prove that $Z_m$ is a field if and only if $m$ is a prime number. Using this analogy, we will give to $(GF(2))^v$ the structure of a field.

Consider the vector space $(GF(2))^v$ over the field $GF(2)$. We can view each vector as a polynomial of degree $\leq v-1$ as follows: the vector $\underline{a} = (a_0, a_1, \ldots, a_{v-1})$ corresponds to the polynomial $a(\alpha) = a_0 + a_1 \alpha + \cdots + \alpha_{v-1} \alpha^{v-1}$.

The goal is to give to $(GF(2))^v$ the structure of a field. We will denote such a field by $GF(2^v)$. The sum in $GF(2^v)$ is the usual sum of vectors in $(GF(2))^v$. We need now to define a product.

Let $f(x)$ be an irreducible polynomial (i.e., it cannot be expressed as the product of two polynomials of smaller degree) of degree $v$ whose coefficients are in $GF(2)$. Let $a(\alpha)$ and $b(\alpha)$ be two elements of $GF(2^v)$. We define the product between $a(\alpha)$ and $b(\alpha)$ in $GF(2^v)$ as the unique polynomial $c(\alpha)$ of degree $\leq v-1$ such that $c(\alpha)$ is the residue of dividing the product $a(\alpha)b(\alpha)$ by $f(\alpha)$ (the notation $g(x) \equiv h(x) \pmod{f(x)}$ means that $g(x)$ and $h(x)$ have the same residue after dividing by $f(x)$, i.e., $g(\alpha) = h(\alpha)$).

The sum and product operations defined above give to $GF(2^v)$ a field structure. The role of the irreducible polynomial $f(x)$ is the same as the prime number $m$ when $Z_m$ is a field. In effect, the proof that $GF(2^v)$ is a field when $m$ is irreducible is essentially the same as the proof that $Z_m$ is a field when $m$ is prime. From now on, we denote the elements in $GF(2^v)$ as polynomials in $\alpha$ of degree $\leq v-1$ with coefficients in $GF(2)$. Given two polynomials $a(x)$ and $b(x)$ with coefficients in $GF(2)$, $a(\alpha)b(\alpha)$ denotes the product in $GF(2^v)$, while $a(x)b(x)$ denotes the regular product of polynomials. Notice that, for the irreducible polynomial $f(x)$, in particular, $f(\alpha) = 0$ in $GF(2^v)$, since $f(x) \equiv 0 \pmod{f(x)}$.

So, the set $GF(2^v)$ given by the irreducible polynomial $f(x)$ of degree $v$ is the set of polynomials of degree $\leq v-1$, where the sum operation is the regular sum of polynomials, and the product operation is the residue of dividing by $f(x)$ the regular product of two polynomials.

**TABLE 18.3** The Finite Field $GF(8)$ Generated by $1 + x + x^3$

| Vector | Polynomial | Power of $\alpha$ | Logarithm |
|---|---|---|---|
| 000 | 0 | 0 | $-\infty$ |
| 100 | 1 | 1 | 0 |
| 010 | $\alpha$ | $\alpha$ | 1 |
| 001 | $\alpha^2$ | $\alpha^2$ | 2 |
| 110 | $1 + \alpha$ | $\alpha^3$ | 3 |
| 011 | $\alpha + \alpha^2$ | $\alpha^4$ | 4 |
| 111 | $1 + \alpha + \alpha^2$ | $\alpha^5$ | 5 |
| 101 | $1 + \alpha^2$ | $\alpha^6$ | 6 |

**Example 5**

Construct the field $GF(8)$. Consider the polynomials of degree $\leq 2$ over $GF(2)$. Let $f(x) = 1 + x + x^3$. Since $f(x)$ has no roots over $GF(2)$, it is irreducible (notice that such an assessment can be made only for polynomials of degree 2 or 3). Let us consider the powers of $\alpha$ modulo $f(\alpha)$. Notice that $\alpha^3 = \alpha^3 + f(\alpha) = 1 + \alpha$. Also, $\alpha^4 = \alpha\alpha^3 = \alpha\ (1 + \alpha) = \alpha + \alpha^2$. Similarly, we obtain $\alpha^5 = \alpha\alpha^4 = \alpha(\alpha + \alpha^2) = \alpha^2 + \alpha^3 = 1 + \alpha + \alpha^2$, and $\alpha^6 = \alpha\alpha^5 = \alpha + \alpha^2 + \alpha^3 = 1 + \alpha^2$. Finally, $\alpha^7 = \alpha\alpha^6 = \alpha + \alpha^3 = 1$.

Note that every nonzero element in $GF(8)$ can be obtained as a power of the element $\alpha$. In this case, $\alpha$ is called a *primitive* element and the irreducible polynomial $f(x)$ that defines the field is called a *primitive* polynomial. It can be proven that it is always the case that the multiplicative group of a finite field is cyclic, so there is always a primitive element.

A convenient description of $GF(8)$ is given in Table 18.3. The first column in Table 18.3 describes the element of the field in vector form, the second one as a polynomial in $\alpha$ of degree $\leq 2$, the third one as a power of $\alpha$, and the last one gives the logarithm (also called Zech logarithm): it simply indicates the corresponding power of $\alpha$. As a convention, we denote by $-\infty$ the logarithm corresponding to the element 0.

It is often convenient to express the elements in a finite field as powers of $\alpha$; when we multiply two of them, we obtain a new power of $\alpha$ whose exponent is the sum of the two exponents modulo $2^v - 1$. Explicitly, if $i$ and $j$ are the logarithms of two elements in $GF(2^v)$, then their product has logarithm $i + j \pmod{(2^v - 1)}$. In the example above, if we want to multiply the vectors 101 and 111, we first look at their logarithms. They are 6 and 5, respectively, so the logarithm of the product is $6 + 5 \pmod 7 = 4$, corresponding to the vector 011.

In order to add vectors, the best way is to express them in vector form and add coordinate to coordinate in the usual way.

## 18.7.5 Cyclic Codes

In the same way we defined codes over the binary field $GF(2)$, we can define codes over any finite field $GF(2^v)$. Now, a code of length $n$ is a subset of $(GF(2^v))^n$, but since we study only linear codes, we require that such a subset is a vector space. Similarly, we define the minimum (Hamming) distance and the generator and parity check matrices of a code. Some properties of binary linear codes, like the Singleton bound, remain the same in the general case. Others, such as the Hamming bound, require some modifications.

Consider a linear code $\mathcal{C}$ over $GF(2^v)$ of length $n$. We say that $\mathcal{C}$ is cyclic if, for any codeword $(c_0, c_1, \ldots, c_{n-1}) \in \mathcal{C}$, then $(c_{n-1}, c_0, c_1, \ldots, c_{n-2}) \in \mathcal{C}$. In other words, the code is invariant under cyclic shifts to the right.

If we write the codewords as polynomials of degree $< n$ with coefficients in $GF(2^v)$, this is equivalent to say that if $c(x) \in \mathcal{C}$, then $xc(x) \bmod (x^n - 1) \in \mathcal{C}$. Hence, if $c(x) \in \mathcal{C}$, then, given any polynomial $w(x)$, the residue of dividing $w(x)c(x)$ by $x^n - 1$ is in $\mathcal{C}$. In particular, if the degree of $w(x)c(x)$ is smaller than $n$, then $w(x)c(x) \in \mathcal{C}$.

From now on, we write the elements of a cyclic code $\mathcal{C}$ as polynomials modulo $x^n - 1$.

**Theorem 2** $\mathcal{C}$ is an $[n, k]$ cyclic code over $GF(2^v)$ if and only if there is a (monic) polynomial $g(x)$ of degree $n-k$ such that $g(x)$ divides $x^n-1$ and each $c(x) \in \mathcal{C}$ is a multiple of $g(x)$, i.e., $c(x) \in \mathcal{C}$ if and only if $c(x) = w(x)g(x)$, $\deg(w) < k$. We call $g(x)$ a generator polynomial of $\mathcal{C}$.

*Proof:* Let $g(x)$ be a monic (i.e., lead coefficient is 1) polynomial in $\mathcal{C}$ such that $g(x)$ has minimal degree. If $\deg(g) = 0$ (i.e., $g = 1$), then $\mathcal{C}$ is the whole space $(GF(2^v))^n$, so assume $\deg(g) \geq 1$. Let $c(x)$ be any element in $\mathcal{C}$. We can write $c(x) = w(x)g(x) + r(x)$, where $\deg(r) < \deg(g)$. Because $\deg(wg) < n$, $g \in \mathcal{C}$ and $\mathcal{C}$ is cyclic, in particular, $w(x)g(x) \in \mathcal{C}$. Hence, $r(x) = c(x) - w(x)g(x) \in \mathcal{C}$. If $r \neq 0$, we would contradict the fact that $g(x)$ has minimal degree, hence, $r = 0$ and $c(x)$ is a multiple of $g(x)$.

Similarly, we can prove that $g(x)$ divides $x^n-1$. Let $x^n - 1 = h(x)g(x) + r(x)$, where $\deg(r) < \deg(g)$. In particular, $h(x)g(x) \equiv -r(x) \bmod (x^n-1)$, hence, $r(x) \in \mathcal{C}$. Since $g(x)$ has minimal degree, $r = 0$, so $g(x)$ divides $x^n-1$.

Conversely, assume that every element in $\mathcal{C}$ is a multiple of $g(x)$ and $g$ divides $x^n-1$. It is immediate that the code is linear and that it has dimension $k$. Let $c(x) \in \mathcal{C}$, hence, $c(x) = w(x)g(x)$ with $\deg(w) < k$. Also, since $g(x)$ divides $x^n-1$, $x^n - 1 = h(x)g(x)$. Assume that $c(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{n-1}x^{n-1}$, then, $xc(x) \equiv c_{n-1} + c_0x + \cdots + c_{n-2}x^{n-1} \pmod{x^n-1}$. We have to prove that $c_{n-1} + c_0x + \cdots + c_{n-2}x^{n-1} = q(x)g(x)$, where $q(x)$ has degree $\leq k-1$. Notice that

$$\begin{aligned} c_{n-1} + c_0x + \cdots + c_{n-2}x^{n-1} &= c_{n-1} + c_0x + \cdots + c_{n-2}x^{n-1} + c_{n-1}x^n - c_{n-1}x^n \\ &= c_0x + \cdots + c_{n-2}x^{n-1} + c_{n-1}x^n - c_{n-1}(x^n - 1) \\ &= xc(x) - c_{n-1}(x^n - 1) \\ &= xw(x)g(x) - c_{n-1}h(x)g(x) \\ &= (xw(x) - c_{n-1}h(x))g(x) \end{aligned}$$

proving that the element is in the code.

**Theorem 2** gives a method to find all cyclic codes of length $n$, simply take all the (monic) factors of $x^n-1$. Each one of them is the generator polynomial of a cyclic code.

### Example 6

Consider the [4,7] cyclic code over $GF(2)$ generated by $g(x) = 1 + x + x^3$. We can verify that $x^7 - 1 = g(x)(1+x)(1+x^2+x^3)$; hence, $g(x)$ indeed generates a cyclic code.

In order to encode an information polynomial over $GF(2)$ of degree $\leq 3$ into a codeword, we multiply it by $g(x)$.

Say that we want to encode $\underline{u} = (1, 0, 0, 1)$, which in polynomial form is $u(x) = 1 + x^3$. Hence, the encoding gives $c(x) = u(x)g(x) = 1 + x + x^4 + x^6$. In vector form, this gives $\underline{c} = (1\ 1\ 0\ 0\ 1\ 0\ 1)$.

It can be easily verified that the [4,7] code given in this example has minimum distance 3 and is equivalent to the Hamming code of Example 4. In other words, the codewords of the code given in this example are permutations of the codewords of the [3,4,7] Hamming code given in Example 4.

The encoding method of a cyclic code with generator polynomial $g$ is then very simple: we multiply the information polynomial by $g$. However, this encoder is not systematic. A systematic encoder of a cyclic code is given by the following algorithm:

**Algorithm 1 (Systematic Encoding Algorithm for Cyclic Codes)** Let $\mathcal{C}$ be a cyclic $[n, k]$ code over $GF(2^v)$ with generator polynomial $g(x)$. Let $u(x)$ be an information polynomial, $\deg(u) < k$. Let $r(x)$ be the residue of dividing $x^{n-k}\,u(x)$ by $g(x)$. Then $u(x)$ is encoded into the polynomial $c(x) = u(x) - x^k\,r(x)$.

We leave as an exercise proving that Algorithm 2 produces indeed a codeword in $\mathcal{C}$.

### Example 7

Consider the [4,7] cyclic code over $GF(2)$ of Example 6. If we want to encode systematically the information vector $\underline{u} = (1, 0, 0, 1)$ (or $u(x) = 1 + x^3$), we have to obtain first the residue of dividing $x^3u(x) = x^3 + x^6$ by $g(x)$. This residue is $r(x) = x + x^2$. Hence, the output of the encoder is $c(x) = u(x) - x^4r(x) = 1 + x^3 + x^5 + x^6$. In vector form, this gives $\underline{c} = (1\ 0\ 0\ 1\ 0\ 1\ 1)$.

### 18.7.6 Reed Solomon Codes

Throughout this subsection, the codes considered are over the field $GF(2^\nu)$. Let $\alpha$ be a primitive element in $GF(2^\nu)$, i.e., $\alpha^{2^\nu-1}=1$,$\alpha^i \neq 1$ for $i$ 0 mod $2^\nu-1$. A Reed–Solomon (RS) code of length $n=2^\nu-1$ and dimension $k$ is the cyclic code generated by

$$g(x) = (x-\alpha)(x-\alpha^2)\cdots(x-\alpha^{n-k-1})(x-\alpha^{n-k})$$

Each $\alpha^i$ is a root of unity, $x-\alpha^i$ divides $x^n-1$, hence, $g$ divides $x^n-1$ and the code is cyclic.

An equivalent way of describing a RS code is as the set of polynomials over $GF(2^\nu)$ of degree $\leq n-1$ with roots $\alpha, \alpha^2, \ldots, \alpha^{n-k}$, i.e., $F$ is in the code if and only if $\deg(F) \leq n-1$ and $F(\alpha)=F(\alpha^2)=\cdots=F(\alpha^{n-k})=0$.

This property allows us to find a parity check matrix for a RS code. Say that $F(x)=F_0+F_1x+\cdots+F_{n-1}x^{n-1}$ is in the code. Let $1 \leq i \leq n-k$, then

$$F(\alpha^i) = F_0 + F_1\alpha^i + \cdots + F_{n-1}\alpha^{i(n-1)} = 0 \tag{18.58}$$

In other words, Eq. 18.58 tells us that codeword $(F_0, F_1, \ldots, F_{n-1})$ is orthogonal to the vectors $(1, \alpha^i, \alpha^{2i}, \ldots, \alpha^{i(n-1)})$, $1 \leq i \leq n-k$. Hence, these vectors are the rows of a parity check matrix for the RS code. A parity check matrix of an $[n, k]$ RS code over $GF(2^\nu)$ is then

$$H = \begin{pmatrix} 1 & \alpha & \alpha^2 & \ldots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \ldots & \alpha^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{n-k} & \alpha^{(n-k)2} & \ldots & \alpha^{(n-k)(n-1)} \end{pmatrix} \tag{18.59}$$

In order to show that $H$ is in fact a parity check matrix, we need to prove that the rows of $H$ are linearly independent. The next lemma provides an even stronger result.

**Lemma 6** Any set of $n-k$ columns in matrix $H$ defined by Eq. 18.59 is linearly independent.

*Proof:* Take a set $0 \leq i_1 < i_2 < \cdots < i_{n-k} \leq n-1$ of columns of $H$. Denote $\alpha^{i_j}$ by $\alpha_j$, $1 \leq j \leq n-k$. Columns $i_1, i_2, \ldots, i_{n-k}$ are linearly independent if and only if their determinant is nonzero, i.e., if and only if

$$\det \begin{pmatrix} \alpha_1 & \alpha_2 & \ldots & \alpha_{n-k} \\ (\alpha_1)^2 & (\alpha_2)^2 & \ldots & (\alpha_{n-k})^2 \\ \vdots & \vdots & \ddots & \vdots \\ (\alpha_1)^{n-k} & (\alpha_2)^{n-k} & \ldots & (\alpha_{n-k})^{n-k} \end{pmatrix} \neq 0 \tag{18.60}$$

Let

$$V(\alpha_1, \alpha_2, \ldots, \alpha_{n-k}) = \det \begin{pmatrix} 1 & 1 & \ldots & 1 \\ \alpha_1 & \alpha_2 & \ldots & \alpha_{n-k} \\ \vdots & \vdots & \ddots & \vdots \\ (\alpha_1)^{n-k-1} & (\alpha_2)^{n-k-1} & \ldots & (\alpha_{n-k})^{n-k-1} \end{pmatrix} \tag{18.61}$$

We call the determinant $V(\alpha_1, \alpha_2, \ldots, \alpha_{n-k})$ a *Vandermonde determinant*—it is the determinant of an $(n-k)\times(n-k)$ matrix whose rows are the powers of vector $\alpha_1, \alpha_2, \ldots, \alpha_{n-k}$, the powers running from 0 to $n-k-1$. By properties of determinants, if we consider the determinant in Eq. 18.60, we have

$$\det\begin{pmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_{n-k} \\ (\alpha_1)^2 & (\alpha_2)^2 & \dots & (\alpha_{n-k})^2 \\ \vdots & \vdots & \ddots & \vdots \\ (\alpha_1)^{n-k} & (\alpha_2)^{n-k} & \dots & (\alpha_{n-k})^{n-k} \end{pmatrix} = \alpha_1\alpha_2\dots\alpha_{n-k}V(\alpha_1, \alpha_2, \dots, \alpha_{n-k}) \tag{18.62}$$

Hence, by Eqs. 18.60 and 18.62, since the $\alpha_j$'s are nonzero, it is enough to prove that $V(\alpha_1, \alpha_2, \dots, \alpha_{n-k}) \neq 0$. A well-known result in literature states that

$$V(\alpha_1, \alpha_2, \dots, \alpha_{n-k}) = \prod_{1<i<j<n-k} (\alpha_j - \alpha_i) \tag{18.63}$$

Because $\alpha$ is a primitive element in $GF(2^v)$, its powers $\alpha^l$, $0 \leq l \leq n-1$ are distinct. In particular, the $\alpha_i$'s, $1 \leq i \leq n-k$ are distinct; hence, the product in the right-hand side of Eq. 18.63 is nonzero.

**Corollary 3** An $[n, k]$ RS code has minimum distance $d = n - k + 1$.

*Proof:* Let $H$ be the parity check matrix of the RS code defined by Eq. 18.59. Notice that, since any $n-k$ columns in $H$ are linearly independent, $d \geq n-k+1$ by Lemma 3.

On the other hand, $d \leq n-k+1$ by the Singleton bound (Corollary 2), so we have equality.

Because RS codes meet the Singleton bound with equality, they are MDS (see second subsection).

**Example 8**

Consider the [3,5,7] RS code over $GF(8)$, where $GF(8)$ is given by Table 18.3. The generator polynomial is

$$g(x) = (x-\alpha)(x-\alpha^2)(x-\alpha^3)(x-\alpha^4) = \alpha^3 + \alpha x + x^2 + \alpha^3 x^3 + x^4$$

Assume that we want to encode the 3-byte vector $\underline{u} = 101\ 001\ 111$. Writing the bytes as powers of $\alpha$ in polynomial form, we have $u(x) = \alpha^6 + \alpha^2 x + \alpha^5 x^2$.

In order to encode $u(x)$, we perform

$$u(x)g(x) = \alpha^2 + \alpha^4 x + \alpha^2 x^2 + \alpha^6 x^3 + \alpha^6 x^4 + \alpha^4 x^5 + \alpha^5 x^6$$

In vector form the output of the encoder is given by 001 011 001 101 101 011 111. If we encode $u(x)$ using a systematic encoder (Algorithm 1), the output of the encoder is

$$\alpha^6 + \alpha^2 x + \alpha^5 x^2 + \alpha^6 x^3 + \alpha^5 x^4 + \alpha^4 x^5 + \alpha^4 x^6$$

which, in vector form, is 101 001 111 101 111 011 011.

Next we make some observations:

- The definition given above for an $[n, k]$ RS code states that $F(x)$ is in the code if and only if it has as roots the powers $\alpha, \alpha^2, \dots, \alpha^{n-k}$ of a primitive element $\alpha$; however, it is enough to state that $F$ has as roots a set of *consecutive* powers of $\alpha$, say, $\alpha^m, \alpha^{m+1}, \dots, \alpha^{m+n-k-1}$, where $0 \leq m \leq n-1$. Although our definition (i.e., $m = 1$) gives the most usual setting for RS codes, often engineering reasons may determine different choices of $m$. It is easy to verify that with the more general definition of RS codes, the minimum distance remains $n-k+1$.
- Given an $[n, k]$ RS code, there is an easy way to shorten it and obtain an $[n-l, k-l]$ code for $l < k$. In effect, if we have only $k-l$ bytes of information, we add $l$ zeros in order to obtain an information string of length $k$. We then find the $n-k$ redundant bytes using a systematic encoder. When writing, of course, the $l$ zeros are not written, so we have an $[n-l, k-l]$ code, called a shortened RS code. It is immediately verified that shortened RS codes are also MDS.

We have defined RS codes, proven that they are MDS and showed how to encode them systematically. The next step, to be developed in the next sections, is decoding them.

### 18.7.7 Decoding of RS Codes: The Key Equation

Through this subsection, $\mathcal{C}$ denotes an $[n, k]$ RS code (unless otherwise stated). Assume that a codeword $F(x) = \sum_{i=0}^{n-1} F_i x^i$ in $\mathcal{C}$ is transmitted and a word $R(x) = \sum_{i=0}^{n-1} R_i x^i$ is received; hence, $F$ and $R$ are related by an error vector $E(x) = \sum_{i=0}^{n-1} E_i x^i$, where $R(x) = F(x) + E(x)$. The decoder will attempt to find $E(x)$.

Let us start by computing the syndromes. For $1 \leq j \leq n-k$, we have

$$S_j = R(\alpha^j) = \sum_{i=0}^{n-1} R_i \alpha^{ij} = \sum_{i=0}^{n-1} E_i \alpha^{ij} \tag{18.64}$$

Before proceeding further, consider Eq. 18.64 in a particular case.

Take the $[n, n-2]$ 1-byte correcting RS code. In this case, we have two syndromes $S_1$ and $S_2$. So, if exactly one error has occurred, say in location $i$, by Eq. 18.64, we have

$$S_1 = E_i \alpha^i \text{ and } S_2 = E_i \alpha^{2i} \tag{18.65}$$

Hence, $\alpha^i = S_2/S_1$, so we can determine the location $i$ in error. The error value is $E_i = (S_1)^2/S_2$.

**Example 9**

Consider the [3,5,7] RS code over $GF(8)$, where $GF(8)$ is given by Table 18.3.

Assume that we want to decode the received vector

$$\underline{r} = (101\ 001\ 110\ 001\ 011\ 010\ 100)$$

which, in polynomial form, is

$$R(x) = \alpha^6 + \alpha^2 x + \alpha^3 x^2 + \alpha^2 x^3 + \alpha^4 x^4 + \alpha x^5 + x^6$$

Evaluating the syndromes, we obtain $S_1 = R(\alpha) = \alpha^2$ and $S_2 = R(\alpha^2) = \alpha^4$. Thus, $S_2/S_1 = \alpha^2$, meaning that location 2 is in error. The error value is $E_2 = (S_1)^2/S_2 = (\alpha^2)^2/\alpha^4 = 1$, which, in vector form, is 100. The output of the decoder is then

$$\underline{c} = (101\ 001\ 010\ 001\ 011\ 010\ 100)$$

which, in polynomial form, is

$$C(x) = \alpha^6 + \alpha^2 x + \alpha x^2 + \alpha^2 x^3 + \alpha^4 x^4 + \alpha x^5 + x^6$$

Let $\mathcal{E}$ be the subset of $\{0, 1, \ldots, n-1\}$ of locations in error, i.e., $E = \{l : E_l \neq 0\}$. With this notation, Eq. 18.64 becomes

$$S_j = \sum_{i \in E} E_i \alpha^{ij}, 1 \leq j \leq n-k \tag{18.66}$$

The decoder will find the error set $\mathcal{E}$ and the error values $E_i$ when the error correcting capability of the code is not exceeded. Thus, if $s$ is the number of errors and $2s \leq n-k$, the system of equations given by Eq. 18.66 has a unique solution. However, this is a nonlinear system, and it is very difficult to solve it directly.

In order to find the set of locations in error $\mathcal{E}$ and the corresponding error values $\{E_i : i \in \mathcal{E}\}$, we define two polynomials. The first one is called the *error locator polynomial*, which is the polynomial that has as roots the values $\alpha^{-i}$, where $i \in \mathcal{E}$. We denote this polynomial by $\sigma(x)$. Explicitly,

$$\sigma(x) = \prod_{i \in \mathcal{E}} (x - \alpha^{-i}) \tag{18.67}$$

If somehow we can determine the polynomial $\sigma(x)$, by finding its roots, we can obtain the set $\mathcal{E}$ of locations in error. Once we have the set of locations in error, we need to find the errors themselves. We define a second polynomial, called the *error evaluator polynomial* and denoted by $\omega(x)$, as follows:

$$\omega(x) = \prod_{i \in \mathcal{E}} E_i \prod_{\substack{i \in \mathcal{E} \\ l \neq i}} (x - \alpha^{-1}) \tag{18.68}$$

An $[n, k]$ RS code corrects at most $(n-k)/2$ errors, so we assume that $|\mathcal{E}| = \deg(\sigma) \leq (n-k)/2$. Notice also that $\deg(\omega) \leq |\mathcal{E}| - 1$, since $\omega$ is a sum of polynomials of degree $|\mathcal{E}| - 1$. Given a polynomial $f(x) = a_0 + a_1 x + \cdots + a_m x^m$ with coefficients over a field $F$, we define the (formal) derivative of $F$, denoted $f'$, as the polynomial

$$f'(x) = a_1 + 2a_2 x + \cdots + m a_m x^{m-1}$$

For instance, over $GF(8)$, if $f(x) = \alpha + \alpha^3 x + \alpha^4 x^2$, then $f'(x) = \alpha^3$ (since $2 = 0$ over $GF(2)$). The formal derivative has several properties similar to the traditional derivative, like the derivative of a product, $(fg)' = f'g + fg'$. Back to the error locator and error evaluator polynomials, we have the following relationship between the two:

$$E_i = \frac{\omega(\alpha^{-i})}{\sigma'(\alpha^{-i})} \tag{18.69}$$

Let us prove some of these facts in the following lemma:

**Lemma 7** The polynomials $\sigma(x)$ and $\omega(x)$ are relatively prime, and the error values $E_i$ are given by Eq. 18.69.

*Proof:* In order to show that $\sigma(x)$ and $\omega(x)$ are relatively prime, it is enough to observe that they have no roots in common. In effect, if $\alpha^{-j}$ is a root of $\sigma(x)$, then $j \in \mathcal{E}$. By Eq. 18.68,

$$\omega(\alpha^{-j}) = \sum_{i \in \mathcal{E}} E_i \prod_{\substack{i \in \mathcal{E} \\ l \neq i}} (\alpha^{-j} - \alpha^{-l}), = E_j \prod_{\substack{i \in \mathcal{E} \\ l \neq i}} (\alpha^{-j} - \alpha^{-l}) \neq 0 \tag{18.70}$$

Hence, $\sigma(x)$ and $\omega(x)$ are relatively prime.

In order to prove (18.69), notice that

$$\sigma'(x) = \sum_{i \in \mathcal{E}} \prod_{\substack{i \in \mathcal{E} \\ l \neq i}} (x - \alpha^{-1})$$

hence

$$\sigma'(\alpha^{-j}) = \prod_{\substack{i\in\mathcal{E}\\ l\neq i}} (\alpha^{-j} - \alpha^{-1}) \tag{18.71}$$

By Eqs. 18.70 and 18.71, Eq. 18.69 follows.

The decoding methods of RS codes are based on finding the error locator and the error evaluator polynomials. By finding the roots of the error locator polynomial, we determine the locations in error, while the errors themselves can be found using Eq. 18.69. We will establish a relationship between $\sigma(x)$ and $\omega(x)$, but first we need to define a third polynomial, the syndrome polynomial. We define the syndrome polynomial as the polynomial of degree $\leq n-k-1$ where coefficients are the $n-k$ syndromes. Explicitly,

$$S(x) = S_1 + S_2 x + S_3 x^2 + \cdots + S_{n-k} x^{n-k-1} = \sum_{j=0}^{n-k-1} S_{j+1} x^j \tag{18.72}$$

Notice that $R(x)$ is in $\mathcal{C}$ if and only if $S(x) = 0$.

The next theorem gives the so-called *key equation* for decoding RS codes, and it establishes a fundamental relationship between $\sigma(x)$, $\omega(x)$, and $S(x)$.

**Theorem 3** There is a polynomial $\mu(x)$ such that the error locator, the error evaluator and the syndrome polynomials verify the following equation:

$$\sigma(x)S(x) = -\omega(x) + \mu(x)x^{n-k} \tag{18.73}$$

Alternatively, Eq. 18.73 can be written as a congruence as follows:

$$\sigma(x)S(x) = -\omega(x) (\text{mod } x^{n-k}) \tag{18.74}$$

*Proof:* By Eqs. 18.72 and 18.66, we have

$$\begin{aligned} S(x) &= \sum_{j=0}^{n-k-1} S_{j+1} x^j \\ &= \sum_{j=0}^{n-k-1} \left( \sum_{i\in\mathcal{E}} E_i \alpha^{i(j+1)} \right) x^j \\ &= \sum_{i\in\mathcal{E}} E_i \alpha^i \sum_{j=0}^{n-k-1} (\alpha^i x)^j \\ &= \sum_{i\in\mathcal{E}} E_i \alpha^i \frac{(\alpha^i x)^{n-k} - 1}{\alpha^i x - 1} \\ &= \sum_{i\in\mathcal{E}} E_i \frac{(\alpha^i x)^{n-k} - 1}{x - \alpha^{-1}} \end{aligned} \tag{18.75}$$

because $\sum_{l=0}^{m} a^l = (a^{m+1} - 1)/(a - 1)$ for $a \neq 1$. Multiplying both sides of Eq. 18.75 by $\sigma(x)$, where $\sigma(x)$ is given by Eq. 18.67, we obtain

$$\sigma(x)S(x) = \sum_{i\in\mathcal{E}} E_i(\,(\alpha^i x)^{n-k} - 1) \prod_{\substack{l\in\mathcal{E}\\ l\neq i}} (x - \alpha^{-1})$$

$$= -\sum_{i\in\mathcal{E}} E_i \prod_{\substack{l\in\mathcal{E}\\ l\neq i}} (x - \alpha^{-1}) + \left(\sum_{i\in\mathcal{E}} E_i \alpha^{i(n-k)} \prod_{\substack{l\in\mathcal{E}\\ l\neq i}} (x - \alpha^{-1})\right) x^{n-k}$$

$$= -\omega(x) + \mu(x)x^{n-k}$$

because $\omega(x)$ is given by Eq. 18.68. This completes the proof.

The decoding methods for RS codes concentrate on solving the key equation. In the next section we describe an efficient decoder based on Euclid's algorithm for polynomials. Another efficient decoding algorithm is the so-called Berlekamp–Massey decoding algorithm [1].

## 18.7.8 Decoding RS Codes with Euclid's Algorithm

Given two polynomials or integers $A$ and $B$, Euclid's algorithm provides a recursive procedure to find the greatest common divisor $C$ between $A$ and $B$, denoted $C = \gcd(A, B)$. Moreover, the algorithm also finds two polynomials or integers $S$ and $T$ such that $C = SA + TB$.

Recall that we want to solve the key equation

$$\mu(x)x^{n-k} + \sigma(x)S(x) = -\omega(x)$$

In the recursion, $x^{n-k}$ will play the role of $A$ and $S(x)$ the role of $B$; $\sigma(x)$ and $\omega(x)$ will be obtained at a certain step of the recursion.

Let us describe Euclid's algorithm for integers or polynomials. Consider $A$ and $B$ such that $A \geq B$ if they are integers, and $\deg(A) \geq \deg(B)$ if they are polynomials. We start from the initial conditions $r_{-1} = A$ and $r_0 = B$.

We perform a recursion in steps $1, 2, \ldots, i, \ldots$. At step $i$ of the recursion, we obtain $r_i$ as the residue of dividing $r_{i-2}$ by $r_{i-1}$, i.e., $r_{i-2} = q_i r_{i-1} + r_i$, where $r_i < r_{i-1}$ for integers and $\deg(r_i) < \deg(r_{i-1})$ for polynomials. The recursion is then given by

$$r_i = r_{i-2} - q_i r_{i-1} \tag{18.76}$$

We also obtain values $s_i$ and $t_i$ such that $r_i = s_i A + t_i B$. Hence, the same recursion is valid for $s_i$ and $t_i$ as well:

$$s_i = s_{i-2} - q_i s_{i-1} \tag{18.77}$$

$$t_i = t_{i-2} - q_i t_{i-1} \tag{18.78}$$

Because $r_{-1} = A = (1)A + (0)B$ and $r_0 = B = (0)A + (1)B$, we set the initial conditions $s_{-1} = 1$, $t_{-1} = 0$, $s_0 = 0$ and $t_0 = 1$.

Let us illustrate the process with $A = 124$ and $B = 46$. We will find gcd(124,46). The idea is to divide recursively by the residues of the division until obtaining a last residue 0. Then, the last divisor is the gcd. The procedure works as follows:

**TABLE 18.4** Euclid's Algorithm for gcd(124,46)

| $i$ | $r_i$ | $q_i$ | $s_i = s_{i-2} - q_i s_{i-1}$ | $t_i = t_{i-2} - q_i t_{i-1}$ |
|---|---|---|---|---|
| −1 | 124 | | 1 | 0 |
| 0 | 46 | | 0 | 1 |
| 1 | 32 | 2 | 1 | −2 |
| 2 | 14 | 1 | −1 | 3 |
| 3 | 4 | 2 | 3 | −8 |
| 4 | 2 | 3 | −10 | 27 |
| 5 | 0 | 2 | 23 | −62 |

$$\begin{aligned} 124 &= (1)124 + (0)46 \\ 46 &= (0)124 + (1)46 \\ 32 &= (1)124 + (-2)46 \\ 14 &= (-1)124 + (3)46 \\ 4 &= (3)124 + (-8)46 \\ 2 &= (-10)124 + (27)46 \end{aligned}$$

Because 2 divides 4, 2 is the greatest common divisor between 124 and 46.

The best way to develop the process above is to construct a table for $r_i$, $q_i$, $s_i$, and $t_i$, using the initial conditions and recursions in Eqs. 18.76 through 18.78. Table 18.4 provides such a table for 124 and 46.

From now on, let us concentrate on Euclid's algorithm for polynomials. If we want to solve the key equation

$$\mu(x)x^{n-k} + \sigma(x)S(x) = -\omega(x)$$

and the error correcting capability of the code has not been exceeded, then applying Euclid's algorithm to $x^{n-k}$ and to $S(x)$, at a certain point of the recursion we obtain

$$r_i(x) = s_i(x)x^{n-k} + t_i(x)S(x)$$

where $\deg(r_i) \leq \lfloor (n-k)/2 \rfloor - 1$, and $i$ is the first with this property. Then, $\omega(x) = -\lambda r_i(x)$ and $\sigma(x) = \lambda t_i(x)$, where $\lambda$ is a constant that makes $\sigma(x)$ monic. For a proof that Euclid's algorithm gives the right solution, see [1] or [5].

We illustrate the decoding of RS codes using Euclid's algorithm with an example. Notice that we are interested in $r_i(x)$ and $t_i(x)$ only.

**Example 10**

Consider the [3,5,7] RS code over $GF(8)$ and assume that we want to decode the received vector

$$\underline{r} = (011\ 101\ 111\ 111\ 111\ 101\ 010)$$

which, in polynomial form, is

$$R(x) = \alpha^4 + \alpha^6 x + \alpha^5 x^2 + \alpha^5 x^3 + \alpha^5 x^4 + \alpha^6 x^5 + \alpha x^6$$

Evaluating the syndromes, we obtain

**TABLE 18.5** Decoding of RS Codes Using Euclid's Algorithm

| $i$ | $r_i = r_{i-2} - q_i r_{i-1}$ | $q_i$ | $t_i = t_{i-2} - q_i t_{i-1}$ |
|---|---|---|---|
| −1 | $x^4$ | | 0 |
| 0 | $\alpha^5 + \alpha x + \alpha^3 x^3$ | | 1 |
| 1 | $\alpha^2 x + \alpha^5 x^2$ | $\alpha^4 x$ | $\alpha^4 x$ |
| 2 | $\alpha^5 + \alpha^2 x$ | $\alpha^2 + \alpha^5 x$ | $1 + \alpha^6 x + \alpha^2 x^2$ |

$$S_1 = R(\alpha) = \alpha^5$$
$$S_2 = R(\alpha^2) = \alpha$$
$$S_3 = R(\alpha^3) = 0$$
$$S_4 = R(\alpha^4) = \alpha^3$$

Therefore, the syndrome polynomial is $S(x) = \alpha^5 + \alpha x + \alpha^3 x^3$.

Next, we apply Euclid's algorithm with respect to $x^4$ and to $S(x)$. When we find the first $i$ for which $r_i(x)$ has degree $\leq 1$, we stop the algorithm and obtain $w(x)$ and $\sigma(x)$. The process is illustrated in Table 18.5.

So, for $i = 2$, we obtain a polynomial $r_2(x) = \alpha^5 + \alpha^2 x$ of degree 1. Now, multiplying both $r_2(x)$ and $t_2(x)$ by $\lambda = \alpha^5$, we obtain $\omega(x) = \alpha^3 + x$ and $\sigma(x) = \alpha^5 + \alpha^4 x + x^2$.

Searching the roots of $\sigma(x)$, we verify that these roots are $\alpha^0 = 1$ and $\alpha^5$; hence, the errors are in locations 0 and 2. The derivative of $\sigma(x)$ is $\sigma'(x) = \alpha^4$. By Eq. 18.69, we obtain $E_0 = \omega(1)/\sigma'(1) = \alpha^4$ and $E_2 = \omega(\alpha^5)/\sigma'(\alpha^5) = \alpha^5$. Adding $E_0$ and $E_2$ to the received locations 0 and 2, the decoder concludes that the transmitted polynomial was

$$F(x) = \alpha^6 x + \alpha^5 x^3 + \alpha^5 x^4 + \alpha^6 x^5 + \alpha x^6$$

which, in vector form, is

$$\underline{c} = (000\ 101\ 000\ 111\ 111\ 101\ 010)$$

If the information is carried in the first three bytes, the output of the decoder is

$$\underline{u} = (000\ 101\ 000)$$

## 18.7.9 Applications: Burst and Random Error Correction

In the previous sections we have studied how to encode and decode RS codes. This subsection will briefly examine how they are used in applications, mainly for correction of bursts of errors. The two main methods for burst and combined burst and random error correction are interleaving and product codes.

In practice, errors often come in bursts. A burst of length $l$ is a vector whose nonzero entries are among $l$ consecutive (cyclically) entries, the first and last of them being nonzero. We consider binary bursts, and we use the elements of larger fields (bytes) to correct them. Below are some examples of bursts of length 4 in vectors of length 15:

0 0 0 1 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 1 0 0

Errors tend to come in bursts not only because the channel is bursty. Normally, both in optical and magnetic recording, data are encoded using a so-called modulation code, which attempts to match the data to the characteristics of the channel. In general, the ECC is applied first to the random data and then the encoded data are modulated using modulation codes (see the Section 18.5 on modulation codes in this chapter). At the decoding, the order is reversed; when data exits the channel, it is first demodulated and then corrected using the ECC. Now, the demodulator tends to propagate errors, even single-bit errors. Although most modulation codes used in practice tend to control error propagation, nevertheless errors have a bursty character. For that reason, we need to implement a burst-correcting scheme, as we will see next.

A well-known relationship between the burst-correcting capability of a code and it redundancy is given by the Reiger bound, to be presented next, and whose proof is left as an exercise.

**Theorem 4 (Reiger Bound)** Let $\mathcal{C}$ be an $[n, k]$ linear code over a field $GF(2^v)$ that can correct all bursts of length up to $l$. Then $2l \leq n - k$.

Cyclic binary codes that can correct bursts were obtained by computer search. A well known family of burst-correcting codes are the so-called Fire codes. Here, we concentrate on the use of RS codes for burst correction. There are good reasons for this. One of them is that, although good burst-correcting codes have been found by computer search, there are no known general constructions giving cyclic codes that approach the Reiger bound. Interleaving of RS codes on the other hand, to be described below, provides a burst-correcting code whose redundancy, asymptotically, approaches the Reiger bound. The longer the burst we want to correct, the more efficient the interleaving of RS codes is. The second reason for choosing interleaving of RS codes, and probably the most important one, is that, by increasing the error-correcting capability of the individual RS codes, we can correct multiple bursts, as we will see. The known binary cyclic codes are designed, in general, to correct only one burst.

Let us start with the use of regular RS codes for correction of bursts. Let $C$ be an $[n, k]$ RS code over $GF(2^b)$ (i.e., $b$-bit bytes). If this code can correct $s$ bytes, in particular, it can correct a burst of length up to $(s-1)b+1$ bits. In effect, a burst of length $(s-1)b+2$ bits may affect $s+1$ consecutive bytes, exceeding the byte-correcting capability of the code. This happens when the burst of length $(s-1)b+2$ bits starts in the last bit of a byte. How good are then RS codes as burst-correcting codes? Given a binary $[n, k]$ code that can correct bursts of length up to $l$, we define a parameter, called the *burst-correcting efficiency* of the code, as follows:

$$e_l = \frac{2l}{n-k} \tag{18.79}$$

Notice that, by the Reiger bound, $e_l \leq 1$. The closer $e_l$ is to 1, the more efficient the code is for correction of bursts. Going back to our $[n, k]$ RS code over $GF(2^b)$, it can be regarded as an $[nb, kb]$ binary code. Assuming that the code can correct $s$ bytes and its redundancy is $n-k=2s$, its burst-correcting efficiency is

$$e_{(s-1)b+1} = \frac{(s-1)b+1}{bs}$$

Notice that, for $s \to \infty$, $e_{(s-1)b+1} \to 1$, justifying our assertion that for long bursts, RS codes are efficient as burst-correcting codes (as a comparison, the efficiency of Fire codes asymptotically tends to 2/3); however, when $s$ is large, there is a problem regarding complexity. It may not be practical to implement a RS code with too much redundancy. Moreover, the length of a RS code is limited, in the case of 8-bit bytes, it cannot be more than 256 (when extended). An alternative would be to implement a 1-byte correcting RS code interleaved $s$ times.

An $[n, k]$ code interleaved $m$ times is illustrated in Fig. 18.63. Each column $c_{0,j}, c_{1,j}, \ldots, c_{n-1,j}$ is a codeword in an $[n, k]$ code. In general, each symbol $c_{i,j}$ is a byte and the code is a RS code. The first $k$

| | | | | |
|---|---|---|---|---|
| $c_{0,0}$ | $c_{0,1}$ | $c_{0,2}$ | $\cdots$ | $c_{0,m-1}$ |
| $c_{1,0}$ | $c_{1,1}$ | $c_{1,2}$ | $\cdots$ | $c_{1,m-1}$ |
| $c_{2,0}$ | $c_{2,1}$ | $c_{2,2}$ | $\cdots$ | $c_{2,m-1}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $c_{k-1,0}$ | $c_{k-1,1}$ | $c_{k-1,2}$ | $\cdots$ | $c_{k-1,m-1}$ |
| $c_{k,0}$ | $c_{k,1}$ | $c_{k,2}$ | $\cdots$ | $c_{k,m-1}$ |
| $c_{k+1,0}$ | $c_{k+1,1}$ | $c_{k+1,2}$ | $\cdots$ | $c_{k+1,m-1}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $c_{n-1,0}$ | $c_{n-1,1}$ | $c_{n-1,2}$ | $\cdots$ | $c_{n-1,m-1}$ |

**FIGURE 18.63** Interleaving $m$ times of code $\mathcal{C}$.

bytes carry information bytes and the last $n - k$ bytes are redundant bytes. The bytes are read in row order, and the parameter $m$ is called the depth of interleaving. If each of the individual codes can correct up to $s$ errors, the interleaved scheme can correct up to $s$ bursts of length up to $m$ bytes each, or $(m-1)b+1$ bits each. This occurs because a burst of length up to $m$ bytes is distributed among $m$ different codewords. Intuitively, interleaving "randomizes" a burst.

The drawback of interleaving is delay. Notice that we need to read most of the information bytes before we are able to calculate and write the redundant bytes. Thus, we need enough buffer space to accomplish this.

Interleaving of RS codes has been widely used in magnetic recording. For instance, in a disk, the data are written in concentric tracks, and each track contains a number of information sectors. Typically, a sector consists of 512 information 8-bit bytes (although the latest trends tend to larger sectors). A typical embodiment would consist in dividing the 512 bytes into four codewords, each one containing 128 information bytes and six redundant bytes (i.e., each interleaved shortened RS codeword can correct up to three bytes). Therefore, this scheme can correct up to three bursts of length up to 25 bits each.

A natural generalization of the interleaved scheme described above is product codes. In effect, we may consider that both rows and columns are encoded into error-correcting codes. The product of an $[n_1, k_1]$ code $\mathcal{C}_1$ with an $[n_2, k_2]$ code $\mathcal{C}_2$, denoted $\mathcal{C}_1 \times \mathcal{C}_2$, is illustrated in Fig. 18.64. If $\mathcal{C}_1$ has minimum distance $d_1$ and $\mathcal{C}_2$ has minimum distance $d_2$, it is easy to see that $\mathcal{C}_1 \times \mathcal{C}_2$ has minimum distance $d_1 d_2$.

In general, the symbols are read out in row order (although other readouts, like diagonal readouts, are also possible). For encoding, first the column redundant symbols are obtained, and then the row redundant symbols. For obtaining the checks on checks $c_{i,j}$, $k_1 \leq i \leq n_1 - 1$, $k_2 \leq j \leq n_2 - 1$, it is easy to see that it is irrelevant if we encode on columns or on rows first. If the symbols are read in row order, normally $\mathcal{C}_1$ is called the outer code and $\mathcal{C}_2$ the inner code. For decoding, many possible procedures are used. The idea is to correct long bursts together with random errors. The inner code $\mathcal{C}_2$ corrects first. In that case, two events may happen when its error-correcting capability is exceeded: either the code will detect the error event or it will miscorrect. If the code detects an error event (that may well have been caused by a long burst), one alternative is to declare an erasure in the whole row, which will be communicated to the outer code $\mathcal{C}_1$. The other event is a miscorrection, that

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $c_{0,0}$ | $c_{0,1}$ | $c_{0,2}$ | $\cdots$ | $c_{0,k_2-1}$ | $c_{0,k_2}$ | $c_{0,k_2+1}$ | $\cdots$ | $c_{0,n_2-1}$ |
| $c_{1,0}$ | $c_{1,1}$ | $c_{1,2}$ | $\cdots$ | $c_{1,k_2-1}$ | $c_{1,k_2}$ | $c_{1,k_2+1}$ | $\cdots$ | $c_{1,n_2-1}$ |
| $c_{2,0}$ | $c_{2,1}$ | $c_{2,2}$ | $\cdots$ | $c_{2,k_2-1}$ | $c_{2,k_2}$ | $c_{2,k_2+1}$ | $\cdots$ | $c_{2,n_2-1}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $c_{k_1-1,0}$ | $c_{k_1-1,1}$ | $c_{k_1-1,2}$ | $\cdots$ | $c_{k_1-1,k_2-1}$ | $c_{k_1-1,k_2}$ | $c_{k_1-1,k_2+1}$ | $\cdots$ | $c_{k_1-1,n_2-1}$ |
| $c_{k_1,0}$ | $c_{k_1,1}$ | $c_{k_1,2}$ | $\cdots$ | $c_{k_1,k_2-1}$ | $c_{k_1,k_2}$ | $c_{k_1,k_2+1}$ | $\cdots$ | $c_{k_1,n_2-1}$ |
| $c_{k_1+1,0}$ | $c_{k_1+1,1}$ | $c_{k_1+1,2}$ | $\cdots$ | $c_{k_1+1,k_2-1}$ | $c_{k_1+1,k_2}$ | $c_{k_1+1,k_2+1}$ | $\cdots$ | $c_{k_1+1,n_2-1}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $c_{n_1-1,0}$ | $c_{n_1-1,1}$ | $c_{n_1-1,2}$ | $\cdots$ | $c_{n_1-1,k_2-1}$ | $c_{n_1-1,k_2}$ | $c_{n_1-1,k_2+1}$ | $\cdots$ | $c_{n-1,n_2-1}$ |

**FIGURE 18.64** Product code $\mathcal{C}_1 \times \mathcal{C}_2$.

cannot be detected. In this case, we expect that the errors will be corrected by the error-erasure decoder of the outer code.

Product codes are important in practical applications. For instance, the code used in the DVD (digital video disk) is a product code where $\mathcal{C}_1$ is a RS code and $\mathcal{C}_2$ is a RS code. Both RS codes are defined over $GF(256)$, where $GF(256)$ is generated by the primitive polynomial $1 + x^2 + x^3 + x^4 + x^8$.

## References

1. R.E. Blahut, *Theory and Practice of Error Control Codes,* Addison-Wesley, Reading, MA, 1983.
2. C. Heegard and S.B. Wicker, *Turbo Coding,* Kluwer Academic Publishers, Dordrecht, the Netherlands, 1999.
3. S. Lin and D.J. Costello, *Error Control Coding: Fundamentals and Applications,* Prentice-Hall, Englewood Cliffs, NJ, 1983.
4. F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error-Correcting Codes,* North-Holland Publishing Company, 1978.
5. R.J. McEliece, *The Theory of Information and Coding,* Addison-Wesley, Reading, MA, 1977.
6. C.E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. Journal,* vol. 27, pp. 379–423 and 623–656, 1948.
7. W. Wesley Peterson and E.J. Weldon, *Error-Correcting Codes,* MIT Press, Cambridge, MA, second edition, 1984.
8. S. Wicker, *Error Control Systems for Digital Communications and Storage,* Prentice-Hall, Englewood Cliffs, NJ, 1995.

# VI

# Operating System

# 19

# Distributed Operating Systems




Peter Reiher
*University of California*


## 19.1 Definitions and Importance

A distributed operating system is a software that runs on several machines and its purpose is to provide a useful set of services, usually to make the collection of machines behave more like a single machine. The distributed operating system plays the same role in making the collective resources of the machines more usable that a typical single-machine operating system plays in making that machine's resources more usable.

Distributed operating systems are usually viewed as running cooperatively on all machines whose resources they control. These machines might be capable of independent operation or they might be usable merely as resources in the distributed system. Unlike parallel operating systems, a distributed operating system typically runs on loosely coupled hardware. Parallel operating systems tend to focus on making all available resources usable by a single large task, whereas distributed operating systems focus on making the collection of resources usable by a set of loosely cooperating users. Network operating systems are sometimes regarded as systems that attempt merely to make the network connecting the machines more usable, without regard for some of the larger problems of building effective distributed systems. The distinctions between parallel, distributed, and network operating systems are somewhat arbitrary, because all the operating systems must handle similar problems.

Distributed operating systems are not in common use today. Although many interesting research systems have been built since the 1970s, and some systems have been in use for many years, they have not

displaced traditional operating systems designed primarily to support single machines; however, some of the components originally built for distributed operating systems have become commonplace in today's systems, notably services to access files stored on remote machines. The failure of distributed operating systems to capture a large share of the marketplace may be primarily due to our lack of understanding on how to build them, or perhaps their lack of popularity stems from users not really needing many distributed services not already provided.

Distributed operating systems are also an important field for study because they have helped drive general research in distributed systems. Replicated data systems, authentication services such as Kerberos, agreement protocols, methods of providing causal ordering in communications, voting and consensus protocols, and many other distributed services have been developed to support distributed operating systems, and have found varying degrees of success outside that field. Popular distributed component services like CORBA owe some of their success to applying hard lessons learned by researchers in distributed operating systems. The popularity of the World Wide Web suggests that users would desire a more global view of the resources available to them than is provided by today's operating systems.

Distributed operating systems are hard to design properly. They must solve inherently hard problems in system design. Further, they must properly trade-off issues of performance, user interfaces, reliability, and simplicity. The relative scarcity of such systems, and the fact that most commercial operating systems' design still focuses on single-machine systems suggests that no distributed operating system yet developed has found the proper trade-off among these issues.

Research continues in distributed operating systems, particularly in certain critical elements of them that have obvious value, especially file systems and other forms of data sharing. Other continuing research in distributed operating systems focuses on their use in important special cases, such as high-performance clustered servers and grid computing. The increasing popularity of portable and handheld computers may lead to more distributed operating system research to support mobile computing. The emerging field of ubiquitous computing offers different hardware, networking, and application characteristics likely to spur further research on distributed operating systems. Peer systems, currently used primarily to share data, are also likely to spur further research in distributed operating systems issues.

## 19.2 Why Are Distributed Operating Systems Hard to Build?

This question touches directly on why distributed operating systems are not commonly used and also helps explain why research continues rapidly in certain areas, whereas it moves more slowly in others.

One core problem for distributed operating system designers is concurrency and synchronization. These issues arise in single-machine operating systems, but they are easier to solve there. Typical single-machine systems run a single thread of control simultaneously, simplifying many synchronization problems. Further, they typically have access to memory, registers, or other useful physical resources that are directly accessible by all processes that they must synchronize. These shared resources allow use of simple and fast synchronization primitives, such as semaphores. Even modern machines that have multiple processors typically include hardware that makes it easier to synchronize their operations.

Distributed operating systems lack these advantages. Typically, they must control a collection of processors connected by a network, most often a local area network (LAN), but occasionally a network with even more difficult characteristics. The access time across this network is orders of magnitude larger than the access time required to reach local main memory and even more orders of magnitude larger than that required to reach information in a local processor cache or register. Further, such networks are not as reliable as a typical bus, so, messages are more likely to be lost or corrupted. At best, this unreliability increases the average access time.

This imbalance means that running blocking primitives across the network is often infeasible. The performance implications for the individual component systems and the system as a whole do not permit widespread use of such primitives. Designers must choose between looser synchronization

(leading to odd user-visible behaviors and possibly fatal system inconsistencies) and sluggish performance. The increasing gap between processor and network speeds suggests that this effect will only get worse.

Theoretical results in distributed systems are discouraging. Research on various forms of the Byzantine general problem and other formulations of the problems of reaching decisions in distributed systems has provided surprising results with bad implications for the possibility of providing perfect synchronization of such systems. Briefly, these results suggest that reaching a distributed decision is not always possible in common circumstances. Even when it is possible, doing so in unfavorable conditions is very expensive and tricky. Although most distributed systems can be designed to operate in more favorable circumstances than these gloomy theoretical results describe (typically by assuming less drastic failure modes or less absolute need for complete consistency), experience has shown that even pragmatic algorithm design for this environment is difficult.

A further core problem is providing transparency. Transparency has various definitions and aspects, but at a high level it simply refers to the degree to which the operating system disguises the distributed nature of the system. Providing a high degree of transparency is good because it shields the user from the complexities of distribution. On the other hand, it sometimes hides more than it should, it can be expensive and tricky to provide, and ultimately it is not always possible. A key decision in designing a distributed operating system is how much transparency to provide, and where and when to provide it.

A related problem is that the hardware, which the distributed operating system must virtualize, is more varied. A distributed operating system must make not only a file on disk appear to be in the main memory, as a typical operating system does, but also a file on a different machine appear to be on the local machine, even if it is simultaneously being accessed on yet a third machine. The system should not just make a multi-machine computation appear to run on a single machine, but should provide observers on all machines with the illusion that it is running only on their machine.

Distributed operating systems also face challenging problems because they are typically intended to continue correct operation despite failure of some of their components. Most single-machine operating systems provide very limited abilities to continue operation if key components fail. They are certainly not expected to provide useful service if their processor crashes. A single processor crash in a distributed operating system should allow the remainder of the system to continue operations largely unharmed. Achieving this ideal can be extremely challenging. If the topology of the network connecting the system's component nodes allows the network to split into disjoint pieces, the system might also need to continue operation in a partitioned mode and would be expected to rapidly reintegrate when the partitions merge.

The security problems of a distributed operating system are also harder. First, data typically moves over a network, sometimes over a network that the distributed operating system itself does not directly control. Second, access control and resource management mechanisms on single machines typically take advantage of hardware that helps keep processes separate, such as page registers. Distributed operating systems cannot rely on this advantage. Third, distributed operating systems are typically expected to provide some degree of local control to users on their individual machines, while still enforcing general access control mechanisms. When an individual user is legitimately able to access any bytes stored anywhere on his own machine, preventing him from accessing data that belongs to others is a much harder problem, particularly if the system strives to provide controlled high-performance access to that data.

Distributed operating systems must often address the issue of local autonomy. In many (but not all) architectures, the distributed system is composed of workstations whose primary job is to support one particular user. The distributed system must balance the needs of the entire collection of supported users against the natural expectation that one's machine should be under one's own control. The local autonomy question not only has clear security implications, but also relates to how resources are allocated, how scheduling is done, and other issues.

In many cases, distributed operating systems are expected to run on heterogeneous hardware. Although commercial convergence on a small set of popular processors has reduced this problem to

some extent, the wide variety of peripheral devices and customizations of system settings provided by today's operating systems often makes supposedly identical hardware behave radically differently. If a distributed operating system cannot determine whether running the same operation on two different component nodes produces the same result, it will face difficulties in providing transparency and consistency.

All the previously mentioned problems are exacerbated if the system scale becomes sufficiently large. Many useful distributed algorithms scale poorly, because the number of messages they require faces combinatorial explosion, or because the delays required to include large numbers of nodes in computations become unreasonable, or because data structures grow in proportion to the number of participants. High scale ensures that partial failures will become more common, and that low probability events will begin to pop up very often. High scale might also imply that the distributed operating system must operate away from the relatively friendly world of the LAN, leading to greater heterogeneity and uncertainty in communications.

An entirely different paradigm of building system software for distributed systems can avoid some of these difficulties. Sensor networks, rather than performing general-purpose computing, are designed only to gather information from sensors and send it to places that need it. The nodes in a sensor network are typically very simple and have low power in many dimensions, from CPU speed to battery. As a result, while inherently distributed systems, sensor network nodes must run relatively simple code. Operating systems designed for sensor networks, like TinyOS [1], are, thus, themselves extremely simple. By proper design of the operating system and algorithms that perform the limited applications, a sensor network achieves a cooperative distributed goal without worrying about many of the classic issues of distributed operating systems, such as tight synchronization, data consistency, and partial failure. This approach does not seem to offer an alternative when one is designing a distributed operating system for typical desktop or server machines, but may prove to be a powerful tool for other circumstances in which the nodes in the distributed systems need to do only very particular and limited tasks.

## 19.3 Components of Distributed Operating Systems

Distributed operating systems consist of components that generally mirror these in their single-machine counterparts. Some of these components are typically little altered from single-machine operating systems. For example, many distributed operating systems schedule their tasks per machine, using standard scheduling mechanisms. Unless distributed shared memory is supported, components of the operating system that support virtual memory, paging, and swapping are typically similar to other operating systems. Support for devices and the use of device drivers is also usually much like that in a single-machine operating system. Note that these similarities are merely the rule in distributed operating systems, and that some such systems handle even these components differently than in single-machine operating systems.

The components that tend to be most different are file systems, interprocess communications, synchronization, reliability and recovery, and security. Also, some distributed operating systems include support for process migration, a facility that makes no sense in an operating system that limits its view to the boundaries of a single machine.

### 19.3.1 File Systems

File systems were quickly recognized as one of the areas of a distributed operating system that required the most attention. Before the development of distributed operating systems, remote file access was limited to explicit copying, perhaps aided by a program such as FTP. Distributed operating system designers recognized the value added to a computer on a network by giving it better access to files stored on other machines. The approach was generally to provide some integrated view of the collection of files stored on the various machines of the distributed system.

A key question in distributed file system design is the degree of transparency provided. At one extreme, services like FTP offer very little transparency. File locations must be explicitly named; the names effectively change if the file is moved to another machine, and files are accessed very differently if they are stored locally or remotely. Such services do not even allow all normal file operations to be performed on a remote file. At the other extreme, some distributed operating systems tried to conceal the actual location of the file whenever possible. The most successful distributed file systems (NFS and the Windows file-sharing services) provide various intermediate points of transparency. NFS does not make file locations explicit in its naming conventions, but a single file may need to be accessed by different names on different machines. The Windows file-sharing service makes file locations explicit, requiring that users access remote files by looking under icons representing remote machines. But both services support their file access interface for both remote and local files, permitting these files to be read, written, and executed. This degree of transparency has proven to be the minimal acceptable amount that users demand. The World Wide Web provides this same amount of transparency, for example, except that it does not allow remote writing.

Other file system issues must be handled by the distributed operating system. Typically, remote access is slower than local access (though in modern systems, accessing data stored in the main memory of a remote machine may be faster than accessing data stored on a local hard disk drive). Many distributed file systems seek to conceal this speed difference, typically by providing a local copy of the data, thereby avoiding expensive remote access. Caching of data blocks from remote files is common, often using the same facilities that cache data blocks fetched from local disks. The Andrew file system [2] uses a different approach, caching whole files for extended periods (including over reboots) at client sites.

Updates cause cache consistency problems. When a local file is written, invalidating locally cached copies of the written block is straightforward and quick. Invalidating remotely cached copies is slower and more difficult, since the local operating system must signal the remote machine's system to perform the actual invalidation. Even being able to do so requires that the local machine keep track of remotely cached copies. If the local machine does keep track of these copies, the remote machine might need to signal the local machine if the cached copy is discarded. Handling all cases of partial and transient failures complicates the problem. The Andrew file system is one system that has successfully handled these problems, aided in part by its use of reliable server machines that store the true permanent copy of all files. If writes are rare, ignoring the consistency problem and accepting occasional reads of stale cached blocks may be a better solution than trying to maintain perfect consistency of the cached copies. The Sun Microsystems implementation of NFS improves on this approach at a reasonable cost by having caching clients periodically check to see if their cached copy is still fresh. This solution still allows use of stale data, but can be tuned to trade-off freshness versus cost. Not caching at all is another solution, but one with serious performance implications.

When file copies are to be permanently stored at a location, they switch from being cached to being replicated. Replicating files has several advantages, including superior fault tolerance, high performance for all file operations, and good support for disconnected operations, which is especially valuable for mobile computing. Replicating files has a significant cost, however. Beyond the mere storage costs of keeping multiple permanent copies of the same bits, maintaining consistency among replicated files is difficult. Generally, replicated file systems must trade the consistency of the replicas against file system performance. Higher consistency is possible using conservative replication methods, which ensure that no inconsistent updates to files are possible. Conservative methods often have high costs and limit file availability, however. Optimistic replication reverses these advantages and disadvantages, offering lower cost and higher availability at the risk of permitting inconsistencies, which must then be handled by some method—typically a reconciliation process that tries to merge the results of conflicting updates. Sample replicated file systems include Coda (which uses optimistic client/server replication) [3] and Ficus (which uses optimistic peer replication) [4]. The systems in wide commercial use tend not to provide full functionality replicated file systems because of their cost and complexity.

File replication is widely used in an informal fashion. For example, programs that automatically distribute new versions of executables to all machines in an office are performing a limited form of file

replication. Mechanisms like the Microsoft Briefcase offer limited forms of file replication for special cases, such as sharing files between a desktop and a portable computer, where typically only one machine is active at any given moment. By leveraging the special circumstances of such uses, many of the harder problems of file replication can be avoided. Such adoption of simple limited expedients for common circumstances is characteristic of much real-world use of distributed operating systems technology.

File migration is another alternative available to distributed file systems to avoid performance problems. Files move to the sites that use them. This paradigm requires a high degree of transparency, since all processes in the system must be able to access all files, local or remote, under the same names. Otherwise, applications might stop working when the file migrates from site to site. Further, such alternatives face difficulties in preventing popular files from moving constantly. This paradigm of remote data access has been more widely used and studied in the field of distributed shared memory.

### 19.3.2 Interprocess Communications Services

Many distributed operating systems are designed to support distributed computations, where cooperating processes run on different nodes. Allowing the processes to cooperate implies that they can exchange information, which in turn implies a need for interprocess communications (IPC) services.

Single-machine systems have the same requirement, so a variety of IPC services exist for such environments, including remote procedure calls, messages, and shared memory. Generally, the implementation of these services in single-machine environments leverages shared hardware. For example, all three of these services are provided in most single-machine operating systems by writing information into main memory that can be read by all communicating processes, or by copying information from the memory of one process into the memory of another, or into buffers provided by the operating system. The shared hardware and the synchronization provided by the shared processor simplify many issues in making these IPC services work.

In a normal distributed system, the participating machines communicate only by passing messages across a network. Such networks typically do not offer the guarantees available from the shared hardware of a single machine. Further, the participating machines are typically under the control of their local processor. No single processor controls the complete collection. These characteristics require different implementations of familiar IPC services.

Message-passing IPC shares many characteristics of the single-machine case. Bytes are gathered into an explicit message. Instead of merely being transferred from one buffer to another on the same machine, the message goes over the network. Failure of the network or the receiver introduces new problems. The issue of how long to block the sender is also different, since the time required to confirm delivery of the message in the remote case can be much longer than in the local case. Issues of message addressing must also be considered. In a single-machine system, all addressable processes are locally known. In a distributed system, some facility must be provided to allow the local process to discover the addressable names of remote processes and to send messages to those names.

Remote procedure call (RPC) faces similar challenges. In a single machine, remote procedures are not that remote. Although they may have a different address space, the local operating system has access to all necessary facilities of both the calling and called processes. In distributed systems, the caller is on one machine and the called process on another. Further, the actual transfer of data must take place via messages crossing the network. One implication is that call-by-reference must be translated to call-by-return-value. Other complexities exist, including some similar to those for message passing. Another issue is handling partial failures. Either the caller or the called process can fail independently of the other, requiring the operating system on the surviving machine to recover.

Shared memory is the hardest common IPC mechanism to provide in distributed operating systems because it relies most heavily on hardware characteristics not present in the distributed environment. Early distributed operating systems did not provide shared memory across the network; however, as LANs became more capable, researchers tackled the difficult problems of providing the semantics of shared memory across the network. This problem spawned much research, which will not be covered in

detail here. A slightly closer look at the concept of distributed shared memory will reveal why this research was necessary.

As before, the distributed system can only communicate via messages. Yet the distributed operating system must provide the illusion that two processes on different machines are sharing a single piece of physical memory. The basic approach is to give each process access to a local copy of the memory, and then have the operating systems work behind the scenes to provide a consistent view between the processes. Another approach is to migrate the memory segments between machines, as needed. This approach can run into difficulties if processes frequently access the same memory locations. Also, because the overheads of handling shared memory at the word level are too extreme, distributed shared memory systems must aggregate words into shared blocks. If the aggregation is too large, false sharing occurs, where one process accesses the first part of a block while another process accesses the second part. Because the two parts are aggregated into a single block, the block must migrate back and forth, despite the processes actually addressing totally different words of memory.

Alternately, memory segments can be replicated. Doing so leads to problems when writes occur. Either the other copies of the segment must be updated (before they are accessed again) or they must be invalidated. Either approach requires much bookkeeping and incurs overheads when writes occur. False sharing effects can also play a role here, since writing to the first word of a block tends to invalidate or cause updates to the entire block.

Much research has been performed on distributed shared memory. Although distributed shared memory has been demonstrated to be feasible, its performance, complexity, and limitations have prevented it from becoming popular. Few systems today provide this facility. Research continues on distributed shared memory, but not as widely as in the past.

### 19.3.3 Naming Services

Names play an important role in operating systems. Many operating systems support several distinct name spaces for different purposes. For example, one name space might describe the file system, while another describes the processes running on a machine, and another name space describes the users permitted to work on the machine. One legacy of UNIX systems is that the file name space is used aggressively to provide name spaces for things that are not classically files, such as devices and interprocess communication services. (One distributed operating system, Plan 9, relies on this abstraction for all its naming needs [5].)

Distributed operating systems have similar naming needs, but now some of the entities that must be named are not located on the local machine. As before, these entities are of various types. The distributed nature of the system leads to different problems, however. For example, in a single machine, one directory can contain the names of all the entities in the system. Scaling and organizational concerns usually lead to breaking the single directory into hierarchical components, but there are relatively few difficulties with maintaining a single name space that describes the name-to-resource mappings of anything currently available in the system.

In a distributed system, independently operating nodes can create, destroy, and change names rapidly. These operations are local, so they are likely to appear instantly in the local name space. But how do the other machines in the system become aware of namespace changes?

One approach is to build a single global namespace for the entire distributed system. The Locus Operating System took this approach, for example [6]. It extended the standard UNIX file system naming convention across multiple machines. By providing a single hierarchical name space, the naming changes made by one machine were available to all machines. An advantage of this approach is high transparency. Files (and other nameable resources) had the same names on all machines, which provided an easier model for users. The difficulties with the Locus approach are that it scales poorly and only works well when all machines tend to be connected most of the time. Maintaining a global name space on multiple machines is very hard. Further, if replication is being used, name space changes in one replica can sometimes conflict with name space changes in other replicas.

The Andrew file system [2] overcame some of these problems by storing all files on reliable servers. Whole file caching was used to provide fast access on machines that interacted directly with users. The Andrew file system has been operated at high scale, but usually in circumstances where one collocated set of servers can access all clients via a high-speed reliable network.

Windows file sharing and NFS provide a more limited form of global name space. In the Windows file-sharing service, each machine has complete autonomy over its own name space, and exports that name space to remote machines explicitly, under its own machine name. NFS allows portions of one machine's name space to be spliced into the name spaces of other machines at fairly arbitrary points. Neither service is as transparent as a true global name space, but many control problems are avoided. As the World Wide Web has demonstrated, such a name space can scale well; however, the World Wide Web's name space also demonstrates some problems with the approach, such as poor results when a resource changes its location, since such a change implies a name change.

### 19.3.4 Recovery, Reliability, and Fault Tolerance Services

Because distributed operating systems are more prone to partial failure, some such systems provide special facilities for recovery. The system itself must have internal mechanisms (typically hidden from normal users) that handle failure problems. These facilities ensure that services like the file system and name spaces continue to exhibit reasonable behavior even in the face of failures. The two-phase commit protocol is commonly used to provide the transactional services such facilities require. The system might also provide process checkpointing facilities, services that allow cooperative processes on different machines to deal with failure of some of their components, or the ability to request replication or backup versions of important processes or data.

Arguably, such services are best provided transparently. Typical users and programmers are not experts in the complexities of distributed computations and failure handling, so few of them can make effective use of any such tools that the system provides. On the other hand, transparent recovery and reliability services are hard to provide, and nontransparent, but powerful, recovery services may be better than weaker transparent ones.

### 19.3.5 Process Migration

Some distributed operating system designers have foreseen value in permitting running processes to be migrated to other nodes, for load balancing, to achieve better performance for high-priority processes, to move processes closer to critical resources (typically files), or to provide improved reliability. Migrating a process at any arbitrary stage in its operations is difficult. The Locus Operating System provided such a facility, but handling all complex cases is tricky. A more common approach is to only migrate processes that are in safe states where the more complex situations cannot arise. Typically, this means they are temporarily quiescent until the migration completes. Not providing process migration at all is even more common. Process migration has not been a popular capability in the systems that do provide it.

### 19.3.6 Security Services

Single-machine operating systems provide some degree of security by relying on the characteristics of the hardware they run, and by leveraging the fact that the operating system trusts itself. Access control mechanisms for files, separation of data belonging to different processes, and authentication of users to the system work on these assumptions. In a distributed operating system, communications often go over insecure shared networks, and the remote operating systems might not be as fully trusted as the local system. The security problems are thus harder to solve, and distributed operating systems sometimes provide facilities to handle the problems.

The use of an insecure network is typically handled by either authenticating or encrypting network traffic. A properly designed cryptographic system can usually make it difficult for outsiders

to improperly inject or alter traffic, or to read secret information in transit. Such a cryptographic approach does not solve all problems, since one system must still rely on a remote system to enforce security restrictions just as the local system would. For example, if a sensitive file is stored at node A, when node B requests access to the file, node A can check that the request was made by a user with the right to view the file; however, if in response node A provides blocks of the file to the proper user on node B, node A must trust that node B will not maliciously or accidentally also provide the blocks to improper users. Node B has concerns, as well, because it cannot determine if node A has properly applied access control to the file. If node A has not done so, node B might provide its user with data that should be inaccessible. These concerns make it relatively difficult to set up a distributed operating system in environments where all participant systems do not completely trust one another.

Assuming that the nodes are all trustworthy to the extent that they will properly handle data that they can legitimately access, the distributed system must still authenticate the participants' requests. Otherwise, a node might tag requests to remote nodes from user X with the identity of user Y, allowing X to access data improperly. The remote node must independently verify that the request really came from user Y. Many cryptographically based mechanisms can provide such authentication. One option is the Kerberos system, which allows machines in a distributed environment to authenticate identities and provide controlled access to services [7]. Security designers are generally happiest with heavily tested and used mechanisms, because they are less likely to have undiscovered security bugs, so Kerberos' long history and the amount of scrutiny applied to it make it popular.

## 19.4 Sample Distributed Operating Systems

### 19.4.1 Locus

The Locus Operating System was an early ambitious attempt to build a distributed operating system that provided all users with a single system image [6]. It was developed at UCLA and the Locus Computing Corporation throughout the 1980s and into the 1990s. Locus was intended to be UNIX-compatible, both in terms of the operating system interface provided and the experience of users. Ideally, a Locus user would be given the illusion of a single large UNIX system vastly more powerful than any single-machine system could be. In actuality, the distributed operating system would run on each component node of the system. The nodes worked together to maintain the single image.

The Locus system achieved some success, but ran into several problems that prevented it from becoming popular. The system demonstrated the value and feasibility of providing high transparency in a distributed operating system, and pioneered concepts such as file replication. But the challenges of providing a true single image were immense. Particularly, handling all of the difficult uncommon cases properly required much complexity. Further, one fundamental mechanism in achieving the single system image was to reach agreement on the set of participating nodes, a task that proved difficult and expensive. The final lesson from the Locus project was that, although transparency was valuable and attractive, too much concentration on providing complete transparency in all circumstances could be counterproductive.

### 19.4.2 Amoeba

Amoeba provides service to a large community of users working at low-powered workstation systems. Amoeba maintains a pool of servers capable of working interchangeably with any of the workstations, as well as some specialized servers [8]. When a user logs in to an Amoeba workstation, he is implicitly logging onto the entire distributed system. The Amoeba system software assigns user tasks to one or more machines in the server pool, handling all issues of communications and synchronization. Because any task can potentially run on any server in the pool, Amoeba must provide a high degree of transparency. Persistent data is typically stored remotely from both the workstation currently occupied by the user and the machines in the server pool working on the request, which also implies a need for high transparency.

Amoeba provides RPC and reliable multicast for interprocess communications. It handles issues of network security by using randomly assigned ports to conceal communications and by requiring cryptographic capabilities to access resources.

Amoeba provides a distributed file system with replication capabilities. The Amoeba file system only permits creation of files, not their alteration. Instead of altering an existing file, a new file is created with the altered contents. This choice simplifies many replication issues, but requires users to adopt a different model of file behavior than is typical.

Amoeba was used for production purposes in several environments, and is available from Vrije Universiteit, where it was developed.

The design philosophy behind Amoeba and many other distributed operating systems is to support operations at one large, well-connected organization. Designers also assume that it is more economical to provide low-powered machines as a front end and perform the system's serious work on pools of servers. When a cheap workstation can provide all the computational and storage resources required by a particular user, there is less advantage in this approach; however, it still has some advantages because of system simplifications inherent in localizing important operations in well-connected, well-maintained servers. Also, this model is not well suited for integrating portable computers, because these machines can be disconnected from the network and are expected to continue providing service. Issues of local autonomy that can be tricky for systems like Locus are often less problematic for systems like Amoeba, since there is no concept of permanent ownership of a particular machine by an individual user.

### 19.4.3 Plan 9

Plan 9 is one of the more recent attempts to build a new distributed operating system [5]. Plan 9's approach to building a distributed operating system shares some similarities to Amoeba's. The system is designed primarily to support a single large organization, using a pool of CPU servers, file servers, and many terminal servers. All machines are connected by a high-speed LAN. All resources in Plan 9 are represented in a name space that resembles a UNIX file system. A user at a terminal requests resources by mounting name space components representing those resources into his own name space. One standard protocol handles access to all resources, be they files, devices, or interprocess communications facilities. Plan 9 is still in use and being studied, with its most recent release being in 2002, but has not achieved widespread popularity.

## 19.5 Recent Research in Distributed Operating Systems

Although distributed operating systems have not become ubiquitous, their goals continue to be tempting, and the success and importance of some distributed operating system components suggest that further research and development is worthwhile. Further, changes in the use of computers, such as mobile computing and ubiquitous computing, demand new research to handle the problems of distribution encountered in these new environments.

Few attempts were made in the past decade to produce revolutionary new general-purpose distributed operating systems akin to Locus, Amoeba, and Plan 9. The focus has been on producing better distributed services and designing distributed operating systems for important special uses.

### 19.5.1 Peer Computing

In the last decade, a method of harnessing the power of multiple machines called peer computing has become popular. While the general field of peer computing is broad, its most common applications are to allow users to share data stored on a very large number of computers among themselves easily and efficiently. Typically, each user in a peer system stores some data, which is made available to others. In some versions of peer computing, the users' machines might also help find data that is not even stored on their machines.

Peer computing is commonly spoken of in two styles, structured and unstructured. Modern peer systems of both styles have significant structure, but those called structured are usually built around precisely defined distributed data structures and algorithms, whereas those called unstructured are more loosely organized and less predictable in their connectivity and behavior.

Structured peer computing is commonly built around a variant of a distributed hash table. Data available in the system is stored at particular places that are chosen from among the participating nodes by performing some kind of hash on the data's name or other attributes. Lookup is performed by hashing the same information, then going to the participating site indicated by the hash result to obtain the content. Since these systems assume that there are large numbers of participants, and that the system can never count on a particular participant to be available at any given time, sophisticated methods are used to optimize the amount of storage required for each participating node to perform lookups and the number of nodes that must be visited to find a copy of the desired data. Different structured peer systems vary significantly in issues of implementation and performance under differing circumstances. Much research has been performed on structured peer systems in recent years. Two of the most popular versions are Chord [9] and Pastry [10]. Practical use of such systems to solve real problems is still in its infancy, however.

Unstructured peer systems present the mirror image of the state of structured file systems. They are very widely used in the real world, but organized research into their proper design and behavior is relatively young, though some work on their performance has been done in recent years [11]. Unstructured peer systems are commonly used to share data among ordinary users. In many, though not all, cases, the data is media files, such as songs or videos. Much of the data that is shared is protected by copyright laws in some nations, and copyright holders often do not give their permission to share their data over these systems. This legal issue has had impacts on the design of these systems. For example, the first popular peer file-sharing service, Napster, used a central directory for pointers to content, with the content itself stored on participating users' systems. Legal challenges in the United States led the company running the central directory to shut it down, which in turn caused the death of the service. Subsequent peer systems have often been designed to avoid any single point that can similarly be shut down by legal challenges.

Designers of unstructured peer systems have faced difficult challenges in providing good functionality in the face of high scale, while not centralizing any features of the system. Many of the solutions that have been adopted seem to be independent rediscoveries of principles of distributed systems design from earlier decades. Some prominent peer file-sharing systems include Gnutella, Kazaa, e-Donkey, and BitTorrent, an interesting exercise in harnessing the combined networking power of multiple machines to speed up data transfer. The popularity of some of these services is extremely broad, with estimates of the total fraction of Internet bandwidth being used to support them ranging as high as 60% [12].

While similar in some ways to earlier distributed file systems, most peer data systems also have significant differences. They are usually not intended as a true file system. They generally do not allow writing of the data in question. They typically are invoked from a special program or command line, rather than from a general file system interface. In many cases, they do not guarantee that they will find the data requested, even if the data is available somewhere in the overall peer network. This restriction is suitable for certain important applications, but does not match the expected behavior of a single machine's file system, and thus is not quite consistent with the distributed operating system designer's dream of a true distributed file system.

### 19.5.2 Server Farms and Grid Computing

In the 1990s, much effort was directed toward building powerful server farms from commodity hardware, primarily to offer Internet services at lower cost than large server solutions. Also, if well designed, these systems could be easily upgraded, as service requirements grew by simply adding more commodity servers. Generally, such systems required some form of load balancing to assign incoming requests to servers in the pool, and often some method of sharing the vital data required to provide the

service. This method might be as simple as replicating static data on all servers in the pool, or as complicated as a distributed database sitting behind the pooled servers.

Such systems were generally built as pure servers designed for whatever particular purpose their owners had. However, as more scientific communities required large quantities of computing power to handle their problems, a more sophisticated method of providing distributed services over the Internet became popular. In essence, a community of users pooled their computing and communications resources, typically scattered all across the Internet, to provide sufficient power to handle large computing problems. Since each user in the community required the full power of the resources only some of the time, the community was able to timeshare its total resources, vastly increasing each user's compute power, while adding little or no cost to his computer facilities. This approach is commonly called grid computing [13].

Forerunners of grid computing include systems that harvested spare cycles on a local area network for use by local users and specialized systems that used spare cycles on home computers to apply to embarrassingly parallel problems, such as SETI@home and cracking cryptographic challenges. Grid computing required more complex distributed systems technology than these systems, because it needed to run truly cooperative processes across a wide range of hardware and network conditions. Issues of proper placement of data, scheduling tasks according to the power of particular pieces of hardware, handling some forms of hardware and software heterogeneity, and ensuring that network capabilities matched the requirements of the division of labor became important. While generally the paradigm assumed that no machines failed during the course of the computation, and that the user who had obtained the resources was able to keep them until finished, or at least for a pre-agreed period of time, some effort to handle partial failures and disconnections was also necessary.

Grid computing is widely used by many communities whose needs are well met by its strengths. A typical grid computing support system is not really a distributed operating system, per se, but certainly must address many of the same issues as distributed operating systems, such as synchronization, data sharing, partial failure, distributed scheduling, interprocess communications, local autonomy, and, in certain aspects, security. The Globus toolkit is one popular and representative approach for solving the problems facing grid computing [14].

### 19.5.3 Distributed Systems for Ubiquitous Computing

Many researchers predict that homes, offices, and other buildings of the future will contain large numbers of objects that have embedded processors and communications devices. For example, all appliances in a house might contain processing and communications capabilities. Also, humans may carry several computing and communications devices on their bodies, in much the same way that most people today wear a watch or carry a cellphone. The purposes and uses of machines in these environments are not yet clear, but they seem likely to differ from the way office workstations in a LAN or machines browsing the Web behave. Plausibly, they will require different sets of system services than today's distributed system services. Various researchers have started examining the requirements of these systems and designing distributed services for them.

One area of interest is naming in such systems. The publish/subscribe model of naming has been proposed and implemented in systems like the Jini service discovery system. In this model, devices on the network publish their capabilities, and other devices that hear the publications subscribe to the services they are interested in. Another proposed model of naming for this environment is intentional naming, where the system provides name resolution and request routing via a self-configured overlay network [15]. In such systems, users name objects by what they want to do with them, and the naming service takes responsibility for forwarding the request to some entity capable of fulfilling their need.

Ubiquitous computing systems require persistent storage, both to access standard persistent data (e.g., files) and to allow the system to keep track of the state required to give users a consistent view of the world. Mobility, limited communications links, security concerns, and varying capabilities of participant machines make providing persistent data harder. One approach is exemplified by OceanStore, which

postulates a utility-like model for providing persistent data handling in a ubiquitous environment [16]. OceanStore uses replication aggressively and relies on a combination of fast (but not always successful) search techniques and slow (but reliable) lookup algorithms. There are some obvious similarities between OceanStore and some of the peer networking techniques discussed earlier.

### 19.5.4 Botnets

For many years, unprincipled Internet users have tried to compromise and control the machines of other users. Their widespread success has led to almost an embarrassment of riches, with collections of hundreds, thousands, or, in some documented cases, tens of thousands of other peoples machines under their control [17]. Simultaneously, criminals have developed ways to make use of large numbers of compromised machines to make money illicitly, from sending vast amounts of spam to performing extortion via distributed denial of service attacks. These activities often require some degree of coordination among the army of compromised nodes, which are often called zombie nodes. In recent years, the criminals who control these zombie nodes have started to turn to distributed system techniques to perform such coordination. A collection of such zombie nodes controlled in this fashion is usually called a botnet.

Currently, the degree of sophistication used in the real world to control botnets is modest, but touches upon many of the issues important in design of distributed operating systems. There is usually a need to provide secure access to the nodes in the botnet, since other criminals will seize them for their own purposes, if they are not protected. New software must be distributed in an effective manner to all botnet nodes, which is a very primitive form of a distributed file system. They must be able to receive and perform commands from their masters, requiring some degree of synchronization and, perhaps, interprocess communications. Botnet nodes are, physically at least, under some degree of control by their true owners, who might remove the botnet code or simply take them off line, requiring some method of handling failure of zombies. When the size of the botnet gets above a few nodes, automated techniques to handle these problems become vital. When the size gets to the thousands, introducing a more structured system, typically hierarchical, is beneficial. Existing botnet control code uses primitive techniques of these kinds.

Botnets are in their infancy, and we should expect to see more sophisticated distributed system techniques applied to them as the needs and ambitions of the criminals controlling them become greater. An interesting distributed systems problem that might well be unique to botnets is that law enforcement and those attacked by them obviously would like to disable them. Relatively little research ever went into explicit attempts to make standard distributed operating systems fail, so this is fertile ground for new research.

## 19.6 Resources for Studying Distributed Operating Systems

Research continues in the field of distributed operating systems. Some of the work is commercial, but much of it is described in the research literature. The principal conferences where distributed operating system research appears most often are the biannual Symposium on Operating Systems (SOSP), the biannual Conference on Operating Systems Design and Implementation (OSDI, held in alternate years with SOSP), and the annual Usenix Technical Conference. Interesting research on more specialized areas often appears in other venues. For example, mobile computing distributed system research often appears in the MOBICOM conference, and distributed systems security research will often appear in security conferences. The annual Usenix Conference on File and Storage Technology (FAST) often has papers on distributed file system issues. A workshop called HotOS publishes early results on new areas of research in operating systems, including distributed operating systems.

The primary journals where distributed operating systems research appears are the *ACM Transactions on Computing Systems* and the *IEEE Transactions on Computers*. Again, research on specialized topics often appears in other journals, such as mobile computing research in mobile networks and applications (MONET).

## References


1. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K., System architecture directions for networked sensors, in *Proceedings of the ACM ASPLOS IX*, November 2000, pp. 93–104.
2. Howard, J. et al., Scale and performance in a distributed file system, *ACM Transactions on Computer Systems,* 6(1): 51–81, February 1988.
3. Satyanarayanan, M. et al., Coda: A highly available file system for a distributed workstation environment, *IEEE Transactions on Computers,* 39(4): 447–459, April 1990.
4. Guy, R. et al., Implementation of the ficus replicated file system, in *Proceedings of the 1990 Usenix Summer Technical Conference,* June 1990, pp. 63–71.
5. Pike, R. et al., The use of name spaces in Plan 9, *Operating Systems Review,* 27(2): 72–76, April 1993.
6. Popek, G. and Walker, B., *The Locus Distributed Operating System,* MIT Press, Cambridge, MA, 1985.
7. Steiner, J., Neuman, C., and Schiller, J., Kerberos: An authentication service for open network systems, in *Proceedings of the Usenix Winter Conference,* February 9–12 1988, pp. 191–202.
8. Tannenbaum, A. et al., Experiences with the Amoeba distributed operating system, *Communications of the ACM*, 33(12): 46–63, 1990.
9. Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishna, H., Chord: A scalable peer-to-peer lookup service for internet applications, ACM SIGCOMM 01, August 2001, pp. 149–160.
10. Rowstran, A. and Druschel, P., Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001, pp. 329–350.
11. Gummadi, K., Dunn, R., Saroiu, S., Gribble, S., Levy, H., and Zahorjan, J., Measurement, modeling, and analysis of a peer-to-peer file-sharing workload, *Symposium on Operating Systems Principles*, pp. 314–329, 2003.
12. Parker, A. P2P in 2005, http://www.cachelogic.com/research/2005_slide01.php, 2004.
13. Foster, I., Kesselman, C., and Tuecke, S., The anatomy of the grid: Enabling scalable virtual organizations, *International Journal of Supercomputer Applications*, 15(3): 200–222, 2001.
14. Foster, I., *The globus toolkit version 4: Software for service-oriented systems*, IFIP International Conference on Network and Parallel Computing, Springer-Verlag, LNCS 3779, pp. 2–13, 2005.
15. Adjie-Winoto, W. et al., The design and implementation of an intentional naming system, in *Proceedings of the 17th Symposium on Operating System Principles*, December 1999, pp. 186–201.
16. Kubiatowicz, J. et al., OceanStore: An architecture for global-scale persistent storage, in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000, pp. 190–201.
17. The Honeynet Project and Research Alliance, Know your enemy: Tracking botnets, http://www.honeynet.org/papers/bots/, March 2005.

# VII

# New Directions in Computing

# 20

# SPS: A Strategically Programmable System




M. Sarrafzadeh
E. Bozorgzadeh
R. Kastner
S.O. Memik
*University of California*


## 20.1 Introduction

Programmability and reconfigurability are considered to be a key ingredient for future silicon platforms [1]. An increase in the complexity of integrated circuits and a shorter time-to-market requirement result in a need to develop hardware platforms shared across multiple applications. In the next generation of electronic systems, it is expected that the conventional embedded systems are unlikely to be sufficient to meet the timing, power, and cost of such systems. Diversity and increasing number of applications do not allow fully customized system design methodology for each application such as ASIC designs. One of the fundamental keys is integrating programmability and reconfiguration in the systems [1,2]. On the other hand, the current *general-purpose* fully programmable solutions cannot satisfy the future aggressive timing and power constraints. Therefore, a new design methodology has to be developed to combine reconfiguration into system design for future applications. One of the techniques to handle the increased complexity in integrated circuits is programmable system-on-a-chip (SoC) design methodology [1]. In this style, there is a combination of IP cores, programmable logic core, and memory blocks on a chip.

System design can be viewed in a variety of levels of granularity from architecture level to logic/interconnection level. Configuration can be applied in different hierarchy levels of a design [1]. For instance, programmability in logic/interconnect level of system is realized via a programmable module such as FPGA chip. Reconfigurable devices provided the necessary flexibility in such systems. FPGAs are mostly the reconfiguration cores. An FPGA is an array of logic blocks placed in an infrastructure of interconnections, which can be configured in logic functionality of logic blocks, interconnection between logic blocks, and input/output (I/O) interface (see Fig. 20.1). SRAM-based

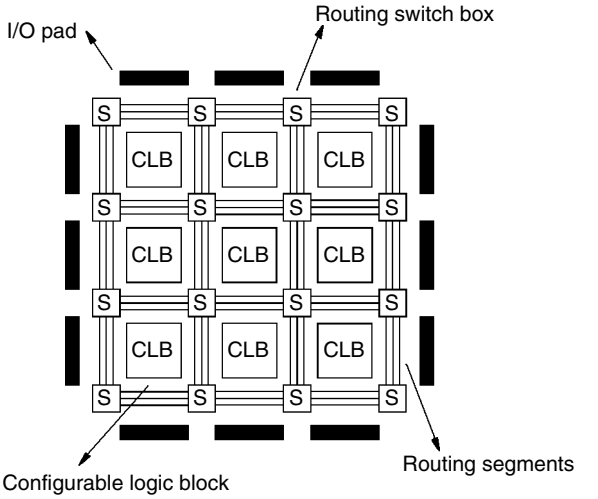



**FIGURE 20.1** An array of logic blocks.

FPGAs allows reconfiguration of the device via string of bits loaded from external source once or several times. An FPGA is programmable at hardware level. Many high-performance operations, such as computationally intensive signal processing functions can be implemented very efficiently using FPGAs.

Xilinx™ first introduced FPGAs in mid-1980s. In the past, considerable research effort was made for developing programmable architectures in the past. Also, numerous commercial programmable devices were introduced; however, not many works have been published on how those architectures were designed. The latest FPGA devices are provided by companies such as Xilinx™, Altera™, Actel™, and Lucent™. Industrial designers are increasingly capturing their designs using hardware description languages such as VHDL and Verilog. There are tools developed for FPGAs, which synthesize designs in VHDL format or other description format. Current synthesis tools for FPGAs are provided by companies Synopsys™, Synplicity™, and Leonado Spectrum™. Physical design tools perform placement and routing for FPGAs. Xilinx™ and Altera™ provide place/route tools for their own FPGA devices.

Today's high-volume applications require a fabric with higher complexity and better performance than FPGAs. Also, shorter development cost and more flexible reconfiguration are required. Several contributions have been made in FPGA devices toward this direction. Capacity of FPGAs has been increased. High-density FPGAs are available in the market offering competitive solutions to ASICs and programmable systems such as DSPs. Hierarchical features have been added into logic and routing architecture of FPGAs. The new generation of FPGAs have a trend towards embedding coarser grain units. Most applications require a large amount of storage. Architectural support for implementation of memory is crucial. Some FPGA devices have embedded memory (RAM, ROM). In addition, implementing general logic in these embedded arrays of memory blocks is viable. In order to support high repetitive and data intensive computation on FPGAs more efficiently, arithmetic resources have been developed. Examples of such enhancement are cascade chain, multipliers, and dedicated adders, etc. Fine-grain FPGA architectures are shifting towards new architectures where memory blocks, hard IPs, and even CPUs are being integrated into FPGAs. In these designs the traditional FPGA is not a co-processor, instead a reconfigurable fabric is embracing all the mentioned components and enabling a much tighter integration among them. Today, FPGA CAD tools provide integration of macro blocks into designs. Macro blocks are optimized for area, delay, or power consumption. In addition, placement of such macro blocks can be predefined in CAD tools such as CoreGEN@ integrated with Xilinx™ design implementation tool. MemGen@ and LogiBlox@ in Xilinx™ tool enable the implementation of embedded memory blocks. Hence, tool vendors are moving to higher-level optimization. There is better integration between synthesis and physical design tool.

Flexibility in reconfigurable systems comes at the expense of the reconfiguration time. The amount of time required to set the function to be implemented on the reconfigurable logic is the configuration time, which can become a serious bottleneck especially in systems where run-time reconfiguration is performed [3,4].

We have introduced a new architecture for a system that uses reconfigurable logic, which is referred as strategically programmable system (SPS) [5]. The basic building blocks of our architecture are parameterized functional blocks that are pre-placed within a fully reconfigurable fabric. They are called versatile parameterizable blocks (VPBs). When implementing an application, operations that can be mapped onto these fixed blocks will be performed by the VPBs; computational blocks that will be instantiated on the fully reconfigurable portion of the chip will perform the remaining operations. Properties of VPBs will be discussed in more detail in later sections. The motivation is to generate reconfigurable architectures that target a set of applications. Such architectures would contain VPBs to specially suit the needs of the particular family of applications. Yet the adaptable nature of our programmable device should not be severely restricted. The SPS will remain flexible enough to implement a very broad range of applications, thanks to the reconfigurable resources. These powerful features help our architecture maintain its tight relation to its predecessors, traditional FPGAs. At the same time the SPS is forming one of the first efforts in the direction of context-specific programmable devices.

Because the VPBs are custom made and fixed on the chip, they do not require configuration, hence there are considerably less switches to program as compared to the implementation of the same design on a traditional FPGA. More important, an instance of our SPS architecture is generated such that for a given set of applications the suitably selected fixed blocks provide the best performance.

In the proceeding sections, we introduce the basic concepts of our architecture and the notion of generating an instance of the SPS architecture for a given set of applications. We present a framework that provides tools for generating SPS instances and implementing applications once a fixed SPS architecture is given. In the following section, we present related work in the field of reconfigurable architecture. Examples of versatile programmable blocks are presented in Sections 20.3 through 20.5 as well as details of our architecture. We complement our architecture with tools that perform the mapping of applications onto the architecture and the actual implementation and tuning for an application on our platform. These two major tasks will be discussed in Section 20.4. In Section 20.6, we present our preliminary results.

## 20.2 Related Work

With the current trend towards hybrid programmable architectures, new systems with embedded reconfigurable cores are also being developed. Among these architectures the basic distinction is due to the level of granularity of the reconfigurable logic.

Commercial FPGAs from several vendors such as Xilinx™, Altera™, and Actel™ are available in the market. Traditional FPGA chips like Xilinx™ 4000 series, or Altera™ Flex family all contain some form of an array of programmable logic blocks. Those blocks usually are not very complex and contain a few LUTs and a small amount of storage elements. They are designed for general-purpose use. Since they only contain fine-grain reconfigurable logic, for a new application to be implemented the whole chip goes through a configuration phase. Although newer devices such as Xilinx™ Virtex FPGA allow partial reconfiguration of selected rows or columns, this is still a critical issue.

Hybrid systems also contain reconfigurable cores as coprocessors. The Garp architecture developed at Berkeley combines a MIPS-II processor with a fine-grained FPGA coprocessor on the same die [6]. Unlike the Garp architecture the main load of hardware implementation lies on the coarse grain parameterized blocks in SPS architecture.

Chimaera [7] is a single chip integration of reconfigurable logic with a host processor. The reconfigurable coprocessor is responsible for performing the reconfigurable functional unit (RFU) instructions. An RFU instruction is any instruction from the program running on the host processor that is

performed by the reconfigurable coprocessor. The Chimaera architecture is for a very specific class of data path applications and still requires a large amount of reconfiguration time.

Another reconfigurable architecture with fine granularity is the dynamically programmable gate array (DPGA) [8]. Although the logic structure is just like existing FPGAs, DPGAs differ from traditional FPGAs by providing on-chip memory for multiple array configurations. The on-chip cache exploits high, local on-chip bandwidth to perform quick reconfiguration.

In addition, several systems with coarse-grain granularity exist, such as RaPiD [9], Raw [10], and Pleiades [11]. RaPiD is a configurable architecture that allows the user to construct custom application-specific architectures in a run-time configurable way. The system is a linear array of uncommitted functional units, which contain datapath registers, three ALUs, an integer multiplier, and some local memory. The RaPiD architecture targets applications that can be mapped to deep pipelines formed from the repeated functional units.

The Reconfigurable Architecture Workstation (Raw) is a set of replicated tiles, where each tile contains a simple RISC-like processor, small amount of bit-level configurable logic, and some memory for instructions and data.

The CS2112 reconfigurable communications processor (RCP) from Chameleon Systems, Inc.™ contains reconfigurable fabric organized in slices, each of which can be independently reconfigured. The CS2112 includes four slices consisting of three tiles. Each tile comprises seven 32-bit datapath units, two $16 \times 24$-bit single-cycle multipliers, four local store memories, and a control logic unit. The RCP uses a background configuration plane to perform quick reconfiguration. This reconfigurable fabric is combined with a 32-bit embedded processor subsystem.

The Pleiades architecture is a processor approach that combines an on-chip microprocessor with an array of heterogeneous programmable computational units of different granularities, connected by a reconfigurable interconnect network. The programmable units are MACs, ALUs, and an embedded FPGA.

Xilinx™ has recently introduced the Virtex-II® devices from the new Xilinx Platform FPGAs. The Virtex-II architecture includes new features such as up to 192 dedicated high-speed multipliers. Designers can use Virtex-II® devices to implement critical DSP elements of emerging broadband systems. This is somewhat a similar effort in the same direction that we are heading. The Virtex-II device is providing the dedicated high-performance multipliers for DSP applications like the VPBs on the SPS, which are intended to improve performance for a set of applications. SPS differs from a Virtex-II device in the following:

- The architecture can contain blocks of various complexities. Depending on the requirements of the applications fixed blocks can be as complex as an FFT block or as simple as an adder or multiplier. Examples of VPBs will be provided in the next section.
- The generation of an SPS instance is automated. Given a set of target applications an architecture generation tool determines the number and types of VPBs to be placed on the chip. Although the Virtex-II device is still general purpose, an instance of an SPS will be more context-defined according to a given set of applications.

## 20.3 Strategically Programmable System

Recently, reconfigurable fabric was integrated into SoCs forming hybrid (re)configurable systems. *Hybrid (re)configurable systems* contain some kind of computational unit, e.g., ALUs, intellectual property units (IPs) or even traditional general-purpose processors, embedded into a reconfigurable fabric (see Fig. 20.2).

One type of hybrid reconfigurable architecture embeds reconfigurable cores as a coprocessor to a general-purpose microprocessor, e.g., Garp [6] and Chimaera [7]. Another direction of new architectures considers integration of highly optimized hard cores and hardwired blocks with reconfigurable fabric. The main goal here is to utilize the optimized blocks to improve the system performance. Such programmable devices are targeted for a specific *context*—a class of similar applications, such as

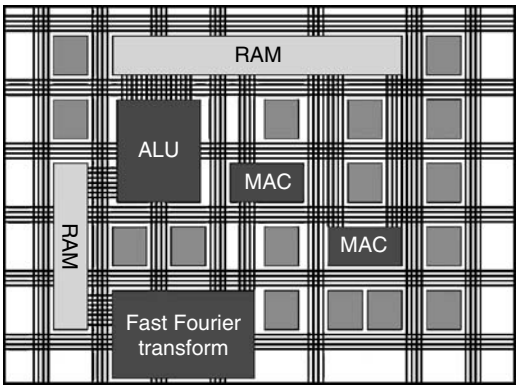



**FIGURE 20.2** A hybrid (re) configurable system.

DSP, data communications (Xilinx Platform Series), or networking (Lucent's ORCA®). The embedded fixed blocks are tailored for the critical operations common to the application class. In essence, the programmable logic is supported with the high-density high-performance cores. The cores can be applied at various levels, such as the functional block level, e.g., fast. Fourier transform (FFT) units, or at the level of basic arithmetic operations (multipliers).

Presently, a context-specific architecture is painstakingly developed by hand. The SPS explores an automated framework, where a systematic method generates context-specific programmable architectures.

The basic building blocks of the SPS architecture are parameterized functional blocks called VPBs. They are preplaced within a fully reconfigurable fabric. When implementing an application, operations can be performed on the VPBs or mapped onto the fully reconfigurable portion of the chip. An instance of our SPS architecture is generated for a given set of applications (specified by C or Fortran code). The functionality of the VPBs is tailored towards implementing those applications efficiently. The VPBs are customized and fixed on the chip; they do not require configuration, hence there are considerably less configuration bits to program as compared to the implementation of the same design on a traditional FPGA.

The motivation is to automate the process of developing hybrid reconfigurable architectures that target a set of applications. These architectures would contain VPBs that specially suit the needs of the particular family of applications. Yet, the adaptable nature of our architecture should not be severely restricted. The SPS remains flexible enough to implement a very broad range of applications due to the reconfigurable resources. These powerful features help the architecture maintain its tight relation to its predecessors, traditional FPGAs. At the same time the SPS is forming one of the first efforts in the direction of context-specific programmable devices.

In general, two aspects are part of the SPS system. The first area involves generating a context-specific architecture given a set of target applications. Once there is a context-specific architecture, one must also be able to map any application to the architecture.

## 20.4 Overview of SPS

### 20.4.1 Versatile Parameterizable Blocks

The main components of SPS are the VPBs. The VPBs are embedded in a sea of fine-grain programmable logic blocks. Consider a lookup table (LUT) based logic blocks commonly referred to as combinatorial logic blocks (CLBs), though it is possible to envision other types of fine-grain logic blocks, e.g., PLA-based blocks.

Essentially, VPBs are hard-wired ASIC blocks that perform a complex function. Because the VPB is fixed resource, it requires little reconfiguration time when switching the functionality of the chip.* Therefore, SPS is not limited by large reconfiguration times like current FPGAs. But, the system must strike a balance between flexibility and reconfiguration time. The system should not consist mainly of VPBs, as it will not be able to handle a wide range of functionality.

*By functionally, we mean the application of the chip can change entirely, e.g., from image detection to image restoration, or part of the application can change, e.g., a different image detection algorithm.

There is a considerable range of functionality for the VPBs. It ranges from high-level intensive tasks such as FFT to a "simple" arithmetic task like addition or multiplication. Obviously, there is a large range of complexity between these two extremes. Because we are automating the architecture generation process, we wish to extract common functionality for the given context (set of applications). The functionality should be as complex as possible while still serving a purpose to the applications in the context. An extremely complex function that is used frequently in only one application of the context would be wasted space when another application is performed on that architecture.

The past decade has brought about extensive research as to the architecture of FPGAs. As we previously discussed, many researchers have spent copious amounts of time analyzing the size and components of the LUT and the routing architecture. Instead of developing a new FPGA architecture, the SPS leverages the abundant body of FPGA architecture knowledge for our system. Embedding the VPBs into the programmable logic is the most important task for the SPS architecture generation.

### 20.4.2 SPS Framework

In this section, the tools and algorithms that actually generate strategically programmable systems and perform the mapping of applications on the architecture are discussed. The architecture formation phase and the architecture configuration phase are the two major parts of the framework. The SPS framework is summarized in Fig. 20.3.

### 20.4.3 Architecture Formation

This task can be described as making the decision on the versatile programmable blocks to place on the SPS chip along with the placement of fine-grain reconfigurable portion and memory elements, given an application or class of applications. In this phase, SPS architecture is customized from scratch given

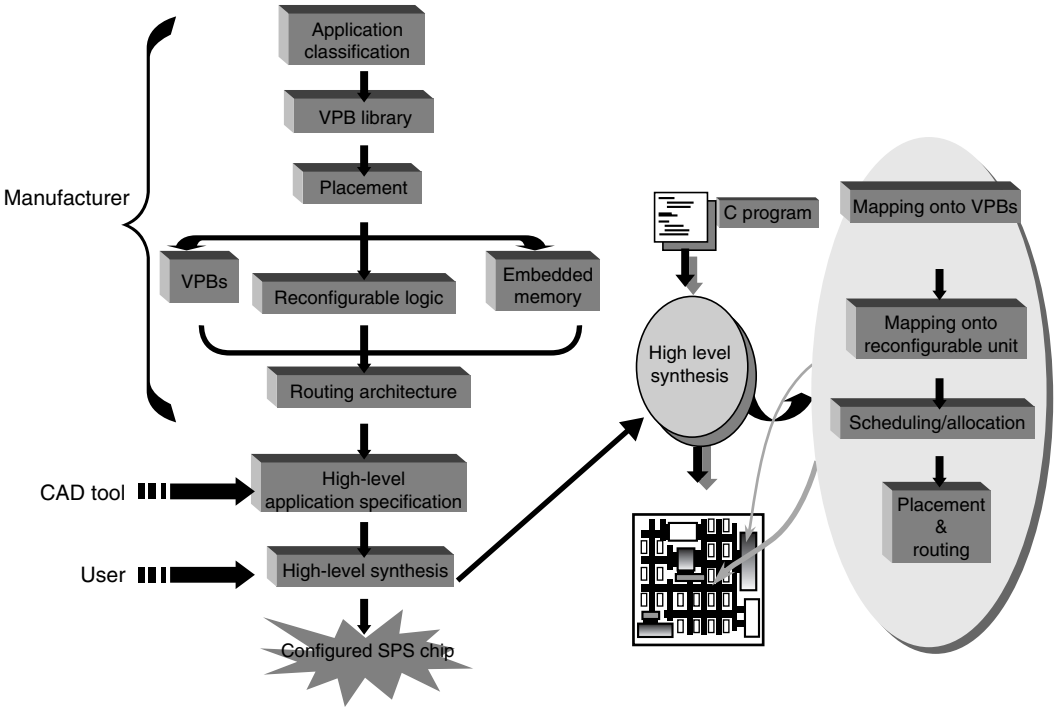



**FIGURE 20.3** Summary of SPS framework.

certain directives. This process requires a detailed description of the target application as an input to the formation process. A tool will analyze these directives and generate the resulting architecture. Again, the relative placement of these blocks on the SPS chip along with the memory blocks and the fully reconfigurable portions need to be done by the architecture formation tool.

Unlike a conventional fine-grain reconfigurable architecture, a uniform distribution of configurable logic blocks does not exist on the SPS. Hence, for an efficient use of the chip area as well as high performance, the placement of VPBs on the chip and the distribution of configurable logic block arrays and memory arrays among those are critical. The fact that the routing architecture supports such hybrid architecture is equally important and requires special consideration. If the routing architecture cannot provide sufficient routing resources between the VPBs and the configurable blocks, the hardware resources will be wasted. The type and number of routing tracks and switches need to be decided such that the resulting routing architecture can support this novel architecture most efficiently. The most important issues here is the routability of the architecture and the delay in the connections.

### 20.4.4 Fixed Architecture Configuration

Another case to be considered is mapping an application onto a given architecture. At this point we need a compiler tailored for our SPS architecture. This SPS compiler is responsible for three major tasks.

The compiler has to identify the operations, groups of operations, or even functions in the given description of the input algorithm that are going to be performed by the fixed blocks. These portions will be mapped onto the VPBs and the information regarding the setting of the parameters of the VPBs will be sent to the SPS chip.

Second, the SPS compiler has to decide how to use the fully reconfigurable portion of the SPS. Based on the information on the available resources on the chip and the architecture, mapping of suitable functions on the fine-grain reconfigurable logic will be performed. Combining these two tasks the compiler will generate the scheduling of selected operations on either type of logic.

Finally, memory and register allocation need to be done. An efficient utilization of the available RAM blocks on the SPS has to be realized.

## 20.5 Target Applications

The first set of applications is DSP applications. Repetitive arithmetic operations on a large amount of data can be very efficiently implemented using hardware. The primary examples focus on several image-processing applications. This soon will be extended to cover other types of applications. First, algorithms that have common properties and operations are grouped together. Such algorithms can use a common set of VPBs for their implementation. The algorithms and the classes to which they belong are summarized in Table 20.1.

Image-processing operations can be classified into three categories. Those that generate an output using the value of a single pixel (point processing), those that take a group of neighboring pixels as input (local processing), and those that generate their output based upon the pixel values of the whole image (global operations) [12]. Point and local processing operations are the most suitable for hardware implementation, because they are highly parallelizable. We have designed three blocks, each representing

**TABLE 20.1** Classification of Algorithms

| Algorithms | Operations | Class |
|---|---|---|
| Image restoration, mean computation, noise reduction, sharpening/smoothing filter | Weighted sum, addition, subtraction, multiplication | Filter operations |
| Image half-toning, edge detection | Comparison | Thresholding |
| Image darkening, image lightening | Addition, subtraction | Pixel modification |

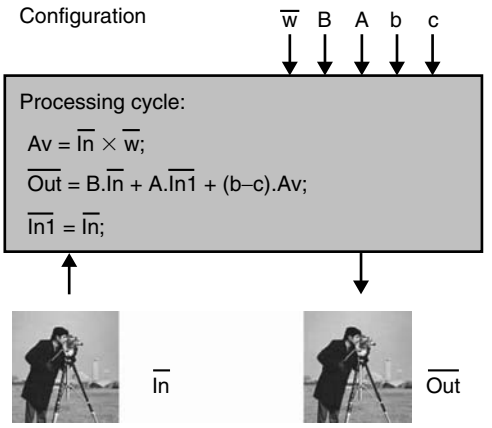



**FIGURE 20.4** Block diagram.

one algorithm class. The blocks that we are currently considering are described in the following. Later, a reference will be made to the implementation of these blocks on fully reconfigurable logic versus the parameterized block realization and present the potential reduction in the number of configuration bits.

### 20.5.1 Filter Operations Block

Many signal-processing algorithms are in essence filtering operations. Basically weighted sum of a collection of pixels indicated by the current position of the mask over the image is computed for each pixel.

The filter block is currently the most complex block we have designed. It is developed to cover an iterative image restoration algorithm, and several other filtering operations such as, mean computation, noise reduction, high pass sharpening, Laplace operator, and edge detection operators (e.g., Prewitt, Sobel). The block diagram is shown in Fig. 20.4.

The general form of the computation that this block performs is given by the following equations:

$$\begin{aligned}\text{weighted\_sum} &= \text{Sum}\{w_i \times \text{input\_pixel}_i\}\\ \text{output\_pixel\_value} &= B \times \text{input\_pixel\_value} + A \times \text{pixel\_value\_from\_prev\_iteration} - b \times \text{weighted\_sum}\\ &\quad + c \times \text{weighted\_sum}\end{aligned}$$

This block takes five parameters that define its operation. The mask coefficients array holds the values of the coefficients. The parameters *B, A*, and *c* all take the value zero for all the functions except the iterative image restoration algorithm.

### 20.5.2 Thresholding Block

The operators in this class produce results based on a single pixel's value. The computation is rather simple; it compares the value of the input pixel to a predetermined threshold value. The output pixel value is determined accordingly. The parameters of this block are the threshold value *T* and the algorithm selection input. For the image halftoning application, the threshold value *T* is set to be 127, where the pixel values range between 0 (black) and 255 (white). If the pixel value is above threshold, output is given as 255, otherwise it is 0. For the edge detection operation, the output is set equal to the input value if the pixel value is above threshold, and to 0 otherwise.

### 20.5.3 Pixel Modification Block

Pixel modification operations are point-processing operations. This block performs darkening, lightening, and negation of images. It takes two parameters, *J* and algorithm selection input ALG. For the darkening operation a positive value *J* is added to the input pixel value. Lightening is achieved by subtracting *J* from each input pixel. The negative of an image is achieved by subtracting the input pixel value from *J*, which takes the value 255 for this case.

## 20.6 Experiments

In this section, the author presents the experiments to estimate the potential gain in reconfiguration time that our SPS architecture will yield. This is an exploration of the reduction in reconfiguration bits that would follow as a result of providing preplaced computation blocks with our coarse grain architecture.

### 20.6.1 Application Profiling

First, a profiling has been done on the image processing benchmarks in order to gain insight into what type of components are used more frequently. Such a profile can give directives to the architecture formation tool and guide it to employ certain blocks. Looking at the numbers and types of components that were selected by our scheduling tool, we have obtained the component usage profile as shown in Fig. 20.5. The first version of the experimental SPS architecture will be created, given the directives from the profiling information. According to Fig. 20.5 the most popular component is the adder with an average of approximately 14 adders per benchmark. Some components such as the 8-bit multiply and bit-wise could not average over 1. These components can be eliminated and we will focus on the rest and decide how many of each of the remaining ones to use on the SPS. If the numbers are normalized according to the constant multiplier, for each constant multiplier there would be one comparator, one subtractor, two right shifters, two left shifters, and seven adders. In the next section, this information will be used to estimate the gain of fixing different numbers of components on the chip. The start point will be the relative usage values given by our analysis.

### 20.6.2 Reconfiguration Time

In the ideal case, the preplaced blocks should cover all the operations in our target applications such that we can fully exploit the benefits of the custom designed high-performance blocks and improved reconfiguration time. In reality it is not possible to create such an architecture that would support every operation that might be encountered in a wide variety of applications. Hence, in cases where provided blocks are not adequate, extra components are instantiated on the reconfigurable fabric. Here, we evaluate the gain that the VPBs would bring to the configuration process. Our scheduler uses the blocks that are made available and as many additional components as necessary for the best latency. As a result it produces an assignment of the operations to the hardware resources.

Compare the implementation of the same design on two architectures: a traditional fine-grain FPGA and a SPS. We are fixing certain blocks on the SPS assuming they are pre-placed blocks on the chip. These blocks will be the gain in reconfiguration time if they are used by the given application. The potential gain is modeled in configuration time by assuming that the number of configuration bits is proportional to the amount of logic to be implemented on the reconfigurable fabric. The higher the contribution of the preplaced blocks is to the total design, the more reduction in reconfiguration time is achieved.

Initially, an architecture is evaluated, which contains low-level blocks such as adders and subtractors. A set of functional blocks fixed on the chip is provided to the scheduler for operation assignment. The scheduler decides on the types and numbers of additional components, if they are needed. The first experiment fixes seven 8-bit adders, one 8-bit subtractor, two right shifters, two left shifters, and one

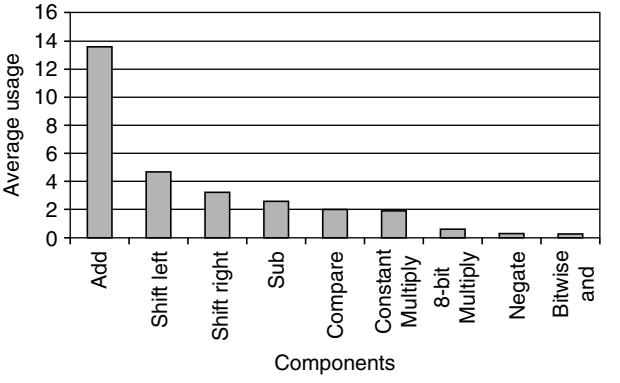



**FIGURE 20.5** Component usage profile.

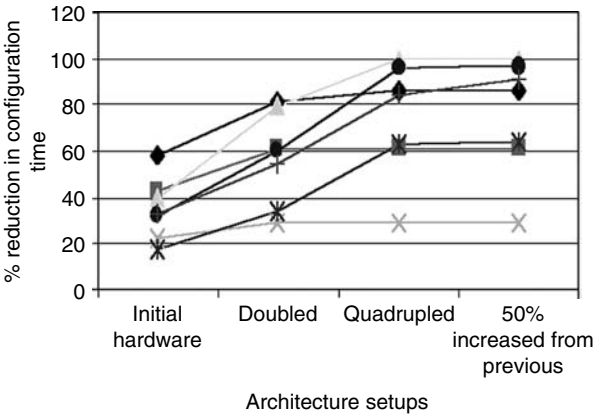



**FIGURE 20.6** Relative gains in configuration times for different setups.

**TABLE 20.2** Programming Bits Required Implementing the VPBs with Reconfigurable Logic

| Parameterizable Block | Size (CLBs) | Programming Bits |
|---|---|---|
| Pixel modification | 30 | 29,910 |
| Thresholding | 11 | 10,967 |
| Filtering | 99 | 98,703 |

constant multiplier. Then we have doubled the amount of hardware fixed in proportion at each step except the last one, where we have increased the fixed hardware 50% from the previous setup. Figure 20.6 presents the relative gains in configuration times for different setups. Observe how the gain in reconfiguration time improves with more logic provided, and how this trend saturates at a certain point. For the initial architecture setup the average reduction in reconfiguration time is 35%. Observe that, as resources available on the chip are duplicated, this reduction goes as high as 75%.

If a VPB can cover all operations of an application, then the largest gain can be obtained. Our image processing blocks presented in Section 20.5 will serve this purpose. They are capable of supporting several different image-processing applications. For the three blocks that we are initially considering, the potential savings in the number of programming bits is shown in Table 20.2. We have synthesized three parameterized designs for these blocks and obtained area information. Using the CLB count for these blocks we can estimate the number of programming bits required, proportional to the size of the designs just as we did for the first set of experiments. We assume that the reconfigurable fabric is similar to a Virtex chip. By using the numbers reported in [13] we derive the number of programming bits required per CLB and hence per parameterizable block.

## 20.7 Conclusion

A novel reconfigurable architecture was presented. The SPS can provide the degree of flexibility required in today's systems. Although offering high-performance for its application set and still a high degree of flexibility for other applications, the architecture promises a good performance and smaller reconfiguration time as well. Experiments indicate that a proper selection of common blocks among a fairly wide range of applications can yield an average reduction of 35% up to 100% in the number of programming bits that need to be transferred to the chip for configuration/reconfiguration. Because the VPBs eliminate a considerable amount of programming switches on the chip, the improvement in delay will

be accompanied by improved power consumption as well. The individual VPBs will be designed targeting the best trade-off between delay and power consumption. Implementation of applications that are within the covering region of this system will highly benefit from these abilities.

## References

1. P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh, "A quick safari through the reconfiguration jungle," in *Proceeding of Design Automation Conference,* June 2001.
2. K. Kuetzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Ciruits and Systems*, Vol. 19, No. 12, Dec. 2000.
3. Ray Bittner and Peter Athanas, "Wormhole run-time reconfiguration," in *Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays*, 1997, pp. 79–85.
4. J.G. Eldredge and B.L. Hutchings, "Run-time reconfiguration: a method for enhancing the functional density of SRAM-based FPGAs," in *Journal of VLSI Signal Processing*, Vol. 12, 1996.
5. S.O. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, "SPS: a strategically programmable system," in *Reconfigurable Architecture Workshop*, April 2001.
6. J.R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a re-configurable co-processor," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.
7. S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, "The chimaera reconfigurable functional unit," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.
8. E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon, "A first generation DPGA implementation," *FPD95, Canadian Workshop of Field-Programmable Devices*, May 29-June 1, 1995.
9. C. Ebeling, D. Cronquist, and P. Franklin, "Configurable computing: the catalyst for high-performance architectures," in *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures, and Proc.*, July 1997, pp. 364–372.
10. E. Waingold et al., "Baring it all to software: The Raw machine," *IEEE Computer*, Sept. 1997.
11. H. Zhang et al., "A IV heterogenous reconfigurable processor IC for baseband wireless applications," ISSCC.
12. B. Wilkinson and M. Allen, *Parallel Programming*, Prentice-Hall, 1999.
13. Xilinx, Inc., "Virtex FPGA series configuration and readback," Application Note: Virtex Series.

# 21

# Reconfigurable Processors

John Morris
*Auckland University*

Danny F. Newport
Don Bouldin
*University of Tennessee*

Ricardo E. Gonzalez
Albert Wang
*Tensilica, Inc.*



## 21.1 Reconfigurable Computing

*John Morris*

### 21.1.1 Preamble

Architects of general-purpose processors face a herculean task: to design a processor that will run *every* application fast. However, applications vary widely in instruction mix, frequency, and patterns of data access, input and output bandwidth requirements, etc. A designer may incorporate elaborate and space-consuming circuitry that simulation shows will dramatically improve performance for one application but has no effect on another—or worse, slows it down. For example, designers will normally incorporate as large a cache as space allows on a die, since cache speeds up most applications; however, the data cache adds nothing to the performance of an application that copies data from one place to another.

Programmable hardware can be used to build systems in which the circuitry matches the structure of the problem. In particular, inherent parallelism in problems, which a general-purpose processor—despite multiple "go-fast" enhancements—cannot exploit, can be exploited in a system in which multiple circuits are used to speed up the computation.

### 21.1.2 Programmable Hardware

Programmable hardware has evolved in capability and performance—tracking processor capabilities for many years now. Designers have a wide spectrum of devices that they can draw upon—from ones that

provide a handful of gates and flip-flops to ones that provide well over a million gates.* In addition, modern devices provide:

- Considerable on-chip memory: this partially overcomes an inability of early devices to effectively solve problems that required more than a few memory bits
- Large numbers of I/O pins—permitting high data bandwidths between a custom processor and its environment
- Multiple I/O protocols, such as LVDS, GTL, and LVPECL—enabling high speed serial channels between the device and other components of a system

Programmability may be provided by a number of technologies:

- Fuses or anti-fuses, in which links are programmed open or closed
- EEPROM, in which a configuration bit is stored in nonvolatile read-only memory, and
- Static RAM cells, which store configuration bits, but, which need to be reloaded every time the device is powered up

Thus, a designer has a broad palette of devices on which to base a system design. All the usual trade-offs apply: in particular, the ability to change the circuit by reprogramming it invariably introduces a speed penalty. A configurable circuit is more complex and has longer propagation delays than a fixed one: this translates to a slower maximum clock frequency. This trade-off is discussed further when we consider whether an application is a good candidate for a reconfigurable processor compared to a general-purpose commodity processor.

A number of terms have been used to describe programmable devices. Simple early devices (ones with a simple programmable and-or array, coupled with ~10 flip-flops and ~20 I/O pins) were commonly called "programmable array logic" chips or PALs, but a host of other similar terms have been used for marketing purposes. The most important group of devices for building processors are now almost universally termed "field programmable gate arrays" (FPGAs)† and this chapter section will focus on them as the key building blocks of a reconfigurable system. As with general-purpose processors, designing a "universal" FPGA is essentially an impossible task and a number of different architectural styles have been proposed and manufactured. The following subsections will describe the key elements of some representative devices.

#### 21.1.2.1 FPGA Architectures

An FPGA's capability can usually be described in terms of three elements:

- *Logic blocks*: These are small blocks of logic, commonly consisting of a small number of simple and-or arrays, some multiplexers for steering signals, one or two flip-flops. Other features such as memory bits, lookup tables, special logic for handling the carry chains in adders, etc., may be present also. Marketing pressures have produced a bewildering array of names for these blocks: fortunately, most of them are readily understood. Examples are logic array blocks (Altera APEX 20k family), logic elements (Altera FLEX 10KE), macrocells (Altera MAX7000/MAX3000), configurable logic blocks (Xilinx), and programmable function units (Lucent ORCA).
- *Routing resources*: A typical FPGA will provide lines of various lengths to interconnect the logic blocks. Short lines provide low propagation delay paths between neighbouring blocks; longer lines connect more distant blocks with low delay. A small number of buffered low delay lines, which can interconnect large groups of logic blocks are usually provided for clocks.
- *I/O buffers*: Special purpose logic blocks provide interfaces to external circuitry. In modern devices, the I/O buffers provide a variety of electrical protocols eliminating the need to use

*2001 value: apply Moore's Law for 2002 and forward.

†Market habits die hard, though: Altera persists in referring to its devices as programmable logic devices (PLDs).

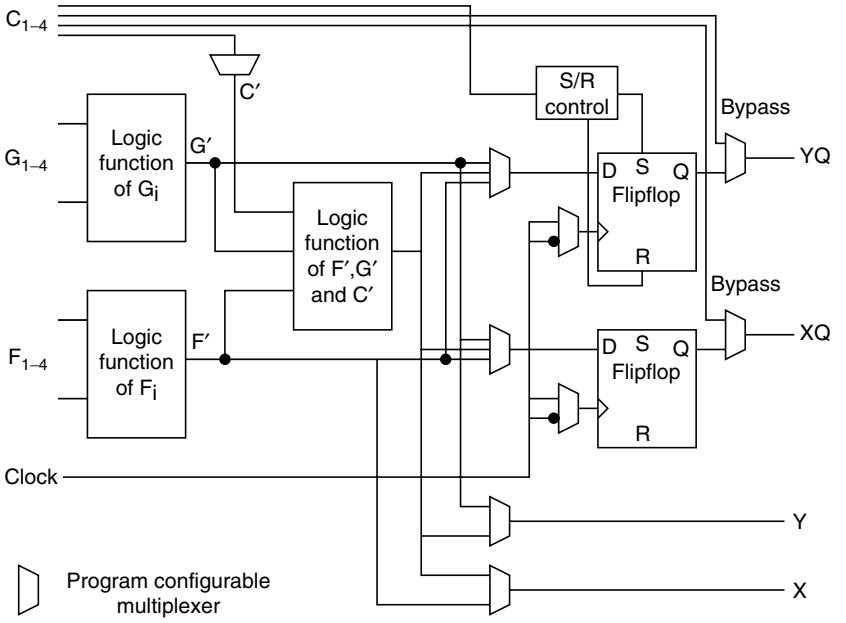



**FIGURE 21.1** Simplified block diagram of the Xilinx XC4000 device control logic block. (The XC4000 CLBs have additional capabilities [1].)

special interface buffers. Reducing chip-to-chip connections provides greater data transfer bandwidth between the reconfigurable processor and its environment.

Recent devices include memory blocks, which may be configured in several ways.

#### *21.1.2.1.1 Xilinx 4000 and on*

Xilinx' 4000 series devices [1] were not the first of their family, there were several antecedents; however, in order to avoid turning this chapter section into a history lesson, I will describe it first. It is a good representative of a number of commercially available devices.

*Control Logic Blocks*—Figure 21.1 shows the essential features of the 4000 series control logic blocks (CLBs). It contains three logic function blocks—each capable of implementing any arbitrarily defined boolean function of its inputs—and two flip-flops controlled by a common clock. "Programming" the device sets the logic functions in the logic function blocks, the signal steering multiplexors and the set/reset control. There are nine basic inputs: $F_{1-4}$, $G_{1-4}$, and $C_4$ (a direct data input to the flip-flops) and four outputs—two registered and two combinatorial. Paths can be chosen which bypass either or both flip-flops. Xilinx's designers have chosen to implement a moderately complex logic block. In contrast, Altera devices have simpler logic blocks with a single flip-flop [2] and Quicklogic's super cells are more complex [3]. Lucent refers to the ORCA logic block as a programmable functional unit (PFU) reflecting its complexity: 19 inputs and 4 flip-flops [4]. Additionally, the logic in the function blocks can be configured to act as a block of RAM, which can be configured as $16 \times 1$, $16 \times 2$, or $32 \times 1$ bit blocks. Without this capability, applications requiring memory are forced to use the CLB flip-flops, using a whole CLB for each 2 bits. This was a significant limitation of early devices, but newer ones, in addition to the 32 bits per CLB provided in the 4000 series, provide dedicated RAM blocks with significant capacities [2,5].

*Routing Resources*—A great challenge to FPGA designers is achieving a good balance in the allocation of die area to programmable logic (the CLBs) versus routing resources. The XC4000 designers provide a combination of short lines which connect each CLB to a programmable switch matrix adjacent to it,

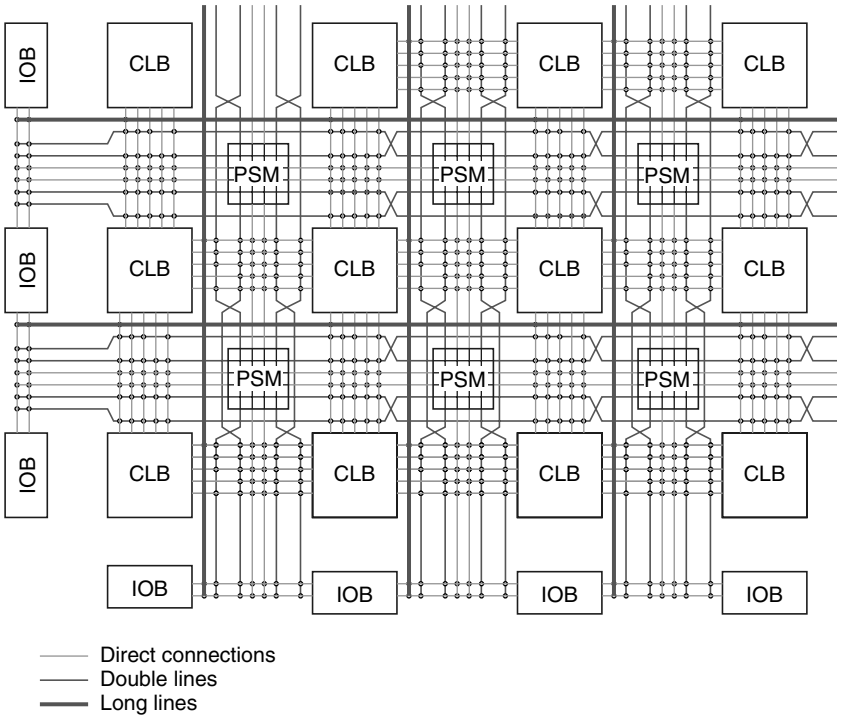



**FIGURE 21.2** Conceptual view of the routing on an XC4000 device showing the pattern of logic blocks (CLBs) embedded in "channels" of routing resources. Direct connections to the programmable switch matrix (PSM) are shown as well as the patterns for double lines connecting every second PSM. Similarly, quad lines (omitted) connect every fourth PSM. Long lines run the length of horizontal and vertical channels. This is a concept diagram only: actual devices may differ in details [1].

double and quad length lines which connect every second (or fourth) switch matrix, and long lines which run the entire length of a device (see Fig. 21.2). Connections through the switch matrices provide ultimate flexibility—any CLB may be connected to any other; however, there is a penalty: the switch points are implemented with pass transistors which add to the propagation delay of any signal passing through them. Thus, the short lines through the switch matrices should not be used for critical signals connecting widely separated CLBs. The double, quad, or long lines need to be used to reduce delays. Predicting the optimal allocation for any application is obviously a hard task and many strategies may be seen in the commercially available devices. For example, Altera's Apex 20K devices employ a hierarchical structure, grouping basic logic elements (LEs) into logic array blocks (LABs), which are in turn grouped into MegaLABs [2]. Each block has appropriate internal routing resources. Copper is also used to reduce resistance and thus propagation delay.

*I/O Buffers*—I/O buffers provide circuitry to interface with external devices. Apart from input buffers and output drivers, the main additional feature is the ability to latch both inputs and outputs. The simplified diagram of an XC4000 I/O buffer (IOB) in Fig. 21.3 shows the output driver, input buffer, registers, several inverters, and the programmable multiplexors. The inverters provide almost all combinations of normal and inverted direct output or latched signals synchronized with direct or inverted clocks: this avoids the need to use resources in the CLBs simply to invert signals. Limited slew rate control was also added to the output buffers—a precursor to the support for multiple electrical protocols now found in more modern designs.

*Additional Features*—Adders, including counters, occur on the critical paths in many calculations, so the 4000 series, like most of its modern counterparts, provides "fast-carry" logic. Ripple carry adders

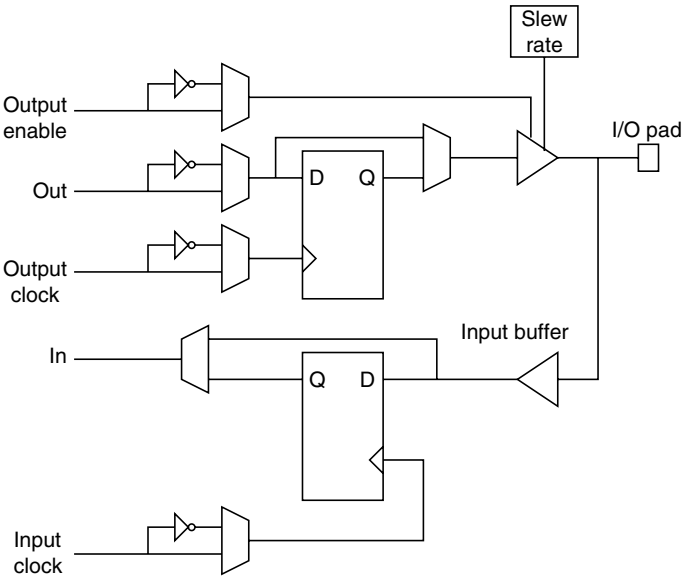



**FIGURE 21.3** Simplified block diagram showing essential features of the Xilinx XC4000 input/output block (IOB). (The XC4000 IOBs have additional capabilities [1].)

are simple, regular, and use minimal logic, but they must wait until a carry bit has "rippled" through all the summing (full adder) blocks. By providing a fast, direct path for carry bits between blocks, the critical delay in a ripple carry adder is significantly reduced. The fast carry logic is so effective that there is no advantage to be gained from more complex adders, such as carry-lookahead ones. A carry-lookahead adder requires a much larger number of CLBs and the signal propagation times between these additional CLBs outweigh any benefit to be gained from a complex adder: trials with carry-lookahead adders show them to be slower than ripple carry adders that use the fast-carry logic [6].

The special needs of global clocks are addressed by providing "semi-dedicated" I/O pads connected to four primary global buffers designed for minimum delay and skew. The clocks of each CLB can be connected to these global buffers, a set of secondary buffers or any other internal signal. Thus multiple global and local clock domains can be established.

Problem diagnosis and boundary scan testing is facilitated through support for IEEE 1149.1 (JTAG) boundary scan logic attached to each I/O buffer.

The CLB structure lends itself to efficient implementation of functions of up to 9 inputs, but address decoders commonly require many more bits. Special decoders accepting up to 132 bits for large XC4000 devices are provided to ensure fast, resource-efficient decoding.

A simple internal oscillator and divider provides clock signals when precise frequencies are not required.

#### *21.1.2.1.2 Virtex*

The Virtex family [5] are enhanced versions of the Xilinx 4000 series. Improved process technology has allowed the gate capacity to exceed one million ($4 \times 10^6$ are claimed for the largest member of the family, requiring 2 MB of configuration data). Supply voltages as low as 1.8 V allow internal clocks up to 400 MHz to be used.

*Memory*—Blocks of dedicated memory are now provided, which can be programmed to a number of single- and dual-port configurations. This will allow considerable performance enhancements for designs which were previously forced to use external memory.

*I/O Buffers*—One of the most dramatic additions to the newest devices from all manufacturers is the support of numerous electrical protocols at the I/O pins. For example, Virtex supports single-ended standards: LVTTL, LVCMOS, PCI-X, GTL, GTLP, HSTL, SSTL, AGP-2X and differential standards: LVDS, BLVDS, ULVDS, LDT, and LVPECL. Support for PCI-X means that a Virtex device can implement the industry-standard PCI interface, considerably reducing the complexity of PCI cards which can now combine interface logic, control logic, some memory, and external bus interfaces (e.g., LVDS) in a single chip.

Virtex devices are also partially reconfigurable: individual columns may be reprogrammed.

#### *21.1.2.1.3 Algotronix*

Algotronix's approach to FPGA design was radically different from the Xilinx* approach. Cells were much simpler *and used for routing as well as logic*. The basic cell in shown in Fig. 21.4. By using a much simpler cell, it becomes possible to fit more logic per silicon die and the XC6264 device [7] was rated as containing $10^5$ gate equivalents early in 1997—about twice as many as devices with distinct logic blocks and routing resources at the same time. Routing—other than between neighboring cells—uses the logic cells programmed simply to route a signal from input to output. Signals routed in this way pass through transmission gates and suffer significant delays, so the XC6200 devices provided a hierarchy of "Fast-LANE" connections, which linked lines of 4 and 16 cells.

These devices also permitted fast reconfiguration: the SRAM cells that hold the configuration data can be directly addressed so that part of an operating circuit may be quickly reconfigured. (By contrast, the XC4000 devices are programmed either with a serial bit stream or byte-by-byte from an EEROM—requiring milliseconds for a complete chip to be reconfigured.)

The "sea-of-gates" approach provided by the XC6200 devices may be viewed as one end of a spectrum stretching from simple cell/no dedicated routing devices to complex cell/dedicated routing devices such

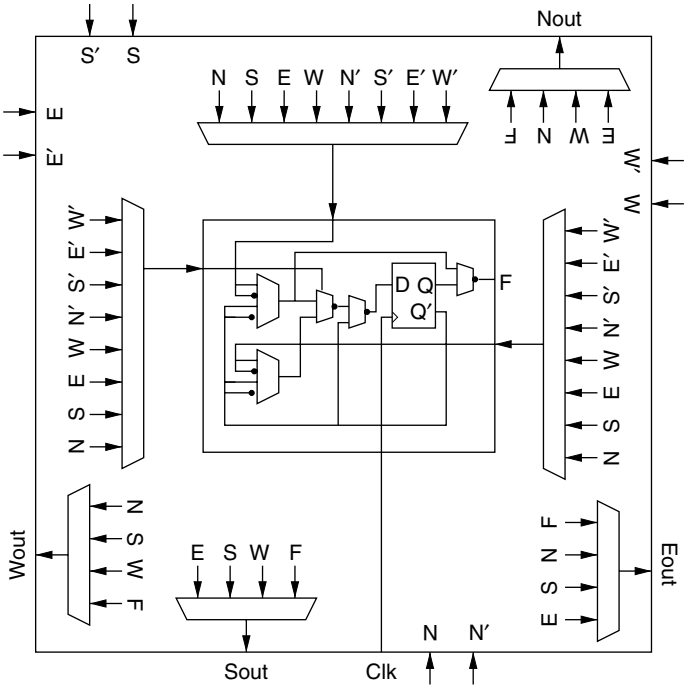



**FIGURE 21.4** Block diagram of the cell in an XC6200 device. This cell is used for routing also.

*However, Algotronix was taken over by Xilinx and its devices appeared as the Xilinx XC6200 series [7].

as the XC4000 and most other commercially available devices. Regular applications requiring large numbers of operations or very wide uniform data paths are likely to match a sea-of-gates device better. Less regular problems with complex control requirements—requiring functions of many signals—are likely to match the complex logic device devices better. The industry, however, appears to have voted strongly for complex logic block devices and the XC6200 series is no longer commercially available.

### 21.1.3 Reconfigurable Systems

Reconfigurable systems are easy to build: a designer has only to decide what interconnection patterns will best serve the needs of target applications and some systems, e.g., UWA's Achilles, even allow that to be deferred. The major proportion of the circuitry may be changed once the basic hardware system has been constructed. As a consequence, an enormous number of experimental and several commercial systems have been built: Guccione has compiled a list, it contains summaries of over 80 systems [8]. An attempt to cover all of these is clearly futile: a small representative sample has been chosen.

#### 21.1.3.1 SPLASH 2

One of the best known systems is SRC's SPLASH 2 [9]. It consists of an array of FPGAs and interface allowing the array to be attached to a SPARC host. The FPGA array itself was composed of a number of array boards, each containing 17 Xilinx XC4010 FPGAs—16 "computing" devices and one interface device. Typical of devices of its time ($\sim$1990), the XC4010 can provide limited amounts of memory itself, so 512 KB of conventional memory were attached to each FPGA. Apart from nearest neighbour connections, a crossbar switch permitted dynamic ("almost" tick-by-tick [9]) connection changes to be made.

#### 21.1.3.2 Programmable Active Memories (PAM)

A dozen copies of the variant named DECPeRLe-1 found their way into research centers around the world and were applied to a diverse set of problems [10]. The computing surface was a $4 \times 4$ array of XC3090 devices with seven additional FPGAs acting as memory and interface controllers. FPGAs in the array were connected directly to each of their four neighbours. Devices in each row and column shared common 16-bit buses, so that there were four 64-bit buses running the length of the array—one for each geographic direction N, S, E, and W. Static RAM was added to provide the storage lacking in the early devices used and FIFOs provided elasticity in a high-speed interface to a host processor. Vuillemin et al. discuss an extensive list of problems to which PAMs were applied [10]: long integer arithmetic, RSA cryptography, molecular biology, finite differences, neural networks, video compression, image classification, image analysis, cluster detection, image acquisition, stereo vision, sound synthesis, and Viterbi decoding.

They consistently applied the following rule in deciding what part of any problem should be allocated to the hardware:

"Cast the inner loop in PAM hardware; let software handle the rest [10]!"

PAM spawned a successor, PAMETTE, a PCI card with 5 Xilinx 4000 series devices on it [11]. One device served as the PCI interface with the remaining four arranged in a $2 \times 2$ matrix. SRAM and DRAM may be added and provision is made for external connections via a daughter board. A large number of similar boards—all with the same basic idea: place a number of FPGAs on a card, which may be inserted into the bus of a suitable host—have been designed by research groups. Several commercial products are also available.

#### 21.1.3.3 SPACE

The Scalable Parallel Architecture for Concurrency Experiments (SPACE) machine was developed at the University of Strathclyde [12]; it was followed by the SPACE-II, built at the University of South Australia [13]. Both variants used fine-grain FPGAs (Algotronix CAL1024s in SPACE and Xilinx XC6216s in SPACE 2) as the primary target was the simulation of highly concurrent systems such as digital circuits, traffic systems, particle flow, and electrical stimuli models of the heart. SPACE 2 processor boards

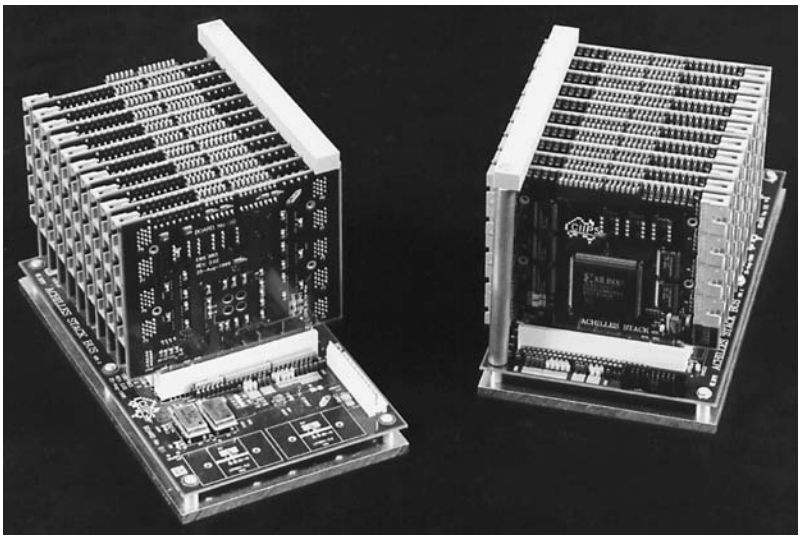


**FIGURE 21.5** Achilles 3-D stack. Each small PCB contains one FPGA: connections are made by cabling between the connectors visible on each small PCB.

contained 8 XC6216 processor FPGAs and an XC4025 providing a PCI interface to an Alpha host. On each board, the fine-grained processors are connected in a mesh in order to provide a seamless array of gates on the board. Additional memory (32 Mb of static RAM) was present on each board. A secondary backplane allowed high-bandwidth connections between SPACE 2 boards.

#### 21.1.3.4 Achilles

The Achilles architecture aims to provide much more flexible interconnection patterns: Figure 21.5 shows the 3-D arrangement in which small PCBs containing a single FPGA are arranged in a vertical "stack" [14,15]. A limited number of fixed bussed interconnections are provided at the base of the stack, committing only about one-third of the available I/O pins to fixed interconnect. A second side of the stack is used for programming and diagnostic connections: this enables the stack to be "gang" programmed—each FPGA is loaded with an identical program—or individually. The remaining two sides have uncommitted connections: connectors are provided for groups of eight signals and ribbon cables are used to connect FPGAs as the target application requires. This system offers wide variations in communication patterns at the expense of manual reconfiguration.

### 21.1.4 Applications

The list of applications, which have been successfully implemented in reconfigurable hardware systems, is long; it includes applications from such diverse areas as:

- Image processing
- Cryptography
- Database and text searching
- Compression
- Signal processing

It is generally straightforward to transfer an algorithm from a general-purpose processor to reconfigurable hardware; synthesizers which convert VHDL or Verilog models into the bit streams necessary to program an FPGA-based system are available and efficient; however, a *successful* transfer must provide a

solution which is more efficient, by some criterion, than the same algorithm running on fast commodity general-purpose processors. Reconfigurable hardware generally runs slower,* consumes more power, and costs more than commodity processors. This remains true at most points in the performance spectrum. At the low performance end, small processors, e.g., Motorola's HC11 series, are available at very low cost and very low power consumption and will thus perform simple control and data processing tasks effectively. Although a modern FPGA may outperform the relatively slow processors available at the low end of the performance spectrum, there are a host of general-purpose embedded processors, e.g., the PowerPC-based devices, which will provide the additional processing power while still consuming less power and costing less than an FPGA. At the high performance end of the spectrum, the internal clock speeds of FPGAs lag behind those of commodity processors and thus their sequential processing capability does not match that of, for example, a state-of-the-art Pentium or SPARC processor; however, although it is clear that reconfigurable hardware will not provide efficient solutions for all problems, there are areas in which it is extremely efficient.

The general characteristics of successful applications are

(a) *Sufficient parallelism*: The processing algorithm must have sufficient inherent parallelism to allow multiple processing pipelines to be created. This parallelism can be either *direct* or *pipelined*.

(b) *Low storage requirements*: Early FPGAs provided very few bits of memory—the flip-flops in logic blocks were an expensive way to provide memory. Later FPGAs have addressed this problem by allowing the configuration bits to be used as lookup tables and thus provides tens of bits per logic block. The newest generation of FPGAs provide blocks of dedicated memory but capacities are measured in megabits, not megabytes. Although external memory can always be added and wide buses employed to provide high bandwidth, this uses valuable I/O pins, the path to external memory is likely to become a bottleneck and limit performance.

(c) "*Decision-free*" *processing patterns*: Multiplexors in the data paths will readily handle simple decisions, which switch the dataflow between down-line functional blocks, but complex decision trees will generally not map efficiently to hardware. When large numbers of branches exist, inevitably many paths are little used and thus expensive to implement in fixed hardware relative to their benefit. In particular, error handling logic will generally be complex relative to its frequency of use. Complex decision logic is efficiently handled in high-performance modern processors, which move common logic to cache at the expense of little used code. When branches have similar probabilities, speculative execution ensures good average rates of instruction completion. However, this criterion for successful hardware implementation should be applied with caution: if high throughput for all possible processing paths is required, then the resources devoted to implementing all paths (including little used ones) may be justified. In the near future, dynamically reconfigurable logic may also provide effective solutions when there are complex decision trees.

(d) Ability to use local (i.e., between neighbouring devices) data paths in problems that are large enough to require multiple devices. Most systems provide high-bandwidth paths between nearest neighbors with lower-bandwidth multiple device buses and global interconnects. The 3-D Achilles design provides more device-device data path flexibility but at a cost—wiring patterns must be set up manually for each application [14].

(e) *Integer arithmetic*: Although it is possible to implement arbitrary precision floating-point processors in FPGAs, the number of logic blocks required and hence the delays introduced by data paths between logic blocks make them expensive in area and low in performance compared to those found in superscalar processors.† On the other hand, the ability to easily implement arbitrary precision integer arithmetic allows a reconfigurable system designer to pack

*However, Tsu et al., argue that there is no inherent reason why an FPGA should be slower [16].

†Superscalar processor manufacturers are also prepared to invest large amounts in order to win benchmark competitions, which allows man-years of effort to be used to optimize individual circuits and layouts.

more functional units of higher performance into a given area by choosing the minimum required word length.

#### 21.1.4.1 Image Processing

Real-time image processing presents a classic application for custom processors. A stream of pixels emanating from a camera can be passed through a wide deep pipeline—performing as many unrelated and complex operations on each pixel as needed. Unrelated operations (e.g., threshholding and masking) are performed in parallel and complex operations (e.g., masking) are performed in deep pipelines. For basic operations, little storage is required and the relatively inefficient memory on an FPGA suffices. A masking operation, such as applying a $3 \times 3$ mask to a group of neighboring pixels, requires the storage of two scanlines in a shift register and thus is feasible in large FPGAs. The reverse process, visualisation, or the processing of machine generated images for display is already the domain of special purpose processors, but market volumes have justified use of ASICs.*

#### 21.1.4.2 Stereo Vision

The matching problem dominates research into fully automated stereo vision systems; it requires the comparison of pixels (or regions of pixels) to determine matches between corresponding segments of two images. The distance between matching regions on the left and right images (the disparity) is combined with camera system geometry to determine the distance to objects in the field of view. Without the apparent ability of a human brain to "jump" to the obvious match, a machine must try all possible disparities in order to find candidate matches between pixels or to correlate regions. Objects close to the camera system have disparities approaching infinity, but one of the major applications of stereo vision is collision avoidance in which it is possible to put a lower bound on the distances of objects from the camera.† In practical camera systems, this results in a need to consider objects with disparities from 0 pixels (i.e., at infinity) to of the order of 10–100 pixels at closest permissible approach. Thus this problem has all of the required attributes for an efficient pipeline parallel implementation:

- Parallelism of 10–100 or more
- Simple calculations (comparing pixel intensities)
- Regular computation (the same correlation operators are applied to each pixel)

Woodfill et al., using the census transform to reduce problems caused by intensity variations and depth discontinuities, programmed a PARTS engine [17] to calculate object depths from pairs of $320 \times 240$ pixel images. With a maximum disparity of 32, their system was able to compute depth at 42 frames per second [18]. They estimated that it was performing about $2.3 \times 10^9$ RISC equivalent operations per second. Piacentino et al. have built a video processing system (Sarnoff Vision Front End 200) in which reconfigurable processing elements are used not only for stereo computations, but for motion estimation and warping also [19]. They estimate that the VFE-200 can provide ~500 GOPS of processing power.

#### 21.1.4.3 Encryption/Decryption

Shand and Vuillemin have used RSA cryptography as a benchmark for their PAM machines; they were able to demonstrate an order of magnitude improvement in performance relative to the best software implementations of the time. In 1992, PAM achieved over 1 Mb/s for 512 bit keys compared to 56 kb/s on a 150 MHz Alpha processor [20]. This relative performance will not change; state-of-the-art FPGAs can now fit the entire PAM system in a single device, giving the reconfigurable hardware system additional speed as it no longer needs to use slower inter-device links or external memory.

---

*However, prototyping designs which are destined for ASICs are a major application for reconfigurable processors. They can be used to ensure that a design is correct and that the silicon will function correctly first time. Some foundries will take FPGA-based designs and convert them directly to ASICs.

†The vehicle carrying the camera system is expected to move away before this bound is violated.

Symmetric encryption algorithms are easily and efficiently implemented in FPGAs; they require a number of "rounds" of application of simple operations. Each round can be implemented as a pipeline stage. Thus, as an example, TwoFish [21] requires 16 rounds of lookup table accesses, which can be implemented as a 16-stage pipeline. This allows a stream of 32-bit input data words to be encrypted at very high input frequencies with a latency of 16 cycles. In a study of four AES candidates, Elbirt et al. report an order of magnitude difference between FPGA-based implementations and the best software ones [22]; however, they also note that for one AES candidate, CAST-256, FPGA implementations were slower than their software counterparts. This result highlights the fact that the performance advantage of commodity processors can only be overcome when the problem matches the capabilities of FPGA-based custom processors. By adding further pipeline stages within each round—24 for TwoFish, for example—Chodowiec et al. were able to achieve throughputs greater than 10 Gb/s for five of the AES candidate algorithms (12 Gb/s using a 95 MHz internal clock for Rijndael, the eventual winner of the AES competition) [23].

Secure communications systems require encryption hardware; placing the encryption subsystem in hardware makes it less susceptible to tampering and enables keys to be hidden in "write-only" registers. Reconfigurable hardware provides an additional capability, algorithm agility [24]. This not only enables an encryption algorithm which has become insecure to be replaced with a secure one, but permits an algorithm independent security protocol to use the hardware effectively, loading the appropriate algorithm on a transaction-by-transaction basis.

#### 21.1.4.4 Compression

Using a systolic array style implementation of the LZ algorithm, Huang et al. were able to obtain throughputs 30 times greater than those achievable with commodity processors [25]. This speedup was obtained even though their FPGAs (Xilinx XC4036s) were clocked at 16 MHz versus 450 MHz for the fastest software implementation. Huang et al. believe that even better relative performance would be obtained from modern FPGAs, e.g., Altera's APEX 20K devices have built-in content addressable memories (CAMs), which would speed up the process of matching input strings with the dictionary.

#### 21.1.4.5 Arithmetic

When designing a reconfigurable system, the widths of arithmetic function units, and hence their propagation delays, can be constrained trivially to the number of bits actually required for the application. This saves space, logic resources, and time. Designers also have considerable flexibility when complex arithmetic expressions must be evaluated; they can choose a single-stage combinatorial circuit or increase throughput by adding registers and forming a pipeline. This can often be done at essentially no cost: the logic blocks contain flip-flops already, so there is no space penalty and negligible time penalty.

An application requiring floating point arithmetic may be a poor candidate for a reconfigurable system—to achieve performance comparable to that offered by a commodity processor will require significant effort; however, reconfigurable systems are excellent at processing streams of data from sensors: this data will be fixed point and readily handled by the same circuits used for integer arithmetic.

##### *21.1.4.5.1 CORDIC*

Even trigonometric functions of fixed-point data are readily implemented using CORDIC arithmetic. CORDIC algorithms are iterative, but require only shifts and adds. Again, the designer has a large space in which to work [26]. Bit-serial designs are simple and compact, but require many cycles; this may not be a problem if the input data rate is relatively slow. An iterative bit-parallel design will require more space but fewer cycles. Finally, the iterative loop can be unrolled by one or more stages to produce the desired throughput/space balance.

#### 21.1.4.6 String and Text Matching

Genetic sequencing technology is just one technology that is producing enormous databases of data that must be searched. Thus, there has been considerable interest in hardware to accelerate the process of

comparing new sequences with those in existing databases. Biologists use a measure known as the edit distance when comparing sequences. A simple implementation of a dynamic algorithm can compute the edit distance in O($mn$) time ($m,n$ = length of source and target sequences, respectively), but if the calculation is carried out on a processor array, then it can be seen that all operations on the diagonal may be performed in parallel. A single board Splash 2 machine achieved a factor of 20 speedup over a CM-2—a massively parallel processor [27]!

Similarly, full text searching of documents for relevance has sufficient parallelism to make FPGA-based hardware effective. When document content cannot be adequately described by keywords, a searcher will supply a list of relevant words and require that every word of every document be checked against the list in order to build a relevance score for each document. Gunther et al. demonstrated that the original SPACE machine was effective in this application [28]. They used a technique called "data folding" in which the data are built into the circuitry. Match circuitry is built for each of the words in the list of relevant words and incorporated into a fixed matching structure. This is an excellent example of the power of partial reconfiguration; circuit patterns corresponding to the relevant words are loaded for each new search. They demonstrate that matching in hardware does not need to be limited to direct character-by-character matching. It is possible to implement simple regular expressions allowing, e.g., matching on the root of a word only. Overall the system is able to test for each word in the relevant list in parallel and aggregate a weighted relevance score as the document is read, results become available at a rate which is basically limited by the rate at which documents can be read from disc.

#### 21.1.4.7 Simulations

Cellular automata map readily to reconfigurable systems. They involve arrays of cells: each cell is a simple finite state machine whose behavior depends only on its current state and the state of cells in its immediate environment. Milne extends the fundamental cellular automata concept by removing the restrictions on identical components, uniform update and synchronization of all updates to create generalized cellular automata (GCA); an example of traffic system simulation is described—digital circuits and forest fire development are further systems, which have suitable characteristics [13].

Petri net models are also used extensively in simulation studies; as with cellular automata, there is abundant low level parallelism to be exploited—the firability of each transition can be evaluated simultaneously. Petri net models are based on simple units—places and transitions. It is possible to create generic models in VHDL for these units [14], paving the way to automatic generation of VHDL code from natural visual representation of Petri nets, which can be compiled and downloaded to suitable hardware. A single Achilles stack is able to accommodate a model containing of the order of 200 transitions [14].

### 21.1.5 Reconfigurable Processors vs. Commodity Processors

Any special purpose hardware has to compete with the rapid increase in performance of commodity processors. Despite the relative inefficiency of a general-purpose processor for many applications, if the special purpose hardware only provides a speedup of, say 2, then Moore's Law will ensure that the advantage of the special purpose hardware is lost in a year.* When assessing whether an application will benefit from use of a reconfigurable processor, one has to keep the following points in mind.

#### 21.1.5.1 Raw Performance

The raw performance of FPGA-based solutions will always lag behind that of commodity processors. This is superficially reflected in maximum clock speeds: an FPGA's maximum clock speed will typically be one-third or less of that of a commodity processor at the same point in time. This is inevitable and will continue: the reconfiguration circuitry loads a circuit and requires space, increasing its propagation delay and reducing the maximum clock speed.

*The author has (somewhat arbitrarily) shortened the "break-even" point from the 18 months of Moore's Law, because the extra cost of additional hardware needs to be factored in versus using cheap commodity hardware.

#### 21.1.5.2 Parallelism

Thus, to realize a benefit from a reconfigurable system, the application must have a considerable degree of inherent parallelism which can be used effectively.

The parallelism may be exploited simply by deploying multiple processing blocks—each processing a separate data element at the same time—followed by some "aggregation" circuitry, which reduces results from the individual processing blocks in some way.

#### 21.1.5.3 Long Pipelines

Alternatively, a long pipeline may be employed in which the same data element transits multiple processing blocks in successive clock cycles. This approach trades latency for throughput: it may take many cycles—the latency—for the first result to appear, but after that new processed data are available on each clock cycle giving high throughput. Many signal processing tasks can effectively use long pipelines.

#### 21.1.5.4 Memory

FPGA devices do not provide large amounts of memory efficiently: recent devices (e.g., Altera's APEX 20K devices [2]) do attempt to address this deficiency and provide significant dedicated memory resources; however, the total number of memory bits remains relatively small and is insufficient to support applications which require large amounts of randomly accessible data. This means that, although preprocessing an image which is available as a pixel stream from a camera for edge detection is feasible, subsequent processing of the image in order to segment it is considerably more difficult. In the first case, to apply a $3 \times 3$ mask to the pixel stream, only two preceding rows of the image need be stored. The application of the $3 \times 3$ mask requires a maximum of nine basic multiply-accumulate operations. Thus, it can be effectively handled in a 9-stage pipeline—easily allowing an edge-detected image to be produced at the same rate as the original image is streamed into the FPGA (allowing for an 8-pixel clock latency before the first result is available). In the second, the whole image needs to be stored and be available for random access. Although an FPGA with auxillary memory might handle this task, it is less likely to offer a significant advantage over a general-purpose processor.

#### 21.1.5.5 Regularity

A processing pipeline with large numbers of decisions (if .. then .. else blocks in a high-level language program) is also not likely to be efficiently implemented in reconfigurable hardware. In such a pipeline, there will generally be a large number of branches which are rarely taken, but all need to be implemented in hardware, taking up considerable space (or numbers of logic blocks). Paths with large numbers of blocks of variable size also present a problem for the fixed routing resources in a device.

#### 21.1.5.6 Power and Cost

FPGAs consume more power and cost more per gate than either commercial processors or custom ASICs. Although FPGA technology tracks processor technology and power consumption is being reduced through lower power supply voltages and reduced transistor size, it is unlikely that one will see reconfigurable technology in micro power applications such as wearable computers; however, experiments are underway to test their viability in spacecraft, where power is limited [29]. The cost factor is generally offset in low volume production by the significantly lower design cost, faster design cycles, and ease with which modifications can be incorporated.

### 21.1.6 Dynamic Reconfiguration

There is considerable interest in the ability to reconfigure a running circuit. This would allow applications (or groups of applications) to load circuits on demand to meet the requirements of a current task. This would make a reconfigurable system a truly general purpose one: able to load processing tasks as demanded by the input data.

Although most commercial devices require a complete new configuration program to be loaded every time, usually by paths with limited bandwidths requiring thousands of cycles to completely reprogram a device, some commercially available devices have had limited dynamic reprogramming capabilities for some time, e.g., the original Algotronix CAL1024, its successor the Xilinx XC6200 (both now out of production), Atmel's AT6000 (now superceded), AT40K, and Xilinx's Virtex family.

These devices extend the standard device programming model by allowing a part of the configuration to be reloaded from an external source: an alternative has been proposed—the DPGA model [30]. A DPGA device would hold several configurations in the configuration memory for each logic block and allow the context to select one dynamically. The flexibility gained from this arrangement allows much more effective gate utilization—at the expense of the additional space for the configuration memory and context selection logic.

Noting some of the limitations introduced by conventional approaches to dynamic reloading of configuration data, Vasilko and Ait-Boudaoud have proposed an optical system for transferring new configuration data to a device [31]. Optical buses not only allow massively parallel data transfer but provide an additional dimension for information transfer and thus reduce the conflict for routing space between data paths and configuration nets. The advent of devices similar to their proposals would remove some of the practical limitations constraining effective dynamically reconfigurable systems.

One requirement for effective run-time reconfiguration is a model which allows a computation to be split into swappable computational blocks. Caspi et al. summarise several proposed models in introducing their SCORE (Stream Computations for Reconfigurable Extension) model [32]. SCORE divides a computation into fixed size pages, which can be swapped between the running hardware and backup store. A dataflow computation model is used, allowing the run-time system to manage resources: a dataflow model is "memory-less" as far as the programmer is concerned. Data flowing between pages is buffered transparently by the run-time system when necessary.

### 21.1.7 Hybrid Systems

Hybrid systems couple a conventional processor and an area of uncommitted logic that may be configured to suit the demands of algorithms in which the conventional processor cannot exploit data or pipeline parallelism. Berkeley's Garp processor is an example of this approach [33]. Garp contains a RISC processor core (MIPS-II) and $32 \times 23$ array of logic blocks. A 24th column of logic blocks is responsible for communication outside the array. Logic blocks take up to four 2-bit inputs and produce 2-bit outputs: a row of the array can thus process up to four 46-bit words. Garp's designers hypothesize that the reconfigurable section may be used effectively to implement the critical kernels found in most code: the ability to hard-wire the control logic will reduce instruction fetch bottlenecks and better exploit parallelism. Memory queues, which handle streaming of data to and from memory, were added because many applications which use reconfigurable systems effectively process streams of data.

Results from the Garp simulator on a wavelet image compression program showed an overall speedup of 2.9 compared to the MIPS processor. Individual kernels within this program showed speedups up to 12, observed when a kernel had high exploitable instruction level parallelism and the configuration loading time could be amortized over many compute cycles. Comparisons of Garp's performance with a 4-issue superscalar processor also showed significant speedups, indicating that Garp was able to exploit more instruction level parallelism, sustaining 10 instructions per cycle in many cases.

### 21.1.8 Programming Reconfigurable Systems

#### 21.1.8.1 High-Level Hardware Design Languages

The design flow for a reconfigurable system is shown in Fig. 21.6; a high-level hardware design language (HDL) is usually used for the software modeling stage: VHDL and Verilog are widely used as excellent support tools are available. The design process is basically identical to that used for any software system: specifications are drawn up and validated, software models created and verified and the compiled

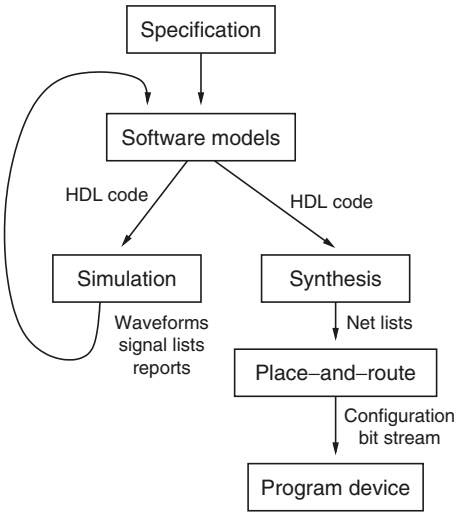



**FIGURE 21.6** Design flow using an HDL (e.g., VHDL). Note the absence of a feedback loop in the synthesis branch: for a design verified in simulation, the synthesis process is a black box.

"program" is loaded onto the target devices or burnt into ROMs. The only significant difference is that two compilers are generally used. A simulator compiles VHDL or Verilog source and produces diagnostic output not only as text to consoles or logged to files, but as waveforms or lists of changes in signal values. When the designer has verified that the models perform in accordance with their specifications under simulation, a synthesizer compiles the source again to a netlist—an intermediate representation of the final circuit. Device-specific place-and-route tools take netlists as input and place logic into logic blocks and configure the FPGA's routing resources to make the necessary connections between logic blocks and I/O pins. The output of this stage is a configuration file—a stream of bits which are loaded onto the device to program its internal registers, multiplexors, etc. For many designs, the whole process (synthesis → place-and-route → configuration bit stream) can be viewed as a single step black-box, which turns verified HDL models into configuration files. Whilst it may take several hours for a complex system, it does not require any input from the user. The designer will usually simply advise the tools whether speed or area is the primary constraint. Significant interaction with the place-and-route tools is needed only if there are performance constraints which cannot be met with default parameters: in this case, manual placement of logic blocks can assist in satisfying the constraints.

#### 21.1.8.2 Other Languages

Other routes from high-level languages are possible: Callahan et al. describe a tool which starts with C [34]: it is aimed at "dusty-deck" systems. Of more potential for effective use of reconfigurable systems are special purpose high-level languages such as Milne's process algebra, CIRCAL [35].

### 21.1.9 Conclusions

In this chapter, instead of providing a litany of praise for reconfigurable computing in all its forms, the author has tried to set out the general characteristics of problems, which a reconfigurable processor might be expected to solve efficiently. A key requirement is clearly sufficient exploitable parallelism, but that may appear either as raw or pipeline parallelism. Raw parallelism is a requirement to perform many simultaneous operations on a single item of data or the same operation on many data items, which may be presented to the reconfigurable hardware at the same instant. Pipeline parallelism, on the other hand, requires many operations to be performed on individual elements of a data stream, allowing a deep

pipeline to process data at its arrival rate and produce results in real time, even if some latency penalty must be paid.

Reconfigurable systems are always competing against the inexorable rise in the power of general-purpose processors. Although reconfigurable devices track the performance gains due to better device technology, they inevitably lack the commercial drive that propels commodity processors forward in performance and thus lag behind their better funded cousins in raw performance. Thus, when considering a special purpose processor for any task, one must keep in mind the performance point at which commodity processors will be when the design is complete. With "multimedia extensions" such as MMX and Altivec, commodity processors even have limited parallel processing capabilities; however, these are limited to very regular computations and a reconfigurable system—with its ability to implement multiple parallel data paths—will generally be better at matching the "shape" of a multiple processing step algorithm. The use of high-level design languages, such as VHDL and Verilog, also shortens design cycles making time-to-completion for projects based on reconfigurable hardware considerably shorter than custom hardware designs.

Thus, although problems that fail to meet the criteria are set out here, and thus will be more effectively solved using commodity processors, the author has shown that many problem domains also exist in which large numbers of individual problems are well suited to reconfigurable processors.

In this chapter, discussion of successes has, for the most part, focussed on systems in which reconfiguration times are long—requiring hours if the time for synthesis software to compile, analyze, and place and route a model expressed in a high-level design language is included; however, there is much active research into dynamically reconfigurable systems, which has the goal of producing hardware whose function may be altered as quickly and conveniently as the general-purpose processors. The significant problems with which researchers in this area are now grappling will be solved eventually. Thus, we can anticipate systems in which parallelism present at some level in virtually all problems, which cannot be exploited now, will be exploited by systems that have been configured on-the-fly. As with statically programmed systems, when data paths can be provided that match problem structures, we will obtain orders of magnitude larger processing powers efficiently. We will not need to use large processor arrays in which many processors are needed for a few vital steps but are idle for much of the time.

## References

1. Xilinx, Inc., XC4000 data book, 1997.
2. Altera Corp., APEX 20K Programmable Logic Device Family Data Sheet, http://www.altera.com/literature/lit-apx.html, 2001.
3. QuickLogic Corp., QuickLogic: Beyond Programmable Logic, Sunnyvale, California, 2001.
4. Lucent Technologies, Inc., ORCA Series 2 Field-Programmable Gate Arrays, June 1999.
5. Xilinx, Inc., Virtex-II 1.5 V Platform FPGA Family, http://www.xilinx.com/partinfo/ds013-2.pdf, 2001.
6. Baskoro, E. and Morris, J., Fast adders in FPGAs, Technical Report TR2001-01, Centre for Intelligent Information Processing Systems, University of Western Australia, 2001.
7. Xilinx, Inc., XC6200 Field Programmable Gate Arrays, 1997.
8. Guccione, S., List of FPGA-based computing machines, www.io.com/~guccione/HW_list.html, 1999.
9. Buell, D.A., Arnold, J.M., and Kleinfelder, W.J., *Splash 2: FPGAs in a Custom Computing Machine,* IEEE Computer Society Press, California, 1996.
10. Vuillemin, J., et al., Programmable active memories: reconfigurable systems come of age, *IEEE Trans. on VLSI Systems*, 4, 1, 56, 1996.
11. Moll, L. and Shand, M., Systems performance measurement on PCI Pamette, in *IEEE Symp. on FPGAs for Custom Computing Machines*, Pocek, K.L. and Arnold, J., Eds., Napa Valley, CA, p. 125, 1997.

12. Cockshott, W.P., Barrie, P., McCaskill, G., and Milne, G.J., Realising massively concurrent systems on the SPACE Machine, in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, Buell, D. and Pocek, K., Eds., IEEE Computer Society Press, 1993.
13. Milne, G.J., Reconfigurable custom computing as a supercomputer replacement, in *Proc. 4th International Conference on High-Performance Computing*, Bangalore, India, p. 260, Dec. 1997.
14. Morris, J., Bundell, G.A., and Tham, S., A reconfigurable processor for Petri net simulation, in *Proc. HICSS-33*, El-Rewini, H. and Helal, S., Eds., Maui, HI, 2000.
15. Tham, S., Achilles: High bandwidth, low latency interconnection for parallel processors, PhD Thesis, Electrical and Electronic Engineering, University of Western Australia, 2001.
16. Motorola Semiconductor Products, MPC500, MPC800 microprocessors, http://e-www.motorola.com/index.html, 2001.
17. Woodfill, J., von Herzen, B., and Zabih, R., Real-time stereo vision on the PARTS reconfigurable computer, in *5th IEEE Symp on FPGAs for Custom Computing Machines,* Pocek, K.L. and Arnold, J., Eds., Napa Valley, CA, p. 201, 1997.
18. Woodfill, J., von Herzen, B., and Zabih, R., Frame-rate robust stereo on a PCI board, http://www.cs.cornell.edu/rdz/Papers/Archive/fpga.pdf, 1998.
19. Piacentino, M.R., van der Wal, G.S., and Hansen, M.W., Reconfigurable elements for a video pipeline processor, in *7th IEEE Symp on Field-Programmable Custom Computing Machines,* Pocek, K.L. and Arnold, J., Eds., p. 82, 1999.
20. Shand, M. and Vuillemin, J., Fast implementation of RSA cryptography, in *Proc 11th IEEE Symposium on Computer Arithmetic,* Windsor, Ontario, 1993.
21. Schneier, B. et al., Twofish: a 128-bit block cipher, http://www.counterpane.com/twofish-paper.html, 2001.
22. Elbirt, A.J. et al., An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists, in *The Third Advanced Encryption Standard Candidate Conference*, New York, April 13–14, 2000.
23. Chodowiec, P., Khuon, P., and Gaj, K., Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining, in *9th ACM Intl Symp on Field-Programmable Gate Arrays*, Schlag, M., Ed., Feb. 2001.
24. Paar, C. et al., An algorithm-agile cryptographic co-processor based on FPGAs, in *Reconfigurable Technology: FPGAs for Computing Applications, Proc. SPIE*, Schewel, J. et al., Eds., 3844, p. 11, 1999.
25. Huang, W.-J., Saxena, N., and McCluskey, E.J., A Reliable LZ data compressor on reconfigurable coprocessors, in *8th IEEE Symposium on Field-Programmable Custom Computing Machines*, Pocek, K.L. and Arnold, J., Eds., p. 175, 2000.
26. Andraka, R., A survey of CORDIC algorithms for FPGA based computers, in *Proc 6th Intl Symp on Field Programmable Gate Arrays*, Kaptanoglu, S., Ed., p. 191, 1998.
27. Hoang, D.T., Searching genetic databases on Splash 2, in Buell, D.A., Arnold, J.M., and Kleinfelder, W.J., *Splash 2: FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, California, 1996.
28. Gunther, B.K., Milne, G.J., and Narasimhan, L., Assessing document relevance with run-time reconfigurable machines, in *4th IEEE Symposium on FPGAs for Custom Computing Machines*, p. 10, 1996.
29. Bergmann, N.W. and Dawood, A., Adaptive interfacing with reconfigurable computers, in *Proc Australasian Computer Systems Architecture Conference*, Heiser, G., Ed., p. 11, 2001.
30. DeHon, A., DPGA utilization and application, in *Proc ACM/SIGDA 4th International Symposium on Field Programmable Gate Arrays*, Ebeling, C., Ed., 1996. Extended version available via anonymous FTP transit.ai.mit.edu:transit-notes/tn129.ps.Z
31. Vasilko, M. and Ait-Boudaoud, D., Optically reconfigurable FPGAs: Is this a future trend?, in *Proc 6th Intl Workshop of Field-Programmable Logic and Applications*, Darmstadt, LNCS, Hartenstein, R.W., and Glesner, M., Eds., 1142, p. 270, 1996.

32. Caspi, E., Chu, M., Huang, R., Yeh, J., Markovskiy, Y., Wawrzynek, J., and André DeHon, A., Stream computations organized for reconfigurable execution (SCORE), in *10th Intl Conference on Field Programmable Logic and Applications, LNCS*, Hartenstein, R.W. and Gruenbacher, H., Eds., p. 1896, 2000 also http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html.
33. Callahan, T.J., Hauser, J.R., and Wawrzynek, J., The Garp architecture and C compiler, *IEEE Computer*, 33, 4, 62, 2000.
34. Callahan, T.J. and Wawrzynek, J., Instruction level parallelism for reconfigurable computing, in *8th International Workshop Field-Programmable Logic and Applications, LNCS*, Hartenstein, R. and Keevalik, A., Eds., p. 1482, 1998.
35. Tsu, W. et al., in *Proc. 7th International Symposium on Field Programmable Gate Arrays*, Trimberger, S., Ed., 1999.
36. Diessel, O. and Milne, G., Compiling process algebraic descriptions into reconfigurable logic, in *Proc 15th IPDPS Workshops, LNCS 1800*, Rolim, J. et al., Eds., p. 916, 2000.

## 21.2 Using Configurable Computing Systems

*Danny F. Newport and Don Bouldin*

### 21.2.1 Introduction

Configurable computing systems use reprogrammable logic components, which are now capable of providing more than 10 million logic gates on a single chip. These systems can be reconfigured at runtime, enabling the same hardware resource to be reused depending on its interaction with external components, data dependencies, or algorithm requirements.

In essence, a specialized instruction set and arithmetic units can be configured as desired by an application designer on an as-needed basis to achieve optimal performance. The location of configurable components within a computing system is one of the keys to achieve maximum efficiency and performance. A variety of architectures driven by the location of configurable components are described in a later section.

### 21.2.2 Configurable Components

Before describing various architectures of configurable computing systems, an understanding of the internal structure of FPGAs, the major component of a configurable computing system is necessary. In 2006, the top two FPGA vendors were Altera Corporation and Xilinx, Inc. The FPGA products from these, and other vendors, differ in their internal structure and programming. However, the basic internal structure for FPGAs can be illustrated as shown in Fig. 21.7. Note that the PLB blocks are programmable logic blocks and the PI blocks are programmable interconnect. A current trend among the FPGA vendors is to also include RAM or a fixed microprocessor core on the same integrated circuit as the FPGA. This enables even greater system flexibility in a design.

The means by which the logic blocks and interconnect are configured for specific functions are of a proprietary nature and specific to each vendor's FPGA families. In general terms, the logic blocks and interconnect have internal structures that "hold" the current configuration and when presented with inputs will produce the programmed logic outputs. This configuration is FPGA specific and contained in a vendor-specific file. A specific FPGA is programmed by downloading the information in this file through a serial or parallel logic connection.

The time required to configure an FPGA is known as the configuration time. Configuration times vary based on the FPGA family and the size of the FPGA. For a configurable computing system composed of several FPGAs, the configuration time is based not only on the configuration time of the individual FPGAs but also on how all the FPGAs are configured. Specifically, the FPGAs could be configured serially, in parallel, or a mixture of serial and parallel depending upon the design of

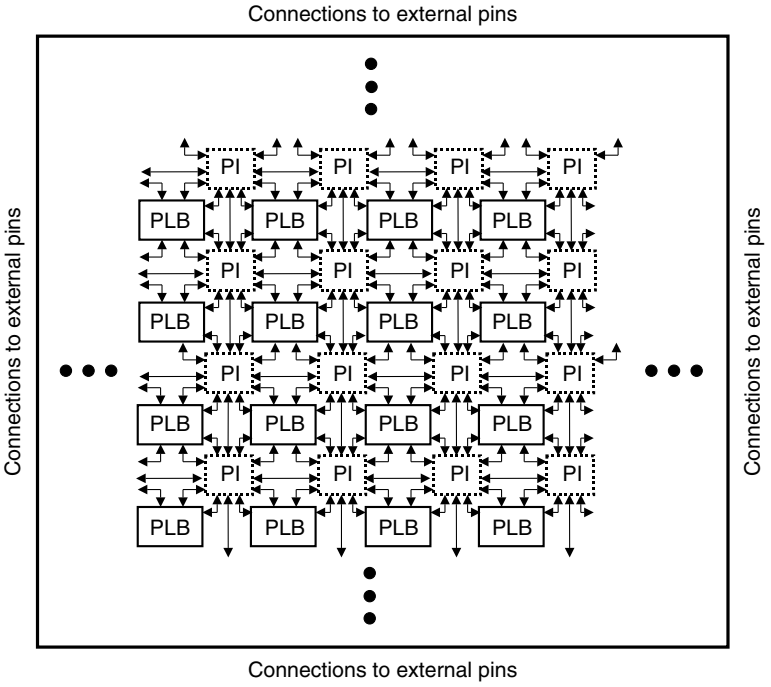



**FIGURE 21.7** Basic internal structure of an FPGA.

the system. Thus, this time can vary from hundreds of nanoseconds to seconds. This directly impacts the types of applications that have improved performance on a particular configurable computing system. A configurable computing system that has a configuration time on the order of seconds is best suited for applications that do not require reconfigurations "on the fly," i.e., applications with a single configuration associated with them or the ones that are pipelined with slow pipeline stages. On the other hand, a configurable computing system that has a very short configuration time can be used for the same applications as one with a slower configuration time and applications that require on-the-fly reconfiguration.

As implied previously, the configurable component of a configurable computing system can be composed of a single FPGA or multiple FPGAs. Many architectures are used for a configurable component composed of multiple FPGAs. Figure 21.8 illustrates the basic architectures from which most of these architectures would be derived. Note that these architectures are very similar, or identical, to those used for parallel processing systems. As a matter of fact, many of the paradigms used in configurable computing systems are derived from parallel processing systems. In many cases, a configurable computing system is the hardware equivalent of a software parallel processing system. Figure 21.8a is a pipelined architecture with the FPGAs hardwired from one to the other. This type of architecture is well suited for functions that have streaming data at specific intervals. Note that variations of this architecture include pipelines with feedback, programmable interconnect between the FPGAs, and RAM associated with each FPGA. Figure 21.8b is an array of FPGAs hardwired to their nearest neighbors. This type of architecture is well suited for functions that require a systolic array. Note, as with the pipelined architecture, that variations of this architecture include arrays with feedback, programmable interconnect between the FPGAs, and RAM associated with each FPGA. Also note that an array of FPGAs is very similar to the internal structure of a single FPGA. Thus, one has a hierarchy of configurability.

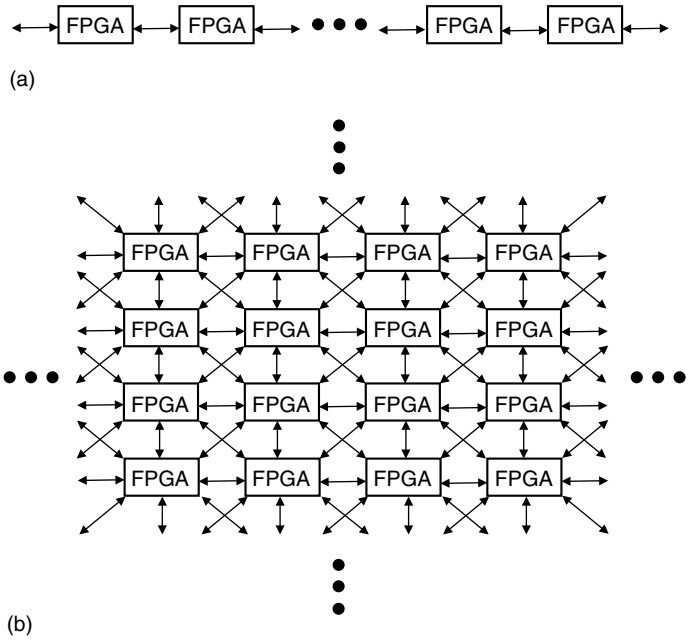



**FIGURE 21.8** Basic architectures for multiple FPGAs.

### 21.2.3 Configurable Computing System Architectures

The placement of one or more configurable components within a computing system is largely determined by the requirements of the application. Several architectures are shown in Fig. 21.9. In some cases as shown in Fig. 21.9a, no additional computing power is required and the component can be utilized in a stand-alone mode. This situation occurs in Internet routing nodes and in some data acquisition

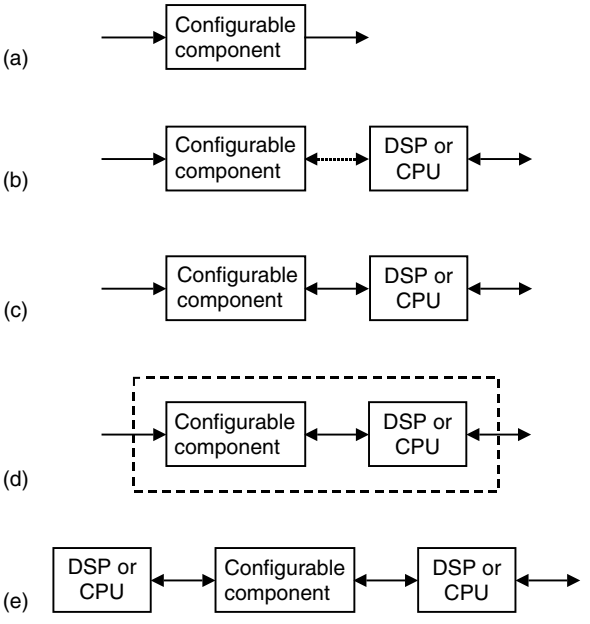



**FIGURE 21.9** CPU/configurable computing architectures.

systems as well as controllers for actuators. Note that the use of FPGAs to replace logic or to be used as state machines is a typical application of this type of architecture. This type of application was the first widespread use of FPGAs.

For configurable computing systems, configurable components are more commonly coupled with conventional DSPs or CPUs such that the processor can accomplish general-purpose computing while acceleration of specialized functions can be performed by the configurable components. The type of general-purpose computing required by the application determines the choice of a DSP or CPU. An application involving signal processing would naturally lead to the use of a DSP. Whereas, an application involving user interaction or user services (disks, etc.) would more likely lead to the use of a general purpose CPU. For this discussion on general configurable computing system architectures, the type of general-purpose processor used is irrelevant; however, it is very relevant when an actual application and system are being developed.

Figures 21.9b–e depict architectures that have configurable components coupled with DSPs or CPUs. The communication requirements between the different types of processors determine the amount of bandwidth and latency provided. If infrequent communication is needed, a serial line or some other slow-speed connection may be sufficient as shown in Fig. 21.9b. For higher bandwidth applications, placing the two types of components on a bus or some other high-speed connection as shown in Fig. 21.9c may be appropriate. In both of these cases, tasks best suited for a particular component can be delegated to that component and sharing of data and results are facilitated. Figure 21.9d depicts the tightest coupling with the lowest latency and highest bandwidth since both types of components are placed inside the same package. Often, the DSP or CPU manages the data, especially when disk storage is involved. When the data is being acquired at a high rate from a sensor, the configurable component is often used to perform initial operations to reduce the size of the data. Thus, the DSP or CPU has only a fraction of the data to be processed or stored. Note that the current trend to include RAM and a fixed microprocessor core on the same integrated circuit as the FPGA is an implementation of this architecture. Another variation of this theme of placing the configurable component within the system just where it is needed can be seen in a network of workstations as shown in Fig. 21.9e. In this case, the configurable component can be inserted into the router itself to perform dedicated operations as the data is passed from processor node to processor node. The processing performed in this manner appears to be "free" as it occurs during message passing.

## 21.2.4 Selected Applications

Several classes of applications have improved performance when implemented on configurable computing systems including image analysis, video compression, and molecular biology. In general, these applications exploit the parallel processing power of configurable computers. Current applications that exert high demand for reconfigurable systems include communications and mobile systems, which utilize FPGAs to provide more flexible operations than ASICs yet higher speed and lower power than CPUs. Applications that can benefit from variable-grain parallelism of FPGAs are hot prospects to emerge as high-volume applications in the near future, especially as improvements in data movement are made.

### 21.2.4.1 Image Analysis

Image analysis requires manipulating massive amounts of data in parallel and performing a variety of data interaction (e.g., point operations, neighborhood operations, and global operations). Many of these operations are ideally suited for implementation on a configurable computing system due to the parallel nature of the operations. Example implementations are image segmentation, convolution, automated target recognition (ATR), and stereo vision. Image segmentation is often the first step in image analysis and consists of extracting important image features. For intensity images (i.e., those represented by point-wise intensity levels), the four popular approaches are threshold techniques, edge-based methods, region-based techniques, and connectivity-preserving relaxation methods. A systolic array of configurable components, similar to that depicted in Fig. 21.8b, can be used to perform an edge-based image

segmentation. Implementation results for various applications have shown that this approach is superior to the conventional digital signal processor approach.

Two-dimensional convolution is commonly used for filtering, edge detection, and feature extraction. The basic idea is that a window of some finite size and shape is scanned across the image. The output pixel value is the weighted sum of the input pixels within the window where the weights are the values of the filter assigned to every pixel of the window itself. Using a systolic array of configurable components, the convolution window can be applied in parallel with an output pixel value being produced as new pixel values are provided. The storage of intermediate pixel values within the window is inherent in the systolic array structure. Implementation results have shown impressive performance gains.

ATR is a computationally demanding application in real-time image analysis problems. The objective of an ATR system is to analyze a digitally represented input scene to locate or identify all objects of interest automatically. Typically, algorithms begin with a preprocessing step to identify regions of interest in the input image. Next, template matching is performed to correlate these regions of interest with a very large number of target templates. The final identification step identifies the template and the relative offset at which peak correlation occurs. The template matching process is the most computationally intensive among these three steps and has the potential of being implemented in a parallel form. Therefore, the template matching is a good candidate to be mapped into a configurable computing system. Implementation results have shown significant performance improvements.

Stereo vision involves locating the same features in each of two images and then measuring the distances to objects containing these features by triangularization. Finding corresponding points or other kinds of features in two images, such that the matched points are the same projections of a point in the scene, is the fundamental computational task. Matching objects at each pixel in the image leads to a distance map. This is very similar to the template matching process in an ATR system and implementation results are similar.

#### 21.2.4.2 Image and Video Compression

Image and video compression are used in many current and emerging products. Image compression is widely used in desktop publishing, graphic arts, color facsimile, and medical imaging. Video compression is at the heart of digital television set-top boxes, DSS, HDTV decoders, DVD players, video conferencing, Internet video, and other applications. Compression reduces the requirements for storage of large archived pictures, less bandwidth for the transmission of the picture from one point to another, or a combination of both. Image and video processing typically require high data throughout and computational complexity. JPEG is widely used for compressing still pictures, and MPEG or wavelets are more appropriate for compressing videos or general moving pictures.

A configurable component using a pipelined approach, as depicted in Fig. 21.8a, provides a much cheaper and more flexible hardware platform than special image compression ASICs, and it can efficiently accelerate desktop computers. A speed improvement over a modern workstation of a factor of 10 or more can be obtained for JPEG image compression.

#### 21.2.4.3 Molecular Biology

Scanning a DNA database is a fundamental task in molecular biology. This operation consists of identifying those sequences in the DNA database that contain at least one segment sufficiently similar to some segment of a query sequence. The computational complexity of this operation is proportional to the product of the length of the query sequence and the total number of nucleic acids in the database. In general, segment pairs (one from a database sequence and one from query sequence) may be considered similar if many nucleotides within the segment match identically. This similarity search may take several hours on standard workstations when using common software that is parameterized for high sensitivity.

One method of performing DNA database searches is to use a dynamic programming algorithm for computing the edit distance between two genetic sequences. This algorithm can be implemented on a configurable computing system configured as two systolic arrays. Execution has been found to be several

orders of magnitude faster than implementations of the same algorithm on a conventional computer. Another method is to use a systolic filter for speeding up the scan of DNA databases. The filter can be implemented on a configurable computing system, which acts as a coprocessor that performs the more intensive computations occurring during the process. An implementation of this system boosted the performances of the conventional workstation by a factor ranging from 50 to 400.

### 21.2.5 Virtual Computing Power

Quantifying computing power is a challenging task due to differing computing architectures and applications. Vuillemin et al. [13] define virtual computing power based on the number of programmable active bits (PABs) and the operating frequency. He defines a "reference" PAB as a 4-input Boolean function. These functions are essentially the core configurable elements of a configurable computing component; however, each vendor defines them differently. For example, Xilinx calls them logic cells (LCs) and organizes them into groups of four called configurable logic blocks (CLBs). Whereas, Altera calls them logic elements (LEs) and organizes them into groups of 10 called logic array blocks (LABs). As newer and larger FPGAs with new architectures are constructed, the vendors will likely rename these logic blocks. But the configurable blocks can always be defined as Boolean functions.

### 21.2.6 Development Systems

Developing applications for microprocessor-based systems is currently far easier than developing applications for a configurable computing system. Microprocessor development systems have been optimized over years, even decades, while those for configurable computing systems are in their infancy. A design for a single FPGA is typically created using tools similar to those used for other digital systems, tools such as schematic capture, VHDL, etc. Owing to the proprietary nature of FPGAs, however, a designer must typically use the design tools available from the FPGA vendor.

A software design environment that facilitates the rapid development of applications on configurable computing systems should permit high-level design entry, simulation, and verification by application designers who need not be familiar with the details of the hardware. Of course, on occasions, it may be necessary to expose to the application designer those hardware details deemed essential to ensure feasible and efficient implementations. Metrics and visualization are desirable to assist the application designer in achieving near-optimal system implementation rapidly. The tools available from the FPGA vendors are currently intended for digital systems designers and not the application designer. Research efforts under way at various universities and start-up companies are producing the first development systems for configurable computing systems similar to those for microprocessor systems.

## Glossary

**Configurable computing systems:** Systems that use reprogrammable logic components, typically field-programmable gate arrays (FPGAs), to implement a specialized instruction set and arithmetic units to improve the performance of a particular application. These systems can be reconfigured, enabling the same hardware resource to be reused depending on its interaction with external components, data dependencies, or algorithm requirements.

**Configuration time:** Time required to program an FPGA or configurable computing system with a given configuration. This time varies from hundreds of nanoseconds to seconds, depending on the system and the FPGAs that are used in the system.

**Field-programmable gate array:** Integrated circuit containing arrays of logic blocks and programmable interconnect between these blocks. The logic blocks can be configured to implement simple or complex logical functions and can be changed as required. Example functions are registers, adders, and multipliers. The programmable interconnect permits the construction of even more complex functions or systems.

**FPGA:** Acronym for field-programmable gate array.

**Reconfigurable computing systems:** Alternate term for configurable computing systems. This term is usually used to indicate that the system can be reconfigured at any time for some desired function.

## To Probe Further

More in-depth information on configurable computing systems is readily available. The first sources of information are the FPGA vendors. A few of these are

1. Altera Corporation, San Jose, CA. http://www.altera.com.
2. Atmel Corporation, San Jose, CA. http://www.atmel.com.
3. Xilinx, Inc., San Jose, CA. http://www.xilinx.com.

Several sites on the World Wide Web are dedicated to configurable computing systems. A search using the terms configurable computing systems or reconfigurable computing systems via any of the search engines will yield a great number of hits. One of these sites is http://www.optimagic.com that provides information on not only configurable computing systems but also programmable logic in general.

Currently, few books focus specifically on configurable computing systems; however, many books about programmable logic provide excellent references for someone interested in configurable computing. Some of these are the following:

4. J. Hamblen, T. Hall and M. Furman, *Rapid Prototyping of Digital Systems*, ISBN 0-387-27728-5, Springer, NY, 2006.
5. M. Gokhale and P. Graham, *Reconfigurable Computing*, ISBN 0-387-26105-2, Springer, NY, 2005.
6. C. Maxfield, *The Design Warrior's Guide to FPGAs*, ISBN 0-750-67604-3, Elsevier, Burlington, MA, 2004.
7. W. Wolf, *FPGA-Based System Design*, ISBN 0-131-42461-0, Prentice-Hall, Englewood Cliffs, NJ, 2004.

Many excellent conferences are held annually to provide the latest information on configurable computing systems from the FPGAs to development systems. Some of these conferences are as follows:

8. Symposium on Field-Programmable Custom Computing Machines (FCCM), http://www.fccm.org.
9. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, (FPGA). http://www.Isfpga.org.
10. International Workshop on Field Programmable Logic and Applications (FPL), http://www.fpl.org.
11. Reconfigurable Architectures Workshop (RAW), http://www.ece.lsu.edu/vaidy/raw06/.
12. Design Automation Conference (DAC), http://www.dac.com.

The proceedings from these conferences contain many articles on not only configurable computing systems but also applications for which configurable computing systems have been shown to be effective. The configurable computing systems are applied in other areas such as cryptography, fingerprint matching, multimedia, and astronomy.

More in-depth information on virtual computing power and a list of applications of configurable computing system is as follows:

13. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, Programmable active memories: Reconfigurable systems come of age, *IEEE Trans. VLSI Syst.*, 4(1): 56–69, 1996.

More information on research and development into design tools for configurable computing may be obtained by visiting the Web sites of the research groups involved. Some of these are:

14. Brigham Young University, Configurable Computing Web page, http://splish.ee.byu.edu and the JHDL Web page, http://www.jhdl.org.
15. University of Cincinnati, REACT Web page, http://www.ececs.uc.edu/∼dal/acs/index.htm.
16. Colorado State University, CAMERON Project Web page, http://cs.colostate.edu/cameron.
17. Northwestern University, A Matlab Compilation Environment for Adaptive Computing Systems Web page, http://www.ece.nwu.edu/cpdc/Match/Match.html.

18. University of Southern California, DEFACTO Web page, http://www.isi.edu/asd/defacto.
19. University of Tennessee, CHAMPION Web page, http://microsys6.engr.utk.edu/∼bouldin/darpa.

## 21.3 Xtensa: A Configurable and Extensible Processor

*Ricardo E. Gonzalez and Albert Wang*

### 21.3.1 Introduction

Until a few years ago, processors were only sold as packaged individual ICs. However, the growing density of CMOS circuits created an opportunity for incorporating the processor as part of a larger system on a chip. Initial processor designs for this market were based on the processor existing as a separate entity, and cores were handcrafted for each manufacturing process technology, resulting in costly and fixed solutions. Furthermore, it was not possible to modify these cores for the particular application, in much the same way that it was not possible to modify a stand-alone prepackaged processor.

Xtensa is a processor core designed with ease of integration, customization, and extension in mind. Unlike previous processors, Xtensa lets the system designer select and size only the features required for a given application. The configuration and generation process is straightforward and lets the designer define new system-specific instructions if preexisting features don't provide the required functionality. Furthermore, Xtensa fits easily into the standard ASIC design flow. Xtensa is fully synthesizeable, and designers can use the most popular physical-design tools during the place-and-route process.

### 21.3.2 Processor Development

Application-specific processor development is an active area of research in the CAD, computer architecture, and VLSI design communities. Early attempts to add application-specific instructions to general-purpose computer engines relied on writable micro-code [1,2]. These techniques dynamically augmented the base instruction set with application-specific instructions.

More recent research focuses on automatic instruction set design [3,4] or on reconfigurable, also called retargetable, processors [5]. These groups, however, try to solve slightly different problems than those addressed by Xtensa. Automatic instruction set design systematically analyzes a benchmark program to derive an entirely new instruction set for a given microarchitecture. Our group—here referred to as "we"—focuses on how to generate a high-performance and low-power implementation of a given microarchitecture with application-specific extensions. In this respect, automatic instruction set design is a good complement to our work. Once the instruction set additions are derived automatically by analyzing the benchmark program, they can be given to the Xtensa processor generator to obtain a high-performance, low-power implementation. Reconfigurable or retargetable processors couple a general-purpose computer engine with various amounts of hardware-programmable logic. In the extreme, the entire processor is implemented using hardware-programmable logic. The technique, however, is limited by the large difference in operating frequency between programmable and nonprogrammable logic. Processors implemented entirely using programmable logic operate an order of magnitude slower than nonconfigurable processors implemented in a comparable process technology. Razdan and Smith present an interesting compromise [5]. Their approach couples a custom-designed high-performance processor with small amounts of hardware-programmable logic. Their system uses compiler-generated information to dynamically reconfigure a small amount of hardware-programmable logic to implement new application-specific functional units. This technique also has limitations due to the disparity in operating frequency of programmable and nonprogrammable logic. Thus, the new functional units must be extremely simple or be deeply pipelined.

The authors' approach is similar to that taken by Razdan and Smith except that we don't attempt to dynamically reconfigure the system. The Tensilica processor generator adds the application-specific functionality at the time the hardware is designed. Thus, the extensions are implemented in the

same logic family as the rest of the processor. This eliminates the disadvantages of using programmable logic for implementing the extensions, but precludes modification of the extensions for different applications.

Due to a lack of automated tools, designers incorporated application-specific functionality in CPUs by adding specialized coprocessors [6,7]. This approach introduces communication overhead between the CPU and the coprocessor, making system design more arduous. Recently, with the advent of synthesizeable processors, some groups have proposed manual modification of the register-transfer level (RTL) description of the processor and the software development tools [8]. This approach is tedious and error prone. Furthermore, the extensions are only applicable to one implementation. If users want to add similar extensions to a future implementation of the same processor, they must modify the RTL again.

The authors' research differs from previous studies because we use a high-level language to express processor extension. This language, called Tensilica Instruction Extension (TIE), expresses the semantics and encoding of instructions. TIE can add new functionality to the RTL description and automatically extend the software tools. This lets the system developer code applications in a high-level language, such as C or C++. TIE imposes restrictions on functions that designers can describe, which greatly simplify verification of the processor and extensions. Because the extensions become an integral part of the processor, there is no communication overhead.

### 21.3.3 Overview of Xtensa

We designed the Xtena instruction set architecture (ISA) to allow ease of extension and configuration. Furthermore, the ISA minimizes code size, reduces power dissipation, and maximizes performance.

The Xtensa ISA consists of a base set of instructions, which exist in all Xtensa implementations, plus a set of configurable options. The designer can choose, for example, to include a 16-bit multiply-accumulate option if it is beneficial to the application. The base ISA defines approximately 80 instructions and is a superset of traditional 32-bit RISC instruction sets [10]. The architecture achieves smaller code size through the use of denser encoding and register windows. The ISA defines 24- and 16-bit instruction formats, as opposed to 32-bit formats found in traditional RISC instruction sets. The Xtensa architecture provides a rich set of operations despite the smaller instruction size. These sophisticated instructions, such as single-cycle compare and branch, enable higher code density and improve performance.

The size of Xtensa instructions is encoded in the instruction, enabling 24- and 16-bit instructions to freely intermix at a fine granularity. The 16-bit instructions are a subset of the 24-bit instructions. Thus, the compiler optimization to reduce code size is trivial: replacing 24-bit instructions with their 16-bit equivalent. The compiler can reduce code size without sacrificing performance.

#### 21.3.3.1 Hardware Implementation

We built the first implementation of Xtensa around a traditional RISC five-stage pipeline, with a 32-bit address space. Many other characteristics of the processor implementation, however, are configurable. The configurability and extensibility of the implementation matches those of the architecture. Figure 21.10 shows a high-level block diagram of Xtensa. The base ISA features correspond to roughly 80 instructions. The designer can size or select configurable options, for example, how many physical registers to include in the implementation, or the size of the instruction and data caches. Optional features, shown as medium-gray in the figure, are selections the designer can make, such as whether to include a 16-bit multiply-accumulate functional unit. Optional and configurable functions let the designer select whether to include that feature and also to size it. For example, whether to include data watch-point registers and, if so, how many. Xtensa optionally supports several data formats such as fixed-point and floating-point. Vectra adds a configurable fixed-point vector coprocessor.

Table 21.1 shows a few of the configuration parameters and associated legal values available in the current Xtensa implementation. Unlike conventional processors, Xtensa gives designers a choice regarding the functionality of the processor.

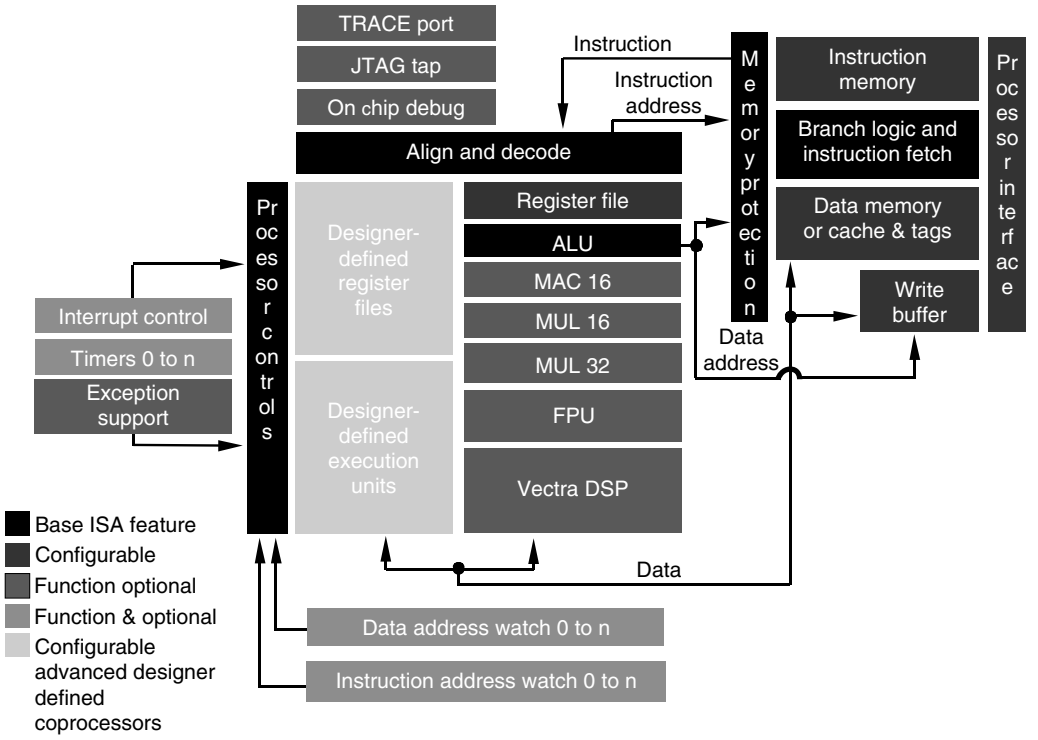



**FIGURE 21.10** Block diagram of Xtensa.

**TABLE 21.1** Xtensa Configuration Parameters

| Parameter | Legal Values |
|---|---|
| Instruction/data cache size | 1–256 KB |
| Instruction/data cache associativity | Direct-mapped, 2-way, 4-way |
| Instruction/data RAM size | 1 KB, 2 KB, 4 KB, 8 KB, 16 KB |
| Instruction/data ROM size | 1 KB, 2 KB, 4 KB, 8 KB, 16 KB |
| Size of windowed register file | 32, 64 |
| Number of interrupts | 0–32 |
| Interrupt levels | 0–3 |
| Timers | 0–3 |
| Memory order | Big-endian, little-endian |

### 21.3.3.2 Configuration

The configuration process begins by accessing the Tensilica processor generator Web page at http://www.tensilica.com. Here, using a standard browser, the designer can select and size the desired features. The site's configuration page gives the designer instant feedback on whether a particular choice will affect the speed, power, or area of the core. The user interface warns the designer of conflicting options or requirements for a particular option.

The designer starts the generation process at the push of a button. The generation process produces the processor's configured RTL description and its configured software development tools. The software tools consist of an ANSI C/C++ compiler, linker, assembler, debugger, code profiler, and instruction set simulator.

The generation process takes approximately one hour to complete. After the process is complete, the designer can download and install the RTL and software development tools. At this point, the designer can either compile an application and measure the performance using the instruction set simulator, or start the hardware synthesis using the RTL description.

The software tools are built on top of the industry-standard GNU tools and include an optimizing C compiler enabling application development in a high-level language. The instruction set simulator and code profiler help the designer quickly identify bottlenecks in application performance. Optionally, the designer can recode the application to work around these bottlenecks or add new instructions to the processor designed to optimize this particular application.

The designer can map the RTL description to a gate-level netlist using industry-standard synthesis tools. Included with the RTL description are a set of synthesis scripts that help automate this process. These scripts let designers quickly obtain a fully optimized gate-level netlist of Xtensa. Tensilica also provides a set of scripts to automate the place-and-route process. It is common for new users to place and route Xtensa within a day or two of downloading the configured RTL.

### 21.3.4 Instruction Set Extension

Hardware designers realized the advantages of extending a general-purpose processor with application specific functional units long time ago [6,7]. Until now, however, the only way to do this was to add the functional units as a coprocessor. This often meant there was some communication overhead between the processor and the application-specific logic. Also, often the coprocessor would require sophisticated control, which had to be implemented with finite state machines or with micro-sequencers.

The Tensilica processor generator provides a more flexible and powerful approach to processor extension. Using TIE the system designer can describe, at a high-level, the functionality of the new functional units. The TIE compiler will then automatically generate an efficient pipelined implementation. The system designer must specify only the functionality of the new hardware and the required (and architecturally visible) storage elements—register files and special-purpose state elements. The pipeline flip-flops and the bypass and interlock detection logic are then automatically generated by the TIE compiler. Furthermore, the TIE compiler (TC) will automatically extend the software tools so that the new hardware is accessible from C/C++.

Using TIE has many advantages over more traditional methods of extension. First, the sophisticated control can now be accomplished using software—making it easier to debug and optimize. Second, the system designer can quickly prototype different design alternatives enabling him (or her) to quickly converge to a good solution to the problem. Third, verification of the new hardware's functionality can be done using the instruction set simulator (ISS), which can simulate hundreds of thousands of instruction per second, rather than on the RTL model, which can only simulate hundreds of cycles per second.

Similar to most previous machine description languages [14], TIE is an instruction set architecture (ISA) description language. It relies on a tool, the TIE compiler, to generate an efficient hardware implementation and required additions to the software tools, including the compiler, ISS, and debugger. TIE is not intended to be a complete processor description language. Instead, the TIE language provides designers simple ways to describe a broad variety of computational instructions, yet allows the TIE compiler to generate efficient hardware. The language is simple enough for a wide range of designers to master, yet general enough to allow description of sophisticated ISAs. The rest of this section describes the capabilities of the TIE language.

TIE lets the designer specify the mnemonic, encoding, and semantics of new instructions. The designer uses a combination of `field`, `opcode`, `operand`, and `iclass` statements to describe the format and encoding of an instruction. The field statement gives a name to a group of bits in the instruction word. The `opcode` statement assigns instruction fields with values. The `operand` statement specifies how an instruction's operand is encoded in an instruction field. The `iclass` statement describes the assembly format for an instruction and lists the input and output operands of the instruction. A large set of predefined instruction fields and operands (used to describe the base Xtensa ISA) can be used directly in the TIE description. The following example describes two

instructions: A4 and S4. These instructions take two 32-bit operands from the core register file, perform four 8-bit additions and subtractions and store the result back to the core register file:

```
opcode   A4   op2 = 0 CUST0
opcode   S4   op2 = 1 CUST0
iclass   RR   {A4, S4} {out arr, in ars, in art}
```

The first two lines define the opcodes for `A4` and `S4` as sub-opcodes of a previously defined opcode `CUST0` with the addition of field `op2` equal to 0 and 1, respectively. The third line makes use of the pre-defined register operands arr, ars, and art, and defines two new assembly instructions,

```
A4 arr, ars, art
S4 arr, ars, art
```

#### 21.3.4.1 Customized Datapath

The computational part of an instruction is specified in a TIE `reference` block. The syntax of a `reference` block is very similar to the Verilog hardware description language. The variables used in the `reference` block are predefined (if they appear in the iclass statement), or locally declared variables. The `reference` block for the `A4` and `S4` instructions defined in the previous section are shown below,

```
reference A4 {
  assign arr = {
  ars[31:24] + art[31:24],
  ars[32:16] + art[23:16],
  ars[15:8] + art[15:8],
  ars[7:0] + art[7:0] };
}
reference S4 {
  assign arr = {
  ars[31:24] - art[31:24],
  ars[32:16] - art[23:16],
  ars[15:8] - art[15:8],
  ars[7:0] - art[7:0]};
}
```

The reference description for the two instructions is simple and direct, yet may not result in the best hardware implementation. For example, the logic for addition and subtraction could be shared between the two instructions. TIE allows the designer to describe this high-level hardware sharing between multiple instructions using the `semantic` block. The `semantic` block is similar to a `reference` block but allows multiple instructions to be described at the same time. The semantics of `A4` and `S4`, for example, can be described as follows:

```
{semantic add sub {A4, S4}}
  assign arr = {
  ars[31:24] + (S4 ? ~art[31:24] : art[31:24]) + S4,
  ars[32:16] + (S4 ? ~art[23:16] : art[23:16]) + S4,
  ars[15:8] + (S4 ? ~art[15:8] : art[15:8]) + S4,
  ars[7:0] + (S4 ? ~art[7:0] : art[7:0]) + S4};
}
```

The `semantic` statements allow more efficient hardware implementation, while the `reference` statements are easier to write, are better suited for inclusion in documentation and are a better source

for simulation code. Thus, TIE allows instructions to have either a `reference` block, a `semantic` block, or both. Most often, designers will write the reference description first. Once they have verified the correctness and usefulness of the instruction they write the semantics to optimize the hardware implementation. TIE allows formal equivalence checking between the semantic and reference description to ensure the implementation is identical.

#### 21.3.4.2 Multi-Cycle Instructions

To keep up with the speed of Xtensa, which is pipelined and runs at a high clock rate, instructions with complex computation may require multiple cycles to complete. Writing and verifying multi-cycle instructions is a challenging task for designer unfamiliar with the processor's pipeline, especially if the designer must add the appropriate data-forwarding and interlock detection logic. TIE provides a `schedule` statement that alleviates this problem. The schedule statement captures the timing requirements of the instruction. The designer can then rely on the TIE compiler to derive the implementation automatically. For example, a multiply-accumulate (MAC) instruction that performs the following operation: `acc = acc + (a*b)` typically requires at least two cycles in a pipelined processor. In order to achieve one MAC per cycle throughput, the hardware must use (read) the `a` and `b` operands at the beginning of the first cycle, use `acc` at the beginning of the second cycle, and produce a new acc at the end of the second cycle. The timing of the instruction can be described in TIE as

```
schedule MAC_SCHEDULE {MAC} {
  use a = 1;
  use b = 1;
  use acc = 2;
  def acc = 2;
}
```

The rest of the implementation, including the efficient insertion of pipeline registers, interlock detection, result bypassing, and generation of good code schedules are all handled automatically by the TIE compiler.

#### 21.3.4.3 Register Files and State Registers

When adding new application-specific datapaths it is often necessary to add new storage elements. Two main reasons exist for adding new storage elements. First, algorithms often require specific bit widths, which may not be efficiently supported by the core register file. And second, some algorithms require higher bandwidth than the core register file provides. In the MAC instruction described in the previous subsection, for example, the machine would require a new state register to hold the value of the accumulator. Otherwise it would require an additional read port in the core register file (the accumulator value would be held in a register). Furthermore, the algorithm may require an accumulator value with more precision.

TIE states are extensions to the software visible programming model. They allow instructions to have more sources and destinations than provided by the read and write ports of the core register file. They can also be used as dedicated registers holding temporary values during program execution. When an application needs a large number of such sharable TIE states, it becomes more efficient to group the state into a register file and rely on the C compiler to assign the variables to register entries.

Describing instructions that use TIE states is simple. The designer must specify, in the `iclass` of the instruction, how the state is used. The state variable is then available in the instruction's `reference` and `semantic` blocks. The following example is a complete description of the MAC instruction,

```
state   acc  40    /* a 40-bit accumulator */
opode   MAC  op2 = 0 CUST0
iclass  MAC_CLASS {MAC} {in ars, in art} {inout acc}
reference {
  assign acc = (ars * art) + acc;
}
```

Using a register file involves one more step: describing the register operands. The following TIE code, for example, adds a 24-bit register file:

```
regfile     GR 24 16    /* 26 entries, 24-bits each*/
operand     gr r {GR[r]}
operand     gs s {GR[s]}
operand     gt t {GR[t]}
```

The three register operands use predefined instruction fields (`r`, `s`, `t`) as indices to access the register file contents. An instruction that uses these operands can be describes as,

```
iclass RF{AVE} {out gr, in gs, in gt}
```

Using TIE it is possible to very quickly develop sophisticated hardware that can significantly enhance the application performance of the processor. Furthermore, the new hardware is easily accessible to the C/C++ programmer.

#### 21.3.4.4 Software Support

One key advantage of TIE is that it allows hardware and software to be extended together. This allows the programmer to access the new hardware from C or C++. This allows the programmer to focus on algorithmic optimization, rather than mapping the algorithm to a fixed processor architecture. Programmers often spend more time designing how to map the algorithm's data-types to the processor data-types than they do on optimizing the algorithm for their application. Using TIE it is possible to extend the hardware and software together so the mapping of the algorithm's data-types is more natural.

In order for this extension to be useful to the programmer, however, it must be complete. The compiler, assembler, simulator, debugger, real-time operating system, and application libraries must be extended to use the new hardware datapaths. The TIE compiler generates dynamically loadable libraries (DLLs) that are used to configure the software tools at runtime. Generation of the DLLs takes less than a minute (even for large TIE descriptions). This allows designers to quickly make changes to the TIE description and evaluate the performance of the system. TIE allows designers (or programmers) to define new C data-types that are mapped to TIE register files. The programmer must also specify, in the TIE description, instruction sequences to load and store these data-types from (to) memory. The programmer can then use these new data-types in C/C++ as if they were built-in data-types. Operations are described via instrinsics (every TIE instruction is available as an intrinsic in C/C++) but register allocation, variable saves and restores, addressing arithmetic, and control flow generation for the new data-types are handled automatically by the C compiler. The C compiler is also aware of any side-effects and pipelining of TIE instructions so it can efficiently schedule the instructions.

The TIE compiler also generates libraries to save and restore processor state on a context switch. The compiler uses the instruction sequences described by the programmer to load and store the new data-types. The libraries are used by commercial operating systems, such a WindRiver's VxWorks™. The operating system is delivered as a pre-built binary with hooks to call the context switch code generated by the TIE compiler.

The TIE compiler must also add knowledge of the new instructions and register files to the instruction set simulator and the debugger. The TIE compiler translates the reference block of each instruction to a C implementation that can be used by the simulator to model the execution of the instruction. The TIE compiler also extends the debugger to allow visualization of new register files and state registers.

### 21.3.5 Application Examples

#### 21.3.5.1 DES

To demonstrate the potential of TIE, we extended Xtensa to improve the performance of the Data Encryption Standard (DES)—a popular encryption and decryption algorithm often used for secure Internet communication. We chose DES for two reasons: its growing popularity in embedded applications

that require secure Internet transactions, and the relatively poor encryption and decryption performance of general-purpose processors.

A simple DES modification, known as Triple-DES, extends the key to 168 bits by iterating the DES algorithm three times with three different keys. Triple-DES has been specified as an encryption algorithm for both the secure shell tools [11] and the Internet protocol for security [12]. Both of these applications require high-speed encryption and decryption and are implemented as part of many of today's interesting embedded systems.

The DES algorithm requires extensive bit permutations, which are difficult to implement efficiently in software. However, designers can efficiently implement these permutations in hardware, since each corresponds to a simple renaming of the wires. The algorithm also specifies rotation on 28-bit boundaries. Even if the processor has a rotate instruction, it often is not usable since it most likely rotates on 32-bit boundaries. Finally, the algorithm requires bit packing, unpacking, and table lookups. These operations are slow in software but easy to implement with hardware. We modified the Xtensa processor to include special instructions to speed up these operations.

Based on run-time profile information, we defined four new instructions and reimplemented the application to use these instructions. We verified the implementation of the TIE instructions by comparing the output of the modified application, which uses the TIE-generated C description of the instructions, with the results of a reference implementation of DES written completely in C. In addition to four new instructions, we also added three new state registers to the processor. The registers hold intermediate values during the encryption and decryption process. Of the four new instructions, one performs the encryption and decryption step using values in the state registers. The other three instructions transfer data to (and from) the processor registers from (and to) the state registers, while concurrently permuting the data values. When compiled for the Xtensa architecture, the new application required only 154 bytes of object code and no static or dynamic data storage. Thus, the original implementation required 36 times more memory than this implementation.

Figure 21.11 shows the speedup of the DES-enhanced Xtensa core compared to an unmodified Xtensa core. The *X*-axis shows the block size used for encryption and decryption. The original DES implementation gains much of its speed by precomputing large tables of values from a fixed key, making key changes very expensive. Thus, small blocks can attain speedup by a greater factor than large blocks (where key changes are less frequent). The modified Xtensa can encrypt and decrypt data at the rate of 377 MB/s. The hardware cost of the TIE instructions is roughly 4,500 equivalent (NAND2) gates

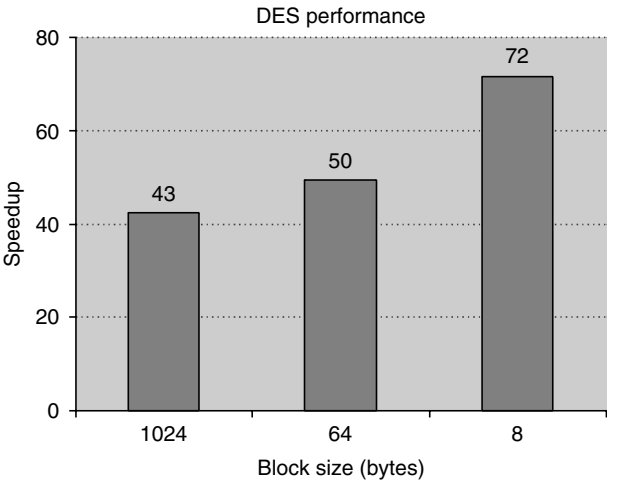



**FIGURE 21.11** DES speedup using TIE.

(measured in a 0.25-$\mu$m process technology). The reduced storage requirements of the application offset this hardware cost. In addition, the new TIE instructions did not increase the cycle time of the machine. DES is only one of the applications that can benefit from specialized hardware.

#### 21.3.5.2 Consumer Multimedia

The EEMBC consumer benchmarks contain a representative sample of multimedia applications of interest today. A baseline configuration of Xtensa contains many features suitable for these applications. At 200 MHz operation Xtensa delivers more than 11 times the performance of the reference processor (ST Microelectronics ST20C2™ at 50 MHz). Performance is measured as the geometric mean of the relative number of iterations per second for each algorithm compared to the reference processor; however, when we added instructions for image filtering and color-space conversion (RGB to YIQ and RGB to CYMB) the average performance increased by 17X (193 times faster then the reference). An AMD K6-III+ at 550 MHz, for comparison, is 34.2 times faster then the reference processor. The base configuration was optimized for 200 MHz operation in a 0.18-$\mu$m technology. The processr was configured with 16 KB two-way set associative caches, 256 KB local data RAM, 16-entry store buffer, and 32-bit multiplier. The total area of the processor was 57,600 NAND2-equivalent gates. The optimized TIE code cost an additional 64,100 NAND2-equivalent gates.

#### 21.3.5.3 DSP Telecommunications

The EEMBC "Telemark" benchmark suite includes many kernels representative of DSP applications. The performance of a base Xtensa processor in this suite is comparable to that of other 32-bit microprocessors (2.3 times faster than the reference). Performance was also measured as the geometric mean of the relative number of iterations per second for each algorithm compared to the reference processor (IDT 32334™ – MIPS32™ architecture at 100 MHz). Adding a fixed-point vector co-processor and a few more specialized instructions, the performance of Xtensa increases by 37X, or a speedup of 85.7X compared to the reference processor. The AMD K6-III+ at 550 MHz has a speedup of 8.7 compared to the reference, while a TI DSP (TMS320C6203) running hand-optimized code at 300 MHz has a 68.5 speedup compared to the reference processor. The base Xtensa configuration was also optimized for 200 MHz operation in 0.18-$\mu$m technology with 16 KB two-way set associative caches, and 16-entry write buffer. The vector coprocessor and new TIE instructions add 180,000 thousand NAND2-equivalent gates.

### 21.3.6 Conclusions

Configurable and extensible processors provide significant advantages compared to traditional hard-wired processors. To take full advantage of extensibility, however, requires a methodology that can extend both the hardware and the software together. We showed that TIE provides a methodology for extension this is complete, fast, and robust.

Using TIE can also help reduce design time by simplifing the hardware verification effort and also by allowing a more natural maping of the algorithm to the hardware implementation.

Furthermore, since the control flow is described in software it is much easier to verify and to enhance. We also showed the extension can significantly increase application performance. We showed that for two different set of application kernels an Xtensa procesor with application-specific extension was 20–40 times faster than a high-performance RISC processor.

## Acknowledgments

The authors thank the entire engineering staff at Tensilica. This article reflects their hard work and dedication. The authors are also grateful to Rick Rudell for the development of the DES TIE code and to Michael Carchia who did the EEMBC benchmarking.

## References


1. A. Abd-alla and D. Kartlgaard. Heuristic synthesis of microprogrammed computer architectures. *IEEE Transactions on Computers*, 23(8): 802–807, Aug. 1974.
2. P. Liu and F. Mowle. Techniques of program execution with a writable control memory. *IEEE Transactions on Computers,* 27(9): 816–827, Sept. 1978.
3. F. Haney. *Using a Computer to Design Computer Instruction Sets.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1968.
4. B. Holmer. *Automatic Design of Computer Instruction Sets.* PhD thesis, University of California, Berkeley, CA, 1993.
5. R. Razdan and M.D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. of Micro-27,* Nov. 1994.
6. O. Nishii, F. Arakawa, K. Ishibashi, et al. A 200 MHz 1.2 W 1.4GFLOPS microprocessor with graphics unit. In *IEEE International Solid-State Circuits Conference,* vol. 41, pp. 288–289, IEEE, Feb. 1998.
7. S. Santhanam, A. Baum, D. Bertucci, et al. A low-cost 300 MHz RISC CPU with attached media processor. In *IEEE International Solid-State Circuits Conference,* vol. 41, pp. 298–299, IEEE, Feb. 1998.
8. http://www.arccores.com.
9. S. Hesley, V. Andrade, R. Burd, et al. A 7th-generation x86 microprocessor. In *IEEE International Solid-State Circuits Conference,* vol. 42, pp. 182–183, IEEE, Feb. 1999.
10. J.L. Hennessy and D.A. Patterson. *Computer Architecture A Quantitative Approach.* First edition, Morgan Kaufmann Publishers, San Mateo, CA, 1990.
11. T. Ylonen, T. Kivinen, M Saarinen, et al. SSH protocol architecture. Internet-Draft, August 1998. *draft-ietf-secsh-architecture-02.txt.*
12. S. Kent, R. Atkinson, "Security architecture for the Internet protocol," RFC 2401, Nov. 1998.
13. G. Hadjiyiannis, S. Hanono, S. Devadas. IDSL: An instruction set description language for retargetability. In *Design Automation Conference,* 1997.
14. V. Zivojnovic, et al. LISA-machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing,* 1997.

# 22

# Roles of Software Technology in Intelligent Transportation Systems




Shoichi Washino
*Tottori University*


## 22.1 Background of Intelligent Transportation Systems

Today, one encounters a lot of traffic congestion and hears of people injured or killed by traffic accidents. Moreover, there has been no remarkable improvement in air pollution due to exhaust gases from vehicles inspite of the stringent regulation for vehicles. It is natural that frequent traffic congestion results in lower mean vehicle speed. Figure 22.1 shows an example of mean averaged vehicle speed in Japan. Vehicle speed in urban area like Tokyo and Osaka is very close to that of a bicycle. On the contrary, vehicle speed in countryside is faster than urban area. Therefore, average vehicle speed of the whole of Japan shows a little bit higher value, as shown in Fig. 22.1.

Traffic accidents are a more serious matter than vehicle speed. Figure 22.2 shows an example of the number of people killed due to traffic accidents in Japan. About ten thousand persons are still killed by traffic accidents. Moreover, the death rate of senior people has become higher in Japan.

A third example of traffic problems—the status of air pollution in Japan—is shown in Fig. 22.3. Both hydrocarbon (HC) and carbon monoxide (CO) have decreased gradually. On the contrary, nitric oxide

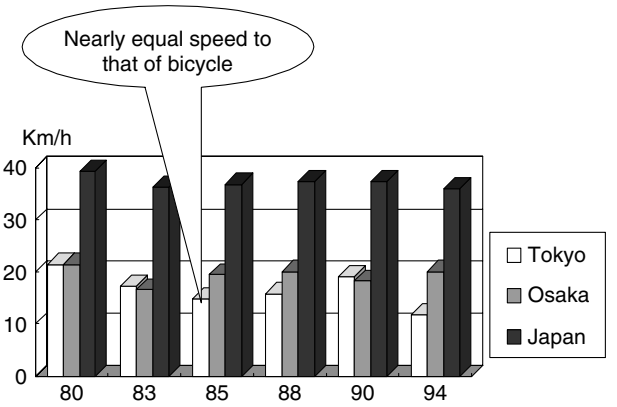



**FIGURE 22.1** Average vehicle speed.

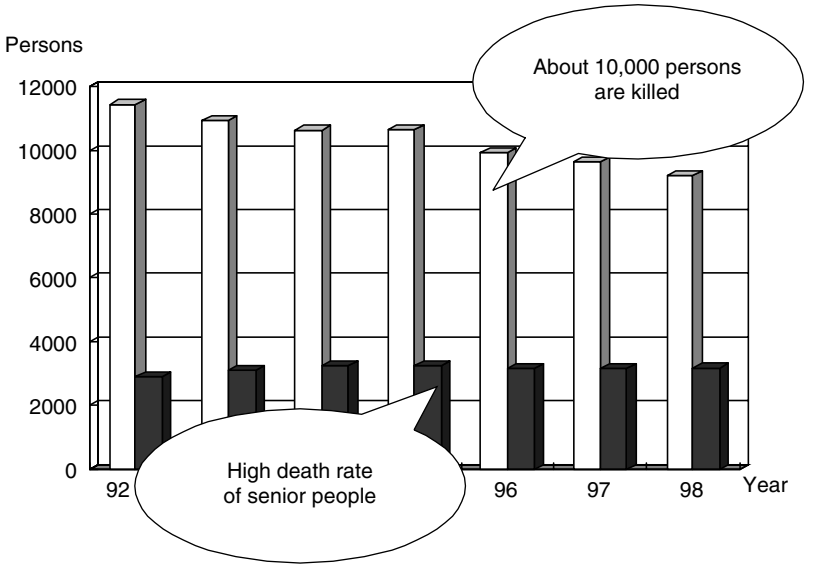



**FIGURE 22.2** Numbers of persons killed by traffic accidents.

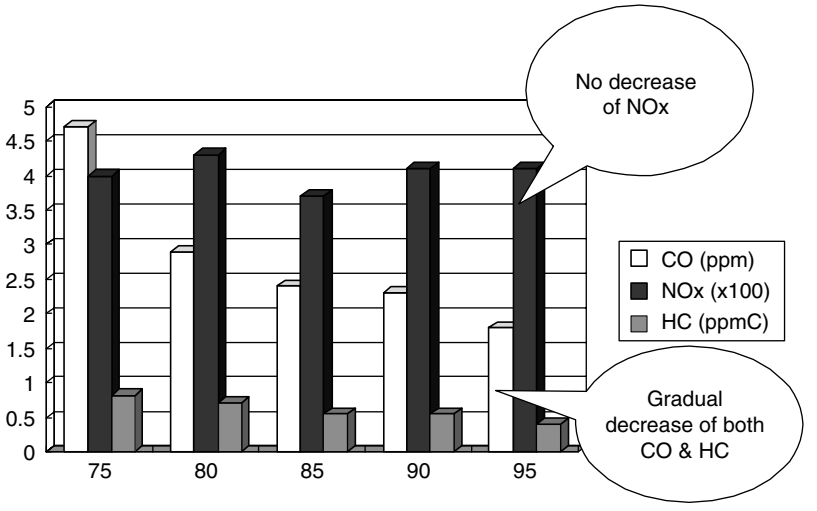



**FIGURE 22.3** Air pollution.

(NOx) has not decreased. It has remained nearly constant in spite of the stringent Japanese exhaust gas regulation. These three examples are related to Japan. But the traffic situation of all countries is very much similar.

In principle, these phenomena are eliminated by constructing new roads because it gives smoother traffic flow. The cost to construct them, however, has become very expensive in every country. So such a conventional way to solve these traffic problems is not available. On the other hand, information technology has progressed remarkably these days. As a result of this, many government officers have embraced an idea to solve these traffic problems by using information technology. Indeed the idea has led to a system image called Intelligent Transportation Systems (ITS), as shown in Fig. 22.4.

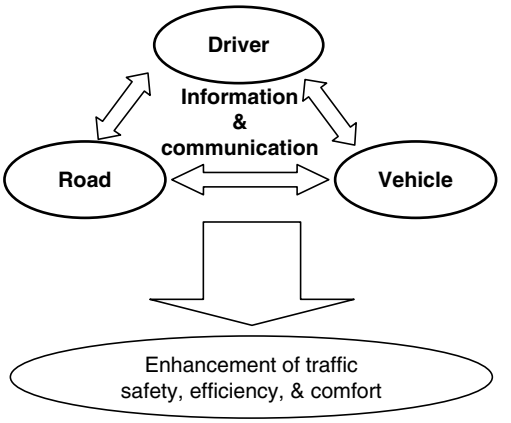



**FIGURE 22.4** Concept of ITS.

## 22.2 An Overview of Japanese ITS and the Nine Developing Fields of Japanese ITS

It was in 1995 that the second ITS World Congress was held at Yokohama. In Japan, several projects on ITS had been carried before 1995. Table 22.1 shows those projects. The first project was called comprehensive automobile traffic control system (CACS) proposed and lead by Ministry of Construction of the Japanese government at that time. Then the Ministry of Transport, Ministry of Post and Telecommunication, Ministry of International Trade and Industry, National Police Agency, and Ministry of Construction proposed several projects on ITS and they had also promoted these projects. These are the major six projects to develop ITS in Japan, as shown in Table 22.1.

VICS means vehicle information & communication system. This mainly provides traffic information such as congestion, road construction, road restriction, traffic accidents, and parking information. VICS supposes an in-vehicle navigation system is loaded in a vehicle; this is already widespread in Japan as shown later.

ETC means electronic toll collection system. One often sees congestion in front of a tollgate due to "stop" and "go" at the tollgate. The aim of this project is to reduce such congestion using communication technologies between vehicle and road infrastructure.

AHS means advanced cruise-assist highway system originally from automated highway system. This system supposes advanced safety vehicle to enhance traffic safety and reduce accidents. This system is a kind of driving support system with collaborating road infrastructure and vehicle.

ASV is advanced safety vehicle to enhance traffic safety and reduce accidents. It has originally aimed to do so without any aid of road infrastructure. But now both ASV and AHS projects are collaborating because the collaboration is more effective from the point of view of both system performance and its cost.

SSVS means super smart vehicle system. Now, its main activity is related to the development of inter-vehicle communication technologies.

**TABLE 22.1** ITS Projects in Japan

- VICS (MOC, NPA, MPT)
  Vehicle Information & Communication System
- ETC (MOC, MPT)
  Electronic Toll collection System
- AHS (MOC)
  Advanced cruise-assist Highway System
- ASV (MOT)
  Advanced Safety Vehicle
- SSVS (MITI)
  Super Smart Vehicle System
- UTMS (NPA)
  Universal Traffic Management System

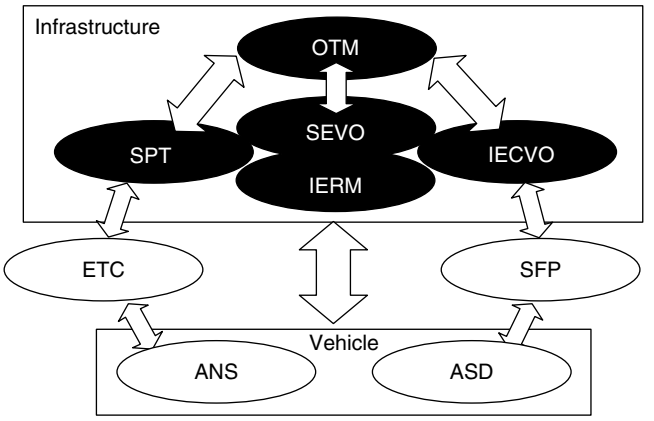



**FIGURE 22.5** Nine fields of Japanese ITS.

**TABLE 22.2** Nine Developing Fields of Japanese ITS

1. Advances in navigation systems (ANS)
2. Electronic toll collection systems (ETC)
3. Assistance for safe driving (ASD)
4. Optimization of traffic management (OTM)
5. Increasing efficiency of road management (IERM)
6. Support for public transport (SPT)
7. Increasing efficiency in commercial vehicle operation (IECVO)
8. Support for pedestrians (SFP)
9. Support for emergency vehicle operation (SEVO)

UTMS is an abbreviation of universal traffic management system. This project has an aim to develop ITS using two-way communication between road and vehicle, particularly using infrared light communication.

In 1995, the Japanese government set nine development fields to promote Japanese ITS. Both Fig. 22.5 and Table 22.2 show the nine development fields. In Fig. 22.5, those fields are expressed dividing the two categories of road infrastructure side and vehicle side.

Vehicle side includes only two fields. One is advances in navigation system (ANS) and the other is assistance for safe driving (ASD). For example, VICS is included in the field of ANS. Five of the nine fields belong to infrastructure side. For example, road maintenance by special cars like removal of snow on roads using snow shoveling car is in these five fields.

ETC and SFP, which means support for pedestrians, locate between infrastructure and vehicle because these systems are very effective only when both infrastructure and vehicle are collaborating.

## 22.3 Status of Japanese ITS Development

It is apparent that setting these developing fields shown in the above accelerates ITS development. As a result of this VICS has started its service to provide vehicles with real-time traffic information such as congestion, road construction, and accidents since 1996. Though this system assumes that in-vehicle navigation systems are loaded in a vehicle, VICS is the first system to be put into practical use all over the world. In 2001 ETC is also to be started at the several tollgates of highways in Tokyo and Osaka area.

As one can easily see, VICS is not likely to reduce traffic congestion and accidents in a direct way. But according to some statistics it is said that VICS also provides people with a kind of comfort when they drive. So VICS is very useful to reduce traffic problems.

At the time the VICS service started, Tokyo, Aichi, and Kansai were the only three available areas to receive the information provided by VICS service. But now the service areas have spread to almost all over Japan. Figure 22.6 shows how rapidly both in-vehicle navigation system and VICS terminals grow in Japan. White bars in Fig. 22.6 show the accumulated number of in-vehicle navigation units in Japan. At the end of the last year, the number reached to about 6 million units. Getting along the

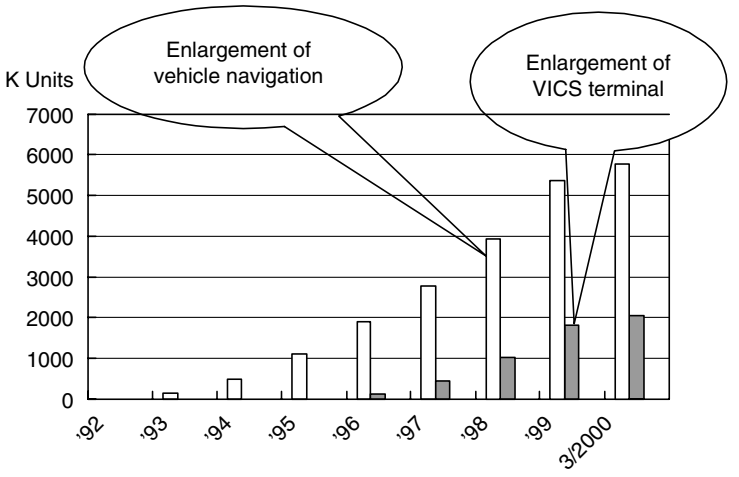



**FIGURE 22.6** Accumulated numbers of units.

**TABLE 22.3** Private Services of Information Provision

| Name | Information Provided & Function | Media | Start | Application Fee | Annual Fee |
|---|---|---|---|---|---|
| Intelligent traffic guidance system | 1. Optimum route to destination, travelling time<br>2. News, weather forecast<br>3. Leasure information | Cellular | April '97 | ¥5,000 | ¥36,000 |
| Moneh | 1. Conjesion, construction, traffic control<br>2. Parking, gas station, restaurant guide<br>3. News, weather forecast | Cellular | April '98 | ¥2,500 | ¥6,000 |
| Inter-navi system | 1. Setting of destination & course<br>2. Connecting to internet<br>3. Parking, gas station, restaurant guide | Cellular | July '98 | ¥2,500<br>Free 99/6 | ¥6,000<br>Free 99/6 |
| Compass link | 1. Parking, gas station, restaurant guide<br>2. News, weather forecast<br>3. Response by operators | Cellular | Sept. '98 | ¥3,500 | ¥30,000 |
| Mobile link | 1. Hotel, restrant, movies, news, TV guide | Cellular | Nov. '97 | Free | Free |

growth of in-vehicle navigation systems VICS terminals also have grown very rapidly, as shown by the grey bars in Fig. 22.6 since 1996.

Besides VICS, five private companies shown in Table 22.3, Benz, Toyota, Nissan, Honda, and Sony, are also providing traffic information and other information such as parking area, weather forecast, sightseeing spots, and so on. Using the Internet, as Sony is doing, means no charge or fee to get information, except the media fee, to receive traffic information. In my opinion, it will be interesting to see which of the companies survive—the automakers or Sony.

## 22.3.1 In-Vehicle Navigation System and VICS

First of all, the Japanese in-vehicle navigation system will be explained more minutely. Figure 22.7 shows the four main functions an in-vehicle navigation system can provide.

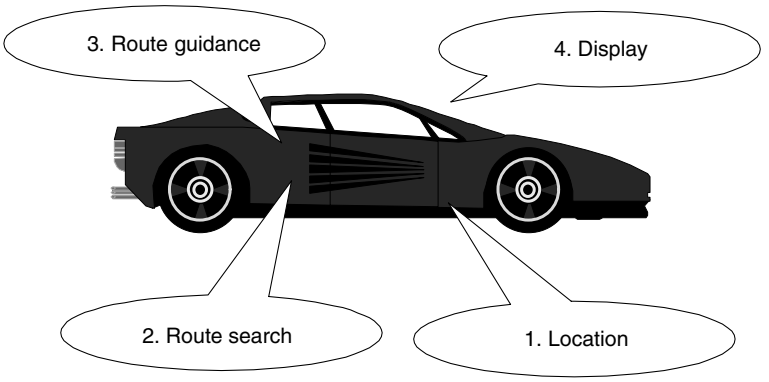



**FIGURE 22.7** Major four functions of car navigation.

The first function is positioning of present vehicle location. Normally, global positioning system (GPS) and map matching technologies have been used to determine the present location of a car. The second function is route search between a present location of a vehicle and a driver's destination. One can easily get an optimum route between the present location and the destination when only the destination is input into an in-vehicle navigation system. The third function is route guidance to guide you to your destination along with the determined way by the route calculation function. Normally, this guidance is performed by both voice and display for driver's safety. The last important function is to display vehicle position, results of route search, and guidance so that drivers can understand them easily at a glance.

A real configuration of a typical in-vehicle navigation system is shown in Fig. 22.8. It normally consists of the six components shown in this picture. A color monitor for a navigation system is also used to display moving pictures from a TV set and a DVD player, as shown at the lower right of Fig. 22.9. An important issue is that VICS supposes an in-vehicle navigation system to be loaded in a vehicle. So, a TV tuner also includes a receiver for VICS, in general. As shown later, VICS uses three major media such as FM multiplex, electromagnetic beacon, and infrared light beacon.

Figure 22.10 is an example of a map display that often appears on a color display monitor of Japanese in-vehicle navigation system. It shows the three major basic results in the minute map: the display of the

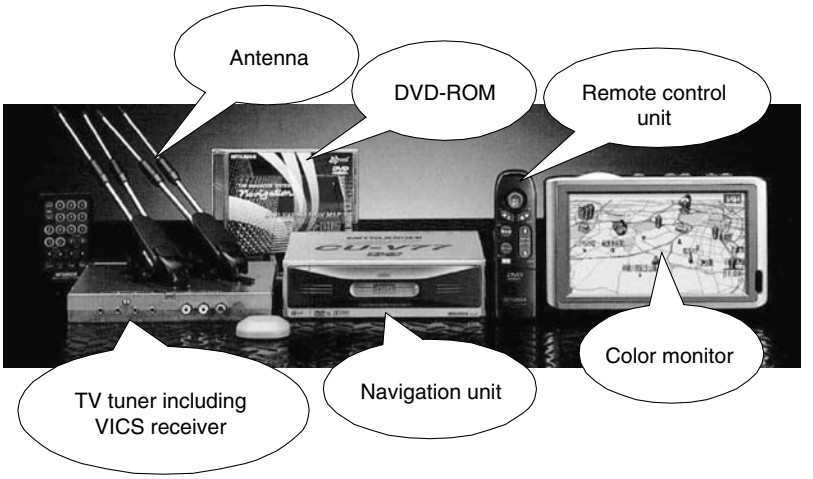



**FIGURE 22.8** Real configuration of in-vehicle navigation system.

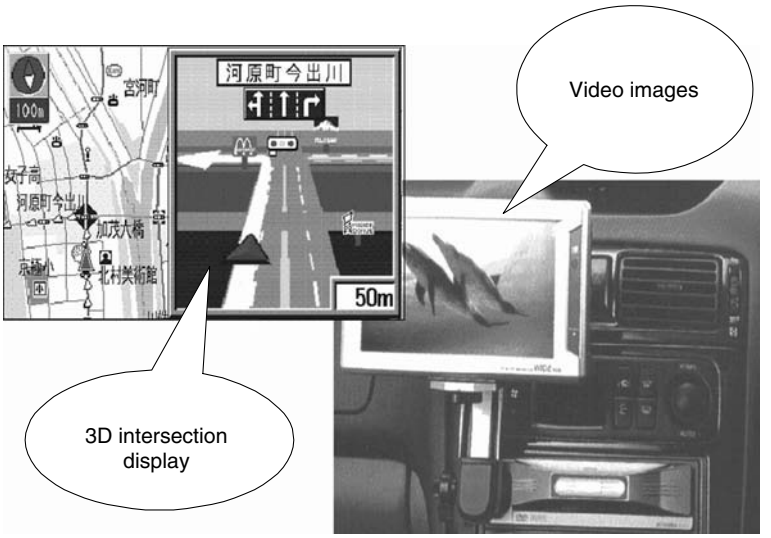



**FIGURE 22.9** Examples of information display.

present vehicle position, the optimum route for the destination, and vehicle guidance. A simplified map in Fig. 22.10 helps drivers to understand the direction they have to follow at the next intersection. The large (red) triangle at the lower left of this figure shows the present vehicle position determined by the location identification technology. A row of small (yellow) triangles shows the optimum route calculated by a navigation system. The right simplified map in this picture enables a driver easily understand which way he or she should take at the next intersection.

Figure 22.11 is another example of map display of in-vehicle navigation system. The left map in this figure shows a real map showing the present vehicle position and the calculated optimum route to the destination. The right picture shows a 3-D, simplified map with several landmarks such as McDonald's, road messages, and a traffic signal. Normally, it is said that a 3-D representation is easily understandable for drivers regarding which way they have to go.

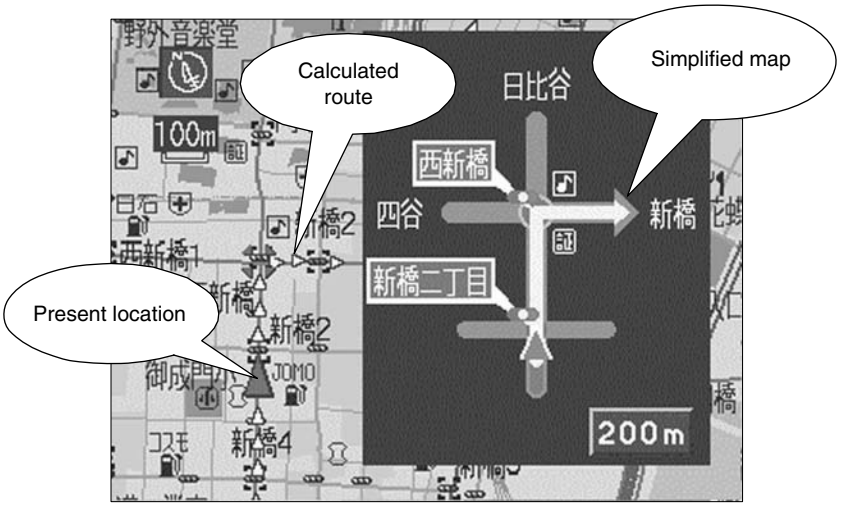



**FIGURE 22.10** One example of a map display.

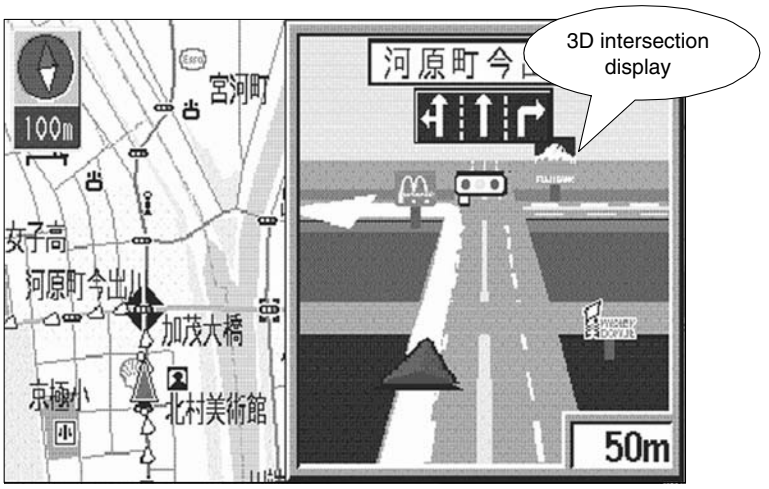



**FIGURE 22.11** Another example of a map display.

The situation is a bit different between Japan and other countries because of people's preferences for navigation displays are a bit different. For example, Japanese people like a map display as shown in Fig. 22.10, but people of other countries like only displays of the directions drivers have to take.

A block diagram of an in-vehicle navigation system is shown in Fig. 22.12. One can easily understand that essential configuration of an in-vehicle navigation system is the same as that of a personal computer excluding both the VICS unit and the sensors to be used to determine the vehicle location. In the CD-ROM, a map database is stored and accessed when a map is displayed, an optimum route is searched, and route guidance is performed. The software in the ROM in Fig. 22.12 provides the four functions—vehicle location, route search, route guidance, and display. For example, vehicle location is determined with processing signals from GPS, a vehicle speed sensor, and a gyro sensor. After vehicle location is fixed, a map database in the CD-ROM is accessed. Then, both the present vehicle location and

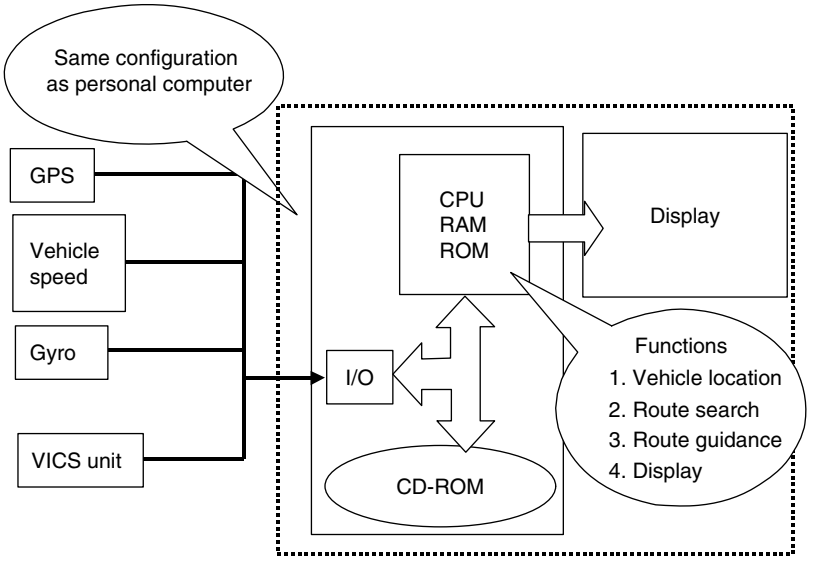



**FIGURE 22.12** Vehicle navigation and VICS.

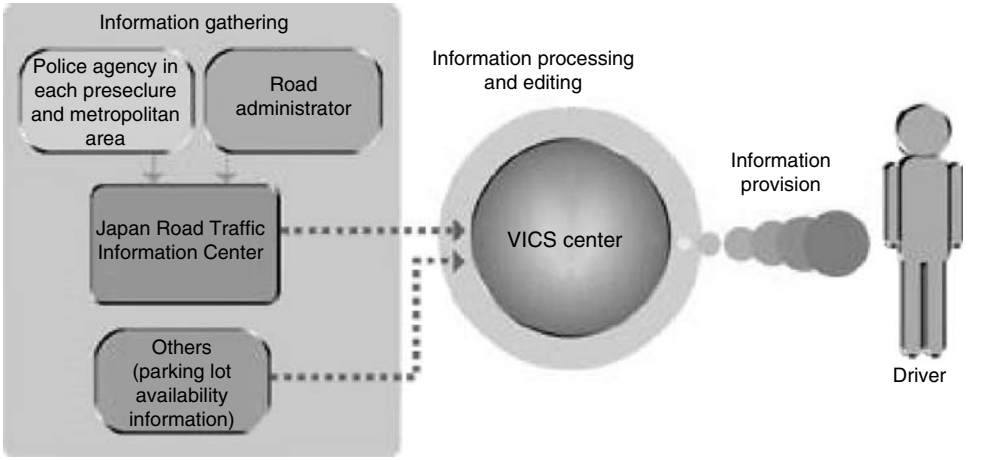



**FIGURE 22.13** Configuration of VICS.

a map near to the present vehicle location are displayed simultaneously on a color display moniter. If a driver inputs his or her destination to the navigation unit through the color monitor, the optimum route calculation is initiated and then the result is displayed. Along with the optimum route, the navigation unit guides the route to the driver in response to the vehicle movement. In addition to this, information such as the locations of restaurants, convenience stores, and gas stations near the vehicle's present location are also displayed.

The four basic functions of an in-vehicle navigation system, including the required function by VICS, are performed by software implemented in a navigation system as previously mentioned. So the scale of the software of an in-vehicle navigation system has become very large, it needs about 10 MB memory. This means it is 100 times as much as that of the software scale of engine control system that can cope with emission regulation of the passenger cars by Japanese government. Therefore, the software development of an in-vehicle navigation system cannot be developed with a conventional way in a short time at low expense.

The whole system of VICS can be explained using Fig. 22.13. Traffic information is collected and edited in VICS center, then it transmitted with the three media—electromagnetic beacon, infrared light beacon, and FM multiplex. One can get information, such as traffic congestion, through display of in-vehicle navigation system with the VICS terminal. VICS is supported by three ministries of Japanese government: the Ministry of Construction, the Ministry of Post & Telecommunication, and the National Police Agency. Figure 22.14 shows an example of a display that shows several pieces of information such as traffic information sent by VICS service. The map display is performed using a map database of an in-vehicle navigation system in Fig. 22.12. A road next to one red line in the left part of Fig. 22.14 shows conjestion in only one direction. In this case up direction of this road is congested. Two red lines along a road shown at the right part in Fig. 22.14 means that the road is congested in two directions (up and down). One can easily detour these congested roads if with such real-time traffic information. VICS provides other information about road construction, road restrictions, and even about parking lots. VICS can also provide real-time information regarding traffic information and other useful information, and it gives a kind of comfort to drivers as a result.

## 22.3.2 Electronic Toll Collection System

The electronic toll collection system (ETC) is an electronic fare collection system. This system was introduced in foreign countries earlier than in Japan. In Japan, the service of ETC starts this year. One of the aims of this system in Japan is to reduce the congestion at highway tollgates thereby resulting in less

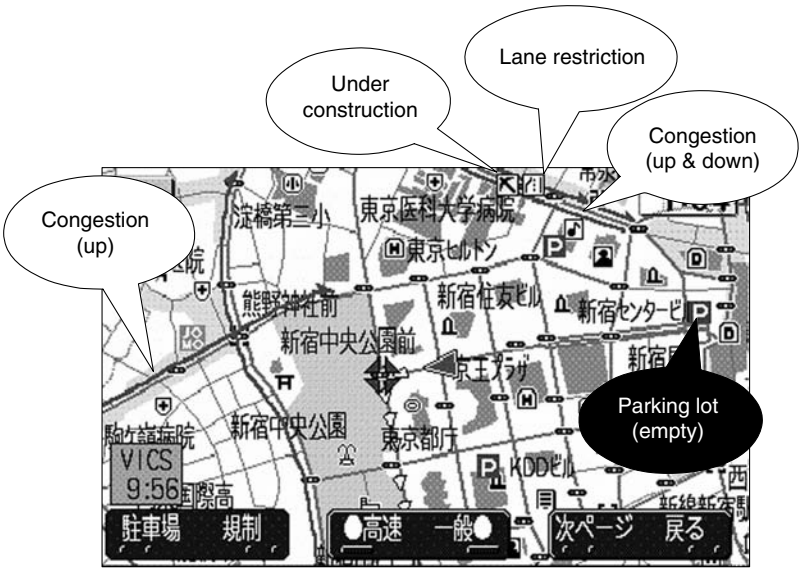



**FIGURE 22.14** Examples of displayed information (VICS).

emission of exhaust. Figure 22.15 shows the whole system of Japanese ETC. Three major technologies are used: network in the infrastructure, automotive terminals for ETC, and road-to-vehicle communication. The infrastructure is composed of a huge network system into which personal information such as credit card numbers are flowing. Therefore, the information flowing into this network is written in code to ensure the security. Information from the automotive terminals for the ETC system is transmitted to the network through road-to-vehicle communication known as dedicated short range communication (DSRC). Of course, it is also written in code to ensure the security. Specification of DSRC is shown in Table 22.4. In the same manner, information about road infrastructure, such as the location of the tollgate, is also transmitted to a vehicle from the infrastructure and through DSRC.

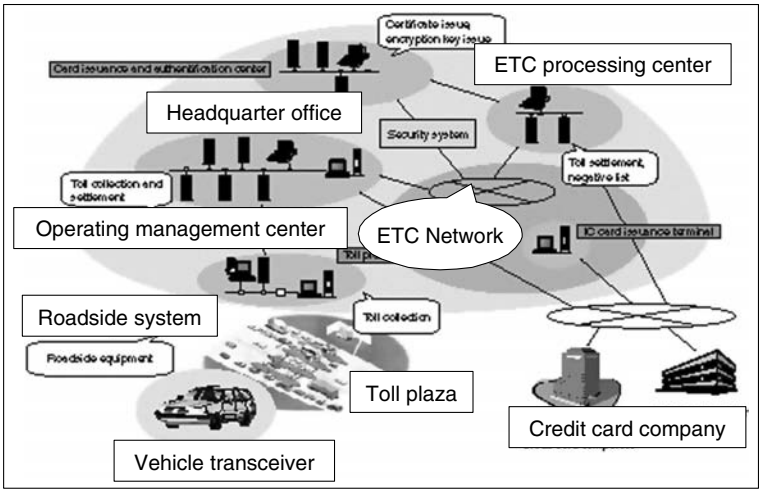



**FIGURE 22.15** Network system for ETC.

**TABLE 22.4** Specification of DSRC

| Item | Description |
|---|---|
| Frequencies | Two pairs in the 5.8 GHz band |
| Bandwidth | 8 MHz max. |
| Transmitter output | 10 mW |
| Communication system | Active type with slotted Aloha |
| Maximum vehicle speed | 80 kph |
| Data transmission rate | 1.024 Mbps |
| Modulation method | Amplitude shift keying |
| Bit error rate | Less than 10 ppm |
| Communication error rate | Less than 1 ppm |
| Encoding method | Manchester encoding |

### 22.3.3 Support of Safe Driving System

The support of safe driving system that appears in Fig. 22.5 is defined as the third development field of Japanese ITS has also been developed in Japan. In the early stages of the safe driving system, automatic driving was considered an ultimate support system. But many technical demonstrations on automatic driving from 1996 to 2000 gave people an impression that there were still many issues to be deployed from the point of view of both drivers and legal aspects. The author will demonstrate this more specifically later. So, recently it appears to the author that the idea of automatic driving is disappearing gradually. Instead of this, various warning systems and assistance systems have been considered. Demo2000 held at Japan last year shows effectiveness of those assistance systems such as AHS-I and ASV. "I" in the term of AHS-I means "information provision." AHS-I is a kind of concept to assure safe driving of vehicle by using information provision of both traffic information and road configuration through road-to-vehicle communication. Not only road-to-vehicle communication but inter-vehicle communication is also useful to ensure traffic safety. A demonstration showing the effectiveness of this technology was also shown in Demo2000 at Tsukuba, Japan. For example, it was shown to form platooning composed of running vehicles using inter-vehicle communication technology.

## 22.4 Issues of ITS Development and Roles of Software Technology

### 22.4.1 Issues of ITS Development

The readers can easily think of technological issues to put into practical with the use of ITS, particularly an assist system of safe driving. Actually, the more important issues to deploy ITS are related to social, legal, and human issues, including responsibility, which are recognized with technical demonstration from 1996 to 2000 of the support system for safe driving. Table 22.5 shows the technical issues for deploying ITS. The importance of these technical issues will be easily understood. Figure 22.16 shows social, legal, and driver's issues summarized by Becker and the author.

**TABLE 22.5** Technological Issues

1. Robust sensing technology (e.g., obstacles, road conditions, traffic flow, and so on)
2. Robust control technology (e.g., vehicle control, traffic control, platoon control, and so on)
3. Human technology (e.g., display, human I/F, drivers intention, and so on)
4. Information and processing technology
5. Communication technology
6. System integration technology

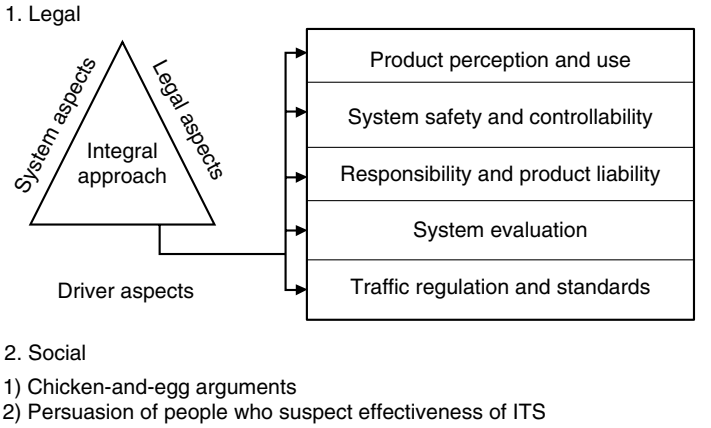



**FIGURE 22.16** Legal and social issues.

Legal and driver's issues are composed of five terms, as shown in Fig. 22.16. The first is "product perception and use." This means how deeply users of the ITS-related system like the assist system for safe driving, can recognize the performance and the use of the system. For example, suppose you buy an air-bag system. You have to understand both its performance and how to use it very well. You have to know how it works depending on magnitude of the crash. If you do not know that, you will be overconfident about the air-bag system, and you may be injured by a traffic accident. In this process, sufficient explanation of a system by a salesman is very important for users to operate the system very well.

The second important issue is about "system safety and controllability" in Fig. 22.16. One can easily understand the importance of this issue if one compares control systems of a car with that of an aircraft. Pilots in an aircraft can operate the control system completely because pilots know how the control systems work well. Moreover, they know how to override the control systems in an emergency. Pilots are highly disciplined people, but drivers are not necessarily such people. So this issue becomes very important in case of drivers.

The third issue is related to "responsibility and product liability." Suppose a crash happens. Who has to be responsible for the crash? Which is more responsible, a control system or a driver? Is it the biggest issue to deploy the ITS properly?

System evaluation, the fourth issue, and traffic regulation and standard, the fifth issue, are also very important. So you can easily understand their importance without any explanation on these issues.

Two social issues are shown in Fig. 22.16. One is the chicken-and-egg argument and the other is persuasion of people who suspect effectiveness of ITS deployment. The chicken-and-egg argument holds in the spread of a system that is composed of both infrastructure and automotive equipments, which rely on infrastructure and vice versa. For example, we consider the case of ETC. There will be no incentive of spreading automotive terminal for ETC without preparation of infrastructure for ETC. On the other hand, there is no incentive for infrastructure to be prepared without automotive terminal of ETC. This is a so called a problem that is first infrastructure or automotive terminals.

Now many people say that ITS is necessary to make smoother traffic flow as well as decrease air pollution as a result. But some people are still suspicious as to the effectiveness of ITS. For example, is it true that ITS decreases traffic congestions? Even if it is true, less congestion makes more people use cars. As a result, less congestion cause more traffic demands. This will lead to more traffic flow. Therefore, only more vehicles can run, thanks to ITS. As a final result, they think that more cars can run on roads after ITS is deployed. So congestion will remain almost invariable even if ITS is introduced. It is a conclusion of the people who are suspicious about the effects of ITS and that traffic congestion will not decrease after ITS is introduced. In case of ETC deployment, several people do not believe the effect,

and they say ETC will change only location of congestion. For example, suppose there is a junction in front of a tollgate. Before ETC is introduced, the tollgate was congested. After ETC is introduced, there will be congestion at the junction instead of the congestion in front of the tollgate. It seems that the location of the congestion shifts from the tollgate to the junction. This is a conclusion of the people who are suspicious as to the effect of ETC.

### 22.4.2 Roles of Software Technology

As discussed earlier, ITS is making full use of both information technology and communication technology. Therefore, the importance of software technology does not need to be explained. For example, in-vehicle navigation system has a size of the software about 10 MB. This size is about one hundred times that of engine control software that meets the stringent exhaust gas regulation. This software can provide important performance, as shown in Fig. 22.7. Besides, this software technology has a great potential to solve these issues mentioned previously.

In advance, to explain the potential of software technology for solving the social and driver's issues, as shown in the preceding section, let us explain the concept to solve these issues. The first issue of "product perception and use" in Fig. 22.16 would be solved by showing both the performance of the product or system to be introduced into the market and how to use it. For this purpose, a simulation program that can simulate the performance is very effective and makes users understand it. For example, a simulation program of the performance of the air-bag system can show both its performance and how it works well. So users can easily understand the performance of an air-bag system or, in some cases, even the limitation of that system. In other words, the issue of product perception and use is realized by software technology.

In a similar way, the second issue of "system safety and controllability" is also solved by software technology. For example, you can very easily learn system safety and controllability if you have a simulation program. For a new aircraft, pilots can learn its control system and can be trained very well with a flight simulator.

As for the third issue of "responsibility and product liability," a simulation program is very helpful in solving this issue. Product liability (PL) is originally set for protecting users of product, but it sometimes makes manufacturers conservative about developing a new product. The solution for this issue, however, is obtained by assuring the safety of a product from the viewpoint of users. This can be done by showing users the safety of the product in every area where the product is supposed to be used. Only a simulation program of the performance of the product can perform this. For the third issue, software technology is also very effective.

The forth issue of "system evaluation" can also be supported with the use of a simulation program. It would not need to explain. The fifth issue of "traffic regulation and standards" is a little bit different compared with the four issues mentioned previously.

Social issues, including both the "chicken-and-egg argument" and persuasion of people who suspect effectiveness of ITS, are also solved by simulation. For example, people can learn about the effectiveness of ETC by a simulation program of ETC. Figure 22.17 is an example of a simulation of the effect of ETC with the ITS simulator that is under development. You can easily understand that the introduction of ETC decreases the congestion in front of a tollgate, as shown in the upper portion of Fig. 22.17. Even conservative people would agree to build ETC in order to see this result.

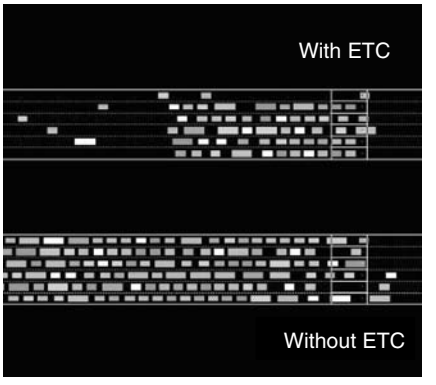



**FIGURE 22.17** Simulation results with ITS simulator.

## 22.5 Practices of Software Development of Both In-Vehicle Navigation System and ITS Simulator

Software scales of both the in-vehicle navigation system and the ITS simulator are relatively large, but there is a difference in both properties between the navigation system and the ITS simulator. In the case of the navigation system it is embedded software that meets the needs of users. On the other hand, the ITS simulator is not embedded. In this section, both the software of the in-vehicle navigation system and the ITS simulator are explained briefly.

### 22.5.1 In-Vehicle Navigation Systems

#### 22.5.1.1 Status of the Development of In-Vehicle Navigation System

One of the most severe issues in the development of the software for in-vehicle navigation is the short development time for the software, which meets customer requirements with low expense.

So some people say that only people who can develop in-vehicle navigation system software very quickly, at very low cost, would control the in-vehicle navigation system market.

As discussed earlier, in-vehicle navigation system software size is relatively large compared with other consumer products like a refrigerator, a washing machine, and an air conditioner. The size has reached nearly 10 MB. Cellular phones have almost the same software size. In general, this means that the cost to build the software has become very high and it needs a long time for development.

As you can easily understand, the market of the in-vehicle navigation system has two aspects. One is the aspect of consumer electronic products mainly sold at after-markets. The other aspect is that of the OEM market where the in-vehicle navigation system is sold and delivered directly to automakers from suppliers such as manufacturers of electric equipments. Table 22.6 shows three important issues to be maintained or met by OEM suppliers of in-vehicle navigation system.

The first issue is customer satisfaction. Here, customer means automakers, not drivers themselves. That is, automakers decide which supplier's navigation system to buy. So, it is necessary for suppliers to meet the various needs of automakers.

The second important issue is keeping the delivery time set by automakers. Automakers have strategies to sell their cars. So the delivery time of navigation systems is set by automakers. The only thing that suppliers can do is to keep the delivery time very strictly. So the lead-time to develop an in-vehicle navigation system is normally very limited for suppliers.

The last important issue to keep in mind is reliability of an in-vehicle navigation system. Of course, reliability is also very important in the case of consumer electronic products; but in the OEM market, reliability is a more important issue because a sold car might have been returned due to the unreliability of the in-vehicle navigation system.

Summarizing the three important issues, we can see a quick development of navigation software, with high reliability is a key point. Currently, many difficulties are present in navigation software development based on conventional methodology; however, as large as the software scale becomes, it does not have any problem if we can reuse the software in response to the needs of each automaker. But each auto-maker has many different needs, for example, different functions of a navigation system, man

**TABLE 22.6** Features of OEM Market as Customers

| | |
|---|---|
| 1. | Customer satisfaction<br>We have to meet the various needs of each automaker. |
| 2. | The date of delivery<br>We have to keep it very strictly. |
| 3. | High reliability<br>It may happen that a sold car is returned to automakers due to unreliability of car navigation system. |

*Note*: Quick development of navigation software with high reliability.

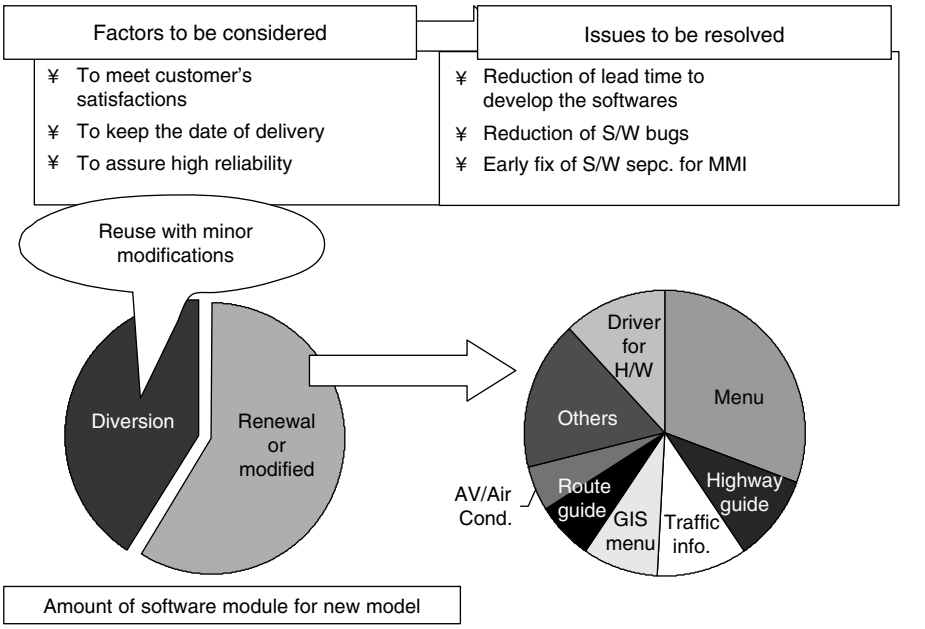



**FIGURE 22.18** Issues of navigation software development.

machine interfaces, desired price, and so on. On the other hand, normally the rate of reused software is relatively low, about 40%. Of the software, 60% has to be developed or modified profoundly in response to the needs of each automaker. Figure 22.18 shows the software volume ratio in developing the new navigation systems. About 60% of the software must be renewed for them, and the human-interface related software occupies a large part of all the renewed software. In addition to this, we have often encountered that software specification of each automaker has not been fixed until the end of development of the software.

To reduce both S/W bugs and lead-time to develop the navigation software, adoption of both middleware architecture and auto code generation is desirable. These techniques originate from the so-called object oriented development of software.

To promote determination of software specification of automakers, particularly man machine interface (MMI) portion of in-vehicle navigation system, it is suitable to introduce man machine builder tool that has already developed in another field, such as public infrastructure. The next subsection explains these two technologies realizing the issues stated earlier.

#### 22.5.1.2 New Methodology to Develop In-Vehicle Navigation Software

The first technology originated from object-oriented software development. Figure 22.19 shows the basic development flow along with the object-oriented software development. In the beginning, that is, the design stage of the object-oriented development model, customer requirements and system functions are decomposed of the object components that represent the basic or initial system. In this stage, various supplemental functions of automakers are decomposed of both each child component and the basic components that will incorporate features as additional components. Because all decomposed components including basic and child components are developed individually, their implementation and tests can be easily performed. For example, both the basic component and child components are implemented and tested separately, and then both implementation and test as a whole system, such as system function, can be performed.

Child components and additional components are also given the same process to assemble the complete system, which meets the customers requirement. In object-oriented development we can easily

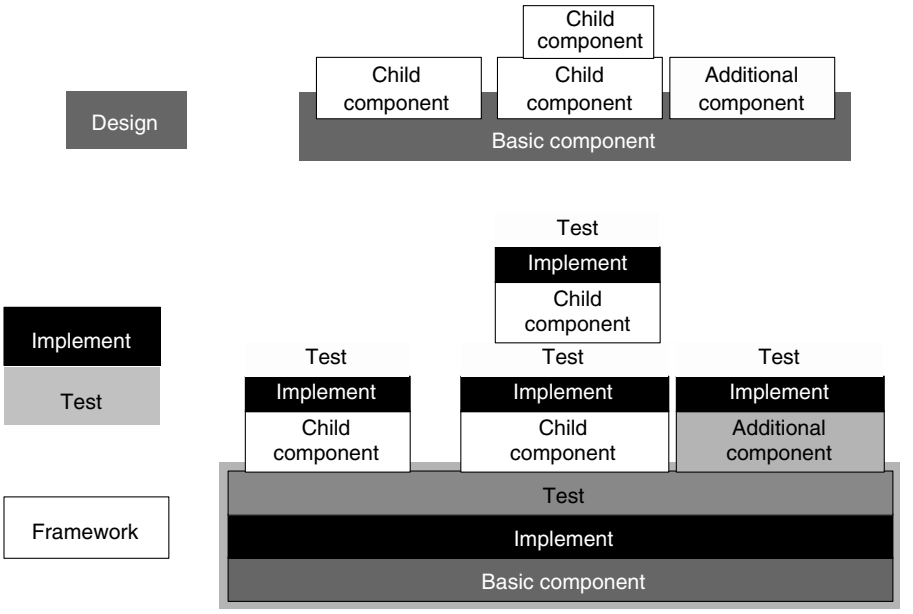



**FIGURE 22.19** Object-oriented development model.

modify with the addition of additional components. So this modification costs relatively less compared with the conventional waterfall software development. From the point of view of both easy debug and implementation to a target machine, we need another means besides adoption of object oriented development model. Moreover, we need a means to make automakers decide their specifications earlier.

Figure 22.20 shows the hierarchical software architecture models to make us debug easily and implement. The architecture has two distinct points. One is this middleware and the other, man machine builder tools serving to build MMI of in-vehicle navigation system. The middleware acts as a

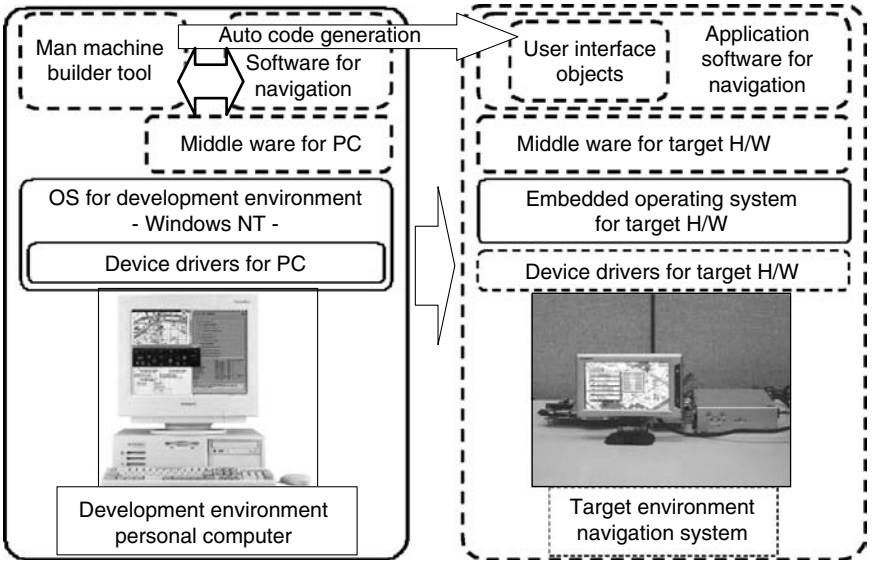



**FIGURE 22.20** Hierarchical software architecture model.

kind of separator between application software for navigation and hardware like personal computer (PC), and target machine. A detail of man machine builder tools will be shown later.

Generally speaking, it is desirable to use a PC as a software development environment because of its lighter weight, portability, and low price. The operating system (OS) for this environment is Windows NT. This OS includes device drivers for the hardware in the left picture of Fig. 22.20. Now let us explain the flow of navigation software development. In the beginning, application software of navigation system, composed of both child and basic components, is built with man machine builder on PCs as software development tools. Then the software is tested and debugged on the PCs. After that, the software is rewritten in terms of a target machine by using auto code generation and implemented. Then the application software is transferred to the target environment. This software definitely works well on the target machine because all necessary tests are already done by using PC and confirmed that the application software works well without any problem. So the remaining development that has to be done is to develop the device drivers for target hardware, which harmonizes the embedded OS with the hardware, as shown in the right picture of Fig. 22.20.

Also, it is necessary to modify the middleware for target hardware. After the modification of the middleware for target machine, the developed application software, excluding user interface objects, is tested to check how well it works. Thus, the application software for navigation can be easily applied onto the target environment by using this middleware.

The man machine builder tool automatically generates the software code for user interface object, and this software code is included into the application software. When one develops navigation software along with the above processes, one can indeed develop both software and hardware concurrently. This reduces the lead-time of developing new navigation system dramatically and enhances reliability of the software. Because the application software of the navigation system is developed on Windows OS, one can easily confirm its operation, in the early phase of development, even if the target hardware is still under development. CRT display in Fig. 22.21 shows an example of the software test in development phase. In this display, one can confirm how the functions of the navigation software work well, instead of the real display device; that is, these functions of application software can operate and confirm with virtual operation panel with dummy signal dialog, and voice guide dialog.

The man machine builder tool is also used to make automakers decide the specification of the man machine interface in the early stage of the software development because man machine builder tool can give them virtual MMI and show how it works. For example, using the virtual display, drivers can input

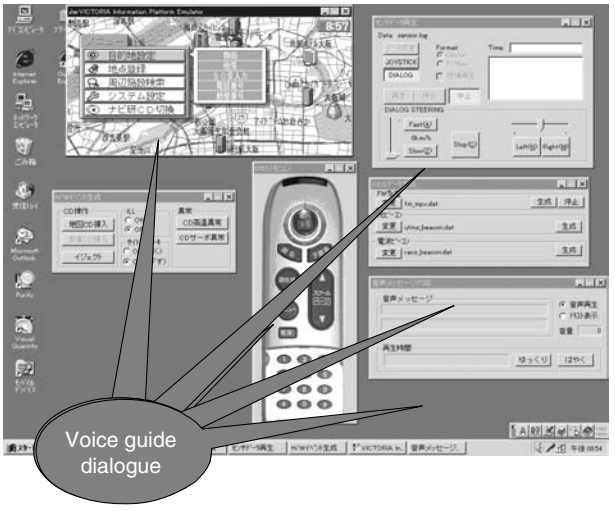



**FIGURE 22.21** Confirmation of operation using PC.

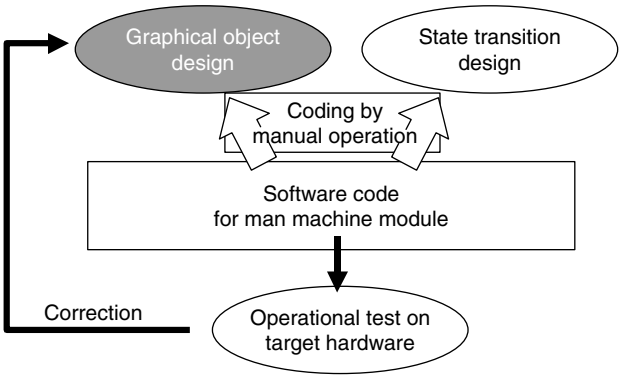



**FIGURE 22.22** Conventional method for man machine module.

their destinations and their requests to search optimum routes or restaurants near their present locations. MMI serves drivers to do these interactively using several icons in Fig. 22.21. Generally speaking, the specification of man machine interface is often not determined until the end of development because the determination needs ergonomic studies. So it is desirable for providers to make a virtual environment to simulate the operation of man machine interface to be developed.

The man machine builder tool consists of both user interface design tool and state chart CASE tool. The missions of the user interface design tool are graphical object design, operation confirmation, and code generation.

On the other hand, the mission of the state chart CASE tool are production of operative transition, graphical confirmation of transition, and code generation of state transition.

Before explaining the function of the man machine builder tool briefly, we will explain about the conventional way to build MMI using Fig. 22.22. In the conventional method, the man machine module including graphical object design for user interface and state transition is designed from requirement analysis in the beginning. After both graphical object design and state transition are proceeded, software code of man machine module are coded separately by manual operation (hand coding).

Then the software code is combined with other application software. Finally, its operational test is performed on a display of target hardware. If some unexpected behavior or shape occurs in the user interface, its correction of design and coding are reprocessed. And, though this is a more important issue, additional requests by customers are often added to the specification to MMI. In order to cope with this, a new environment to build MMI differently from the conventional method is indispensable.

By using the new development environment of building MMI, one can generate the man machine module throughout from design to software code. With this tool, a graphical object is designed with the menu object editor, as shown in Fig. 22.23. This means both the graphic objects for MMI and the animation of their graphic objects are defined simultaneously with production of objects.

State transition is defined in the state chart editor in the right part of Fig. 22.23. In the state chart editor, both the animation parameters of graphic objects and navigation function behavior are defined following the state transition. After the graphical confirmation of both these graphic behaviors and state transition, these tools can generate the software code for man machine module automatically. Thus, you can show automakers how the user interface under development works well, using the virtual display. And you can get necessary suggestions including the additional needs from them. Thus, one can easily understand that this tool can accept various requests from customers interactively. So the specification of MMI is easily determined, unlike the conventional method, by using this tool.

The menu object editor in Fig. 22.23 has two main functions. One is the production of the graphical object of the user interface of the navigation system using the basic and special components. The other is the production of animation in the user interface. In Fig. 22.24 basic components, such as polygonal line, polygon and ellipse are shown in the upper dotted box of the left picture in Fig. 22.24. Special

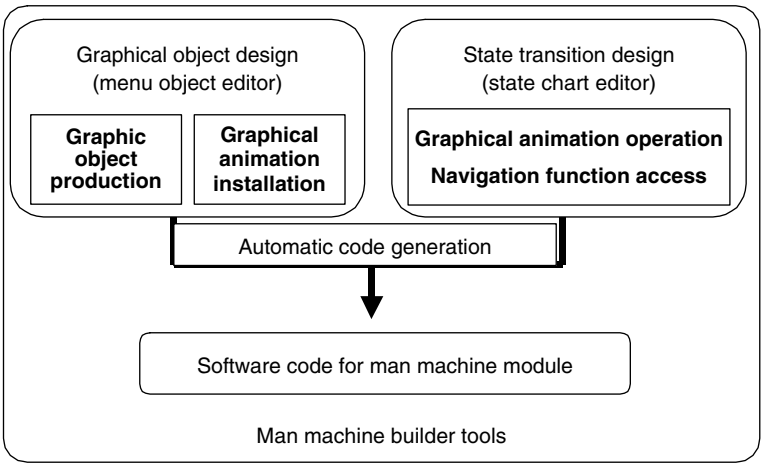
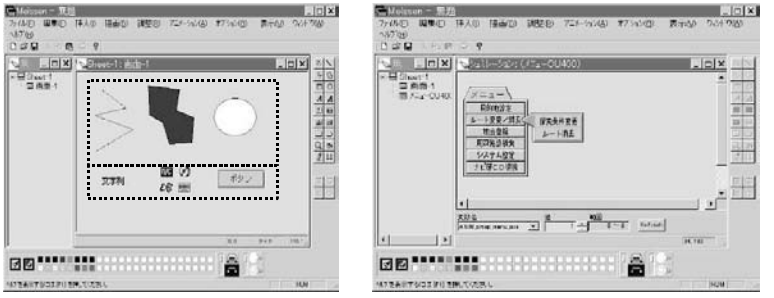



**FIGURE 22.23** Development with man machine builder tool.



- Graphical object production by basic/special components
  - – Design navigation menu objects by polygonal line, polygon, ellipse, character string, bitmap, etc.
- "Operation" setting in diagram (production of animation)
  - – Transform each diagram in relation to its variable.

**FIGURE 22.24** Function of menu object editor.

components are only needed for the navigation object, such as character strings, bitmaps representing the specified functions of the system, and the button object for the menu interface are generated. In addition to the object design, animation of the object can also create its transition, and this animation can be simulated on this tool, as shown in Fig. 22.25. This menu object editor also has a function of automation code generation for designed object and animation.

Use of state chart CASE tool to state transition design has these major points. The first major point is the easy description of graphical menu transition. Designed transition can be visually confirmed with the visible state chart, which is the second major point. The third point is easy customizing of the state transition. It can be modified in the visual tool by moving the transition line in the design window in the right picture of Fig. 22.25. So one can introduce it to advanced environment for navigation software development. This CASE tool can also output the software code of the state transition with an automatic code generation function.

Here, we show the operation flow of making the entire software using the MMI builder tool. In Fig. 22.26, the man machine module provides the navigation software code. When the graphical object

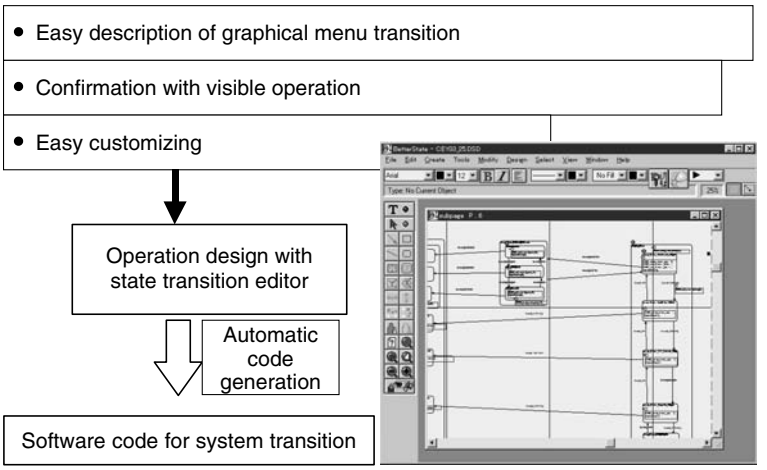



**FIGURE 22.25** State transition design.

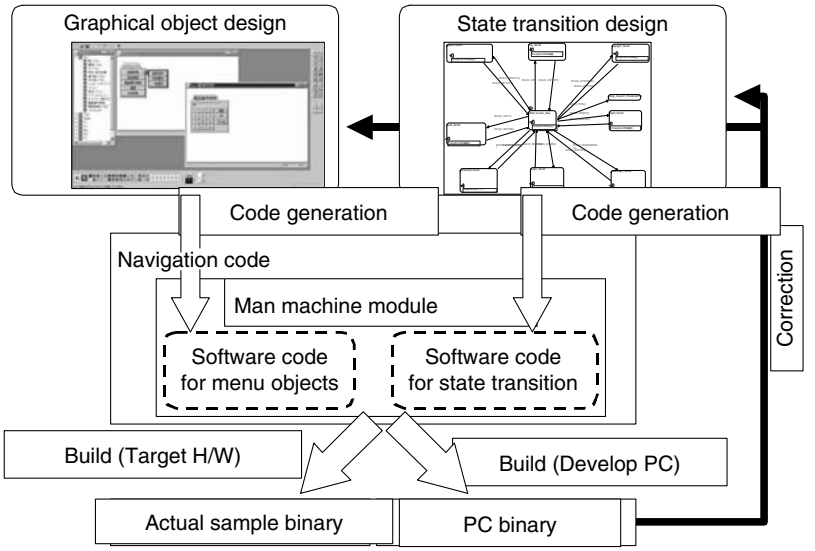



**FIGURE 22.26** Operation flow with man machine builder.

design is finished, the software code of the menu objects is generated automatically. The software code of the state transition is also generated in the same manner. Both the codes are included in man machine module, then the binary code of the navigation software is built to confirm its function on the PC. This sequence is continued until achievement of all requirements, and it builds for the target hardware to release the actual sample.

### 22.5.1.3 ITS Simulator and Its Functions

ITS simulator is a kind of driving simulator. So real configuration of the ITS simulator is very similar to that of the driving simulator. Figure 22.27 shows the configuration that is composed of computer, screen, projector, and a half actual vehicle. Although the configuration of the ITS simulator is similar to a driving simulator, the functions are very different from those of a driving simulator. The ITS simulator can simulate both several effects and effectiveness of ITS deployment as a result of simulating the

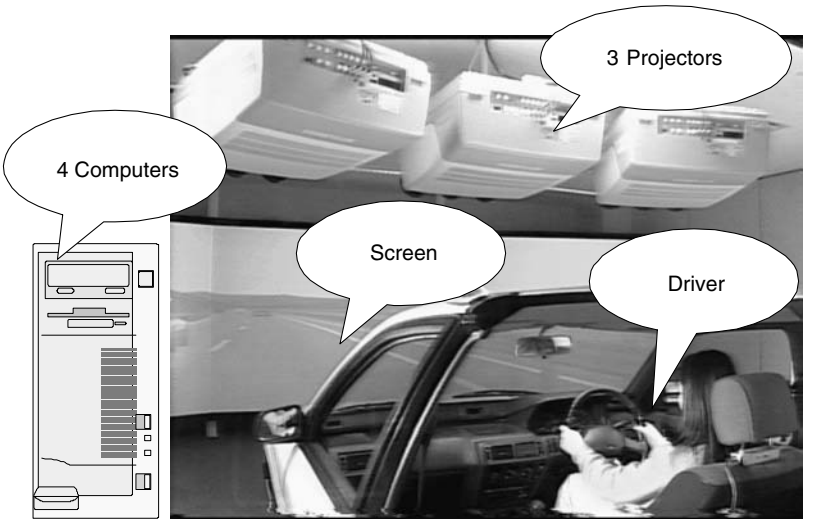



**FIGURE 22.27** Physical configuration of ITS simulator.

interaction between road infrastructure and vehicles. This is one of the greatest different points between a driving simulator and ITS simulator.

Figure 22.28 shows the architecture of ITS simulator. It consists of four major modules, vehicle dynamics simulator module, micro traffic simulator module, 3D road environmental simulator module, and system control module.

The first module of vehicle dynamics simulator provides the movement of the half driving vehicle dynamics in real-time. This vehicle dynamics model has nine degrees of freedom of movements. Therefore, you can feel as if you were in a real car in the half vehicle of ITS simulator in the same manner of a driving simulator.

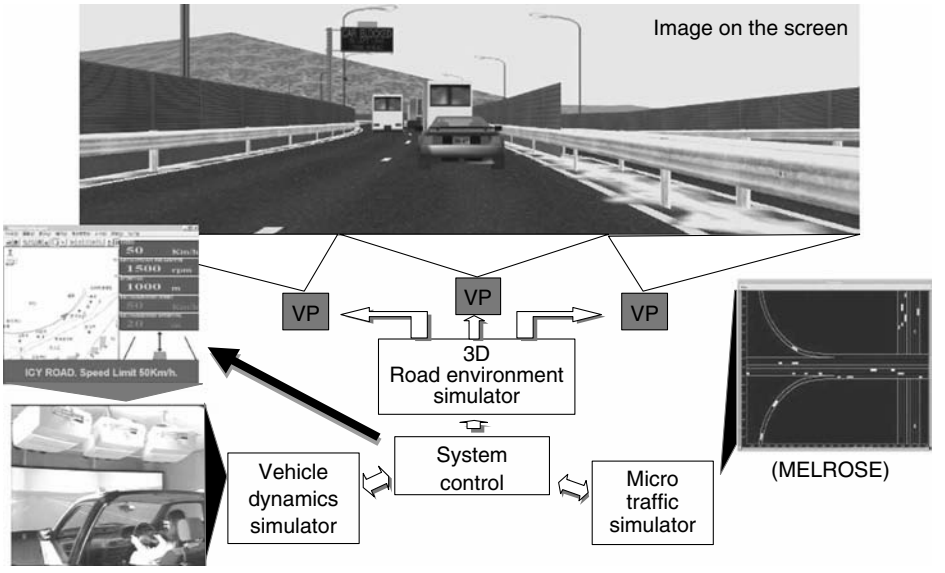



**FIGURE 22.28** Modules of ITS simulator.

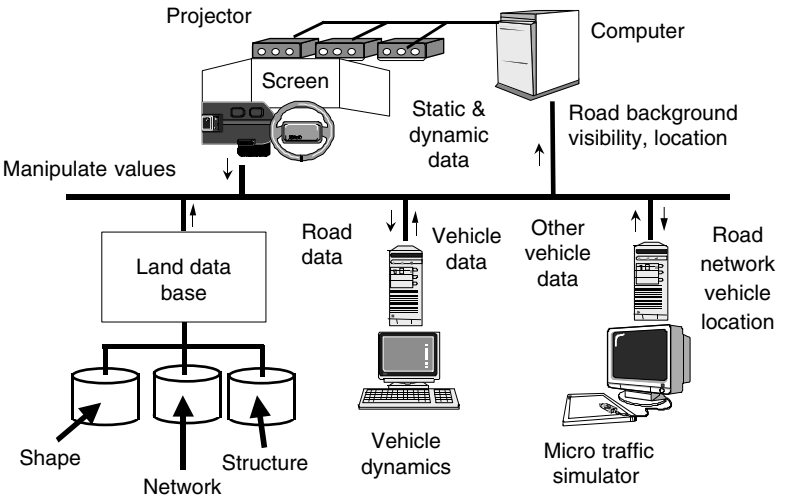



**FIGURE 22.29** Physical architecture of ITS simulator.

The second module of microscopic road traffic simulator, which is called MELROSE, generates a virtual road traffic environment based on an autonomous driving model. Every vehicle provided by this module acts as surrounded vehicles of the simulated vehicle of the vehicle dynamics simulator, and also every vehicle can run with its own origin and destination. So these vehicles can act as surrounded vehicles.

This third module is a 3D-computer graphics road environment simulator. This simulator creates actual geographical conditions and road configurations in response to the vehicle location of the simulated vehicle. For these, various road data like width, gradient, numbers of lane, road shape, and so on are stored in the memories of this module. In response to the vehicle movement the stored road data are accessed to form virtual road environment with 3D graphics based on real road data.

The final module is to control the operation of the above three module and to make the total modules operate as the ITS simulator. A representative action is to synchronize other three modules because all time units to compute are different in each module.

Figure 22.29 shows an architecture and data flow among all four modules. The land database means the stored data as shown before. The upper right computer in Fig. 22.29 means system control module and 3D graphic processing of road environment simulator.

A simulation result on how congestion is formed by a blocking vehicle is shown in Fig. 22.30. The reader can easily understand a process of congestion due to a blocking vehicle in a circle. A driving

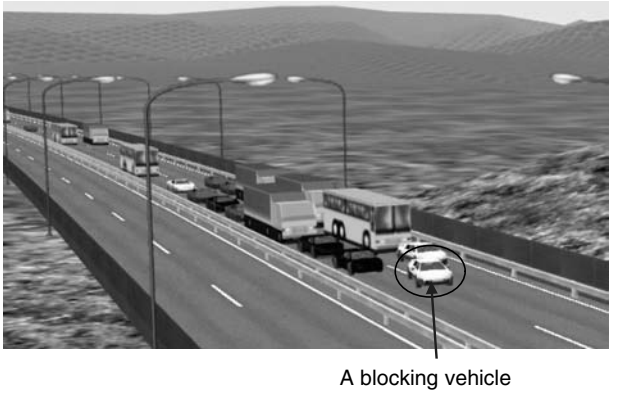



**FIGURE 22.30** Generation of congestion due to accident.

simulator cannot simulate such a situation. The next simulation result is shown in Fig. 22.31. It simulates a kind of the interaction effects between provisions of traffic information and vehicles or drivers, that is, it simulates the differences of traffic flow rate with and without traffic information with a message board to drivers. For example, a roadside message board in the circle in Fig. 22.31 says that there is an icy road surface ahead, so please slow down. This message is also sent through road to vehicle communication to vehicle and shown in the in-vehicle display like in-vehicle navigation system in Fig. 22.31. Now a smart driver follows this advice to get smooth traffic flow. Some driver might not follow. As a result of this, congestion would occur to cause bad traffic flow. Therefore, traffic flow rates between these two cases are very different. ITS simulator can simulate such a situation and show the effectiveness of the provisions of traffic information.

Figure 22.32 shows another example of the simulation with ITS simulator. A foggy situation with bad visibility is simulated. For example, a vehicle with an automatic driving system can run safely in spite of such a foggy condition. Though several modifications are needed, ITS simulator can simulate both every considerable traffic situation and considerable situation of use of control systems like automatic driving.

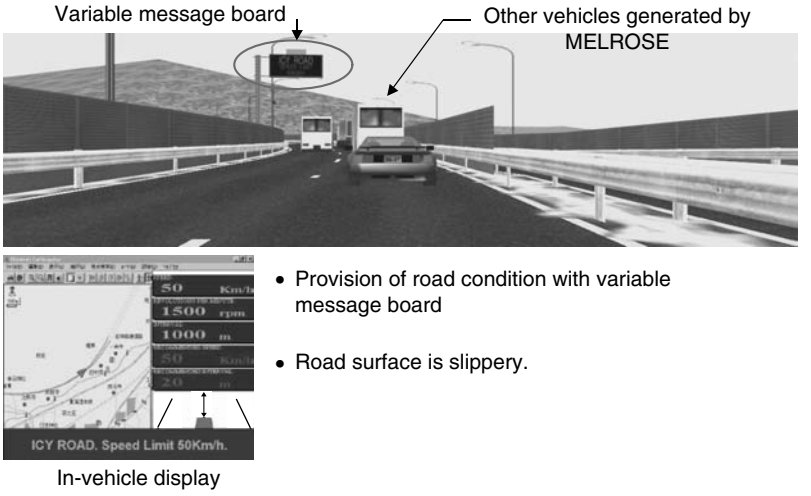



**FIGURE 22.31** Simulation of effectiveness of AHS-i.

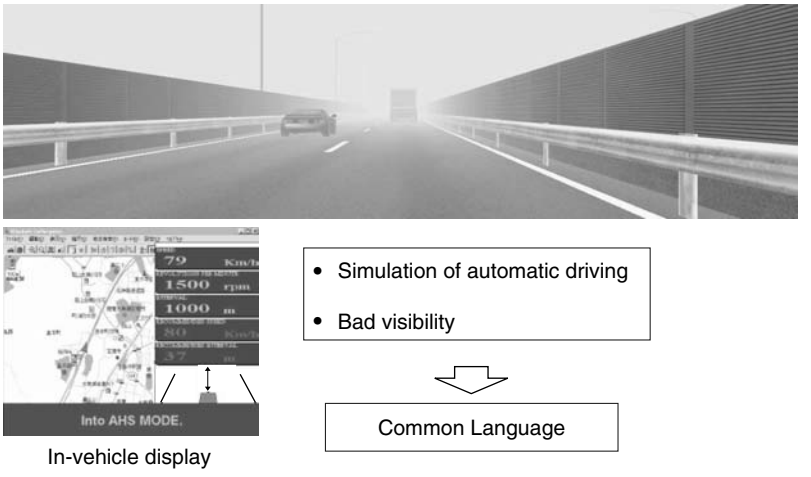



**FIGURE 22.32** Simulation of effectiveness of AHS-a.

As I mentioned earlier, this means that ITS simulator has a great advantage to show both the effects and the effectiveness of ITS deployment.

The microscopic traffic simulator, that is a part of the ITS simulator, alone can simulate several situations. For example, Fig. 22.17 shows the effects on congestion in front of a tollgate of a highway between with and without ETC. The upper case in Fig. 22.17 shows congestion in front of a tollgate with ETC. The lower shows the case without ETC. Each colored rectangle shows each vehicle. So the length of congestion is given by that of the row of the rectangles on each lane. You can easily see the difference between with and without ETC. Even people who suspect the effect of introducing ETC can understand the effect and would agree to the introduction of ETC.

## 22.6 Conclusion

We have the following conclusions:

1. In Japan in-vehicle navigation system are widespread and the VICS service to provide real-time traffic information is also widely spread. As for ETC, its service is about to start.
2. In the development of navigation software, adoption of both an hierarchical architecture and man machine builder into the development environment of navigation software is powerful enough to develop the software very quickly with high reliability.
3. ITS simulator is a powerful tool to solve the issues of ITS deployment.

### Acknowledgments

The author expresses his great thanks to Dr. Yukio Goto, Mr. Yoshihiko Utsui, Mr. Masahiko Ikawa, Mr. Akio Uekawa, Dr. Hiroyuki Kumazawa, Mr. Mitsuo Shimotani, Mr. Minoru Ozaki, Mr. Akira Sugimoto, and other many researchers and engineers at the Industrial Electronics and Systems Laboratory of Mitsubishi Electric Corporation, for their great assistance and encouragement.

### To Probe Further

In this field, there has been much rapid progress. So we cannot find comprehensive literature on Japanese ITS. Therefore, below is some literature about specific fields.

Japanese ITS in general:

"ITS Handbook" edited by 2000

and the following several website are available for your further study:

http//:www.vics.or.jp/;www.moc.go.jp/;www.mpt.go.jp/;www.npa.go.jp/;www.miti.go.jp/;www.mot.go.jp/

In-vehicle navigation system:

For example, "Car-Navigation Systems" K. Yokouchi, H. Ideno, and M. Ota, Mitsubishi Electric Advance Vol. 91/Sep. 2000, 2000.

ITS deployment and ITS simulator:

1. "Driver Assistance Systems—industrial, psychological, and legal aspects" S. Becker, D. Randow, and J. Feldges, In *Proceeding of the Intelligent Vehicle Symposium,* 1998.
2. "Simulation environment for ITS—a real-time 3D simulator" M. Ikawa, H. Kumazawa, Y. Goto, H. Furusawa, and Y. Akemi, In *Proceeding of the 5th ITS World Congress,* 1998.
3. "A prototype of smart ways in ITS simulator" Y. Goto, M. Ikawa, and H. Kumazawa, In *Proceeding of the 6th ITS World Congress,* 1999.

# 23

# Media Signal Processing




Ruby Lee
*Princeton University*

Gerald G. Pechanek
*BOPS, Inc.*

Thomas C. Savell
*Creative Advanced Technology Center*

Sadiq M. Sait
Habib Youssef
Mohammad Faheemuddin
*King Fahd University of Petroleum & Minerals*


## 23.1 Instruction Set Architecture for Multimedia Signal Processing

*Ruby Lee*

### 23.1.1 Introduction

Multimedia signal processing, or media processing [1], is the processing of digital multimedia information in a programmable processor. Digital multimedia information includes visual information like images, video, graphics, and animation, audio information like voice and music, and textual information like keyboard text and handwriting. With general-purpose computers processing more multimedia information, multimedia instructions for efficient media processing have been defined for the instruction set architectures (ISAs) of microprocessors. Meanwhile, digital processing of video and audio data

in consumer products has also resulted in more sophisticated media processors. Traditional digital signal processors (DSPs) in music players and recorders and mobile telephones are becoming increasingly sophisticated as they process multiple forms of multimedia data, rather than just audio signals. Video processors for televisions and video recorders have become more versatile as they have to take into account high-fidelity audio processing and real-time three-dimensional (3-D) graphics animations. This has led to the design of more versatile media processors, which combine the capabilities of DSPs for efficient audio and signal processing, video processors for efficient video processing, graphics processors for efficient 2-D and 3-D graphics processing, and general-purpose processors for efficient and flexible programming. The functions performed by microprocessors and media processors may eventually converge. In this chapter, some of the key innovations in multimedia instructions added to microprocessor ISAs are described, which have allowed high-fidelity multimedia to be processed in real-time on ubiquitous desktop and notebook computers. Many of these features have also been adopted in modern media processors and DSPs.

### 23.1.2 Subword Parallelism

Workload characterization studies on multimedia applications show that media applications have huge amounts of data parallelism and operate on lower-precision data types. A pixel-oriented application, for example, rarely needs to process data that is wider than 16 bits. This translates into low computational efficiency on general-purpose processors where the register and datapath sizes are typically 32 or 64 bits, called the width of a word. Efficient processing of low-precision data types in parallel becomes a basic requirement for improved multimedia performance. This is achieved by partitioning a word into multiple *subwords,* each subword representing a lower-precision datum. A *packed data type* will be defined as data that consists of multiple subwords packed together. These subwords can be processed in parallel using a single instruction, called a *subword-parallel* instruction, a *packed* instruction, or a *microSIMD* instruction. SIMD stands for "single instruction multiple data," a term coined by Flynn [2] for describing very large parallel machines with many data processors, where the same instruction issued from a single control processor operates in parallel on data elements in the parallel data processors. Lee [3] coined the term microSIMD architecture to describe an ISA—where a single instruction operates in parallel on multiple subwords within a single processor.

Figure 23.1 shows a 32-bit integer register that is made up of four 8-bit subwords. The subwords in the register can be pixel values from a grayscale image. In this case, the register is holding four pixels with values 0xFF, 0x0F, 0xF0, and 0x00. The same 32-bit register can also be interpreted as two 16-bit subwords, in which case, these subwords would be 0xFF0F and 0xF000. The subword boundaries do not correspond to a physical boundary in the register file; they are merely how the bits in the word are interpreted by the program. If we have 64-bit registers, the most useful subword sizes will be 8-, 16-, or 32-bit words. A single register can then accommodate 8, 4, or 2 of these different sized subwords, respectively.

To exploit subword parallelism, packed parallelism, or microSIMD parallelism in a typical word-oriented microprocessor, new subword-parallel or packed instructions are added. (The terms "subword-parallel," "packed," and "microSIMD" are used interchangeably to describe operations, instructions and architectures.) The parallel processing of the packed data types typically requires only minor modifications to the word-oriented functional units, with the register file and the pipeline structure remaining unchanged. This results in very significant performance improvements for multimedia processing, at a very low cost (see Fig. 23.2).

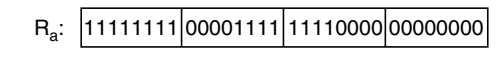



**FIGURE 23.1** 32-bit integer register made up of four 8-bit subwords.

Typically, packed arithmetic instructions such as `packed add` and `packed subtract` are first introduced. To support subword parallelism efficiently, other classes of new instructions such as subword permutation instructions are also needed. Typical subword-parallel instructions

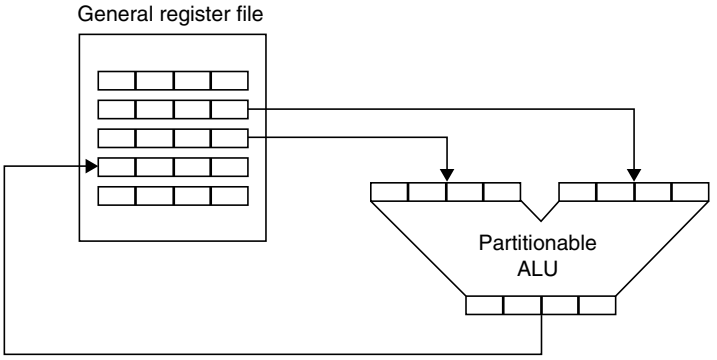



**FIGURE 23.2** MicroSIMD parallelism uses packed data types and a partitionable ALU.

are described in the rest of this chapter, pointing out interesting arithmetic or architectural features that have been added to support this style of microSIMD parallelism. In Section 23.1.3, `packed add` and `packed subtract` instructions described are, as well as several variants of these. These instructions can all be implemented on the basic Arithmetic Logical Units (ALUs) found in programmable processors, with minor modifications. Such partitionable ALUs are described in Section 23.1.3.1, *Saturation arithmetic*—one of the most interesting outcomes of subword-parallel additions—for efficiently handling overflows and performing in-line conditional operations is also described. A variant of packed addition is the `packed average` instruction, where unbiased rounding is an interesting associated feature. Another class of packed instructions that can use the ALU is the `parallel compare` instruction where the results are the outcomes of the subword comparisons.

Section 23.1.4 describes how packed integer multiplication is handled. Also described are different approaches to solving the problem of the products being twice as large as the subword operands that are multiplied in parallel. Although subword-parallel multiplication instructions generally require the introduction of new integer multiplication functional units to a microprocessor, the special case of multiplication by constants, which can be achieved very efficiently with `packed shift and add` instructions that can be implemented on an ALU with a small preshifter, is described.

Section 23.1.5 describes `packed shift` and `packed rotate` instructions, which perform a superset of the functions of a typical shifter found in microprocessors, in parallel, on packed subwords.

Section 23.1.6 describes a new class of instructions, not previously found in programmable processors that do not support subword parallelism. These are subword permutation instructions, which rearrange the order of the subwords packed in one or more registers. These permutation instructions can be implemented using a modified shifter, or as a separate permutation function unit (see Fig. 23.3).

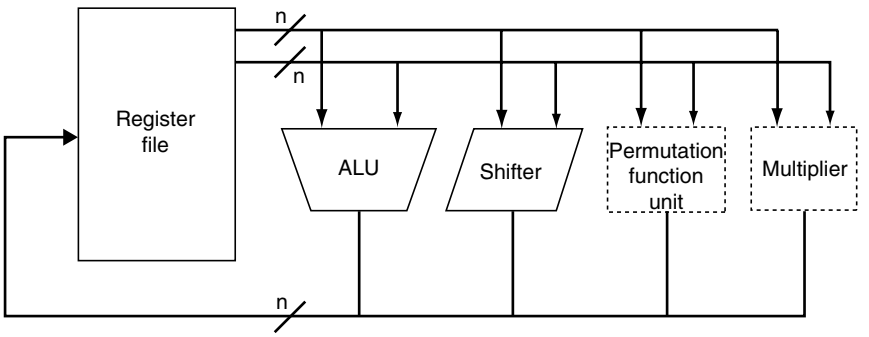



**FIGURE 23.3** Typical datapaths and functional units in a programmable processor.

To provide examples and illustrations, the following first and second generation multimedia instructions in microprocessor ISAs are used:

- IA-64 [4,5], MMX [6,7], and SSE-2 [8] from Intel
- MAX-2 [9,10] from Hewlett-Packard
- 3DNow!* 1 [11,12] from AMD
- AltiVec [13] from Motorola

#### 23.1.2.1 Historical Overview

The first generation multimedia instructions focused on subword parallelism in the integer domain. These are described and compared in [14]. The first set of multimedia extensions targeted at general-purpose multimedia acceleration, rather than just graphics acceleration, was MAX-1, introduced with the PA-7100LC processor in January 1994 [15,16] by Hewlett-Packard. MAX-1, an acronym for "multimedia acceleration extensions," is a minimalist set of multimedia instructions for the 32-bit PA-RISC processor [17]. An application that clearly illustrated the superior performance of MAX-1 was MPEG-1 video and audio decoding with software, at real-time rates of 30 frames per second [18]. For the first time, this performance was made possible using software on a general-purpose processor in a low-end desktop computer. Until then, such high-fidelity, real-time video decompression performance was not achievable without using specialized hardware. MAX-1 also accelerated pixel processing in graphics rendering and image processing, and 16-bit audio processing.

Next, Sun introduced VIS [19], which was an extension for the UltraSparc processors. VIS was a much larger set of multimedia instructions. In addition to packed arithmetic operations, VIS provided very specialized instructions for accessing visual data, stored in predetermined ways in memory.

Intel introduced MMX [6,7] multimedia extensions in the dominant Pentium microprocessors in January 1997, which immediately legitimized the valuable contribution of multimedia instructions for ubiquitous multimedia applications.

MAX-2 [9] was Hewlett-Packard's multimedia extension for its 64-bit PA-RISC 2.0 processors [10]. Although designed simultaneously with MAX-1, it was only introduced in 1996, with the PA-RISC 2.0 architecture. The subword permutation instructions introduced with MAX-2 were useful only with the increased subword parallelism in 64-bit registers. Like MAX-1, MAX-2 was also a minimalist set of general-purpose media acceleration primitives.

MIPS also described MDMX multimedia extensions and Alpha described a very small set of MVI multimedia instructions for video compression.

The second generation multimedia instructions initially focused on subword parallelism on the floating-point (FP) side for accelerating graphics geometry computations and high-fidelity audio processing. Both of these multimedia applications use single-precision, floating-point numbers for increased range and accuracy, rather than 8-bit or 16-bit integers. These multimedia ISAs include SSE and SSE-2 [8] from Intel and 3DNow! [11,12] from AMD. Finally, the PowerPC's AltiVec [13] and the Intel-HP IA-64 [4,5] multimedia instruction sets are comprehensive integer and floating-point multimedia instructions. Today, every microprocessor ISA and most media and DSP ISAs include subword-parallel multimedia instructions.

### 23.1.3 Packed Add and Packed Subtract Instructions

`Packed add` and `packed subtract` instructions are similar to ordinary `add` and `subtract` instructions, except that the operations are performed in parallel on the subwords of two source registers. `Add` (nonpacked) and `packed add` operations are shown in Figs. 23.4 and 23.5, respectively. The `packed add` in Fig. 23.5 uses source registers with four subwords each. The corresponding subwords from the two source registers are summed up, and the four sums are written to the target register. A `packed subtract` operation operates similarly.

*3DNow! may be considered as having two versions. In June 2000, 25 new instructions were added to the original 3DNow! specification. In this text, this extended 3DNow! architecture will be considered.

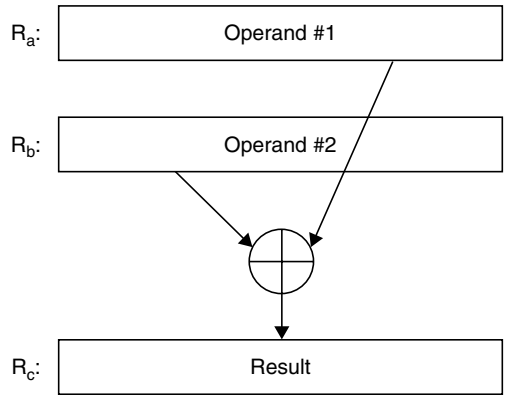



**FIGURE 23.4** `ADD` $R_c$, $R_a$, $R_b$: `Ordinary add` instruction.

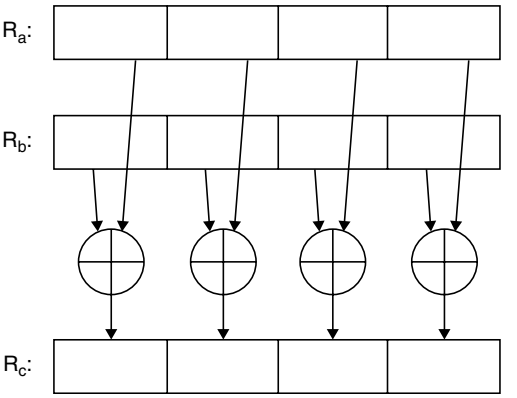



**FIGURE 23.5** `PADD` $R_c$, $R_a$, $R_b$: `Packed add` instruction.

### 23.1.3.1 Partitionable ALUs

Very minor modifications to the underlying functional units are needed to implement `packed add` and `packed subtract` instructions. Assume that we have an ALU with 32-bit integer registers, and we want to extend this ALU to perform a `packed add` that will operate on four 8-bit subwords in parallel. To achieve this, the carry propagation across the subword boundaries has to be blocked. Because each subword is interpreted as being independent of the neighboring subwords, by stopping the carry bits from affecting the neighboring subwords, the `packed add` operation can be realized.

In Fig. 23.6, the packed integer register $R_a = [0xFF|0x0F|0xF0|0x00]$ is being added to another packed register $R_b = [0x00|0xFF|0xFF|0x0F]$. The result is written to the target register $R_c$. In an ordinary add instruction, the overflows generated by the addition of the second and third subwords will propagate into the first two sums. The correct sums, however, can be achieved easily by blocking the carry bit propagation across the subword boundaries, which are spaced 8-bits apart from one another.

As shown in Fig. 23.7, a 2-to-1 multiplexer placed at the subword boundaries of the adder can be used to control the propagation or the blocking of the carry bits. If the instruction is a `packed add`, the multiplexer control is set such that a zero is propagated into the next subword. If the instruction is an ordinary add, the multiplexer control is set such that the carry from the previous stage is propagated. By placing such a multiplexer at each subword boundary and adding the control logic, partitionable ALUs are achieved at insignificant cost.

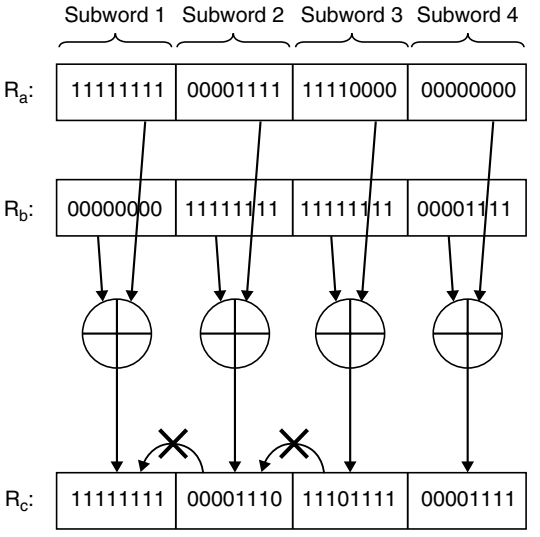



**FIGURE 23.6** In the `packed add` instruction, the carry bits are not propagated.

By using 3-to-1 multiplexers instead of 2-to-1 multiplexers, we can also implement `packed subtract` instructions. The multiplexer control is set such that:

- For `packed add` instructions, zero is propagated into the next stage.
- For `packed subtract` instructions, one is propagated into the next stage.
- For ordinary `add/subtract` instructions, the carry/borrow bit from the previous stage is propagated into the next stage.

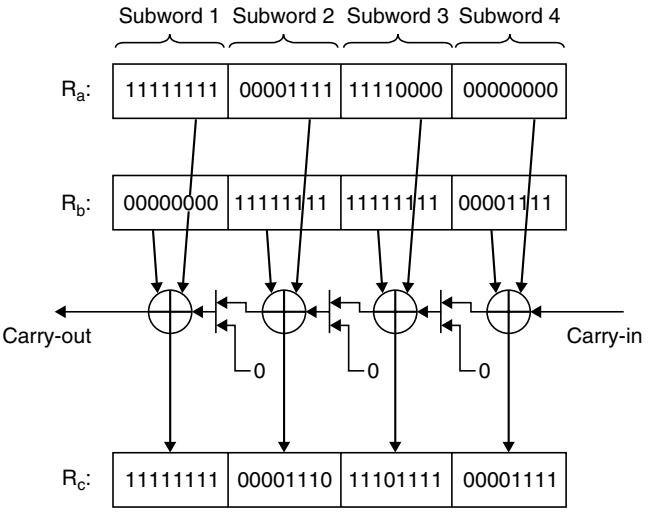



**FIGURE 23.7** Partitionable ALU: In `packed add` instructions, the multiplexers propagate zero; in ordinary `add` instructions, the multiplexers propagate carry-out from the previous stage into the carry-in of the next stage.

When a zero is propagated through the boundary into the next subword in the `packed add` instructions, we are essentially ignoring any overflow that might have been generated. Similarly, when a one is propagated through the boundary into the next subword in the `packed subtract` instructions, we are essentially ignoring any borrow that might have been generated. Ignoring overflows is equivalent to using modular arithmetic in `add` operations. Although modular arithmetic can be necessary or useful, other occasions arise when the carry bits should not be ignored and have to be handled differently.

### 23.1.3.2 Handling Parallel Overflows

Overflows in `packed add/subtract` instructions can be handled in the following ways:

- The overflow may be ignored (modular arithmetic).
- A *flag* bit may be set if at least one overflow is generated.
- Multiple flag bits (i.e., one flag bit for each addition operation on the subwords) may be set.
- A software overflow trap can be taken.
- Saturation arithmetic: the results are limited to a certain range. If the outcome of the operation falls outside this range, the corresponding limiting value will be the result.

Most nonpacked integer `add/subtract` instructions choose to ignore overflows and perform modular arithmetic. In modular arithmetic, the numbers wrap around from the largest representable number to the smallest representable number. For example, in 8-bit modular arithmetic, the operation $254 + 2$ will give a result of 0. The expected result, 256, is larger than the largest representable number, which is 255, and therefore is wrapped around to the smallest representable number, which is 0.

In multimedia applications, modular arithmetic frequently gives undesirable results. If the numbers in the previous example were pixel values in a grayscale image, by wrapping the values from 255 down to 0, white pixels would have converted into black ones. One solution to this problem is to use overflow traps, which are implemented in software.

A flag bit is an indicator bit that is set or cleared depending on the outcome of a particular operation. In the context of this discussion, an overflow flag bit is an indicator that is set when an `add` instruction generates an overflow. Occasions arise where the use of the flag bits are desirable. Consider a loop that

iterates many times and in each iteration, executes many add instructions. In this case, it is not desirable to handle overflows (by taking overflow trap routines) as soon as they occur, because this would negatively impact the performance by interrupting the execution of the loop body. Instead, the overflow flag can be set when the overflow occurs, and the program flow continues as if the overflow did not occur. At the end of each iteration, however, this overflow flag can be checked and the overflow trap can be executed if the flag turns out to be set. This way, the program flow would not be interrupted while the loop body executes.

An overflow trap can be used to *saturate* the results so that the aforementioned problems would not occur. A result that is greater than the largest representable value is replaced by that largest value. Similarly, a result that is less than the smallest representable value is replaced by that smallest value. One problem with this solution will be its negative effects to performance. An overflow trap is handled in software and may take many clock cycles to resolve. This can be acceptable only if the overflows are infrequent. For nonpacked `add/subtract` instructions, generation of an overflow on a 64-bit register by adding 8-bit quantities will be rare, so a software overflow trap will work well. This is not the case for packed arithmetic operations. Causing an overflow in an 8-bit subword is much more likely than in a 64-bit register. Also, since a 64-bit register may hold eight 8-bit subwords, multiple overflows can occur in a single execution cycle. In this case, handling the overflows by software traps could easily negate any performance gains from executing packed operations. The use of saturation arithmetic solves this problem.

#### 23.1.3.3 Saturation Arithmetic

Saturation arithmetic implements in hardware the work done by the overflow trap described above. The results falling outside the allowed numeric ranges are saturated to the upper and lower limits by hardware. This can handle multiple parallel overflows efficiently, without operating system intervention. Two types of overflows for arithmetic operations are:

- A *positive overflow* occurs when the result is larger than the largest value in the defined range for that result
- A *negative overflow* occurs when the result is smaller than the smallest value in the defined range for that result

If saturation arithmetic is used in an operation, the result is clipped to the maximum value in its defined range if a positive overflow occurs, and to the minimum value in its defined range if a negative overflow occurs.

For a given instruction, multiple saturation options may exist, depending on whether the operands and the result are treated as signed or unsigned integers. For an instruction that uses three registers (two for source operands and one for the result), there can be eight different saturation options. Each one of the three registers can be treated as containing either a signed or an unsigned integer, which gives $2^3$ possible combinations. Not all of the eight possible saturation options are equally useful. Only three of the eight possible saturation options are used in any of the multimedia ISAs surveyed:

a) ***sss*** (signed result–signed first operand–signed second operand): In this saturation option, the result and the two operands are all treated as signed integers. The most significant bit is considered the sign bit. Considering $n$-bit subwords, the result and operands are defined in the range $[-2^{n-1}, 2^{n-1}-1]$. If a positive overflow occurs, the result is saturated to $2^n-1$. If a negative overflow occurs, the result is saturated to $-2^{n-1}$. In an addition operation that uses the *sss* saturation option, since the operands are signed numbers, a positive overflow is possible only when both operands are positive. Similarly, a negative overflow is possible only when both operands are negative.

b) ***uuu*** (unsigned result–unsigned first operand–unsigned second operand): In this saturation option, the result and the two operands are all treated as unsigned integers. Considering $n$-bit integer subwords, the result and the operands are defined in the range $[0,2^n-1]$. If a positive overflow occurs, the result is saturated to $2^n-1$. If a negative overflow occurs, the result is

saturated to zero. In an addition operation that uses the *uuu* saturation option, since the operands are unsigned numbers, negative overflow is not a possibility; however, for a subtraction operation using the *uuu* saturation, negative overflow is possible, and any negative result will be clamped to zero as the smallest value.

c) ***uus*** (unsigned result–unsigned first operand–signed second operand): In this saturation option, the result and the first operand are treated as unsigned numbers, and the second operand is treated as a signed number. Although this may seem like an unusual option, it is very useful because it allows the addition of a signed increment to an unsigned pixel. It also allows negative numbers to be clipped to zero. Its implementation also has logical symmetry to the *sss* case.

In addition to the efficient handling of overflows, saturation arithmetic also facilitates several other useful computations. For instance, saturation arithmetic can also be used to clip results to arbitrary maximum or minimum values. Without saturation arithmetic, these operations could normally take up to five instructions for each pair of subwords. That would include instructions to check for upper and lower bounds and then to perform the clipping. Using saturation arithmetic, however, this effect can be achieved in as few as two instructions for all the pairs of packed subwords.

Saturation arithmetic can also be used for in-line conditional execution, reducing the need for conditional branches that can cause significant performance degradations in pipelined processors. Some examples are the `packed maximum` and `packed absolute difference` operations shown in Figs. 23.8a, b.

Table 23.1 contains examples of operations that can be performed using saturation arithmetic [15]. All of the instructions in the table use three registers. The first register is the target register. The second and the third registers hold the first and the second operands respectively. PADD and PSUB denote `packed add` and `packed subtract` instructions. The three-letter field after the instruction mnemonic specifies which saturation option is to be used. If this field is empty, modular arithmetic is assumed. All the examples in the table operate on 16-bit integer subwords.

Table 23.2 contains a summary of the register and subword sizes and the saturation options found in different multimedia ISAs. Table 23.3 is a summary of the `packed add/subtract` instructions in

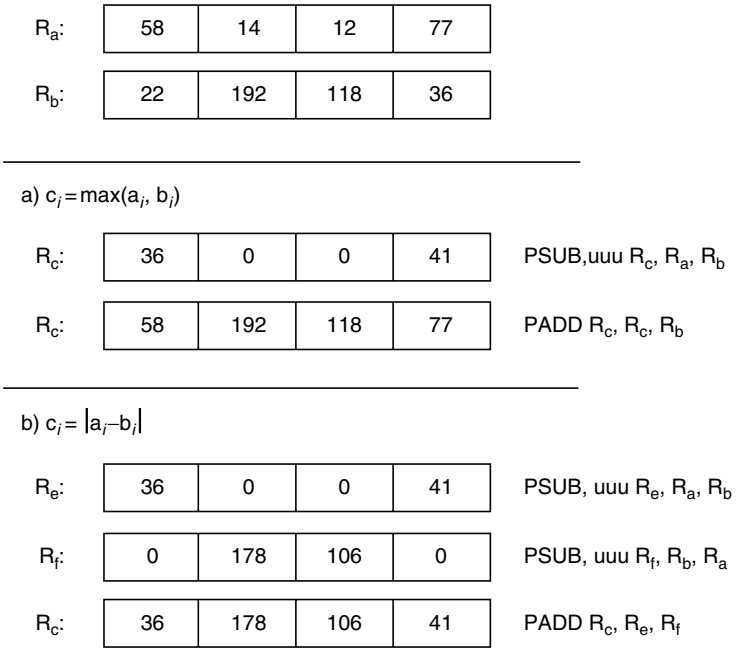



**FIGURE 23.8** (a) `Packed maximum` operation using saturation arithmetic. (b) `Packed absolute difference` operation using saturation arithmetic.

**TABLE 23.1** Examples of Operations That Are Facilitated by Saturation Arithmetic

| Operation | Instruction Sequence | Notes |
|---|---|---|
| Clip $a_i$ to an arbitrary maximum value $v_{max}$, where $v_{max} < 2^{15} - 1$. | PADD.*sss* $R_a$, $R_a$, $R_b$ | $R_b$ contains the value $(2^{15} - 1 - v_{max})$. If $a_i > v_{max}$, this operation clips $a_i$ to $2^{15} - 1$ on the high end. |
| | PSUB.*sss* $R_a$, $R_a$, $R_b$ | $a_i$ is at most $v_{max}$. |
| Clip $a_i$ to an arbitrary minimum value $v_{min}$, where $v_{min} > -2^{15}$. | PSUB.*sss* $R_a$, $R_a$, $R_b$ | $R_b$ contains the value $(-2^{15} + v_{min})$. If $a_i < v_{min}$, this operation clips $a_i$ to $-2^{15}$ at the low end. |
| | PADD.*sss* $R_a$, $R_a$, $R_b$ | $a_i$ is at least $v_{min}$. |
| Clip $a_i$ to within the arbitrary range $[v_{min}, v_{max}]$, where $-2^{15} < v_{min} < v_{max} < 2^{15} - 1$. | PADD.*sss* $R_a$, $R_a$, $R_b$ | $R_b$ contains the value $(2^{15} - 1 - v_{max})$. This operation clips $a_i$ to $2^{15} - 1$ on the high end. |
| | PSUB.*sss* $R_a$, $R_a$, $R_d$ | $R_d$ contains the value $(2^{15} - 1 - v_{max} + 2^{15} - v_{min})$. This operation clips $a_i$ to $-2^{15}$ at the low end. |
| | PADD.*sss* $R_a$, $R_a$, $R_e$ | $R_e$ contains the value $(-2^{15} + v_{min})$. This operation clips $a_i$ to $v_{max}$ at the high end and to $v_{min}$ at the low end. |
| Clip the signed integer $a_i$ to an unsigned integer within the range $[0, v_{max}]$, where $0 < v_{max} < 2^{15} - 1$. | PADD.*sss* $R_a$, $R_a$, $R_b$ | $R_b$ contains the value $(2^{15} - 1 - v_{max})$. This operation clips $a_i$ to $2^{15} - 1$ at the high end. |
| | PSUB.*uus* $R_a$, $R_a$, $R_b$ | This operation clips $a_i$ to $v_{max}$ at the high end and to zero at the low end. |
| Clip the signed integer $a_i$ to an unsigned integer within the range $[0, 2^{16}]$. | PADD.*uus* $R_a$, 0, $R_a$ | If $a_i < 0$, then $a_i = 0$ else $a_i = a_i$. |
| | | If $a_i$ was negative, it gets clipped to zero, else remains same. |
| $c_i = \max(a_i, b_i)$ | PSUB.*uuu* $R_c$, $R_a$, $R_b$ | If $a_i > b_i$, then $c_i = (a_i - b_i)$ else $c_i = 0$. |
| `Packed maximum` operation | PADD $R_c$, $R_b$, $R_c$ | If $a_i > b_i$, then $c_i = a_i$ else $c_i = b_i$. |
| $c_i = \lvert a_i - b_i \rvert$ | PSUB.*uuu* $R_e$, $R_a$, $R_b$ | If $a_i > b_i$, then $e_i = (a_i - b_i)$ else $e_i = 0$. |
| `Packed absolute difference` operation | PSUB.*uuu* $R_f$, $R_b$, $R_a$ | If $a_i <=$ then $f_i = (b_i - a_i)$ else $f_i = 0$. |
| | PADD $R_c$, $R_e$, $R_f$ | If $a_i > b_i$, then $c_i = \lvert a_i - b_i \rvert$, else $c_i = \lvert b_i - a_i \rvert$. |

*Note*: $a_i$ and $b_i$ are the subwords in the registers $R_a$ and $R_b$, respectively, where $i = 1, 2, \ldots, k$, and $k$ denotes the number of subwords in a register. Subword size $n$, is assumed to be two bytes (i.e., $n = 16$) for this table.

several multimedia ISAs. The first column contains descriptions of common packed instructions. The symbols $a_i$ and $b_i$ represent the corresponding subwords from the two source registers. The symbol $c_i$ represents the corresponding subword in the target register.

The IA-64* architecture has 64-bit integer registers. `Packed add` and `packed subtract` instructions are supported for subword sizes of 1, 2, and 4 bytes. Modular arithmetic is defined for all subword sizes whereas the saturation options (*sss, uuu*, and *uus*) exist for only 1 and 2-byte subwords.

**TABLE 23.2** Summary of the Integer Register, Subword Sizes, and Subtraction Options Supported by the Different Architectures

| Architectural Feature | IA-64 | MAX-2 | MMX | SSE-2 | AltiVec |
|---|---|---|---|---|---|
| Size of integer registers (bits) | 64 | 64 | 64 | 128 | 128 |
| Supported subword sizes (bytes) | 1, 2, 4 | 2 | 1, 2, 4 | 1, 2, 4, 8 | 1, 2, 4 |
| Modular arithmetic | Y | Y | Y | Y | Y |
| Supported saturation options | *sss, uuu, uus* for 1, 2 byte | *sss, uus* for 2 byte | *sss, uuu* for 1, 2 byte | *sss, uuu* for 1, 2 byte | *uuu, sss* for 1, 2, 4 byte |

*All the discussions in this chapter consider Intel's IA-64 as the base architecture. Evaluations of the other architectures are generally carried out by comparisons to IA-64.

**TABLE 23.3** Summary of the `packed add` and `packed subtract` Instructions and Variants

| Integer Operations | IA-64 | MAX-2 | MMX | SSE-2 | 3DNow! | AltiVec |
|---|---|---|---|---|---|---|
| $c_i = a_i + b_i$ | √ | √ | √ | √ | | √ |
| $c_i = a_i + b_i$ (with saturation) | √ | √ | √ | | √ | |
| $c_i = a_i - b_i$ | √ | √ | √ | √ | | √ |
| $c_i = a_i - b_i$ (with saturation) | √ | √ | √ | | | √ |
| $c_i = average\ (a_i, b_i)$ | √ | √ | | √ | √ | √ |
| $c_i = average\ (a_i, -b_i)$ | √ | | | | | |
| $[c_{2i}, c_{2i+1}] = [a_{2i} + a_{2i+1}, b_{2i} + b_{2i+1}]$ | | | | | | √ |
| $lsbit(c_i) = carryout(a_i + b_i)$ | | | | | | √ |
| $lsbit(c_i) = carryout(a_i - b_i)$ | | | | | √ | |
| $c_i = compare(a_i, b_i)$ | √ | | √ | | | √ |
| `Move mask` | | | | √ | √ | |
| $c_i = \max(a_i, b_i)$ | √ | √[a] | | √ | √ | √ |
| $c_i = \min(a_i, b_i)$ | √ | √[a] | | √ | √ | √ |
| $c = \Sigma\lvert a_i - b_i\rvert$ | √ | √[a] | | √ | √ | |

[a] This operation is realized by using saturation arithmetic.

The PA-RISC MAX-2 architecture also has 64-bit integer registers. `Packed add` and `packed substract` instructions operate on only 2-byte subwords. MAX-2 instructions support modular arithmetic, and the *sss* and *uus* saturation options.

The IA-32 MMX architecture defines eight 64-bit registers for use by the multimedia instructions. Although these registers are referred to as separate registers, they are aliased to the registers in the FP data register stack. Supported subword sizes are 1, 2, and 4 bytes. Modular arithmetic is defined for all subword sizes whereas the saturation options (*sss* and *uus*) exist for only 1- and 2-byte subwords.

The IA-32 SSE-2 technology introduces a new set of eight 128-bit FP registers to the IA-32 architecture. Each of the 128-bit registers can accommodate four single-precision (SP) or two double-precision (DP) numbers. Moreover, these registers can also be used to accommodate packed integer data types. Integer subword sizes can be 1, 2, 4, or 8 bytes. Modular arithmetic is defined for all subword sizes whereas the saturation options (*sss* and *uus*) exist for only 1- and 2-byte subwords.

The PowerPC AltiVec architecture has thiry-two 128-bit registers for multimedia instructions. `Packed add/subtract` instructions are supported for 1-, 2-, and 4-byte subwords. Modular or saturation arithmetic (*uuu* or *sss*) can be used, although *sss* saturation is only supported for `packed add`.

### 23.1.3.4 Packed Average

`Packed average` instructions are very common in media applications such as pixel averaging in MPEG-2 encoding, motion compensation, and video scaling. In a `packed average`, the pairs of corresponding subwords in the two source registers are added to generate intermediate sums. Then, the intermediate sums are shifted right by one bit, so that any overflow bit is shifted in on the left as the most significant bit. The beauty of the average operation is that no overflow can occur, and two operations (add followed by a one bit right shift) are performed in one operation. In a `packed average` instruction, $2n$ operations are performed in a single cycle, where $n$ is the number of subwords. In fact, even more operations are performed in a `packed average` instruction, if the rounding applied to the least significant end of the result is considered. Here, two different rounding options have been used:

- *Round away from zero*: A one is added to the intermediate sums, before they are shifted to the right by one bit position. If carry bits were generated during the addition operation, they are inserted into the most significant bit position during the shift right operation (see Fig. 23.9).
- *Round to odd*: Instead of adding one to the intermediate sums, a much simpler OR operation is used. The intermediate sums are directly shifted right by one bit position, and the last two bits of each of the subwords of the intermediate sums are ORed to give the least significant bit of the final result. This makes sure that the least significant bit of the final results are set to 1 (odd) if at least one of the two least-significant bits of the intermediate sums are 1 (see Fig. 23.10).

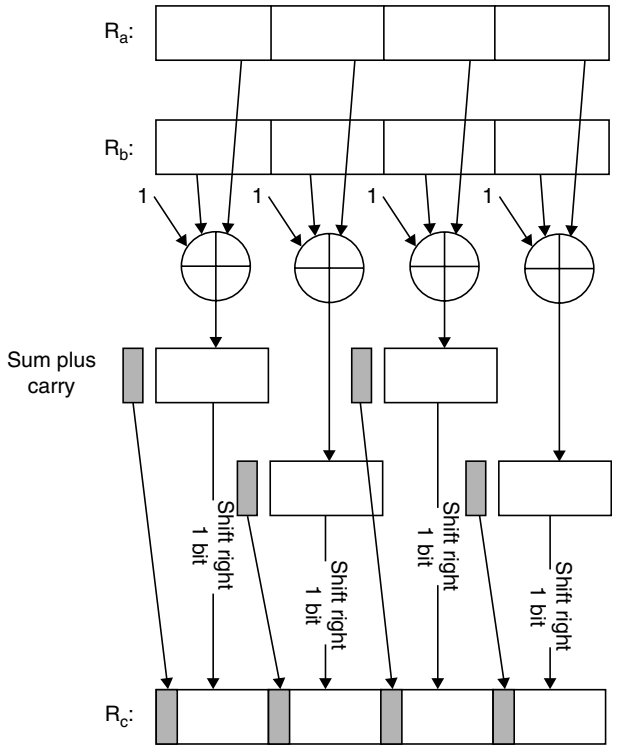



**FIGURE 23.9** `PAVG` $R_c$, $R_a$, $R_b$: `Packed average` instruction using the *round away from zero* option.

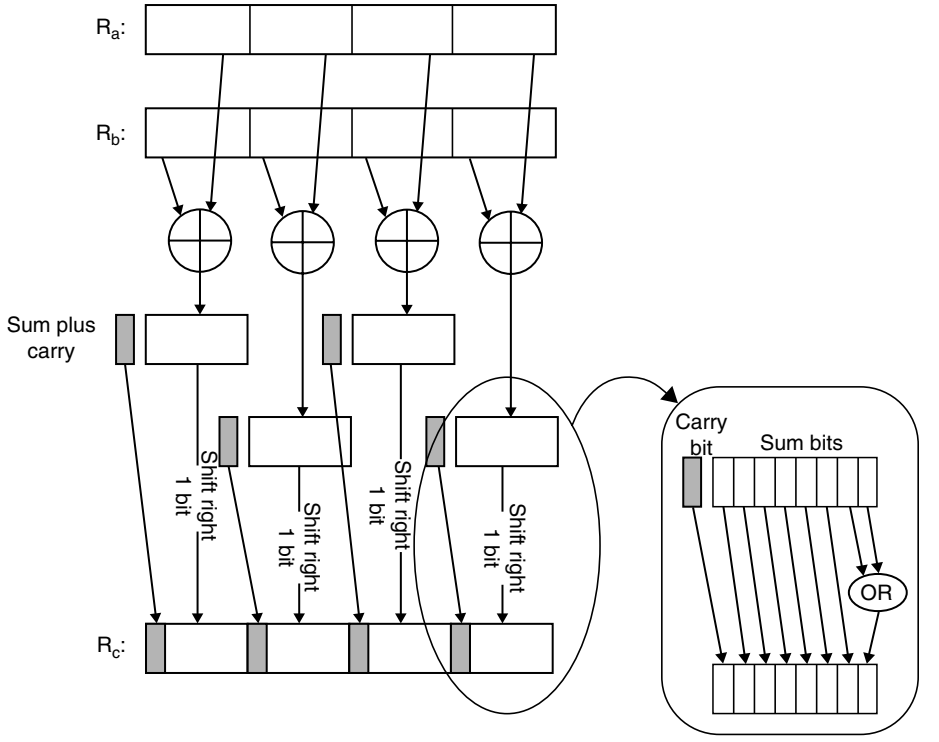



**FIGURE 23.10** `PAVG` $R_c$, $R_a$, $R_b$: `Packed average` instruction using the *round to odd* option. (From Intel, IA-Architecture Software Developer's Manual, Vol. 3, Instruction Set Reference, Rev. 1.1, July 2000. With permission.)

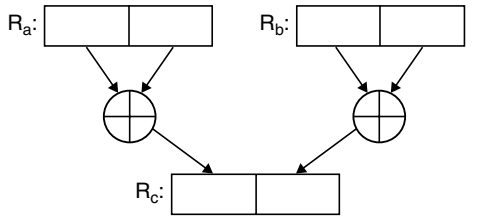



**FIGURE 23.11** `ACC` $R_c$, $R_a$, $R_b$: `Accumulate integer` working on registers with two subwords.

This rounding mode also performs *unbiased rounding* under the following assumptions. If the intermediate result is uniformly distributed over the range of possible values, then half of the time the bit shifted out is zero, and the result remains unchanged with rounding. The other half of the time the bit shifted out is one: if the next least significant bit is one, then the result loses −0.5, but if the next least significant bit is a zero, then the result gains +0.5. Because these cases are equally likely with a uniform distribution of the result, the *round to odd* option tends to cancel out the cumulative averaging errors that may be generated with repeated use of the averaging instruction.

### 23.1.3.5 Accumulate Integer

Sometimes, it is useful to add adjacent subwords in the same register. This can, for example, facilitate the accumulation of streaming data. An `accumulate integer` instruction performs an addition of the subwords in the same register and places the sum in the upper half of the target register, while repeating the same process for the second source register and using the lower half of the target register (Fig. 23.11).

### 23.1.3.6 Save Carry Bits

This instruction saves the carry bits from a `packed add` operation, rather than the sums. Figure 23.12 shows such a `save carry bits` instruction in AltiVec: a `packed add` is performed and the carry bits are written to the least significant bit of each result subword in the target register. A similar instruction saves the borrow bits generated when performing `packed subtract` instead of `packed add`.

### 23.1.3.7 Packed Compare Instructions

Sometimes, it is necessary to compare pairs of subwords. In a `packed compare` instruction, pairs of subwords are compared according to the relation specified by the instruction. If the condition is true for a subword pair, the corresponding field in the target register is written with a 1-mask. If the condition is false, the corresponding field in the target register is written with a 0-mask. Alternatively, a true or false

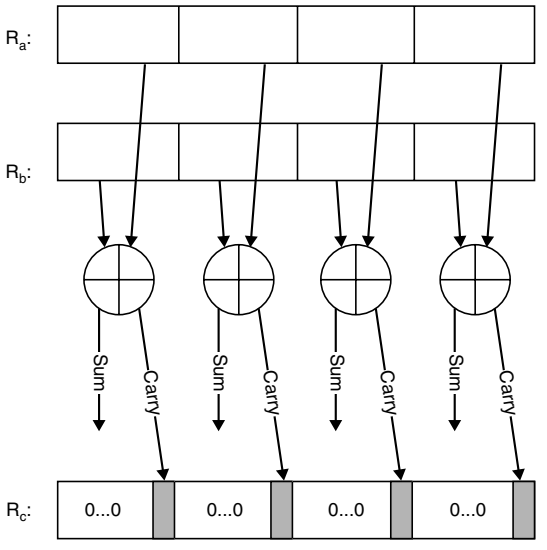



**FIGURE 23.12** `Save carry bits` instruction.

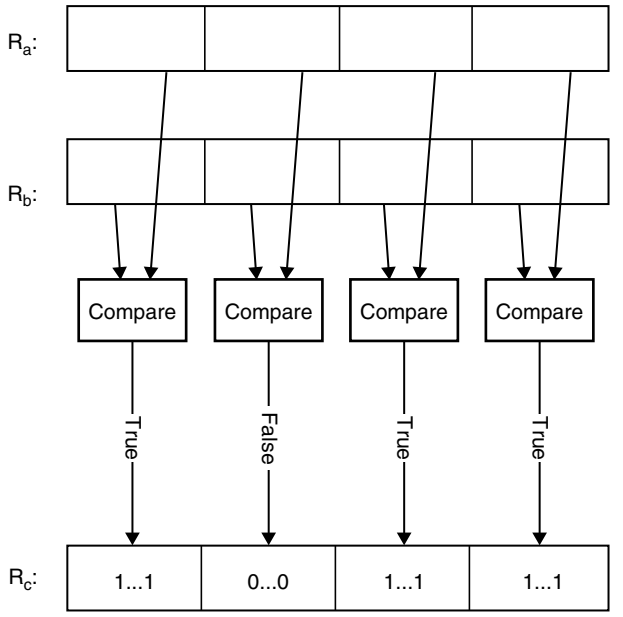



**FIGURE 23.13** `Packed compare` instruction. Bit masks are generated as a result of the comparisons made.

bit is generated for each subword, and this set of bits is written into the least significant bits of the result register. Some of the architectures have compare instructions that allow comparison of two numbers for all of the 10 possible relations,* whereas others only support a subset of the most frequent relations. A typical `packed compare` instruction is shown in Fig. 23.13 for the case of four subwords.

When a mask of bits is generated as in Fig. 23.13, often a `move mask` instruction is also provided. In a `move mask` instruction, the most significant bits of each of the subwords are picked, and these bits are placed into the target register, in a right aligned field (see Fig. 23.14). In different algorithms, either the subword mask format generated in Fig. 23.13 or the bit mask format generated in Fig. 23.14 is more useful.

Two common comparisons used are finding the larger of a pair of numbers, or the smaller of a pair of numbers. In the `packed maximum` instruction, the greater of the subwords in the compared pair gets written to the corresponding subword in the target register (see Fig. 23.15). Similarly, in the `packed`

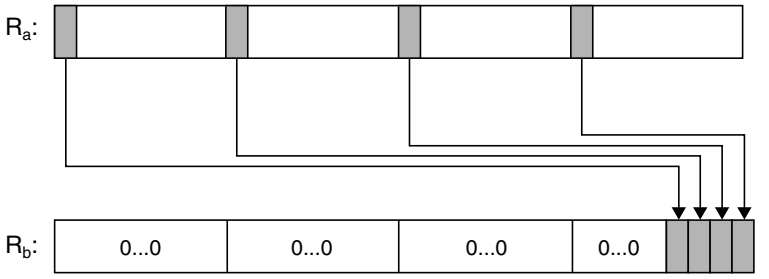



**FIGURE 23.14** `Move mask` $R_b$, $R_a$.

*Two numbers *a* and *b* can be compared for one of the following 10 possible relations: equal, less-than, less-than-or-equal, greater-than, greater-than-or-equal, not-equal, not-less-than, not-less-than-or-equal, not-greater-than, not-greater-than-or-equal. Typical notation for these relations are as follows respectively: =, <, <=, >, >=, !=, !<, !<=, !>, !>=.

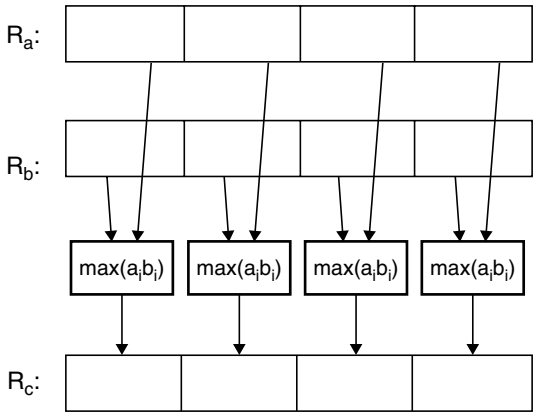



**FIGURE 23.15** `Packed maximum` instruction.

`minimum` instruction, the smaller of the subwords in the compared pair gets written to the corresponding subword in the target register. As described in the earlier section on saturation arithmetic, instead of special instructions for `packed maximum` and `packed minimum`, MAX-2 performs `packed maximum` and `packed minimum` operations by using `packed add` and `packed subtract` instructions with saturation arithmetic (see Fig. 23.8). An ALU can be used to implement comparisons, `maximum` and `minimum` instructions with a subtraction operation; comparisons for equality or inequality is usually done with an exclusive-or operation, also available in most ALUs.

#### 23.1.3.8 Sum of Absolute Differences

A more complex, multi-cycle instruction is the `sum of absolute differences (SAD)` instruction (see Fig. 23.16). This is used for motion estimation in MPEG-1 and MPEG-2 video encoding, for example. In a SAD instruction, the two packed operands are subtracted from one another. Absolute values of the resulting differences are then summed up.

Although useful, the SAD instruction is a multi-cycle instruction with a typical latency of three cycles. This can complicate the pipeline control of otherwise single cycle integer pipelines. Hence, minimalist multimedia instruction sets like MAX-2 do not have SAD instructions. Instead, MAX-2 uses generic `packed add` and `packed subtract` instructions with saturation arithmetic to perform the SAD operation (see Fig. 23.8b and Table 23.1).

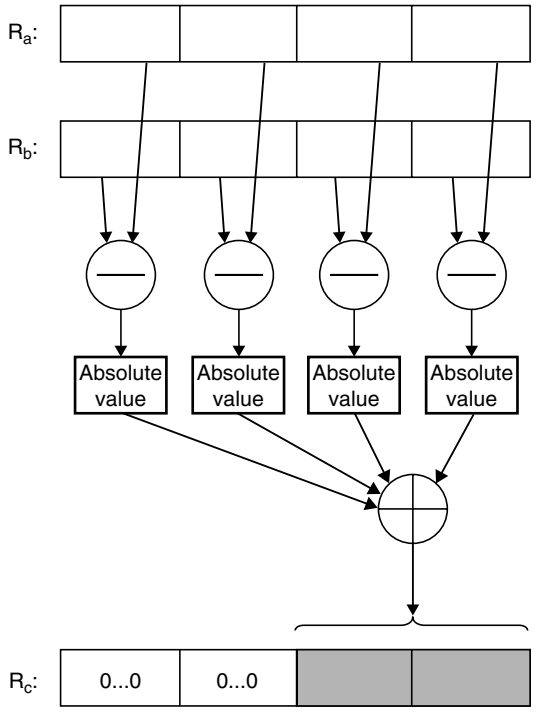



**FIGURE 23.16** `SAD` $R_c$, $R_a$, $R_b$: `Sum of absolute differences` instruction.

### 23.1.4 Packed Multiply Instructions

#### 23.1.4.1 Multiplication of Two Packed Integer Registers

The main difficulty with packed multiplication of two *n*-bit integers is that the product is twice as long as each operand. Consider the case where the register size is 64 bits and the subwords are 16 bits. The result of the packed multiplication will be four 32-bit products, which cannot be accommodated in a single 64-bit target register.

One solution is to use two `packed multiply` instructions. Figure 23.17 shows a `packed multiply high` instruction, which places only the more significant upper halves of the products into the target register. Figure 23.18 shows a `packed multiply low` instruction, which places only the less significant lower halves of the products into the target register.

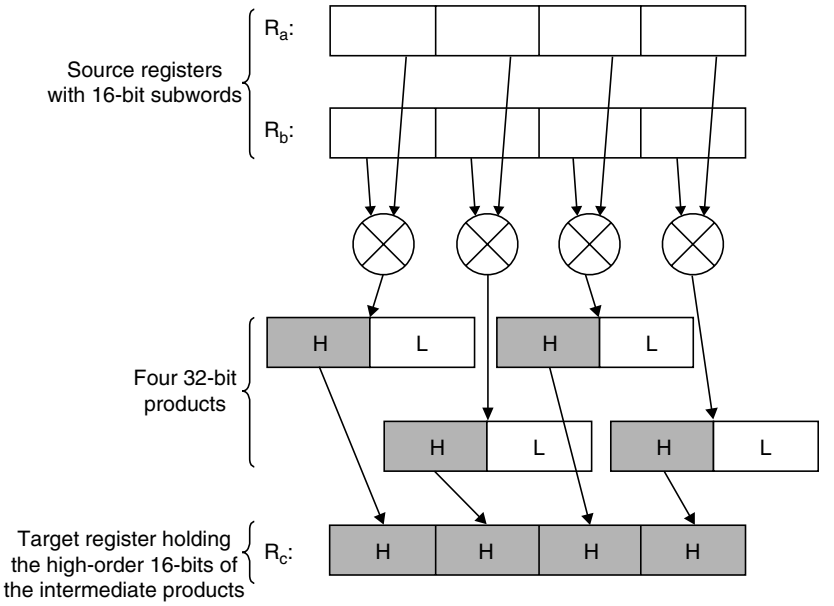



**FIGURE 23.17** `Packed multiply high` instruction.

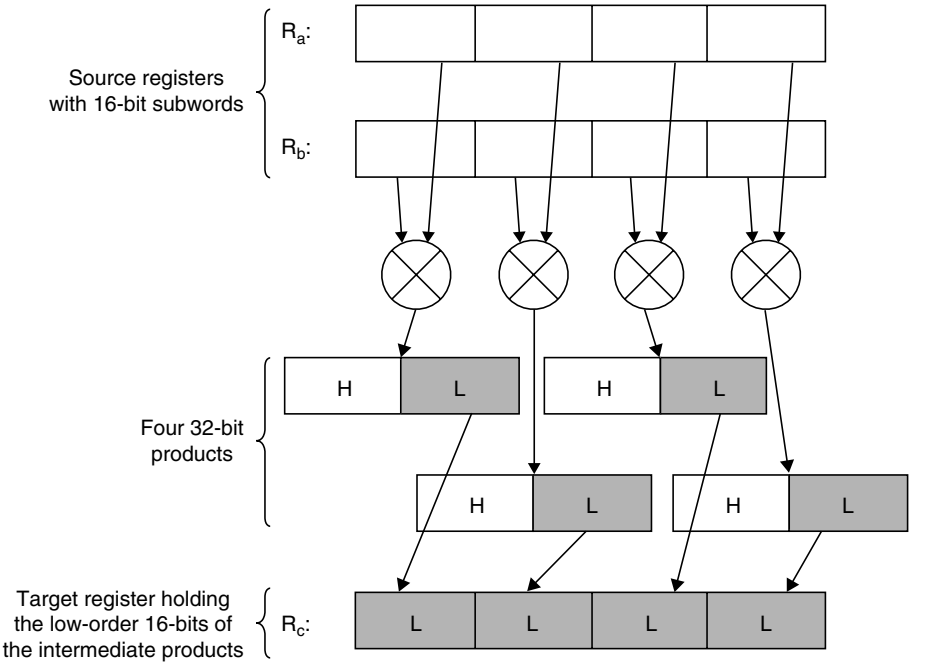



**FIGURE 23.18** `Packed multiply low` instruction.

IA-64 generalizes this with its `packed multiply and shift right` instruction (see Fig. 23.19), which does a parallel multiplication followed by a right shift. Instead of being able to choose either the upper or the lower half of the products to be put into the target register, it allows multiple* different

*In IA-64 the right-shift amounts are limited to 0, 7, 15, or 16 bits, so that only 2 bits in the `packed multiply and shift right` instruction are needed to encode the four shift amounts.

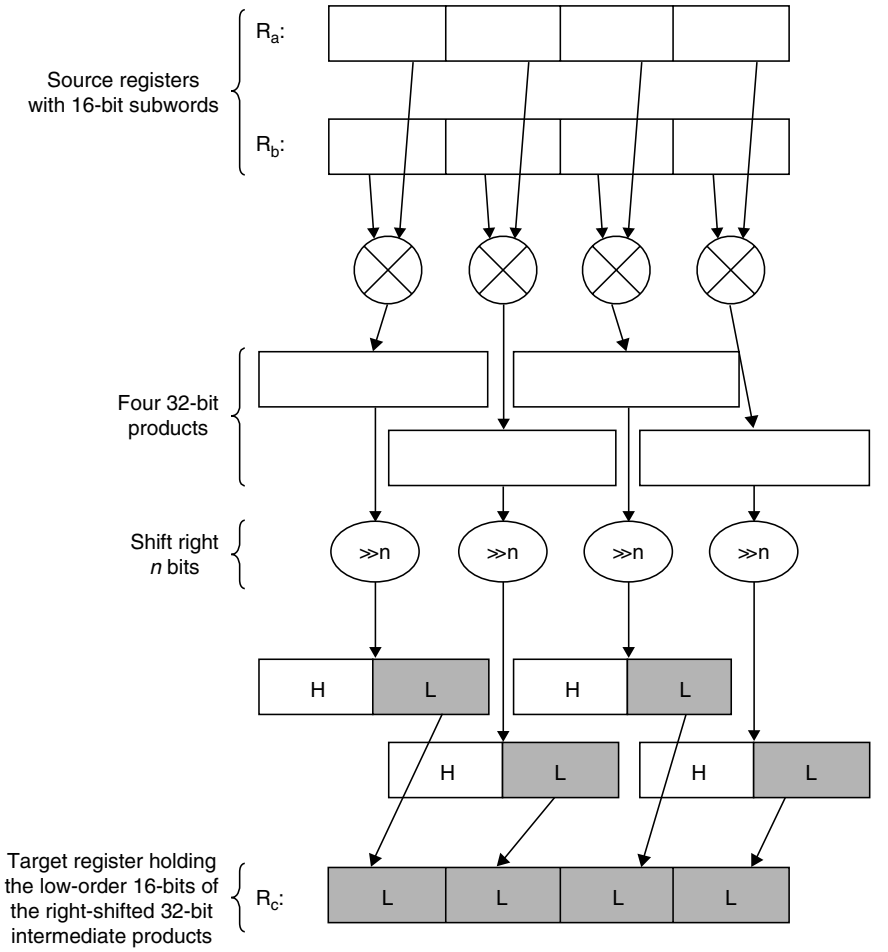



**FIGURE 23.19** The generalized `packed multiply and shift right` instruction.

16-bit fields from each of the 32-bit products to be chosen and placed in the target register. Ideally, saturation arithmetic is applied to the shifted products, to guard for the loss of significant "1" bits in selecting the 16-bit results.

IA-64 also allows the full product to be saved, but for only half of the pairs of source subwords. Either the odd or the even indexed subwords are multiplied. This makes sure that only as many full products as can be accommodated in one target register are generated. These two variants, the `packed multiply left` and `packed multiply right` instructions, are depicted in Figs. 23.20 and 23.21.

Another variant is the `packed multiply and accumulate` instruction. Normally, a `multiply and accumulate` operation requires three source registers. The `PMADDWD` instruction in MMX requires only two source registers by performing a `packed multiply` followed by an addition of two adjacent subwords (see Fig. 23.22).

Instructions in the AltiVec architecture may have up to three source registers. Hence, AltiVec's `packed multiply and accumulate` uses three source registers. In Fig. 23.23, the instruction packed multiply high and accumulate starts just like a `packed multiply` instruction, selects the more significant halves of the products, then performs a `packed add` of these halves and the values from a third register. The instruction `packed multiply low and accumulate` is the same, except that only the less significant halves of the products are added to the subwords from the third register.

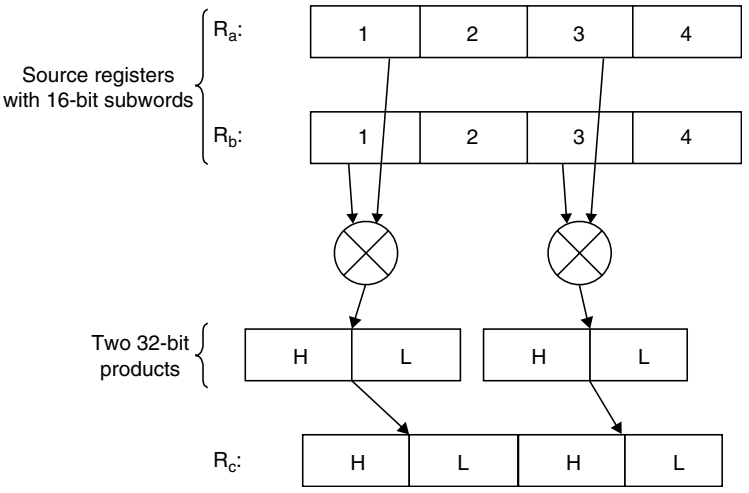



**FIGURE 23.20** `Packed multiply left` instruction where only the odd indexed subwords of the two source registers are multiplied.

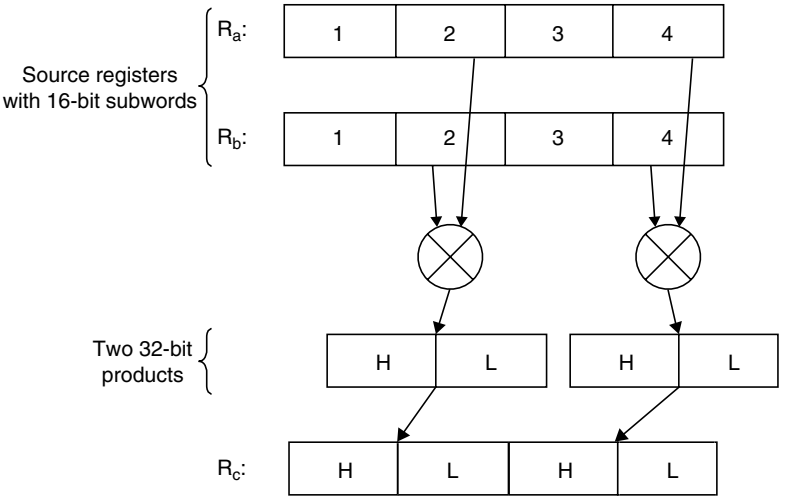



**FIGURE 23.21** `Packed multiply right` instruction where only the even indexed subwords of the two source registers are multiplied.

### 23.1.4.2 Multiplication of a Packed Integer Register by an Integer Constant

Many multiplications in multimedia applications are with constants, instead of variables. For example, in the inverse discrete cosine transform (IDCT) used in the compression and decompression of JPEG images and MPEG-1 and MPEG-2 video, all the multiplications are by constants. This type of multiplication can be further optimized for simpler hardware, lower power, and higher performance simultaneously by using `packed shift and add` instructions [14,15,20]. Shifting a register left by $n$ bits is equivalent to multiplying it by $2^n$. Since a constant number can be represented as a binary sequence of ones and zeros, using this number as a multiplier is equivalent to a left shift of the multiplicand of $n$ bits for each $n$th position where there is a 1 in the multiplier and an add of each shifted value to the result register.

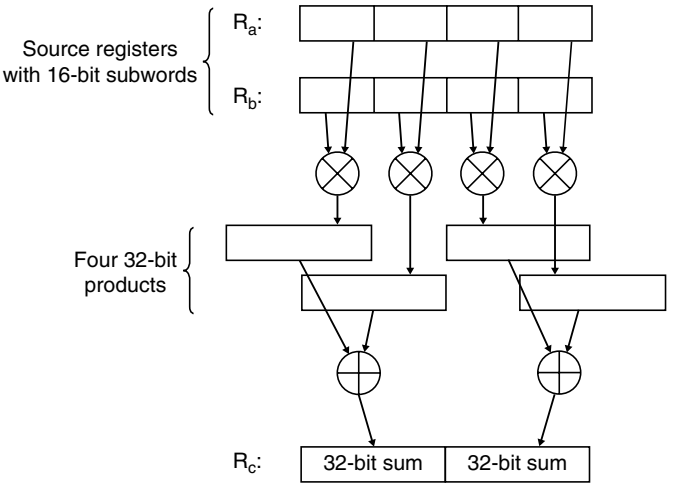



**FIGURE 23.22** `Packed multiply and accumulate` instruction in MMX.

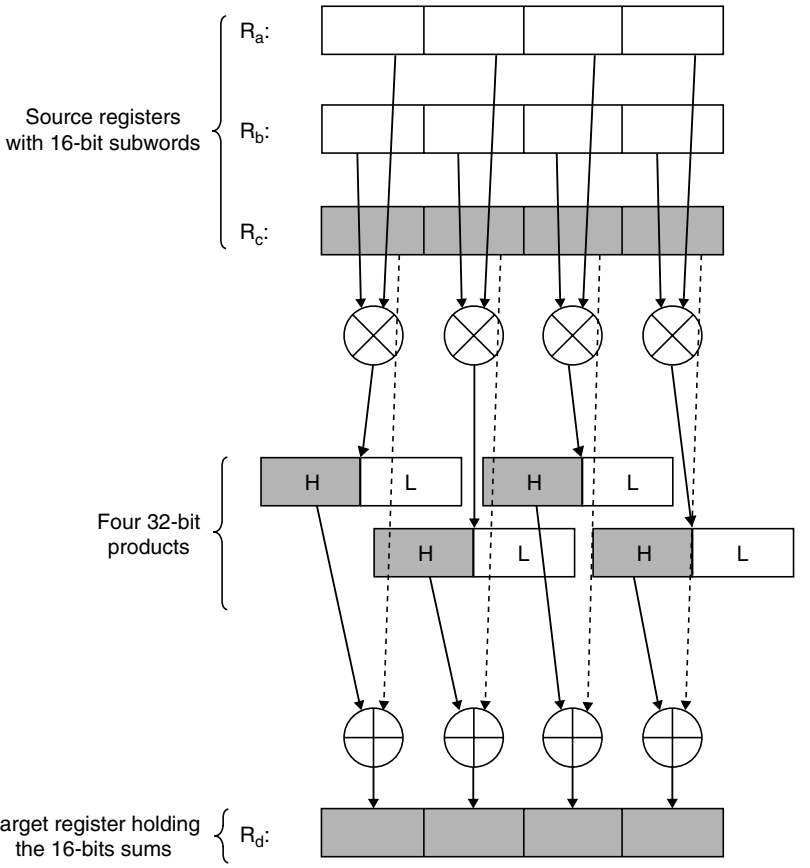



**FIGURE 23.23** In the `packed multiply high and accumulate` instruction in AltiVec, only the high-order bits of the intermediate products are used in the addition.

As an example, consider multiplying the integer register $R_a$ with the constant $C = 11$. The following instruction sequence performs this multiplication. Assume $R_a$ initially contains the value 6.

*Initial values:* $C = 11 = 1011_2$ and $R_a = 6 = 0110_2$

| Instruction | Operation | Result |
|---|---|---|
| `Shift left 1 bit` $R_b$ `,` $R_a$ | $R_b = R_a \ll 1$ | $R_b = 1100_2 = 12$ |
| `Add` $R_b$ `,` $R_b$ `,` $R_a$ | $R_b = R_b + R_a$ | $R_b = 1100_2 + 0110_2 = 010010_2 = 18$ |
| `Shift left 3 bit` $R_c$`,` $R_a$ | $R_c = R_a \ll 3$ | $R_c = 0110_2 * 8 = 110000_2 = 48$ |
| `Add` $R_b$ `,` $R_b$ `,` $R_c$ | $R_b = R_b + R_c$ | $R_b = 010010_2 + 110000_2 = 1000010_2 = 66$ |

This sequence can be shortened by combining the `shift left` and the `add` instructions into one new `shift left and add` instruction. The following new sequence performs the same multiplication in half as many instructions and uses one less register.

*Initial values:* $C = 11 = 1011_2$ and $R_a = 6 = 0110_2$

| Instruction | Operation | Result |
|---|---|---|
| `Shift left 1 bit and add` $R_b$`,` $R_a$`,` $R_a$ | $R_b = R_a \ll 1 + R_a$ | $R_b = 18$ |
| `Shift left 3 bit and add` $R_b$`,` $R_a$`,` $R_b$ | $R_b = R_a \ll 3 + R_b$ | $R_b = 66$ |

Multiplication of packed integer registers by integer constants uses the same idea. The `shift left and add` instruction becomes a `packed shift left and add` instruction to support the packed data types. As an example consider multiplying the subwords of the packed integer register $R_a = [1|2|3|4]$ by the constant $C = 11$. The instructions to perform this operation are:

*Initial values:* $C = 11 = 1011_2$ and $R_a = [1|2|3|4] = [0001|0010|0011|0100]_2$

| Instruction | Operation | Result |
|---|---|---|
| `Shift left 1 bit and add` $R_b$ `,` $R_a$ `,` $R_a$ | $R_b = R_a \ll 1 + R_a$ | $R_b = [3\|6\|9\|12]$ |
| `Shift left 3 bit and add` $R_b$ `,` $R_a$ `,` $R_b$ | $R_b = R_a \ll 3 + R_b$ | $R_b = [11\|22\|33\|44]$ |

The same reasoning used for multiplication by integer constants applies to multiplication by fractional constants. Arithmetic right shift of a register by $n$ bits is equivalent to dividing it by $2^n$. Using a fractional constant as a multiplier is equivalent to an arithmetic right shift of the multiplicand by $n$ bits for each $n$th position where there is a 1 in the multiplier and an add of each shifted value to the result register. By using a `packed arithmetic shift right and add` instruction, the `shift` and the `add` instructions can be combined into one to further speed such computations. For instance, multiplication of a packed register by the fractional constant $0.011_2$ ($= 0.375$) can be performed by using just two `packed arithmetic shift right and add` instructions.

*Initial values:* $C = 0.375 = 0.011_2$ and $R_a = [1|2|3|4] = [0001|0010|0011|0100]_2$

| Instruction | Operation | Result |
|---|---|---|
| `Arithmetic shift right 3 bit and add` $R_b$ `,` $R_a$ `, 0` | $R_b = R_a \gg 3 + 0$ | $R_b = [0.125\|0.25\|0.375\|0.5]$ |
| `Arithmetic shift right 2 bit and add` $R_b$ `,` $R_a$ `,` $R_b$ | $R_b = R_a \gg 2 + R_b$ | $R_b = [0.375\|0.75\|1.125\|1.5]$ |

Only two single-cycle instructions are required to perform the multiplication of four subwords by a constant, in this example. This is equivalent to an effective rate of two multiplications per cycle. Without subword parallelism, the same operations would take at least four integer `multiply` instructions.

Furthermore, the `packed shift and add` instructions use a simple ALU with a small preshifter, whereas the integer `multiply` instructions need a more complex multiplier functional unit. In addition, each multiplication operation takes at least three cycles of latency compared to one cycle of latency for a preshift and add operation. Hence, for this example, the speedup for multiplying four subwords by a constant is six times faster ($4 \times 3/2$), comparing implementations with one (non-pipelined) subword multiplier versus one partitionable ALU with preshifter.

MAX-2 in PA-RISC and IA-64 are the only multimedia ISAs surveyed that have these efficient `packed shift left and add` instructions and `packed shift right and add` instructions. The preshift amounts allowed are by one, two, or three bits, and the arithmetic is performed with signed saturation, for 16-bit subwords.

### 23.1.4.3 Vector Multiplication

So far, this chapter has examined relatively simple `packed multiply` instructions. These instructions all take about the same latency as a single `multiply` instruction, which is typically 3–4 cycles compared to an `add` instruction normalized to one cycle latency. For better or worse, some multimedia ISAs have included very complex, multiple-cycle operations. For example, AltiVec has a `packed vector multiply and accumulate` instruction, using three 128-bit packed source operands and a 128-bit target register (see Fig. 23.24). First, all the pairs of bytes within a 32-bit subword in two of the source registers are multiplied in parallel and 16-bit products are generated. Then, four 16-bit products are added to each other to generate a "sum of products" for every 32 bits. A 32-bit subword from the third source register is added to this "sum of products." The resulting sum is placed in the corresponding 32-bit subword field of the target register. This process is repeated for each of the four

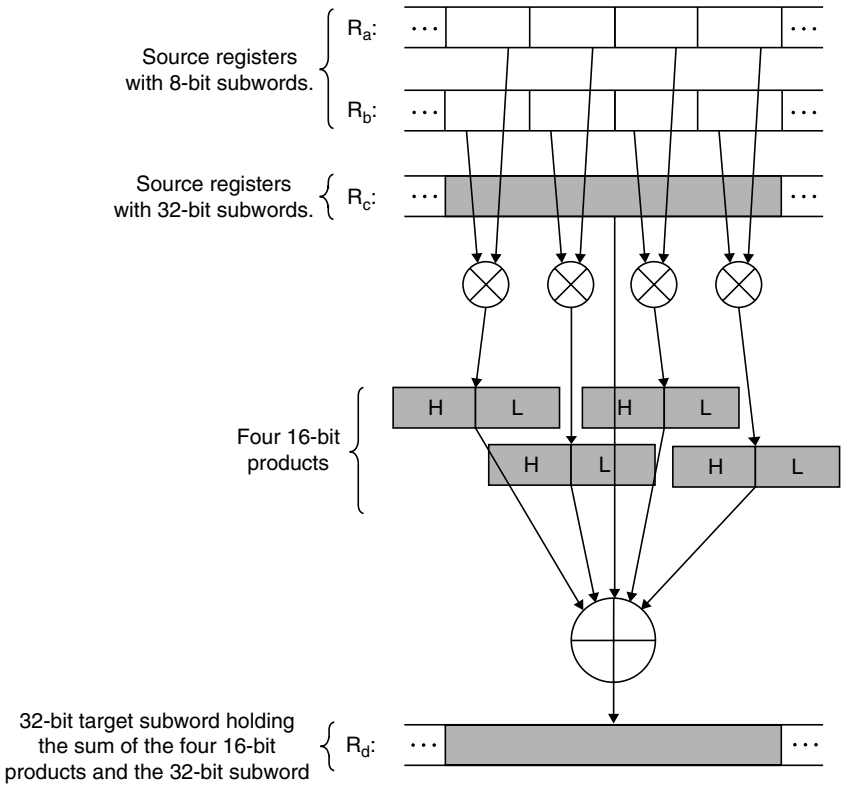



**FIGURE 23.24** AltiVec's `VSUMMBM` instruction: only one-fourth of the instruction is shown. Each box represents a byte. This process is carried out for each 32-bit word in the 128-bit source registers.

**TABLE 23.4** Packed Integer Multiplication Instructions

| Integer Operations | IA-64 | MAX-2 | MMX | SSE-2 | 3DNow! | AltiVec |
|---|---|---|---|---|---|---|
| $c_i = lower\_half(a_i * b_i)$ | √ | | √ | √ | √ | √ |
| $c_i = upper\_half(a_i * b_i)$ | √ | | √ | √ | √ | √ |
| $c_i = lower\_half[(a_i * b_i) \gg n]$ | √[a] | | | | | |
| `Packed multiply left` $[c_{2i}, c_{2i+1}] = a_{2i} * b_{2i}$ | √ | | | | | |
| `Packed multiply right` $[c_{2i}, c_{2i+1}] = a_{2i+1} * b_{2i+1}$ | √ | | | | | |
| `Packed multiply and accumulate` $[c_{2i}, c_{2i+1}] = a_{2i} * b_{2i} + a_{2i+1} * b_{2i+1}$ | | | √ | | | |
| $d_i = upper\_half(a_i * b_i) + c_i$ | | | | | | √ |
| $d_i = lower\_half(a_i * b_i) + c_i$ | | | | | | √ |
| `Packed shift left and add`[b] $c_i = (a_i \gg n) + b_i$, for $n = 1$, 2 or 3 bits. | √ | √ | | | | |
| `Packed shift right and add`[c] $c_i = (a_i \ll n) + b_i$, for $n = 1$, 2 or 3 bits. | √ | √ | | | | |
| `Packed vector multiply and accumulate (VSUMMBM)` $[d_{4i}, d_{4i+1}, d_{4i+2}, d_{4i+3}] = [c_{4i}, c_{4i+1}, c_{4i+2}, c_{4i+3}] + \sum_{j=1}^{4} a_{4i+j} * b_{4i+j}$ | | | | | | √ |
| `VMSUMxxx` instructions of AltiVec (general form) $[d_{2i}, d_{2i+1}] = a_{2i} * b_{2i} + a_{2i+1} * b_{2i+1} + [c_{2i}, c_{2i+1}]$ | | | | | | √ |

[a] Shift amounts are limited to 0,7,15, or 16 bits.
[b] For use in multiplication of a packed register by an integer constant.
[c] For use in multiplication of a packed register by a fractional constant.

32-bit subwords. This is a total of sixteen 8-bit integer multiplies, twelve 16-bit additions, and four 32-bit additions, using four 128-bit registers, in a single `VSUMMBM` instruction. This can perform a $4 \times 4$ matrix times a $4 \times 1$ vector multiplication, where each element is a byte, in a single instruction, but this single complex instruction takes many cycles of latency. While a multiplication of a $4 \times 4$ matrix with a $4 \times 1$ vector is a very frequent operation in graphics geometry processing, the precision required is usually that of 32-bit single-precision floating-point numbers, not 8-bit integers. Whether the complexity of such a compound `VSUMMBM` instruction is justified depends on the frequency of such $4 \times 4$ matrix-vector multiplications of bytes. Table 23.4 summarizes the packed integer multiplication instructions described.

### 23.1.5 Packed Shift and Rotate Operations

Most microprocessors have one or more shifters in addition to one or more ALUs (see Fig. 23.3). Just as the ALU is partitionable, so is the shifter, for subword-parallel operation. A `packed shift` instruction performs blocking shifts of the subwords packed in a register. Any bits shifted to the left are blocked from affecting the adjacent subword on the left; any bits shifted to the right are blocked from affecting the adjacent subword on the right.

For the `packed shift` instruction, the shift can be logical (zeros substituted for vacated bits) or arithmetic (zeros substituted for vacated bits on the right and sign-bit replicated for vacated bits on the left). The shift amount can be given by an immediate operand or by a register operand. When the shift amount is given by a register, each subword is usually shifted by the same amount, given by the least significant $\log_2 n$ bits of a second source register, for shifting the $n$ bits of a first source register (see Fig. 23.25). In a more complicated, but more versatile form, each subword in a packed register can be shifted by a different amount (see Fig. 23.26).

Similarly, the `packed rotate` instruction performs rotations on each subword in parallel. The amount to be rotated can be specified by an immediate in an instruction, by a single rotate amount in a

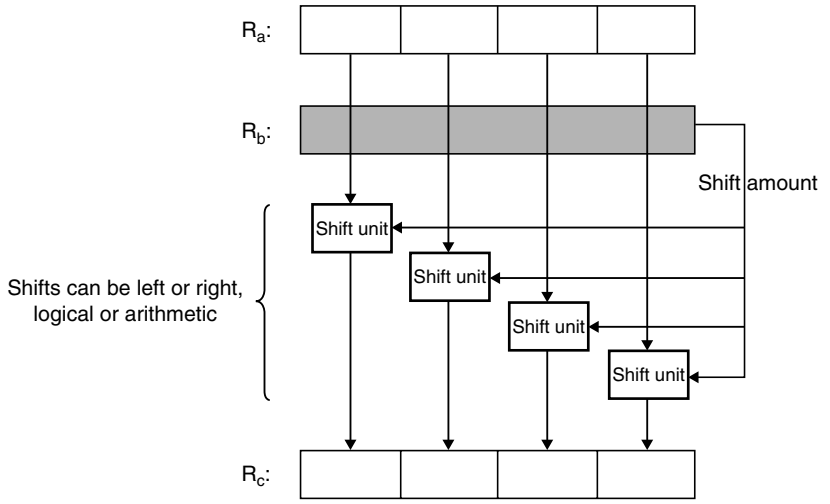



**FIGURE 23.25** `Packed shift` instruction. Shift amount is given in the second operand. Each subword is shifted by the same amount.

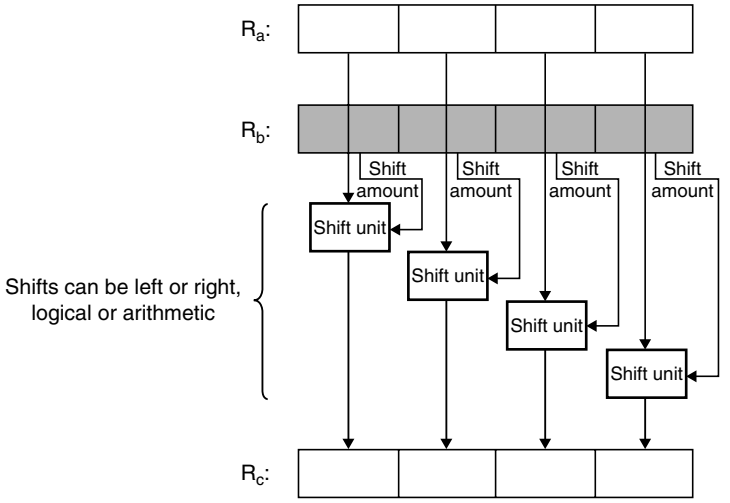



**FIGURE 23.26** `Packed shift` instruction. Shift amount is given in the second operand. Each subword can be shifted by a different amount.

register, or by different rotate amounts for each subword (see Fig. 23.27). Data-dependent rotations, where the single rotate amount is given in a register, have been proposed for symmetric cryptography algorithms like RC5.

`Packed shift` instructions may also be used to multiply or divide subwords by a constant that is a power of two. When used in this way, it may be necessary to apply saturation arithmetic with parallel left shifts used for multiplication. It may also be desirable to apply rounding with parallel arithmetic right shifts. Such saturation and rounding complicate the circuitry for the shifter functional unit, and is not implemented by any of the current multimedia ISAs. Hence, `packed shift` instructions should be used for multiplication or division only when no overflow can occur on left shifts, and sufficient precision can be preserved on right shifts. For multiplication by an integer or fractional constant, `packed shift and add` instructions, described in Section 23.1.4.2 are preferable. These can better control accuracy in the multiplication.

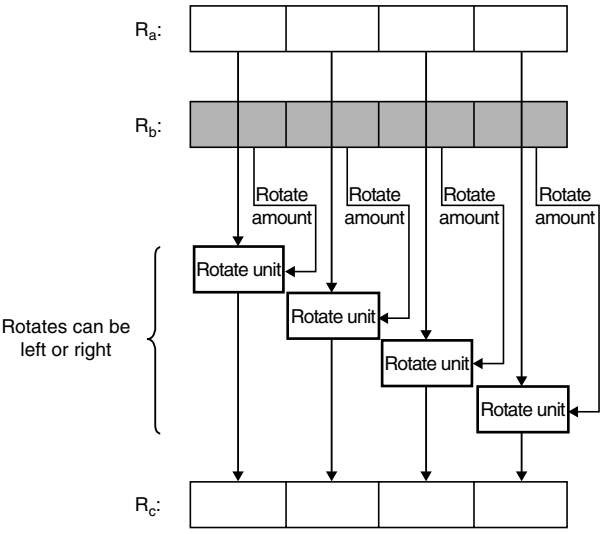



**FIGURE 23.27** `Packed rotate` instruction. Rotate amount is given in the second operand. Each subword can be rotated by a different amount.

**TABLE 23.5** Summary of `packed shift` and `packed rotate` Instructions

| Integer Operations | IA-64 | MAX-2 | MMX | SSE-2 | 3DNow! | AltiVec |
|---|---|---|---|---|---|---|
| $c_i = a_i \ll n$ | √ | √ | √ | | | |
| $c_i = a_i \ll b$ | √ | | √ | | | |
| $c_i = a_i \ll b_i$ | | | | | | √ |
| $c_i = a_i \gg n$ | √ | √ | √ | | | |
| $c_i = a_i \gg b$ | √ | | √ | | | |
| $c_i = a_i \gg b_i$ | | | | | | √ |
| $c_i = a_i \lll n$ | | | | | | |
| $c_i = a_i \lll b$ | | | | | | |
| $c_i = a_i \lll b_i$ | | | | | | √ |

Table 23.5 summarizes the multimedia instructions involving `packed shift` and `packed rotate` operations. In the table, $n$ is used to represent a shift or rotate amount that is specified in the immediate field of an instruction. For example, in the operation denoted as $c_i = a_i \ll n$, each subword of $c$ is shifted to the left by the amount given in the immediate field of the corresponding instruction. Similarly, in the operation $c_i = a_i \ll b$, each subword of $c$ is shifted to the left by the amount specified in the source register $b$. In $c_i = a_i \ll b_i$, each subword of $c$ is shifted to the left by the amount specified in the corresponding subword of the source register $b$. Shift left is represented by $\ll$, shift right by $\gg$, and rotate by $\lll$.

## 23.1.6 Subword Permutation Instructions

Initially, the rearrangement of subwords in registers manifested only as packing and unpacking operations. MAX-2 first introduced general-purpose subword permutation instructions for more versatile reordering of subwords packed into one or more registers [9].

### 23.1.6.1 Pack Instructions

`Pack` instructions convert from larger subwords to smaller subwords. If the value in the larger subword is greater than the maximum value that can be represented by the smaller subword, saturation arithmetic is performed, and the resulting subword is set to the maximum value of the smaller subword. Figure 23.28 shows how a register with smaller packed subwords can be created from two registers

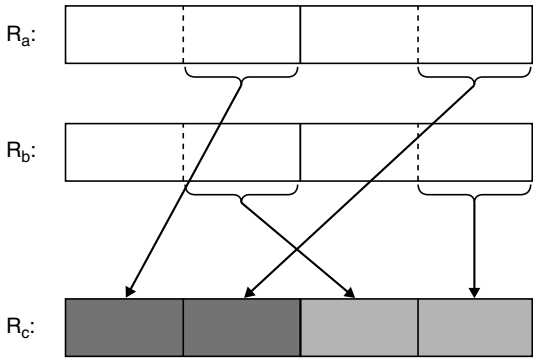



**FIGURE 23.28** `Pack` instruction converts larger subwords to smaller ones.

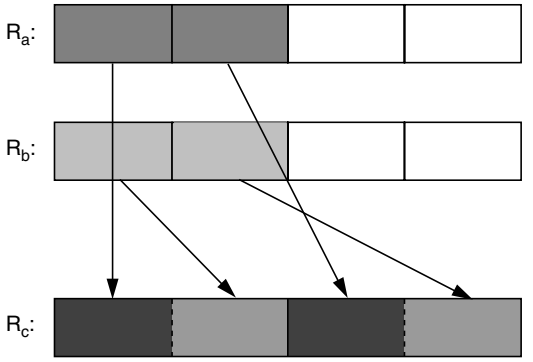



**FIGURE 23.29** `Unpack high` instruction.

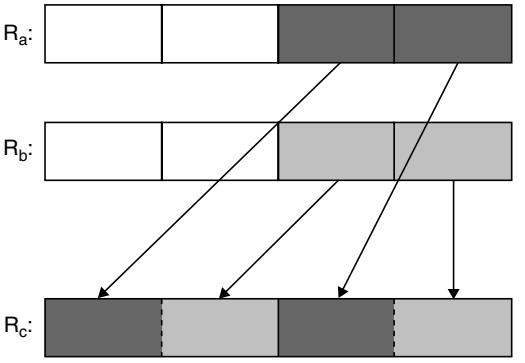



**FIGURE 23.30** `Unpack low` instruction.

with subwords that are twice as large. `Pack` instructions differ in the size of the supported subwords and in the saturation options used.

#### 23.1.6.2 Unpack Instructions

`Unpack` instructions are used to convert smaller packed data types to larger ones. The subwords in the two source operands are written sequentially to the target register in alternating order. Because, only one-half of each of the source registers can be used, the `unpack` instructions come with two variants `unpack high` or `unpack low` (Figs. 23.29 and 23.30). The `unpack high/low` instructions select and unpack the `high` or `low` order subwords of a source register, when used with register zero as the second source register.*

### 23.1.7 Subword Permutation Instructions

Ideally, it is desirable to be able to perform all possible permutations on packed data. This is only possible for small numbers of subwords. When the number of subwords increases, the number of control bits required to specify arbitrary permutations becomes too large to be encoded in an instruction. For the case of $n$ subwords, the number of control bits used to specify a particular permutation of these $n$ subwords is $n \log_2(n)$. Table 23.6 shows how many control bits are required to specify any arbitrary permutation for different numbers of subwords. When the number of subwords is 16 or greater, the number of control bits exceeds the number of the bits available in the instruction, which is typically 32 bits. Therefore, it becomes necessary to use a second register† to contain the control bits used to specify the permutation. By using this second register, it is possible to get any arbitrary permutation of up to 16 subwords in one instruction.

Because AltiVec instructions have three 128-bit source registers, a subword permutation can use two registers to hold data, and the third register to hold the control bits. This allows

*Register zero gives a constant value of "zero" when used as a source register.

†This second register needs to be at least 64-bits wide to fully accommodate the 64 control bits needed for 16 subwords.

**TABLE 23.6** Number of Control Bits Required to Specify an Arbitrary Permutation

| Number of Subwords in a Packed iData Type | Number of Control Bits Required to Specify an Arbitrary Permutation for a Given Number of Subwords |
|---|---|
| 2 | 2 |
| 4 | 8 |
| 8 | 24 |
| 16 | 64 |
| 32 | 160 |
| 64 | 384 |
| 128 | 896 |

any arbitrary selection and re-ordering of 16 of the 32 bytes in the two source registers in a `vperm` instruction.

#### 23.1.7.1 `Mux`, `Permute`, and `Mix` Instructions

Only a small subset of all the possible permutations is achievable with one subword permutation instruction, so it is desirable to select permutations that can be used as primitives to realize other permutations. A subword permutation instruction can have one or two source registers as operands. In the latter case, only half of the subwords in the two source operands may actually appear in the target register. Examples of these two cases are the `mux` and `mix` instructions respectively, in both IA-64 and MAX-2.

`Mux` in IA-64 operates on one source register. It allows all possible permutations of four packed 16-bit subwords, with and without repetitions (see Fig. 23.31). An 8-bit immediate field in the instruction is used to select one of the 256 possible permutations. This is the same operation performed by the permute instruction in the earlier MAX-2.

In IA-64, the `mux` instruction can also permute eight packed 8-bit subwords. For the 8-bit subwords, `mux` has five variants, and only the following permutations are implemented in hardware (see Fig. 23.32):

- `Mux. rev` (reverse): Reverses the order of bytes.
- `Mux. mix` (mix): Performs the `Mix` operation (see below) on the bytes in the upper and lower 32-bit halves of the 64-bit source register.
- `Mux. shuf` (shuffle): Performs a perfect shuffle on the bytes in the upper and lower halves of the register.
- `Mux.alt` (alternate): Selects first the even* indexed bytes, placing them in the upper half of the result register, then selects the odd indexed bytes, placing them in the right half of the result register.

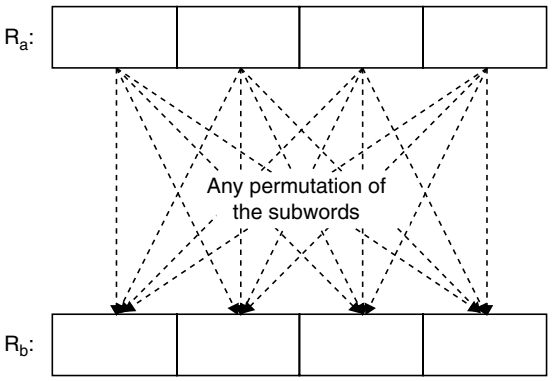



**FIGURE 23.31** `Mux` instruction in IA-64 or `Permute` instruction in Max-2.

*The bytes indexed from 0 to 7.0 corresponds to the most significant byte, which is on the left end of the registers.

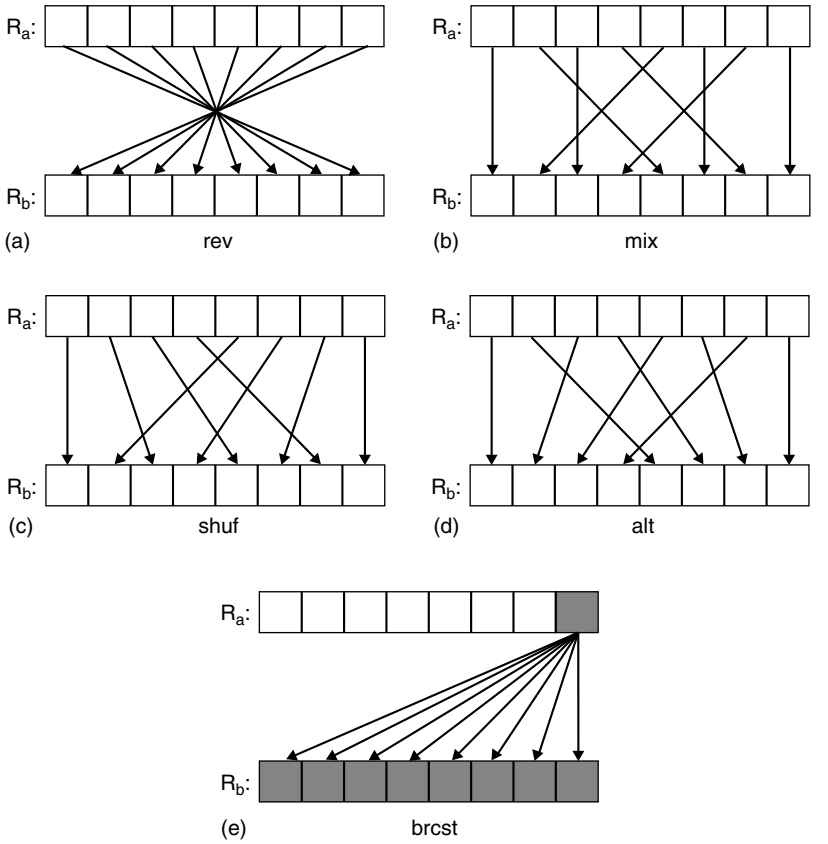



**FIGURE 23.32** `Mux` instruction in IA-64 has five permutation options for 8-bit subwords. (From Intel, IA-Architecture Software Developer's Manual, Vol. 3, Instruction Set Reference, Rev. 1.1, July 2000. With permission.)

- `Mux. brcst` (broadcast): Replicates the least significant byte into all the byte locations of the result register.

`Mix` is a very useful permutation operation on two source registers. A `mix left` instruction picks even subwords alternately from the two source registers and places them into the target register (see Fig. 23.33). A `mix right` instruction picks odd subwords alternately from the two source registers and places them into the target register (see Fig. 23.34).

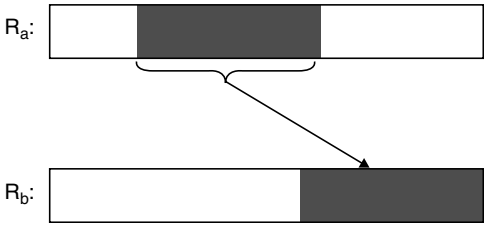



**FIGURE 23.33** `Mix Left` instruction.

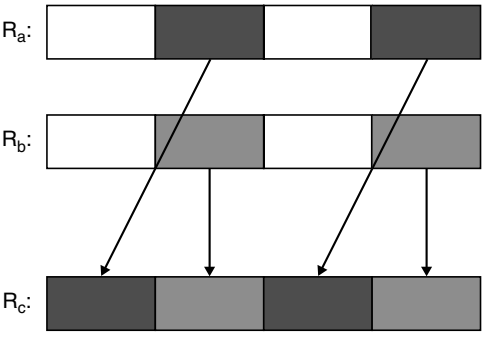



**FIGURE 23.34** `Mix Right` instruction.

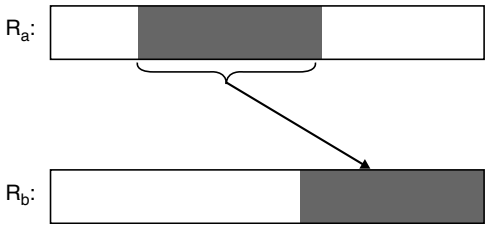



**FIGURE 23.35** `Extract` bit-field instruction.

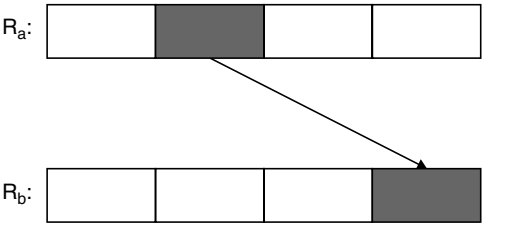



**FIGURE 23.36** `Extract subword` instruction.

The versatility of `Mix` is demonstrated [9,14], for example, in performing a matrix transpose. `Mix` can also be used to perform an unpacking function similar to that done by `Unpack High` and `Unpack Low`. The usefulness of `Mix` and `Mux` (`or Permute`) has also been validated in [21] as general-purpose subword permutation primitives for processing of two-dimensional data in microSIMD architectures.

#### 23.1.7.2 Extract, Deposit, and Shift Pair Instructions

A more sophisticated shifter can also perform `extract` and `deposit` bit-field operations, as in PA-RISC [17,10]. An `extract` instruction picks an arbitrary contiguous bit-field from the source operand and places it right aligned into the result register (Fig. 23.35). `Extract` instructions may be limited to work on subwords instead of bit-fields (Fig. 23.36). `Extract` instructions clear the upper bits of the target register.

A `deposit` instruction picks a right-aligned contiguous bit-field from the source register and patches it into an arbitrary location in the target register (Fig. 23.37). The unpatched bits of the target register remain unchanged. Alternatively, they are cleared to zeros in a `zero and deposit` instruction [17]. `Deposit` instructions may be limited to work on subwords instead of arbitrarily long bit-fields and arbitrary patch locations (Fig. 23.38).

A very useful instruction for rearranging subwords from two registers is the `shift pair` instruction in IA-64 (see Fig. 23.39). This instruction, which was first introduced in the PA-RISC ISA [10,17], is essentially a `shift` instruction for bit-strings that span more than one register. `Shift pair` concatenates the two source registers to form a 128-bit intermediate value, which is shifted to the right by $n$ bits. The least significant 64 bits of the shifted value is written to the result register. If the same register is specified for both operands, the result is a `rotate` operation. Rotates can be realized this way, so IA-64 does not have a separate `rotate` instruction. This `shift pair` instruction is more general than a `rotate`, allowing flexible combination of two bit-fields from separate registers. Table 23.7 summarizes the subword permutation instructions on packed data types.

### 23.1.8 Floating-Point MicroSIMD Instructions

High-fidelity audio and graphics geometry processing require the higher precision and range of floating-point numbers. Usually, single-precision (32-bit) floating-point (FP) numbers are sufficient, but 16-bit

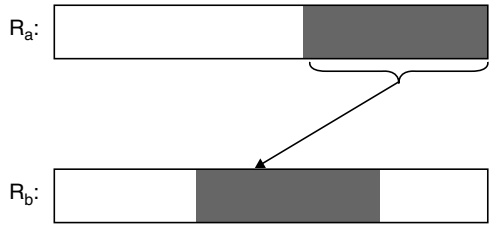



**FIGURE 23.37** `Deposit bit-field` instruction.

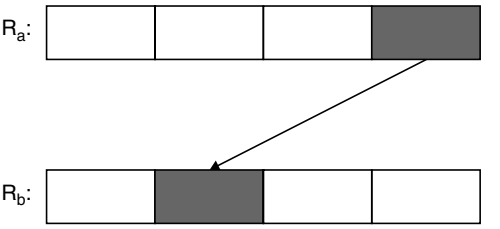



**FIGURE 23.38** `Deposit subword` instruction.

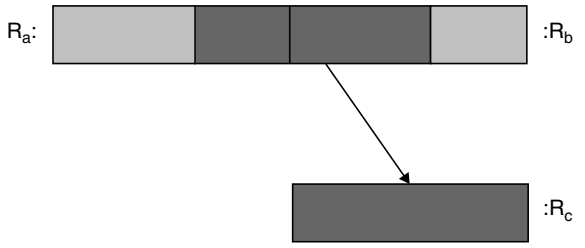



**FIGURE 23.39** `Shift pair` instruction in IA-64.

integers or fixed-point numbers are not. Double-precision (64-bit) floating-point numbers are not really needed for such multimedia computations.

Because floating-point registers are at least 64-bits wide in microprocessors to support double-precision (DP) FP numbers, it is possible to pack two single-precision (SP) FP numbers in a 64-bit register, to support subword parallelism, or packed parallelism, or microSIMD parallelism on the FP functional units and registers. The precision levels supported by different ISAs are shown in Table 23.8. SP and DP numbers are 32 and 64 bits long, respectively, as defined by the IEEE-754 FP number standard. Only SSE-2 supports packed DP FP numbers. MAX-2 and MMX do not support packed FP instructions.

### 23.1.8.1 Packed Floating-Point Arithmetic Instructions

#### *23.1.8.1.1 Packed FP Add*

Figure 23.40 shows a `packed FP add`, where four pairs of single-precision FP numbers in two 128-bit registers are added using floating-point addition. `Packed FP subtract` instructions are similar. While the packed FP instruction looks very similar to the packed integer equivalents (see Fig. 23.5), implementation of `packed FP add` is not as simple as blocking carries at the subword boundary as in packed integer addition (see Fig. 23.7). It is much more difficult to partition a FP functional unit for subword parallelism because of the nature of FP arithmetic acting on FP numbers represented in sign,

**TABLE 23.7** Subword Permutation Instructions

| Integer Operations | IA-64 | MAX-2 | MMX | SSE-2 | 3DNow! | AltiVec |
|---|---|---|---|---|---|---|
| `Pack` | √ | | √ | | | √ |
| `Unpack low` | √ | | √ | √ | | √ |
| `Unpack high` | √ | | | √ | | √ |
| `Permute` *n* subwords | √ ($n = 4$) | √ ($n = 4$) | | √ ($n = 4$) | √ ($n = 4$) | √ ($n = 16,32$)[a] |
| `Mux.rev` | √ | | | | | |
| `Mux.mix` | √ | | | | | |
| `Mux.shuffle` | √ | | | | | |
| `Mux.alt` | √ | | | | | |
| `Mux.brcst` | √ | | | | | |
| `Mix left` | √ | √ | | | | √ |
| `Mix right` | √ | √ | | | | √ |
| `Extract bit-field` | √ | √ | | | | |
| `Extract subword` | | | | | √ | |
| `Deposit bit-field` | √ | √ | | | | |
| `Deposit subword` | | | | | √ | |
| `Shift pair` $R_c$, $R_a$, $R_b$ | √ | √ | | | | |

[a] This is the `vperm` instruction, and it has some limitations for $n = 32$. See text for more details on this instruction. Subword size for this instruction is 8 bits regardless of whether $n$ is 16 or 32.

**TABLE 23.8** Supported Precision Levels for the Packed FP Operations

| Architecture | IA-64 | SSE-2 | 3DNow! | AltiVec |
|---|---|---|---|---|
| FP register size | 82 bits | 128 bits | 128 bits | 128 bits |
| Allowed packed FP data types | 2 SP | 4 SP or 2 DP | 4 SP | 4 SP |

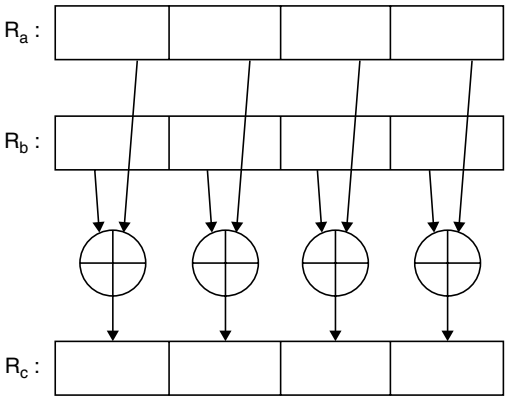



**FIGURE 23.40** PFPADD $R_c$, $R_a$, $R_b$: Packed FP add instruction.

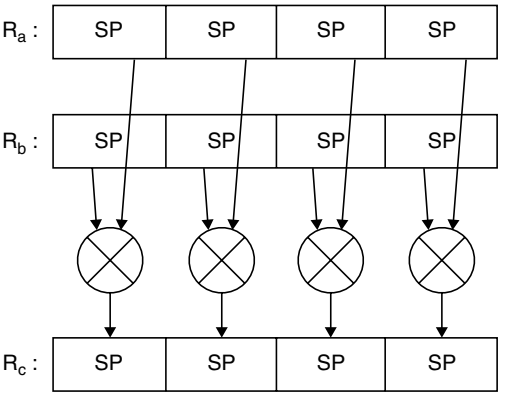



**FIGURE 23.41** PFPMUL $R_c$, $R_a$, $R_b$: Packed FP multiply instruction.

mantissa, and exponent format. Another difference is that in floating-point number representation, considerations like modular arithmetic or saturation arithmetic are not applicable.

#### *23.1.8.1.2 Packed FP Multiplication*

Multiplication of two packed FP registers involves multiplication of corresponding FP subwords from the source registers, where the products are written to the corresponding subword in the target register (see Fig. 23.41). In multiplication of two single-precision numbers, the product is also single-precision, and hence the same width. Therefore, packed FP multiply does not have the problem associated with packed integer multiply instructions, where the product is twice the width of the operands.

#### *23.1.8.1.3 Packed FP Multiply and Add*

The most important FP operation in audio, graphics, and digital signal processing is the FP multiply and accumulate operation. Recognizing this, many ISAs have implemented this as the basic FP operation, needing three source registers. For example, IA-64 implements packed FP multiply and add (FPMA), packed FP multiply and subtract (FPMS), and packed FP negative multiply and add (FPNMA). It then realizes packed FP add, packed FP subtract, and packed FP multiply operations by using FPMA and FPMS instructions. IA-64 architecture specifies 128 FP registers, which are numbered FR0 through FR127. Of these registers, FR0 and FR1 are special. FR0 always returns the value +0.0 when sourced as an operand, and FR1 always reads +1.0. When FR0 or FR1 are used as source operands, the FPMA and FPMS instructions can be used to realize packed FP add or packed FP subtract operations and packed FP multiply operations (see Table 23.9).

**TABLE 23.9** IA-64 uses FPMA and FPMS Instructions for packed FP add, packed FP subtract, and packed FP multiply

| IA-64 Instruction | Operation | Equivalent Instruction |
|---|---|---|
| FPMA $R_d$, FR1, $R_b$, $R_c$ (packed FP multiply and add) | $R_d = FR1 * R_b + R_c$<br>$= 1.0 * R_b + R_c$<br>$= R_b + R_c$ | Packed FP add |
| FPMS $R_d$, FR1, $R_b$, $R_c$ (packed FP multiply and subtract) | $R_d = FR1 * R_b - R_c$<br>$= 1.0 * R_b - R_c$<br>$= R_b - R_c$ | Packed FP subtract |
| FPMA $R_d$, $R_a$, $R_b$, FR0 (packed FP multiply and add) | $R_d = R_a * R_b + FR0$<br>$= R_a * R_b + 0.0$<br>$= R_a * R_b$ | Packed FP multiply |

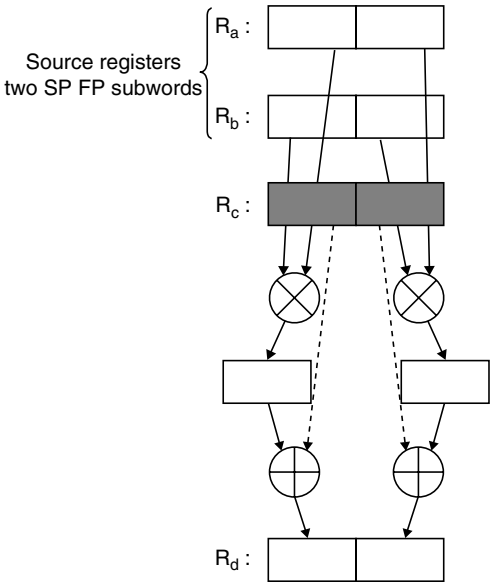



**FIGURE 23.42** `Packed FP multiply and add` instruction in IA-64.

The `format` of the `FPMA` (Fig. 23.42) instruction is `FPMA` $R_d$, $R_a$, $R_b$, $R_c$ and the operation it performs is $R_d = R_a * R_b + R_c$. If FR1 is used as the first or the second source operand, a `packed FP add` operation is realized. Similarly, a `FPMS` instruction can be used to realize a `packed FP subtract` operation. Using FR0 as the third source operand in `FPMA` or `FPMS` results in a `packed FP multiply` operation.

Table 23.10 is a summary of the packed FP instructions supported by multimedia ISAs. Several packed FP instructions operate like their packed integer equivalents, except that they operate on packed FP subwords rather than packed integer (or fixed-point) subwords. These include `packed FP add`, `packed FP subtract`, `packed FP multiply`, `packed FP negate`, `packed FP absolute value`, `packed FP compare`, `packed FP maximum`, and `packed FP minimum`. IA-64 also has the `packed FP maximum absolute value` and the `packed FP minimum absolute value`. These put the larger or smaller of the absolute values of the pairs of FP subwords into the result subwords in the target register, respectively.

#### *23.1.8.1.4 Packed FP Compare*

The `packed FP compare` instruction compares pairs of FP subwords according to the relation specified by the instruction. If the condition is true for a subword pair, the corresponding field in the target register is written with a 1-mask. If the condition is false, the corresponding field in the target register is written with a 0-mask. The only difference is that two additional relations, ordered and unordered, are possible for floating-point numbers in addition to the 10 relations already specified for comparing integers (see Section 23.1.3.7). Some ISAs have `packed FP compare` instructions that allow all the 12 possible relations,* whereas others support a more limited subset of relations.

#### *23.1.8.1.5 Packed FP Compare Bounds*

An interesting comparison instruction is the `packed FP compare bounds (VCMPBFP)` instruction of AltiVec. This instruction compares corresponding FP subwords from the two source registers, and depending on the relation between the compared numbers, it generates a two-bit result, which is written to the target register. The resulting two-bit field indicates the relation between the two compared FP numbers. For instance, in `VCMPBFP` $R_c$, $R_a$, $R_b$, the FP number pairs $(a_i, b_i)$ are compared, and a two-bit field is written into $c_i$ such that:

- Bit 0 of the two-bit field is cleared if $a_i <= b_i$, and is set otherwise.
- Bit 1 of the two-bit field is cleared if $a_i >= (-b_i)$, and is set otherwise.
- Both bits are set if any of the compared FP numbers is a NaN.

The two-bit result field is written to the high-order two bits of $c_i$; the remaining bits of $c_i$ are cleared to 0. Table 23.11 gives examples of input pairs that result in each of the four different possible outputs for this instruction.

*Two floating-point numbers a and b can be compared for one of the following 12 possible relations: equal, less-than, less-than-or-equal, greater-than, greater-than-or-equal, unordered, not-equal, not-less-than, not-less-than-or-equal, not-greater-than, not-greater-than-or-equal, ordered. Typical notation for these relations are as follows respectively: $=$, $<$, $<=$, $>$, $>=$, ?, $!=$, $!<$, $!<=$, $!>$, $!>=$, !?.

**TABLE 23.10** Summary of FP microSIMD Instructions

| Packed FP Instructions | IA-64 | SSE-2 | 3DNow! | AltiVec |
|---|---|---|---|---|
| $c_i = a_i + b_i$ | √[a] | √ | √ | √ |
| $c_i = a_i - b_i$ | √[b] | √ | √ | √ |
| $c_i = a_i * b_i$ | √[c] | √ | √ | |
| $d_i = -a_i * b_i$ | √ | | | |
| $d_i = a_i * b_i + c_i$ (FPMA) | √ | | | √ |
| $d_i = a_i * b_i - c_i$ (FPMS) | √ | | | |
| $d_i = -a_i * b_i + c_i$ (FPNMA) | √ | | | √ |
| $c_i = -a_i$ | √ | | | |
| $c_i = \|a_i\|$ | √ | | | |
| $c_i = -\|a_i\|$ | √ | | | |
| $c_i = \text{compare}(a_i, b_i)$ | √ | √ | √ | √ |
| $c_i = \max(a_i, b_i)$ | √ | √ | √ | √ |
| $c_i = \min(a_i, b_i)$ | √ | √ | √ | √ |
| $c_i = \max(\|a_i\|, \|b_i\|)$ | √ | | | |
| $c_i = \min(\|a_i\|, \|b_i\|)$ | √ | | | |
| $c_i = \text{VCMPBFB}(a_i, b_i)$[e] | | | | √ |
| $c_i = \sqrt{a_i}$ | | √ | | |
| $c_i = 1/\sqrt{a_i}$ | √ | √ | | √ |
| $c_i = 1/a_i$ | √ | √ | | √ |
| $c_i = \log_2 a_i$ | | | | √ |
| $c_i = 2^{a_i}$ | | | | √ |
| `Permute` *n* FP subwords | √ | ($n = 2,4$) | | |
| `Swap` FP subwords (optionally negate left or right subword) | √ | | | |
| `Mix_Left, Mix_Right, Mix_Left_Right` | √ | | | |
| `Unpack_high, Unpack_low` | | √ | | |
| `Pack` | √ | √ | | |

[a] This operation is realized by using the `FPMA` instruction.
[b] This operation is realized by using the `FPMS` instruction.
[c] This operation is realized by using the `FPMA` or `FPMS` instruction.
[d] This operation is realized by using the `FPNMA` instruction.
[e] This is the `packed FP compare bounds` instruction, which is explained in the text.

The SSE-2 architecture also includes a `packed FP square root` instruction. This instruction operates on packed single-precision or double-precision numbers and computes the square roots to SP or DP accuracy. IA-64 has the `packed FP reciprocal square root` instruction and the `packed FP reciprocal` instruction. Both are very useful for graphics computations.

### 23.1.8.2 FP Subword Permutation Instructions

#### *23.1.8.2.1 FP Permutation Instructions*

SSE-2 has an `FP permute` (see Fig. 23.43) instruction that allows any arbitrary permutation of the four 32-bit SP subwords in one of its 128-bit multimedia registers. This operates just like the `permute` instruction in MAX-2 and the `mux` instruction (2-byte subword version) in IA-64 (see Fig. 23.31).

**TABLE 23.11** Result of the `VCMPBFP` Instruction for Different Input Pairs

| Input | | Output | |
|---|---|---|---|
| $a_i$ | $b_i$ | Bit 0 | Bit 1 |
| 3.0 | 5.0 | 0 | 0 |
| −8.0 | 5.0 | 0 | 1 |
| 8.0 | 5.0 | 1 | 0 |
| 3.0 | −5.0 | 1 | 1 |

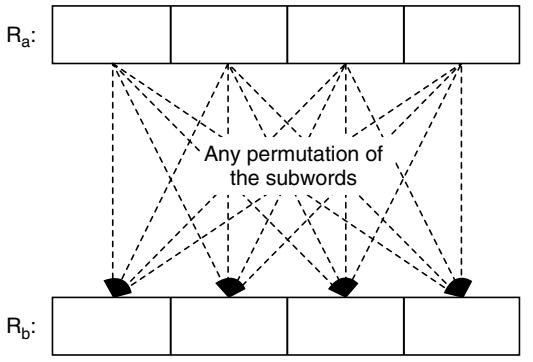



**FIGURE 23.43** `FP permute` $R_b$,$R_a$: `FP permute` instruction.

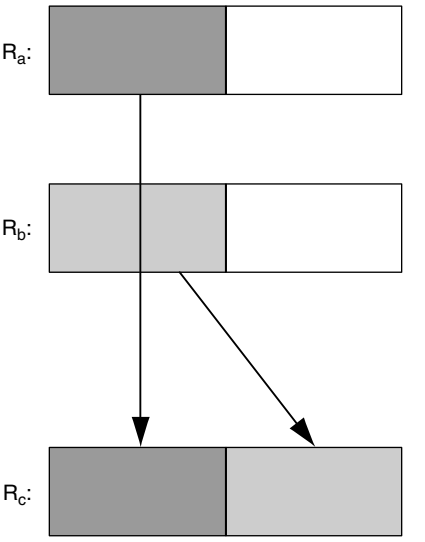



**FIGURE 23.44** `FP mix left` $R_c$, $R_b$, $R_a$: `FP mix left` instruction in IA-64.

IA-64 only has two single-precision subwords in its packed format, so all possible permutations of two subwords can be achieved with a much simpler operation, `FP swap`. This instruction just exchanges the two subwords. IA-64 also allows two variants of this: after swapping the subwords, the sign of either the left or the right FP value is negated.

`FP mix` is a useful operation that performs a permutation on two packed FP registers. A `FP mix` instruction picks alternating subwords from two source registers and places them into the target register. `FP mix` in IA-64 appears in three variants. The first one (Fig. 23.44) is called the `FP mix left` and uses the odd indexed FP subwords of the source registers in the permutation, starting from the leftmost subword. The second variant, `FP mix right` (Fig. 23.45) uses the even indexed FP subwords of the source registers, ending with the rightmost subword. The third variant, `FP mix left right` (Fig. 23.46) uses the odd indexed FP subword of the first source register, and the even indexed subword of the second source register. These three FP mix instructions, together with the `Shift Pair` instruction described earlier, allow any one of the four combinations of the SP subwords packed into two IA-64 registers to be achieved with only one instruction.

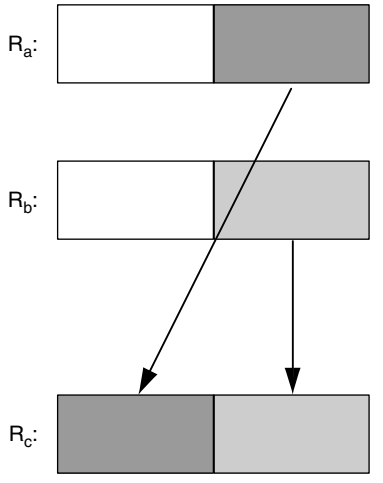



**FIGURE 23.45** `FP mix right` $R_c$, $R_b$, $R_a$: `FP mix right` instruction in IA-64.

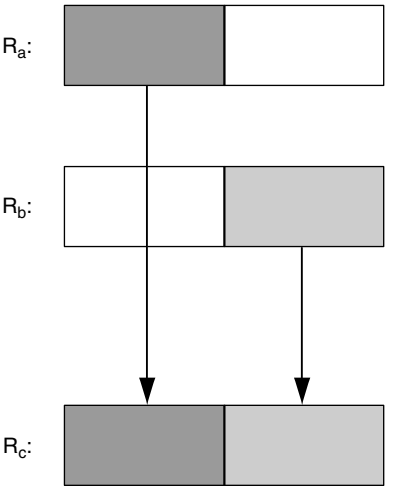



**FIGURE 23.46** `FP mix left right` $R_c$, $R_b$, $R_a$: `FP mix left right` instruction in IA-64.

#### *23.1.8.2.2 FP Unpack*

Packing and unpacking subwords has a different interpretation for FP numbers than for integers. In general, there is sufficient precision in single-precision numbers, and there is no need to `unpack` it to a double-precision number; however, the `FP unpack` can be regarded as a useful subword permutation instruction like `FP mix`. It performs a `shuffle` by interleaving the subwords from two registers. The `FP unpack` instructions operate just like the equivalent integer `unpack` instructions (see Figs. 23.29 and 23.30). They come in two "flavors": `FP unpack high` and `FP unpack low`. Note that the SSE-2 employs `FP unpack`, after `unpack` in MMX, and IA-64 employs `FP mix`, after `mix`, in MAX-2.

#### *23.1.8.2.3 FP Pack*

In the integer domain, `pack` instructions are used to create smaller packed data types from larger data types. The `FP pack` instruction in IA-64 creates two packed SP numbers from two 82-bit source registers. All IA-64 FP registers are 82-bit extended precision FP format with two extra guard bits for computational accuracy. First, the two 82-bit numbers are converted to standard 32-bit SP representation. These two SP numbers are then concatenated and the result is stored in the significand field (which is 64 bits) of the 82-bit target FP register. The exponent field of the target register is set to the biased exponent for $2.0^{63}$, which indicates a packed FP format, and the sign bit is set to zero, indicating a positive number.

## 23.1.9 Conclusions

Section 23.1 described multimedia instructions for programmable processors by broad classes according to the functional units used, first in the integer domain then in the floating-point domain. For integer subwords, `packed add` and `packed subtract` instructions, and different variants of these, use the ALU. `Packed multiply` instructions use the multiplier functional unit, although very efficient multiplication by constants can be implemented with `packed shift and add` instructions, which only need an ALU with a preshifter. `Packed shift` and `packed rotate` instructions use the shifter. Packed subword permutation instructions can either be implemented on a modified shifter or in a new permutation unit. For packed floating-point instructions, less leverage of hardware seems possible. The basic functional units are a floating-point adder, multiplier, and FP subword permutation unit. IA-64 combines the FP adder and multiplier into an FP multiply-add unit. For each of these instruction classes, interesting multimedia instructions introduced in current microprocessors were described, for example, in the IA-64, MMX, and SSE-2 from Intel; MAX-2 from Hewlett-Packard; 3DNow! from AMD; and AltiVec from Motorola.

The key feature in these multimedia instructions is the concept of subword parallelism, also called packed parallelism or microSIMD parallelism. This is implemented for packed integers or fixed-point numbers in the integer datapaths, and for packed floating-point numbers in the floating-point datapaths. Visual multimedia data like images, video, graphics rendering and animation involve pixel processing, which can fully exploit subword parallelism on the integer datapath. Higher-fidelity audio processing and graphics geometry processing require single-precision floating-point computations, which exploit subword parallelism on the floating-point datapath. Typical DSP operations such as `multiply and accumulate` have also been added to the multimedia repertoire of general-purpose microprocessors. These multimedia instructions have embedded DSP and visual processing capabilities into general-purpose microprocessors, providing native signal processing (sometimes referred to as NSP) for multimedia data. In fact, most DSPs and media processors have also adopted subword parallelism in their architectures, as well as other features often first introduced in microprocessors for multimedia signal processing.

More unusual computer arithmetic issues arising from subword-parallel multimedia instructions in microprocessors are saturation arithmetic, integer rounding alternatives, integer multiplication problems and solutions, and subword permutation instructions.

Some of the multimedia ISAs introduced in microprocessors adhere to the "less is more" minimalist architecture approach, defining as few instructions as necessary for high-performance, with each instruction executable in a single pipeline cycle. Others embody the "more is better" approach, where complex sequences of operations are represented by a single multimedia instruction, with such an instruction taking many cycles for execution. An example is the `packed vector multiply and accumulate` instruction in AltiVec (Fig. 23.24). These two trends represent different stylistic preferences, akin to reduced instruction set computer (RISC) and complex instruction set computer (CISC) architectural preferences. In fact, sometimes, RISC-like multimedia instructions have been added to CISC processor ISAs, and CISC-like multimedia instructions to RISC processor ISAs. The remarkable fact is that subword-parallel multimedia instructions have achieved such rapid and pervasive adoption in both RISC and CISC microprocessors, DSPs and media processors, attesting to their undisputed cost-effectiveness in accelerating multimedia processing in software.

To simplify software compatibility and interoperability of multimedia software across different processors, it is highly desirable to refine the best ideas from the different multimedia ISAs into a coherent set of subword-parallel instructions. If this is a small yet powerful set, it is more likely to be implemented in all future microprocessors and media processors, allowing algorithm and compiler optimizations to exploit microSIMD parallelism with confidence that benefits would be realized across almost all processors. While slight differences in multimedia instructions across processors may not affect the potential performance provided by each ISA, they make it difficult to design an optimal algorithm and a set of compiler optimizations that achieve the best multimedia performance for every processor. The challenge for the next phase of multimedia ISA design is to understand which ISA features are truly effective for multimedia signal processing, and encapsulate these insights into the design of third-generation multimedia ISA for both microprocessors and media processors.

## Acknowledgments

The author thanks her student, A. Murat Fiskiran, for surveying SSE-2, 3DNow! and AltiVec, and for his invaluable help in preparing the figures and tables.

## References

1. Ruby Lee and Michael Smith, "Media processing: a new design target," *IEEE Micro,* Vol. 16, No. 4, pp. 6–9, Aug. 1996.
2. Michael Flynn, "Very high-speed computing systems," *Proceedings of the IEEE,* No. 54, Dec. 1966.
3. Ruby Lee, "Efficiency of MicroSIMD architectures and index-mapped data for media processors," *Proceedings of IS&T/SPIE Symposium on Electric Imaging: Media Processors 99,* pp. 34–46, Jan. 1999.
4. Intel, "IA-64 architecture software developer's manual, volume 3: instruction set reference," Revision 1.1, July 2000, Order Code 245319-002.
5. Ruby Lee, Murat Fiskiran, and Abdulla Bubshait, "Multimedia instructions in IA-64," Invited paper. *Proceedings of the 2001 IEEE International Conference on Multimedia and Exposition,* Aug. 22–24, 2001.
6. Alex Peleg and Uri Weiser, "MMX technology extension to the intel architecture," *IEEE Micro,* Vol. 16, No. 4, pp. 10–20, Aug. 1996.
7. Intel, "Intel architecture software developer's manual, volume 2: instruction set reference," 1999, Order Code 243191.
8. Intel, "IA-32 intel architecture software developer's manual with preliminary willamette architecture information, volume 2: instruction set reference," 2000.
9. Ruby Lee, "Subword parallelism with MAX-2," *IEEE Micro,* Vol. 16, No. 4, pp. 51–59, Aug. 1996.
10. G. Kane, *PA-RISC 2.0 Architecture*, 1996, Prentice-Hall, Englewood Cliffs, NJ.
11. AMD, "3DNow! technology manual," March 2000, Order Code 21928G/0.

12. AMD, "AMD extensions to the 3DNow! and MMX Instruction Sets Manual," March 2000, Order Code 22466D/0.
13. Motorola, "AltiVec technology programming environments manual," Revision 0.1, November 1998, Order Code ALTIVECPEM/D.
14. Ruby Lee, "Multimedia extensions for general-purpose processors," Invited paper. *Proceedings of the IEEE Signal Processing Systems: Design and Implementation,* pp. 9–23. Nov. 1997.
15. Ruby Lee, "Accelerating multimedia with enhanced microprocessors," *IEEE Micro,* Vol. 15, No. 2, pp. 22–32, April 1995.
16. Ruby Lee, John Beck, Joel Lamb, and Ken Severson, "Real-time software MPEG video decoder on multimedia-enhanced PA7100LC processors," *Hewlett-Packard Journal,* Vol. 46, No. 2, pp. 60–68, April 1995.
17. Ruby Lee, "Precision architecture," *IEEE Computer,* Vol. 22, No. 1, pp. 78–91, Jan. 1989.
18. Vasudev Bhaskaran, Konstantine Konstantinides, Ruby Lee and John Beck, "Algorithmic and architectural enhancements for real-time MPEG-1 decoding on a general purpose RISC workstation," *IEEE Transactions on Circuits and Systems for Video Technology,* Vol. 5, No. 5, pp. 380–386, Oct. 1995.
19. Mark Tremblay, J.M. O'Connor, V. Narayanan, and H. Liang, "VIS speeds new media processing," *IEEE Micro,* Vol. 16, No. 4, pp. 10–20, Aug. 1996.
20. Zhen Luo and Ruby Lee, "Cost-effective multiplication with enhanced adders for multimedia applications," *Proceedings of ISCAS 2000, IEEE International Symposium on Circuits and Systems,* Vol. I, pp. 651–654, May 2000.
21. Ruby Lee, "Subword permutation instructions for two-dimensional multimedia processing in Micro-SIMD architectures," *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors,* pp. 3–14, July 2000.

## 23.2 DSP Platform Architecture for SoC Products

*Gerald G. Pechanek*

### 23.2.1 Introduction

The development of wireless, networking, communications, video, and consumer products has shifted toward low-power high-functionality systems-on-chip (SoC) semiconductors [1]. Driving this development is the availability of deep sub-micron technology allowing more complete system designs to be embedded in silicon. Some of these improvements include increasing on-chip memory capacity, the use of more fully programmable solutions using DSPs, and the inclusion of specialized interfaces and functions.

To make these high-value SoC products widely available at low cost requires the use of standard design practices that allow them to be fabricated at multiple semiconductor suppliers. This means that custom designed SoCs, optimized to a particular manufacturing process, cannot be used. Consequently, as the complexity and functionality of SoC products continues to increase with stringent power requirements, the standard approach of increasing clock speed on an existing design to meet higher performance requirements is infeasible.

The need to support multiple standards, and to quickly adapt to changing standards, has become a product requirement [2]. To satisfy this need, programmable DSPs and control processors are being increasingly used as the central SoC design component. These processors form the basis of the SoC product platform and permeate the overall system design including the on-chip memory, DMA, internal busses, etc. Consequently, choosing a flexible and efficient processor, which can be manufactured by multiple semiconductor suppliers, is arguably the most important intellectual property (IP) decision that needs to be made in the creation of an SoC product.

In recent years, a class of high-performance programmable processor IP has emerged that is appropriate for use in high-volume embedded applications such as digital cellular, networking, communications, and console gaming [3,4]. Section 23.2 briefly describes the ManArray thread coprocessor as an example of the architectural features needed for demanding SoC requirements. The next subsection provides a brief description of the ManArray thread coprocessor architecture. Section 23.2.3 describes how the ManArray architecture fulfills SoC application requirements, with focus on the implementation, compiler, and tools. Section 23.2.4 presents performance results, and Section 23.2.5 concludes the section.

## 23.2.2 The ManArray Thread Coprocessor Architecture

In numerous application environments there is a need to significantly augment the signal processing capabilities of a MIPS, ARM, or other host processor. In addition, many applications require low power consumption at very high performance levels to accomplish the tasks of emerging applications, such as wireless LAN (i.e., 802.11a) for battery-powered Internet devices. The BOPS SoC cores provide streamlined coprocessor attachment to MIPS, ARM, or other hosts for this purpose. Through selectable parallelism, the ManArray SoC cores achieve high performance at low clock rates, which minimizes power requirements. The compiler or programmer can select from packed data, indirect VLIW, PE array SIMD, and multiple threaded forms of parallelism to provide the best product solution. Further, BOPS provides a complete solution by providing a comprehensive top-down design methodology for delivering the SoC solutions.

The ManArray processor is an array processor using a sequence processor (SP) array controller and an array of distributed indirect VLIW processing elements (PEs) (see Fig. 23.47). By varying the number of PEs on a core, an embedded scalable design is achieved with each core using a single architecture. This embedded scalability makes it possible to develop multiple products that provide a linear increase in performance and maintain the same programming model by merely adding array processor elements as needed by the application. As the processing capability is increased, the memory-to-PE bandwidth is increased, and the system DMA bandwidth may be increased as well. Embedded scalability drastically reduces development costs for future products because it allows for a single BOPS software development kit (SDK) to support a wide range of products.

In addition to the embedded scalability, ManArray cores are configurable in the number and type of cores included on a chip, instruction subsetting for application optimization, the sizes of each SP's instruction memory, the distributed iVLIW memories, the PE/SP data memories, and the I/O buffers, selectable clock speed, choice of on-chip peripherals, and DMA bus bandwidth. The ManArray cores provide a lower cost, more optimized signal processing solution than reconfigurable processors designed using FPGA technology [5]. Multiple ManArray cores provide optimized scalable multiprocessing by including multiple BOPS cores on an SoC product. These multiple ManArray cores can be organized to

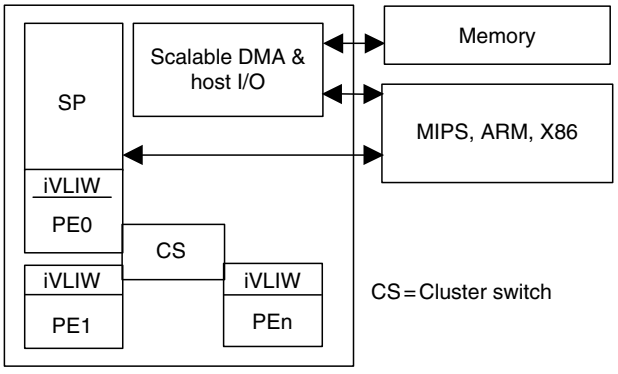



**FIGURE 23.47** ManArray architectural elements.

provide data pipeline processing between SP/PE-array cores and the parallelization of sub-application tasks (thread parallelism) with a centralized host-based control to be described later in this section.

Generally speaking, the ManArray processor combines PEs in clusters that also contain a SP, uniquely merged into the PE array, and a cluster-switch, Fig. 23.47. The SP provides program control, contains the instruction and data address generation units, and dispatches instructions to the processor array. In this manner, the ManArray processor is designed for scalability with a single architecture definition and a common tool set. The processor and supporting tools are designed to optimize the needs of a SoC platform by allowing a designer to balance an application's sequential control requirements with the application's inherent data parallelism. This is accomplished by having a scalable architecture that begins with a simple uniprocessor model and continues through multi-array processor implementations. In the design flow we ensured that the ManArray architecture supported a reasonably large array processor as well as a simple stand-alone uniprocessor that could act as an array controller. In more detail, a SP merged with PE0 (SP/PE0) and an additional PE (PE1), referenced as a $1 \times 2$, are shown in Fig. 23.48.

The ManArray architecture uses a distributed register file model where the SP and each PE contain their own independent register space, up to eight execution units (five shown), a distributed very long instruction word memory (VIM), local SP instruction memory, local data memories, and an application-optimized DMA and bus I/O control unit. In the Manta™ core, an available $2 \times 2$ implementation of the ManArray architecture, and its $1 \times 1$ and $1 \times 2$ subsets (available by software masking of selected PEs), a 64-entry register file space is used in the SP and each PE. The register space consists of a reconfigurable compute register file (CRF), which can act as a $32 \times 32$-bit or $16 \times 64$-bit register file for the execution units on a cycle-by-cycle basis, totally integrated into the instruction set architecture, an $8 \times 32$-bit address register file (ARF), and a $24 \times 32$-bit miscellaneous register file (MRF).

In the ManArray architecture, the address registers are separated from the compute register file. This approach maximizes the number of registers for compute operations and guarantees a minimum number of dedicated address registers. This approach does not require any additional ports from the compute register file to support the load and store address generation functions, and it still allows independent PE memory addressing for such functions as local data dependent table lookups.

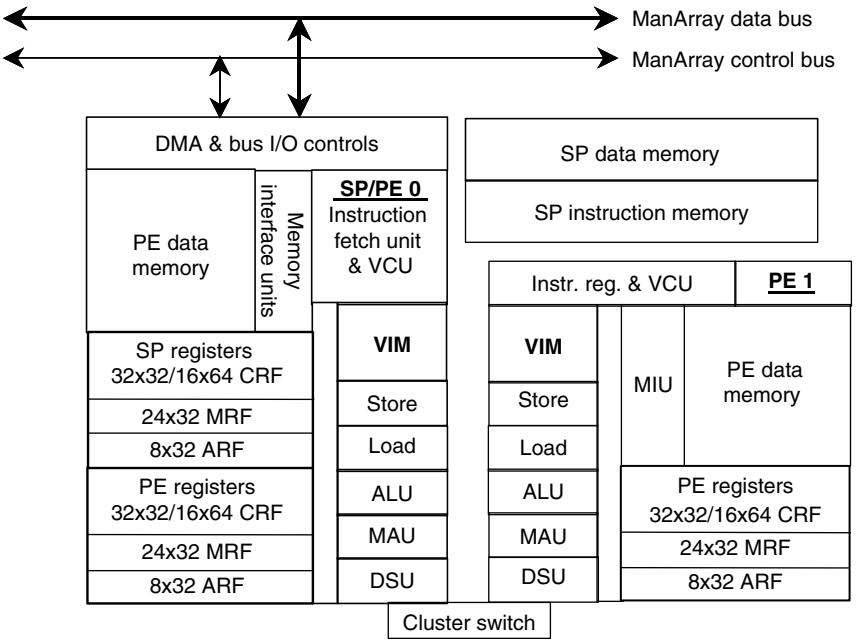



**FIGURE 23.48** ManArray $1 \times 2$ core elements.

The Manta chip supports both 32-bit data types including quad byte, dual halfword, and word; and 64-bit data types including octal byte, quad halfword, dual word, and double word. The balanced architectural approach taken for the compute register file provides the high performance features needed by many applications. It supports octal byte and quad halfword operations in a logical $16 \times 64$-bit register file space without sacrificing the 32-bit data type support in the logical $32 \times 32$-bit register file. Providing both allows optimum usage of the register file space and minimum overhead in manipulating packed data items. By adding PEs, the packed data support grows such that a $1 \times 2$ effectively provides 128-bit packed data support, a $2 \times 2$ provides 256-bit packed data support, etc., growing the level of parallelism needed by appropriate choice of the selected core.

The ManArray instruction set is partitioned into four groups using the high two bits of the instruction format—a control group, an arithmetic group, a load/store group, and a reserved proprietary instruction group. Figure 23.49 shows 32-bit simplex instructions in groupings that represent the five execution unit slots of the Manta chip, the first ManArray implementation, plus a control group (01). The execution units include store and load units, an arithmetic logic unit (ALU), a multiply accumulate unit (MAU), and a data select unit (DSU). The load and store instructions support base plus displacement, direct, indirect, circular, and table addressing modes. The ALU, MAU, and DSU support basic add/subtract, multiply, and data type manipulations such as shift, rotate, and permute, respectively. In addition, many application specific instructions are used for improved signal processing efficiency. An example of this are the multiply complex instructions for improved FFT performance described in reference [6].

The control and branch instructions are executed by the SP. It is also capable of indirectly executing VLIWs that are local to the SP and in each PE. To minimize the effects of branch latencies, a short variable pipeline is used consisting of Fetch, Decode, Execute, and ConditionReturn for non-iVLIWs and Fetch, PreDecode, Decode, Execute, and ConditionReturn for iVLIWs. The PreDecode pipeline stage is used to indirectly fetch VLIWs from their local VIMs. Note that VLIWs are stored locally in VIMs in each PE and in the SP and are fetched by a 32-bit execute VLIW (XV) instruction. In addition, an extensive scalable conditional execution approach is used in each PE and the SP to minimize the use of branches.

All loads/stores and arithmetic instructions execute in one or two cycles with no hardware interlocks. Further, all arithmetic and load/store instructions can be combined into VLIWs, stored locally in the SP and in each PE, and can be indirectly selected for execution from the small distributed VLIW memories (VIMs). Using the load iVLIW (LV) instruction, the programmer or compiler loads individual instruction slots with the 32-bit simplex instructions optimized for the algorithm being programmed. These VLIWs are used for algorithm performance optimization, are re-loadable, and require only the use of a 32-bit execute VLIW (XV) instructions in the program stored in the SP instruction memory.

A dedicated bit in all instruction formats controls whether an instruction is executed in parallel across the array of PEs or sequentially in the SP. To more optimally support a multiple PE array containing the distributed register files, the ManArray network is integrated into the architecture providing single-cycle data transfers within PE clusters and between orthogonal clusters of PEs. The DSU communications instructions can also be included into VLIWs, thereby overlapping communications with computation operations, which in effect reduces the communication latency to zero. The ManArray network

| Control | Store | Load | ALU | MAU | DSU |
|---|---|---|---|---|---|
| Group 01 | Group 10 | Group 10 | Group 11 | Group 11 | Group 11 |
| Call | Base+Disp. | Base+Disp. | ADD/SUB | ADD/SUB | Copy |
| Jump | Direct | Direct | Butterfly | Butterfly | Shift/rotate |
| EventPoint loops | Indirect | Indirect | Compare | MPY/MPYA | Permute |
| Return | Circular | Circular | AbsoluteDiff | MPYCmplx | Bit operations |
| Load VLIW | Table | Table | Min/Max | MPYCmplxA | Divide |
| Execute VLIW | ARF group | ARF group | Logicals | SUM2P/SUM2PA | Communications PEeXchange |
| | | Immediate | | | |
| | | Broadcast | | | |

**FIGURE 23.49** 32-bit simplex instructions.

operation is independent of background DMA operations, which provide a data streaming path to peripherals, such as a global memory.

The inherent scalability of the ManArray processor is obtained in part through the advanced ManArray network which interconnects the PEs. Consider by way of example, a two-dimensional (2D) $4 \times 4$ torus and the corresponding embedded 4D hypercube, written as a $4 \times 4$ table with both row, column, and hypercube node labels. (See Fig. 23.50a.)

In Fig. 23.50a, the $PE_{i,j}$ cluster nodes are labeled in gray-code as follows: $PE_{G(i),G(j)}$ where $G(x)$ is the gray code of $x$. First, columns 2, 3, and 4 are rotated one position down. Next, the same rotation is repeated with columns 3 and 4, and then with column 4. The resulting 4D ManArray table is shown in Fig. 23.50b.

Notice that the row elements in Fig. 23.50b, for example {(1,0), (0,1), (3,2), (2,3)}, contain the transpose PE elements. By grouping the row elements in clusters of four PEs each, and completely interconnecting the four PEs, connectivity among the transpose elements can be obtained. Notice also that, in the new matrix of PEs, the east and south wires, as well as the north and west wires, are connected between adjacent clusters. For example, using Fig. 23.50a note that node (2,3) connects to the east node (2,0) with wraparound wires in a torus arrangement. Node (2,3) also connects to the south node (3,3). Now, using Fig. 23.50b, note that nodes (2,0) and (3,3) are both in the same cluster adjacent to the cluster containing node (2,3). This same pattern occurs for all nodes in the new matrix. This means that the east and south wires can be shared and, in a similar manner, the west and north wires can be shared between all clusters. This effectively cuts the wiring in half as compared to a standard torus, and without affecting the performance of any SIMD array algorithm.

The rotating algorithm maintains the connectivity between the PEs, so the normal hypercube connections still remain as shown in one example in Fig. 23.50b as PE (1,0/0100) can communicate to its nearest hypercube nodes {(0000), (0101), (0110), (1100)} in a single step. Note also that the longest paths in a hypercube, where each bit in the node address changes between two nodes, are all contained in the completely connected clusters of processors nodes. For example, the circled cluster contains node pairs {(0100), (1011)} and {(0001), (1110)}, which would take four steps to communicate between each pair in previous hypercube processors, takes only one step to communicate in the new ManArray network. These properties are maintained in higher dimensional ManArray networks containing higher dimensional tori, and thus hypercubes, as subsets of the ManArray connectivity matrix. We have also shown that the complexity of the ManArray network is small and that the diameter, the largest distance between any pair of nodes, is 2 for all $d$ where $d$ is the dimension of the subset hypercube [7].

Application-specific instructions are included in the various execution units, such as multiply complex [6] and other video, graphics, and communications unique instructions. Any of the four groups of instructions can be mixed on a cycle-by-cycle basis. The single ManArray instruction set architecture supports the entire ManArray family of cores from the single merged SP/PE0 $1 \times 1$ to any of the highly parallel multi-processor arrays ($1 \times 2$, $2 \times 2$, $2 \times 4$, $4 \times 4$, etc.), for more details see references [8] and [9].

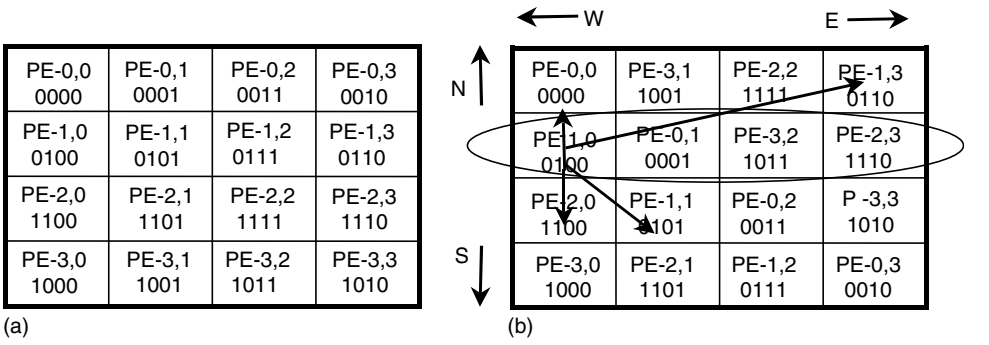



**FIGURE 23.50** Hypercube interconnection scheme.

### 23.2.3 The ManArray Thread Coprocessor Platform

The ManArray thread coprocessors are designed to act as independent coprocessors to ARM, MIPS, or other hosts. The programmer's view is a shared memory sequentially coherent model where multiple processors operate on independent processes. With this model, an SoC developer can quickly utilize the signal processing capabilities of the ManArray core subsystem since the operating system already runs on the host processors. In its role as a digital signal coprocessor, the ManArray core is subservient to the host processor. A core driver running on the host operating system manages all the DSP resources on the core. The ManArray system interface allows multiple BOPS cores to be attached to a single host processor as shown, for example, in Fig. 23.51. For wireless and media processing applications the $1 \times 1$ MOCARay-I mobile communications accelerator and the $1 \times 2$ MICORay-I imaging communications engine are designed to work separately or jointly, as shown in Fig. 23.51, to provide ultra low-power baseband and media DSP services for 3G mobile products. Figure 23.51 shows a multimode Smart Phone or PDA with MOCARay-I providing the GPRS/EDGE and/or UMTS mode while MICORay-I provides support for video MPEG-4, JPEG 2000 photo imaging, speech decode/encode, sprite-based rendering in a gaming mode, audio processing MP3, etc.

Another example of the use of BOPS cores as thread coprocessors is in voice-over-Internet protocol (VoIP) products. For this application, an integrated dual $1 \times 2$ arrangement with a common DMA controller is used as the basic platform unit. One, two, or four of these dual $1 \times 2$ units are provided as an SoC DSP "farm" with an on-board host engine, e.g. MIPS.

Figure 23.52 illustrates the configuration with eight $1 \times 2$ cores. This system arrangement allows the workload to be partitioned appropriately allowing extant applications to run on existing host OSs in the MIPS controller. This lowers the risk of migrating an existing code base, and no new OS ports are

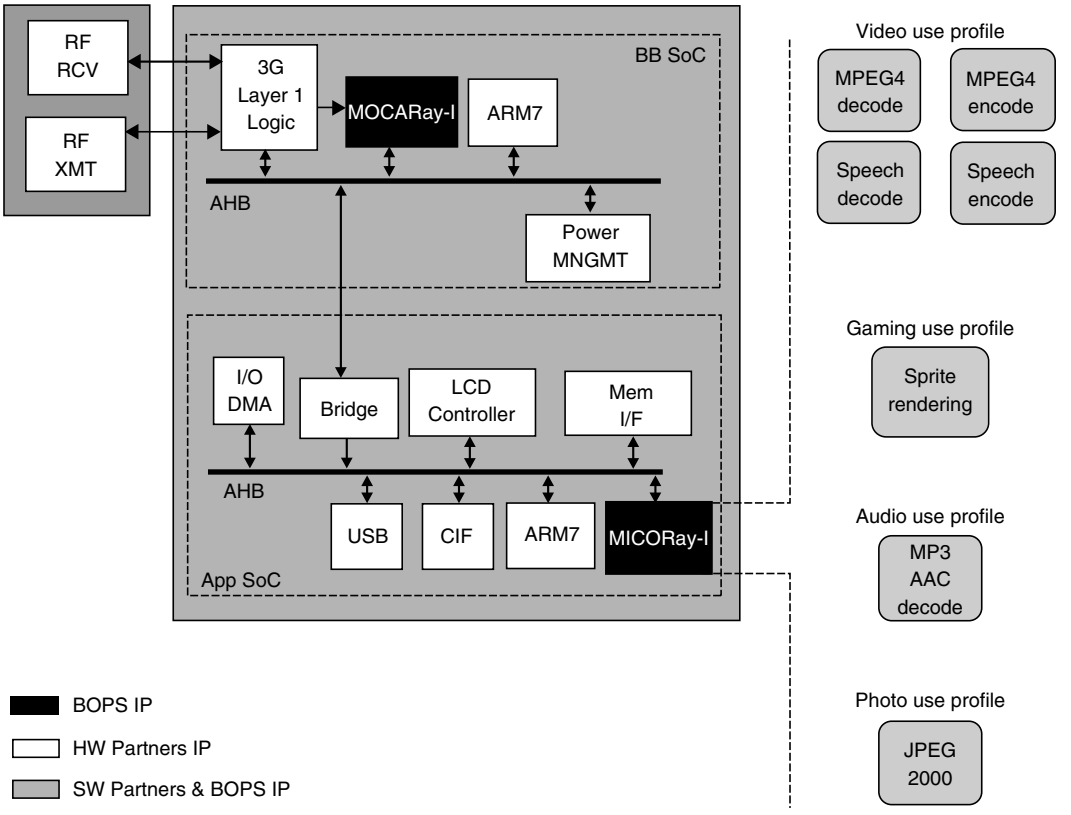



**FIGURE 23.51** Application of multiple BOPS cores to 3G wireless.

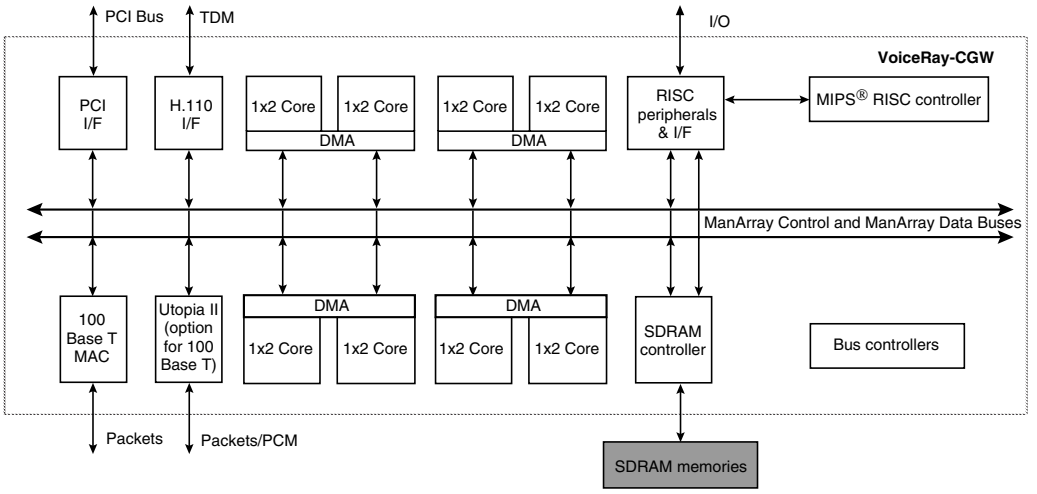



**FIGURE 23.52** Application of multiple BOPS cores to VoIP.

required to support the ManArray cores. To complement the configurable hardware, there is a BOPS library of both DSP and control software routines to perform the desired VoIP gateway functions. In addition, existing host optimized compilers are used for the sequential code that remains resident on the host allowing the parallel code to be optimized for the ManArray cores.

A driver API allows host applications to initialize, control, and communicate with multiple ManArray coprocessors attached to the host. A standard message interface exists for all coprocessors to/from the host. Specifically, the ManArray core DSPs are described as thread coprocessors because the host processor dispatches entire threads of execution to the cores. The host driver loads programs, schedules context switches, and manages data streams into and out of the various coprocessors. The high-performance DMA engine, scaled appropriately for the application, autonomously transfers data streams to and from the host. In addition, data streams can be "pushed/pulled" from one coprocessor to another, or to/from peripherals (such as an H.100 interface) and coprocessors, without host intervention, using the ManArray DMA-to-DMA interconnection protocol.

The DMA subsystem consists of the DMA controller, a ManArray control bus (MCB), and a ManArray data bus (MDB). The MDB provides the high-bandwidth data interface between the cores and other system peripherals including system memory. The MDB consists of multiple identical lanes and is scalable by increasing the number of lanes and/or increasing the width of the lanes. Specifically, the MDB uses time division multiplexing of multiple independent buses or lanes to transfer data. The MCB is a low latency coprocessor-to-coprocessor/peripheral messaging bus, which runs independently and in parallel with the MDB. This system of multiple independent application task-optimized cores is designed to have each core run an independent thread supported by the programmable DMA engines [10].

Figure 23.53 illustrates the host-DSP software layers. The multiple ManArray cores support the multiple independent program threads that are managed by the host OS through remote procedure

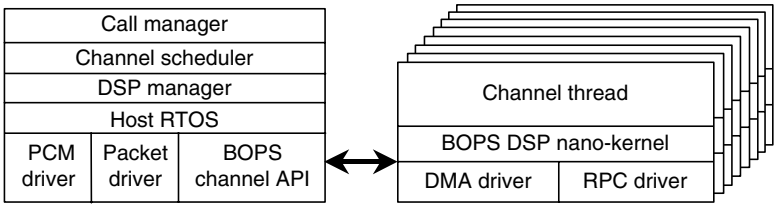



**FIGURE 23.53** Host DSP software layers.

calls (RPC) scheduled by the RTOS driver running in the host. The BOPS channel processing API provides a standard interface for allocating voice channel processing to the multiple $1 \times 2$ cores. On the ManArray core side, a thin DSP nano-kernel supports thread load/unload with DMA transfers overlapping computation. The RPC and DMA drivers provide standard host-DSP communication and data transfer support.

Supporting this scalable platform for VoIP solutions is BOPS SoC design flow as shown in Fig. 23.54. Four parallel processes are shown supporting both hardware and software design efforts for the thread coprocessors, peripherals, host software, and DSP software developments.

Once the SoC functional specification and a basic system design is determined the next development steps can be done in parallel. The ManArray core RTL and other peripheral RTL are done in parallel, being designed to the ManArray interface specifications. At the same time, due to the use of a cycle accurate system simulator and other supporting tools, the host CPU software and DSP software development are done in parallel.

To streamline development, verification and debug, BOPS provides a range of modeling and prototyping platforms to support system modeling, and software and hardware system development including:

- A cycle-accurate C-simulator, which can be used to develop ManArray DSP and system software. This can be used directly with other C simulations, or with control processor tools and bus models to provide a software simulation model of an entire system.
- A software development toolkit (SDK) including the BOPS ANSI-C Halo™ parallelizing C compiler.
- The Jordan™/Manta™ PCI card, which can be used to model and test DSP software at 100 MHz processor speeds and under actual DMA I/O conditions.

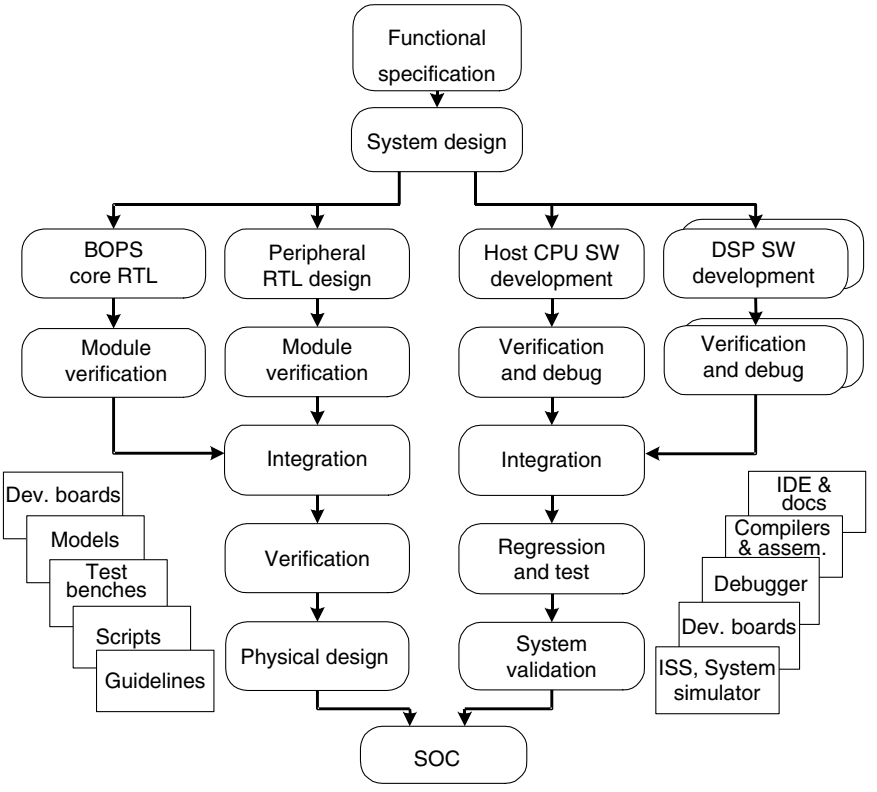



**FIGURE 23.54** BOPS SoC design flow.

- The Travis™/Mantax™ prototyping board, which can be used to prototype entire SoC systems at actual speeds.
- The Xemulator™ emulation board, which can be used to model RTL emulations of an entire SoC system.

The same DSP debugging GUI is shared by the C-simulator, Jordan, Travis, and Xemulator boards. The Jordan development and Travis prototyping boards are provided using a $2 \times 2$ Manta core that contains features needed in many applications. The Travis system prototyping board uses the standard configurations available on Manta cores $1 \times 1$, $1 \times 2$, and $2 \times 2$. All of the normal fixed resources such as host microcontroller, oscillators, memories, power supplies, configuration controls, debug, peripheral and PCI interfaces are also on board. In addition, a large FPGA accommodates all unique logic circuits allowing for rapid design, testing, and debug. The Jordan board with the Manta chip provides real time operation of standard cores in a MIPS host system with off board system prototyping. Also included on the Jordan board is a MIPS microcontroller with interrupts and boot ROM. With these development boards the software can be integrated and tested. BOPS also provides verification tools and supporting scripts and guidelines for the physical design.

The SoC emulator board, the Xemulator, allows MHz speed emulation in FPGAs of the many possible RTL hardware and software configurations of the scalable ManArray architecture. This is useful when bus sizes, bus protocols, and external interfaces are changed from the standard core configurations. Likewise, I/O and DMA controllers may need to be altered for certain applications. Additions, subsetting, and other changes to the instruction set can be explored on the Xemulator by modifying the downloaded core's FPGA description.

### 23.2.4 Performance Evaluation

To illustrate the power of the highly parallel ManArray architecture, a simple example is presented: Two vectors are to be added and the result stored in a third vector.

$$\begin{aligned} &\text{for } (i = 0; i < 256; i{+}{+}) \\ &\quad A[i] = B[i] + C[i] \end{aligned}$$

In a sequential-only implementation there would be required a loop of four instructions, two loads to move a $B$ and a $C$ element to registers, an add of the elements, and a store of the result to register $A$. The sequential implementation takes $(4 * 256)$ iterations $= 1024$ cycles, assuming single cycle load, add, and store instructions.

Assuming the data type is 16-bits and quad 16-bit packed data instructions are available, the vector sum would require $(4 * 64)$ iterations $= 256$ cycles.

Further assuming an array processor of four PEs where each PE is capable of the packed data operations, then the function can be partitioned between the four PEs and run in parallel requiring $(4 * 16)$ iterations $= 64$ cycles.

Finally, assuming a VLIW processor such as the ManArray processor, a software pipeline technique can be used with the VLIWs to minimize the instructions issued per iteration such that $(2 * 16)$ iterations $= 32$ cycles are required. This represents a 32x improvement over the sequential implementation.

ManArray architecture allows a programmer or compiler to select the level of parallelism appropriate for the task at hand. This selectable parallelism includes packed data operations ($4 \times 16$-bits and $8 \times 8$-bits in one 64-bit operation on the Manta core), parallel array PEs (performance scales linearly with the addition of PEs), and instruction level parallelism (iVLIW concurrent store/load, ALU, MAU, and DSU instructions).

By use of the three levels of parallelism available on each core, including the use of single-cycle PE communications, scalable conditional execution, and background data streaming DMA, the following benchmarks, Fig. 23.55, can be obtained on the Manta $2 \times 2$ thread coprocessor, which can also function as subset array $1 \times 1$ and $1 \times 2$ array processors.

| Benchmark | Data type | Performance |
|---|---|---|
| 256 pt. Complex FFT (2x2) | 16-bit real & imaginary | 383 cycles |
| 256 pt. Complex FFT (1x1) | 16-bit real & imaginary | 1115 cycles |
| 1024 pt. Complex FFT (2x2) | 16-bit real & imaginary | 1513 cycles |
| 1024 pt. Complex FFT (1x1) | 16-bit real & imaginary | 5221 cycles |
| 2048 pt. Complex FFT (2x2) | 16-bit real & imaginary | 3182 cycles |
| 2D 8x8 IEEE IDCT [11] (2x2) | 8-bit | 34 cycles |
| 2D 8x8 IEEE IDCT (1x1) | 8-bit | 176 cycles |
| 256 tap Real FIR filter, M samples (2x2) | 16-bit | 16*M + 81 cycles |
| 256 tap Real FIR filter, M samples (1x1) | 16-bit | 64*M + 78 cycles |
| 4x4 Matrix*4x1 vector (2x2) | IEEE 754 Floating Point | 2 cycles / output vector |
| 3x3 Correlation (720col) (2x2) | 8-bit | 271 cycles |
| 3x3 Median Filter (720col) (2x2) | 8-bit | 926 cycles |
| 8x8 Block Motion Est. (H=64, V=32) (2x2) | 8-bit | 4611 cycles |
| Horizontal Wavelet (N Rows = 512) (2x2) | 16-bit | 1029 cycles |

**FIGURE 23.55** Manta $2 \times 2$ thread coprocessor benchmarks.

### 23.2.5 Conclusions and Future Extensions

The pervasive use of processor IP in embedded SoC products for consumer applications requires a stable design point based on a scalable processor architecture to support future needs with a complete set of hardware and software development tools. The ManArray cores are highly scalable, using a single architecture definition that provides low power and high performance. Target SoC designs can be optimized to a product by choice of core type, $1 \times 1, 1 \times 2, 2 \times 2, \ldots$ and by number of cores. The BOPS tools and SoC development process provides a fast path to delivering verified SoC products. Future plans include architectural extensions, representing a superset of the present design, which greatly improve performance in the intended applications.

## Acknowledgments

The author thanks Dr. Steve Walsh, Dr. Dave Strube, Dr. Sergei Larin, and Carl Lewis for their comments in preparing this paper. The author also thanks the BOPS, Inc. development team for their drive, excitement, and creative technical skills in developing the ManArray processor, tools, and their supporting functions, as well as for making BOPS, Inc. a fun place to work.

### Trademark Information

BOPS and ManArray are registered trademarks of BOPS, Inc. Manta, MoCARay, MICoRay, Jordan, Travis, Xemulator, and Halo are trademarks of BOPS, Inc. All other brands or product names are the property of their respective holders.

## References

1. The Design and Implementation of Signal-Processing Systems Technical Committee, edited by Jan M. Rabaey, with contributions from W. Gass, R. Brodersen, and T. Nishitani, "VLSI design and implementation fuels the signal-processing revolution," *IEEE Signal Processing Magazine*, pp. 22–37, Jan., 1998.
2. Alan Gatherer, Trudy Stetzler, Mike McMahan, and Edgar Auslander, "DSP-based architectures for mobile communications: past, present, and future," *IEEE Communications Magazine*, pp. 84–90, Jan., 2000.
3. Krishna Yarlagadda, "The expanding world of DSPs," *Computer Design*, pp. 77–89, March, 1998.
4. Ichiro Kuroda and Takao Nishitani, "Multimedia processors," *Proceedings of the IEEE*, Vol. 86, No. 6, pp. 1203–1227, June, 1998.

5. Bruce Schulman and Gerald G. Pechanek, "A 90k gate "CLB" for Parallel Distributed Computing," in *Proceedings of EHPC. IPDPS Workshops 2000,* pp. 831–838.
6. Nikos P. Pitsianis and Gerald G. Pechanek, "High-performance FFT implementation on the BOPS ManArray parallel DSP," *Advanced Signal Processing Algorithms, Architectures and Implementations IX,* Volume 3807, pp. 164–171, SPIE International Symposium, Denver, CO, USA, July, 1999.
7. Gerald G. Pechanek, Stamatis Vassiliadis, and Nikos P. Pitsianis, "ManArray interconnection network: an introduction," in *Proceedings of EuroPar '99 Parallel Processing,* Lecture Notes in Computer Science, Vol. 1685, pp. 761–765, Toulouse, France, Aug. 31–Sept. 3, 1999.
8. Gerald G. Pechanek and Stamatis Vassiliadis, "The ManArray embedded processor architecture," in *Proceedings of the 26th Euromicro Conference: "Informatics: inventing the future,"* Maastricht, The Netherlands, September 5–7, 2000, Vol. I, pp. 348–355.
9. BOPS, Inc. corporate Web site (www.bops.com).
10. David Baker, "BOPS DSPs as co-processors," BOPS Internal Technical Report, April 4, 2001.
11. Gerald G. Pechanek, Charles Kurak, and Bruce Schulman, "Design of MPEG-2 function with embedded ManArray cores," in *Proceedings of DesignCon 2000,* Jan. 31–Feb. 3, 2000.

## 23.3 Digital Audio Processors for Personal Computer Systems

*Thomas C. Savell*

### 23.3.1 Introduction

The audio subsystem of the personal computer (PC), once an almost unnecessary component, has become an integral part of the operating systems and applications software that run on them. The evolution of the PC itself has led to a complex audio system, requiring simultaneous playback and recording while applying advanced signal processing. The best PC audio systems employ one or more specialized digital audio processors to off-load the main processor and guarantee artifact-free audio.

### 23.3.2 Brief History and Evolution

The early PCs could only generate simple tones and beeps. In the early 1980s, the system designers of the original IBM-PC used the Intel 8253 digital timer to generate a series of pulses at a regular rate, usually a square wave within the audio range of less than 20 kHz. The output of this chip drove the base of a transistor to switch on and off a small speaker as shown in the schematic representation of Fig. 23.56. This simple, cost-effective solution effectively off-loaded the 4.77 MHz Intel 8088 main processor from the task of generating a tone. The 8253 timer had three independent channels, and the system designers used channel 0 to keep track of the time of day and channel 1 to generate DRAM refresh cycles. Thus, the otherwise unused timer channel 2 provided an essentially cost-free audio processor.

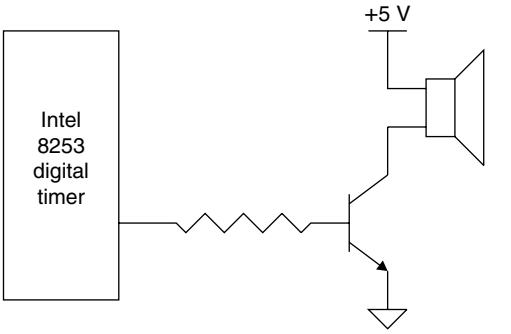



**FIGURE 23.56** Simplified schematic of IBM-PC speaker circuit.

Although the most common use of this primitive audio system was to alert the user to an event, clever programmers were able to play simple melodies using it. Eventually, they discovered how to use pulse-width modulation coupled with the reactance of the circuit to create a low-quality digital-to-analog converter (DAC), enabling the playback of digitally sampled waveforms. Audio

created using this method, however, was very noisy and presented a significant load on the main processor.

Later in the 1980s, add-in cards appeared with an integrated music synthesizer capable of playing back polyphonic music. One such early card, the AdLib sound card, used a form of music synthesis known as frequency modulation (FM) synthesis. As described by John Chowning in 1973, FM synthesis creates complex sounds using simple sine waves to modulate the frequency of other sine waves [1]. The AdLib sound card used the simple Yamaha OPL2 FM synthesis chip, which used only two sine waves per voice to synthesize complex waveforms. It could create satisfactory, yet unrealistic synthesis of natural musical instruments, as well as a limited spectrum of special sound effects.

The immensely popular AdLib-compatible SoundBlaster (Creative Technology, Ltd.) was introduced in 1989 by Creative Labs. In addition to AdLib's FM synthesis capabilities, it added a simple method of playing and recording digital audio encoded as a pulse-code modulated (PCM) stream. Perhaps as important to its success, Creative Labs provided software development support to computer game developers free of charge, resulting in widespread software support for the SoundBlaster. The new PCM audio capabilities added the possibility of using any sound as an effect in a game. This important enhancement led to the requirement for PCM audio on all future sound cards.

PCM audio was transferred to and from the sound card using the Intel 8237 direct memory access (DMA) controller on the main system motherboard, as shown in Fig. 23.57. The early SoundBlaster cards could only transfer 8-bit PCM audio, resulting in a dynamic range of only about 48 dB. Later, the SoundBlaster 16 card added support for 16-bit PCM audio with a much better 96 dB dynamic range, using the 16-bit DMA controller of the newer computers.

As time progressed, wavetable synthesis replaced FM synthesis. Wavetable synthesis is capable of synthesizing musical instrument sounds that are nearly indistinguishable from real instruments except to the trained ear. It works by triggering digital recordings of notes played on actual instruments in response to keys played on a keyboard. To synthesize the sound of a piano, the wavetable synthesizer stores a series of digital recordings of a real piano, and plays them back on command. Although the sound quality is far superior to that produced by the earlier FM sound cards, the high price of the early wavetable implementations was prohibitive to widespread market acceptance. The normal market forces eventually drove the price down, and sound cards that could only produce FM became obsolete.

An important force in the evolution of both graphics and audio in PCs is the computer game. The continually increasing realism of game graphics, with players able to navigate through virtual three-dimensional (3-D) environments, created demand for more realistic game audio. This demand led to the advent of 3-D positional audio, allowing accurate placement of sound sources within a virtual 3-D environment rendered on stereo speaker systems. It ultimately led to full environmental simulation, with the ability to simulate a sound in various environments such as a carpeted room, a large hall, and even under water.

A 3-D audio experience is difficult to achieve using two speakers. The smallest head movement of the listener can often destroy the effect. Movie theaters overcame this problem using multi-speaker sound

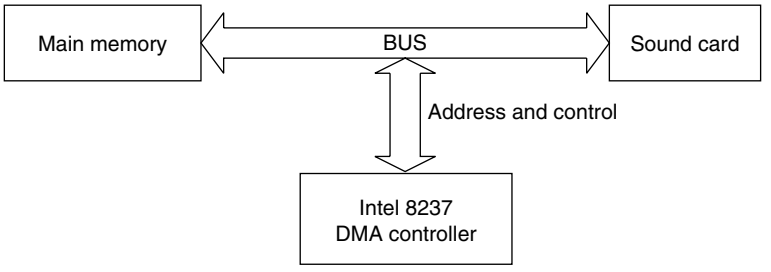



**FIGURE 23.57** Slave DMA using the Intel 8237 DMA controller.

systems that placed speakers to the sides and rear of the listener. This eventually migrated to home theater systems and finally to computer gaming systems.

The best systems now have a 7.1 channel audio system supporting high channel-count content, such as Dolby Digital Plus (Dolby Laboratories) coupled to a 3-D rendering sound card with environmental simulation capabilities. These systems provide an audio experience that immerses the listener in the environment, helping to create the illusion of realism.

### 23.3.3 Today's System Requirements

Today's systems use a layered approach, with applications able to produce audio with little or no knowledge of the underlying hardware. Layers of software hide most of the hardware-specific features. Applications use a query mechanism to determine the features present, enabling considerable freedom in hardware implementation. Many features of the audio system can also be rendered in software, guaranteeing the application developer a minimum feature set and performance level, nearly independent of the installed hardware. Thus, today's architecture is scalable, allowing the user to choose hardware acceleration for better performance, or software emulation for lowest cost.

Audio on a PC can be divided into several general categories, including operating system interaction, music, gaming, and voice applications. Each of these categories has unique properties, but with proper architecture, a single solution can apply to all of them.

Operating system interaction is generally limited to alerting the user to various events, such as starting up the system, selecting an invalid choice, or receiving new e-mail messages. In the early days of PCs, simple beeps communicated all of these items. Now, these events can be associated with any sound recording, and each association can be unique. Whenever an event occurs, the operating system instructs the sound card to play back the associated sound recording.

Music applications are much more complex. The sound card is required to provide a wavetable synthesizer responsive to musical instrument digital interface (MIDI) commands [2]. In addition, it must be able to play back streaming audio in various formats, including PCM, MP3 (MPEG-1 Layer 3 Audio), and Dolby Digital (5.1-channel home theater audio). Finally, it must be capable of recording in CD quality, or 16-bit stereo PCM at a sample rate of 44.1 kHz. And the most advanced systems support multichannel recording at sample rates up to 192 kHz and resolution up to 24-bits. Each of these major features must be independent and operate simultaneously.

Gaming applications often require a very sophisticated audio system. Many games place sound sources in a virtual 3-D space. The user expects the system to render this 3-D space on any number of speaker systems, ranging from headphones to stereo speakers to 7.1-channel home theater systems. The virtual 3-D space includes not only positional cues, but environmental cues as well. A game player must be able to move a character from an open outdoor space into a small wooden room and seamlessly hear the environmental cues, such as the short reverberation of a small room. Objects in motion produce the well-known Doppler effect, increasing the apparent frequencies of the sounds emitted by objects moving toward the listener and decreasing those of objects moving away [3]. The most sophisticated audio systems can reproduce the Doppler effect on both the objects in motion and their reflections.

Voice applications, although not new, have yet to gain the widespread availability of operating system, music, and gaming applications. Because of the large memory requirements, voice recognition algorithms are better suited to the main processor and use limited, if any, preprocessing by the sound card in the record path. Moreover, automatic voice recognition is still unreliable, except when restricted to isolated words from a limited vocabulary. Another class of voice applications is voice communication. The emergence of the Internet has brought with it the promise of low-cost worldwide telephony. The implementation of Internet telephony requires sophisticated noise-cancellation and echo-cancellation algorithms that are often best suited to run on the sound card. In a related application, many multiplayer games, such as those played over the Internet provide players the ability to talk to each other within the game. The games record a voice microphone and compress it for transmission across the network to the other players in the game. Upon receipt of transmitted voice from another player, the game

decompresses the stream and places it in 3-D space at the same coordinates as the visual representation of the other player, providing a compelling gaming experience. This requires that the audio system provide simultaneous recording capability while rendering 3-D audio.

## 23.3.4 Hardware Architecture

The hardware of the PC audio system satisfies these system requirements with a simple model. Much like the entire computer system, it consists of three major subsystems: storage, processing, and input/output (I/O). The storage subsystem can include local memory, system memory, and disk storage, such as hard drives and compact discs, but the audio processor does not usually interface directly with a disk storage device. The processing subsystem includes both the main processor and a processor located on the sound card to provide hardware acceleration. The I/O subsystem may consist of an analog interface, such as the Audio CODEC'97 (AC97) standardized DAC and ADC, or a newer standard such as an HD-Audio CODEC. Higher quality systems use expensive high resolution converters to provide signal-to-noise ratios (SNR) in excess of 110 dB. In addition, digital interfaces, such as the Sony/Philips digital interface (S/PDIF) are often included. By dividing the audio system into three logical blocks, the system designer faces the simplified task of creating each block while optimizing the interfaces between them. The audio processor designer is concerned with the processing capabilities of the chip as well as the I/O system interface and the memory bus interface.

### 23.3.4.1 Memory

Local memory connects directly to the audio processor. This includes both ROM and RAM of various types located on the sound card, generally used to store wavetables for wavetable synthesis and digital delay lines for environmental simulation algorithms. Local memory provides the highest system performance for wavetable synthesis and environmental simulation since it need not share bandwidth with the main processor and other hardware, such as disk, video, and networking interfaces; however, local memory costs money, and cost is often a major consideration in market-driven engineering. The emergence of the RAM-less sound card, which stores audio in system memory rather than local memory, is primarily due to the need to decrease costs.

Creation of a RAM-less sound card requires that the system memory stores most audio data. A relatively small amount of RAM is still required on the audio processor chip for algorithms that require high-bandwidth access to memory. System memory connects to the main processor of the PC through bus bridging logic, and stores the programs and data that make up the operating system and application programs. When the audio processor requires access to system memory, it generates a memory access request on the add-in card bus. If the main processor or any other device is currently accessing system memory, the audio processor must wait.

The early PCs used a relatively low-performance add-in card bus known as ISA (Industry Standard Architecture). The sound cards that plugged into the ISA bus accessed system memory through the Intel 8237 DMA controller. The 8237 DMA controller contains auto-incrementing address registers and uses a request/acknowledge handshake protocol to communicate with requesting devices. Because it generates the memory address and controls the direction of the transfer, it is the bus master.

The sound card operates as a slave to the 8237 DMA controller, which is limited to a single address per requesting device and only performs single-word transfers. In addition, certain channels of the DMA controller are limited to 8-bit transfers, and others can perform 16-bit transfers. The sound cards that could support 16-bit samples had to allocate two DMA channels: one for 8-bit audio and the other for 16-bit audio. These limitations were acceptable for sound cards that did not require system memory to store wavetables or digital delay lines. These sound cards either had local memory or supported only FM synthesis. They used this slave DMA system for streaming audio, which is generally a recording of music or other sounds.

Applications such as games that create virtual environments generate a continuous stream of audio; however, it is not as simple as playing a static recording. The content of the stream changes based on the

actions of the user. As the user interacts with the virtual environment, virtual objects move in relation to the listener, and the sounds they produce may change over time. Each sound an object can make is usually stored as a short recording. The process of creating the continuous audio stream that represents the virtual environment entails summing all the sound sources within the listener's range. The use of slave DMA for these types of applications requires software to create the continuous audio stream. The software for positioning objects in a virtual 3-D space is nontrivial, so the applications generally simplify the problem when using an ISA bus sound card.

A better solution is to place a powerful DMA controller directly on the sound card. When the sound card contains a DMA controller, it becomes the bus master, and can overcome the limitations of the 8237A DMA controller. For example, it can have a large number of independent address generators, enabling both wavetable synthesis and 3-D hardware acceleration for audio stored in system memory. A sound card that supports wavetable synthesis or environmental simulation using system memory must have a bus-mastering DMA controller.

The audio processor designer must consider the memory bandwidth requirements, bus bandwidth availability, and bus transfer latency to determine whether bus-mastering DMA is a viable design choice. Given the number of simultaneous audio channels, the sample rate of each channel and the number of bytes in each sample, the designer can easily calculate the memory bandwidth requirements. For example, a processor supporting 64 audio channels with a sample rate of 48 kHz and 2 bytes (16-bit) per sample requires 6,144,000 bytes/s. The available bus bandwidth must be greater than that for it to be a viable design choice.

Calculating available bus bandwidth is much more difficult. It depends on the bus bandwidth capability, the reserved bandwidth for other transactions on the bus, and any transaction overhead not accounted for in the bus bandwidth capability. The bus bandwidth capability is straightforward to calculate. The simplest method is to use the data transfer rate times the bus width.

For example, a single-lane 2.5 GHz PCI Express [4] bus has a raw bandwidth of 250 MB/s, since it uses 10-bits per byte. This, however, ignores the per-transaction overhead inherent in the bus protocol. For example, the overhead for a memory read transaction is 24 bytes. Thus, a minimum data transfer of 4 bytes consumes 28 bytes of bus bandwidth. If all transactions are minimum 4-byte data transfers, the available bus bandwidth is only about 36 MB/s, ignoring latency and control. Burst data transfers reduce the effect of the per-transaction overhead. Transferring 64 bytes in a burst transaction increases the theoretical bus bandwidth to about 181 MB/s. Computers represent a waveform as an array of numeric values in memory, so audio is well suited to burst transactions. But larger burst sizes increase the audio latency, which becomes unacceptable beyond a few milliseconds, especially for interactive applications. And owing to other factors in both the system and protocol, actual bus bandwidth may be significantly less, perhaps 100 MB/s.

The least quantified of all the factors is the reserved bandwidth for other transactions in the system. Other devices such as the CPU and graphics adapters compete with the audio system for access to main memory, delaying transactions initiated by the audio system. Even the bandwidth needed to program the audio processor can reduce available bandwidth for memory access. The reduction in bandwidth because of other devices is unknown, since it depends on the configuration of each user's individual system. Even the bandwidth needed to program the audio processor is difficult to quantify, since it depends on the peculiarities of the software device driver, the operating system, and the application programs that ultimately generate the audio. Any method of determining the amount to reserve seems entirely arbitrary, since variable quantities determine the optimal amount.

Instead of relying on an arbitrary decision based on a guess, one could make measurements of the bus bandwidth available on the bus in a typical system. This may require building a prototype card to emulate the performance of the audio system. Although measurements are by no means a guarantee that any particular system will provide enough bandwidth, one can assume that a similarly equipped typical system will provide a similar amount of bandwidth. The designer can also estimate the bandwidth needed to program the audio processor. Clearly, there is a known overhead to start up a single channel of audio. The bandwidth needed to program the processor includes at least this overhead multiplied by the

number of channels. There is additional bandwidth required to maintain a channel of audio. For example, if an object in a virtual 3-D environment moves, the processor must reprogram the portion of the audio processor that positions the object. Numerous other facets are part of this problem, and the audio processor designer should consult with the software engineers to obtain a reasonable estimate of the true bandwidth needed to program the processor.

Often the available bus bandwidth exceeds the requirements of the audio system, yet audio defects still occur. This happens when the system exceeds the latency tolerance of the audio device. To implement burst data transfers and gain the associated efficiency boost, the audio system pre-fetches some number of audio samples. If it plays through all of the pre-fetched samples before the next memory transaction completes, an audio defect will occur. For example, if an audio system pre-fetches 16 samples and then requests another 16 samples, it will tolerate a transaction latency of 16 sample periods without any defects. This example system has a latency tolerance of 16 sample periods, or 0.333 ms given a 48 kHz sample rate.

With high performance buses such as PCI Express, latency can be more of a problem than bandwidth. And latency has a cumulative effect on a high channel-count application such as positional 3-D audio. The problem here is that a single long latency transaction could cause many other streams to grow near the starvation point, since their transactions must wait as well. For example, with 32 simultaneous streams, a single transaction with 15 sample periods of latency could cause the other 31 streams to require immediate service within a single sample period, assuming all streams have a 16 sample period latency tolerance.

Given estimates of the memory bandwidth required for audio data transfer, the available bus bandwidth, and the expected average and maximum transaction latencies, the designer can determine the limits at which the system will fail. On the basis of this information, the processor implementation or the target system requirements may need to change.

### 23.3.4.2 Mixing Multiple Sources

The basic system requirements and user expectations require that the sound system sum together multiple audio sources with an independent level control for each source. The audio term for this summation process is mixing. The operating system usually provides software for a simple audio mixer that enables the user to control the relative levels of the compact disc, line in, microphone, and various internally generated sound sources. The system often uses a small digitally programmable analog mixer for the analog sources, such as line in and microphone; however, the wavetable synthesizer and 3-D gaming applications require mixing a relatively large number of channels under real-time software control, as shown in Fig. 23.58. These applications use an all-digital mixer due to the large number of channels.

On the surface, a digital audio mixer sounds like a trivial exercise in multiply–accumulate operations; however, in order to sum together sampled waveforms, they must all have the exact same sample rate. Consider two sampled waveforms, each 1 s in length. The first has a sample rate of 48 kHz and the second has a sample rate of 24 kHz. Although they both represent 1 s of time, the first waveform consists

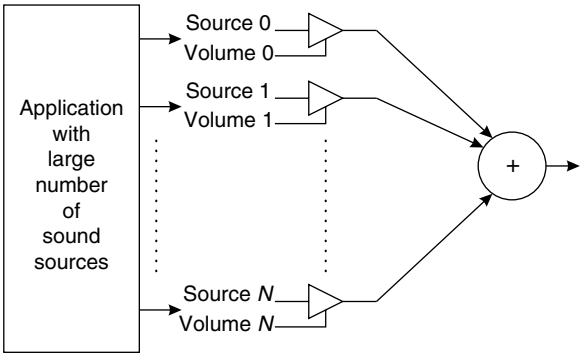



**FIGURE 23.58** Mixing a large number of sound sources under software control.

of 48,000 points and the second waveform consists of 24,000 points. In order to mix them, they must have the same number of points representing the same amount of time.

The solution is to use a sample rate converter before the mixer. The sample rate converter has a fixed output sampling-rate and a variable input sampling-rate. This allows the digital mixer to operate on multiple waveforms of different sample rates. The software programs the sample rate converter with the ratio of input to output, also known as the pitch. Pitch is a musical term that relates to the frequency of a note. A higher pitch corresponds to a higher frequency. The sample rate converter performs a double duty as a pitch shifter, enabling a single-recorded note to reproduce many notes on the instrument. This provides effective data compression by reducing the number of recordings required to reproduce the sound of an instrument. In addition, it enables a variety of musical effects, such as vibrato, pitch bend, and portamento. Finally, the pitch shifting effect of the sample rate converter emulates the Doppler effect needed by 3-D environmental audio. Thus, the sample rate converter is a fundamental building block used by nearly all facets of the digital audio system in the PC.

#### *23.3.4.2.1 Sample Rate Converters*

Sample rate converters come in several varieties, offering different levels of conversion quality. Higher quality conversion requires more computation, and comes at a correspondingly higher cost. Drop-sample converters require almost no computation to implement and offer the lowest quality. Linear interpolation converters require more computation and offer reasonably good quality, especially for downward pitch shift. Multipoint interpolation converters require the most computation and memory bandwidth, but provide the highest quality; however, there can be considerable variation in the quality of multipoint interpolation converters.

To understand sample rate conversion, it is necessary to understand discrete-time sampling theory as described by Nyquist and Shannon [5,6]. To sample a signal properly, the sample rate must be at least twice the highest frequency component in the signal. The Nyquist frequency is one-half the sample rate, and indicates the highest frequency component that a particular sample rate can represent. Sampling of frequency components above the Nyquist frequency results in aliases in the sampled waveform that are not present in the original signal. Sampling systems such as digital recorders typically use a low-pass filter at the input of the analog-to-digital converter to avoid aliasing.

The relationship between the frequencies of the aliases and those of the original out-of-band signal is simple. A sine wave at a frequency $F$ between the Nyquist, $N$, and the sample rate, $2N$, will alias to a frequency of $2N - F$. Consider a signal consisting of two sine waves, one at 28,000 Hz and another at 45,000 Hz. Using a sample rate of 48,000 Hz, the resulting sampled waveform would consist of an alias of the 28,000 Hz sine wave at 20,000 Hz, and an alias of the 45,000 Hz sine wave at 3,000 Hz. The sampling process has lost the original signal and created a new signal. Figure 23.59 illustrates the frequency domain spectrum of the original signal and the aliases created by sampling at too low of a rate.

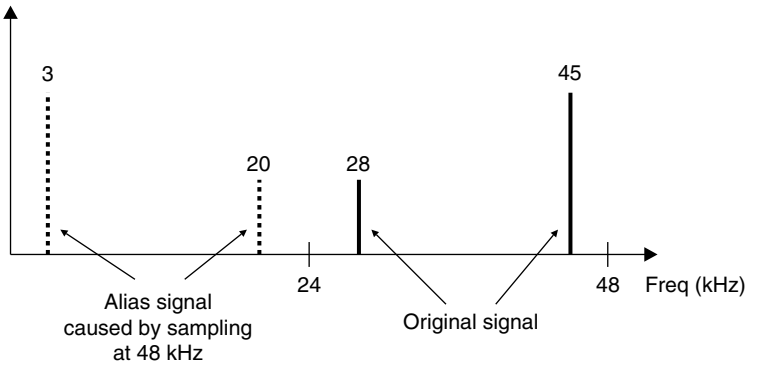



**FIGURE 23.59** Aliasing caused by sampling at too low a rate.

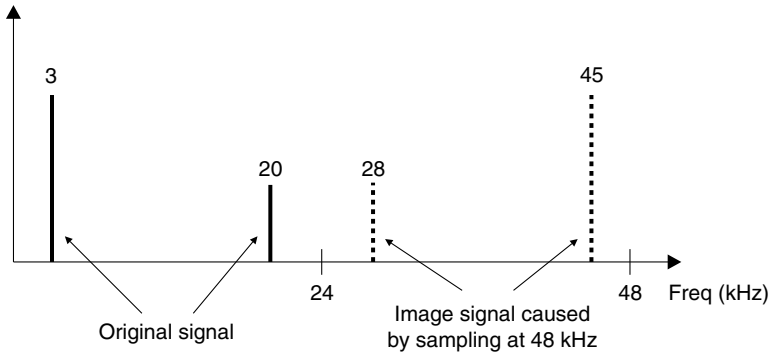



**FIGURE 23.60** Images above the Nyquist frequency.

In-band signals also create a type of alias, known as an image. Images and aliases are the converse of one another. A properly sampled sine wave at frequency $F$ has an image at $2N - F$. It also has images at $2N + F$, $4N - F$, $4N + F$, and so on up to infinity. Consider a signal consisting of two sine waves, one at 20,000 Hz, and the other at 3,000 Hz. The spectrum of this signal is identical to the one generated by sampling sine waves at 28,000 Hz and 45,000 Hz, as shown in Fig. 23.60. One cannot determine by inspection whether the sampled waveform represents true in-band signals, aliases of out-of-band signals, or some combination of the two.

The images are quite important when performing sample rate conversion. At the original sample rate, the images fold back into the passband at exactly the same frequencies of the in-band signal; however, changing the sample rate causes the images to fold back onto different frequencies in the passband, creating aliasing distortion. Figure 23.61 illustrates the effect of changing the sample rate on the images of the in-band signal. The sample rate converter must remove these images to obtain high-quality conversion.

It is easy to deceive a naïve observer by a sampled waveform. Consider the following time series:

$$0.707 \quad 0.707 \quad -0.707 \quad -0.707 \quad 0.707 \quad 0.707 \quad -0.707 \quad -0.707$$

As shown in Fig. 23.62, the waveform might appear to represent a square wave of peak magnitude 0.707 at exactly one-half the Nyquist frequency. This is incorrect. A true square wave consists of an infinite series of frequencies at $F$, $3F$, $5F$, $7F, \ldots, (2n+1)F$, where $n$ reaches infinity. However, the first partial, $3F$, is above the Nyquist frequency. Therefore, this must represent a sine wave. However, its peak magnitude is not 0.707, but is instead equal to 1.0, as shown in Fig. 23.63.

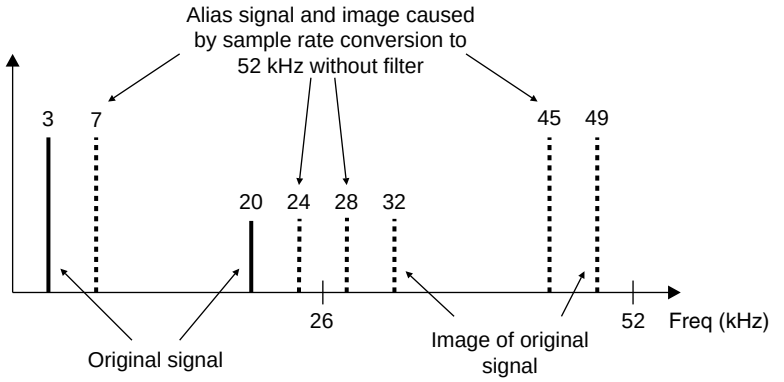



**FIGURE 23.61** Aliases and images from sample rate conversion.

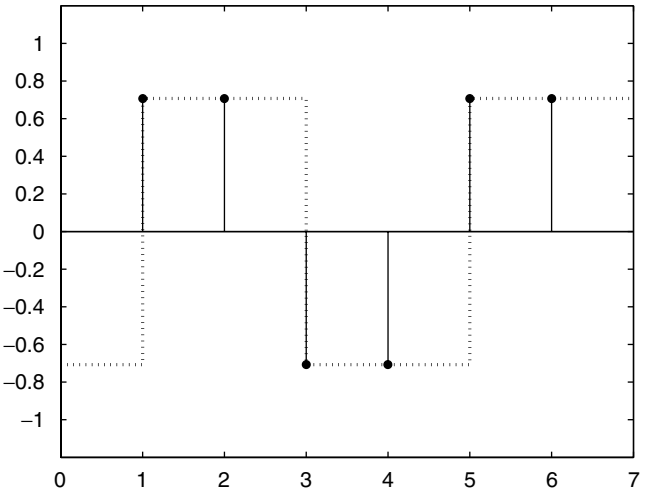



**FIGURE 23.62** Simple time series that deceptively appears to represent a square wave.

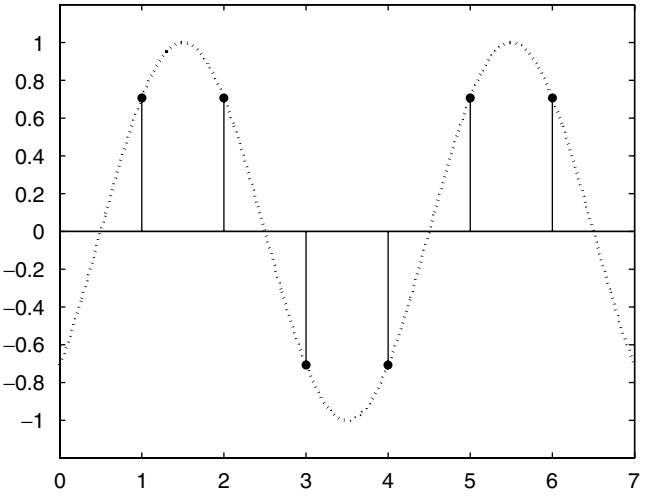



**FIGURE 23.63** True signal represented by simple time series.

The ultimate goal of sample rate conversion is to create a new time series at a new sample rate that correctly represents the true original signal. An ideal sample rate converter creates a time series that is indistinguishable from that derived by resampling the original signal. Because it requires an infinite-length time series, no real sample rate converter achieves this ideal; however, it is possible to come arbitrarily close, given enough computation.

To create a new time series at a new sample rate, it is necessary to interpolate values between the input samples. To do this, the sample rate converter maintains an output phase that represents the time index of the output samples relative to the input samples. It maintains the phase by accumulating the pitch. To double the number of output samples, use a pitch of 0.5. On each output sample period, the sample rate converter adds the pitch to the phase accumulator. If the initial value for the phase accumulator is 0.0, the resulting phase over time will be the following:

$$0.0 \quad 0.5 \quad 1.0 \quad 1.5 \quad 2.0 \quad 2.5 \quad 3.0 \quad 3.5 \quad \ldots$$

The integer portion of the phase accumulator is the address of the input samples to convert, and the fractional portion is the interpolation factor that indicates how far between input samples to generate an output sample. In the previous example phase accumulator, the fraction alternates between 0.0 and 0.5. A fraction of 0.5 indicates that the output sample should be halfway between two input samples. This definition is somewhat imprecise when using multipoint interpolation, but in a logical sense, it still applies.

Usually, a hardware implementation of a sample rate converter uses a fixed-point representation for both the pitch and the phase accumulator. The fixed-point representation enables very simple extraction of both the integer and fractional portions, and minimizes the size of the adder used to maintain the phase accumulator. The number of integer bits in the phase accumulator limits the amount of memory addressable by the sample rate converter. Integer widths of at least 24-bit are common.

The number of integer bits required for the pitch is much smaller than the number required for the phase accumulator. Upward pitch shifting, which is equivalent to conversion to a lower sample rate, requires filtering to a lower cutoff frequency than downward pitch shifting. Often, much more distortion occurs when performing upward pitch shifting. The additional filtering and distortion of upward pitch shifting place limits on its usefulness. Upward pitch shifts of more than three octaves extend into the realm of special effects. When viewed as pure sample rate conversion, a three-octave upward shift is equivalent to converting from a 48 to a 6 kHz sample rate. The lowest sample rate commonly used for audio is 8 kHz. Thus, it is usually acceptable to limit the number of integer bits in the pitch to two or three, providing a two- to three-octave upward shift capability.

The magnitude of the least-significant bit (LSB) of the pitch fraction, indicated by the number of fractional bits, determines the frequency ratio resolution. For example, the LSB of a 12-bit fraction is equal to 1/4096. The perceptual unit of measurement for pitch is cents, or 1/100 of a semitone. This is equal to a ratio of $2^{1/1200}$ or 1.00058. The just-noticeable-difference (JND) for pitch is around 8 cents, or 1.0046, indicating the acceptable frequency error [3]. Given this, it would seem that a 12-bit fraction is sufficient and even generous, since the ratio 4097/4096 is equal to 1.00024, much better than the JND of 1.0046; however, an effective method of data compression for sampled waveforms is to lower the sample rate such that the highest frequency of interest in the signal is near the Nyquist frequency of the lower sample rate. Consider a sine wave consisting of only four points. When played back at unity pitch on a system with a 48 kHz output rate, the frequency of the sine wave is 12 kHz. A more useful frequency in the human hearing range is 125 Hz. To generate this, the pitch must be 0.0104. The closest available ratio with 12-bits of fraction is 0.0105, generating a frequency ratio error of 1.0078, more than the JND. In practice, a minimum of 14-bits of fraction is required for acceptable results across a wide range of input rates and pitches. Ideally, the number of fractional bits in both the phase accumulator and the pitch should match.

#### *23.3.4.2.2 Drop-Sample Interpolation*

Drop-sample interpolation, sometimes called nearest-neighbor interpolation, is the simplest type of sample rate converter. The drop-sample interpolator simply rounds the phase accumulator to the nearest integer and chooses the input sample at the resulting integer address to be the output sample. This requires very little hardware as illustrated in Fig. 23.64. It also requires access to only a single input

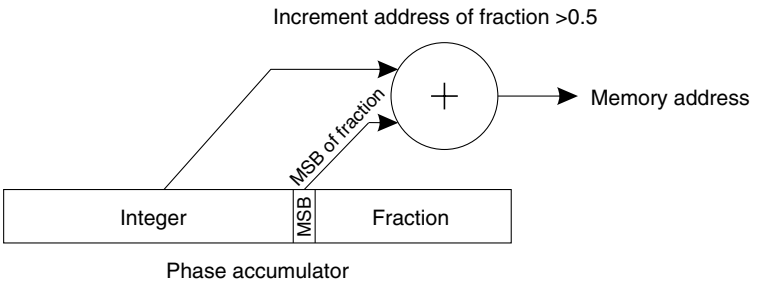



**FIGURE 23.64** Address generator for drop-sample interpolator.

sample to create an output sample, whereas all other forms of sample rate conversion require access to more than one input sample to create a single output sample, but the result can be of very poor quality.

#### *23.3.4.2.3 Linear Interpolation*

Linear interpolation may be the most common type of sample rate converter. The quality is good, and the cost is relatively low. It requires access to two input samples to create a single output sample. The computational cost is one multiply, one add, and one subtract. It is possible to implement the entire linear interpolator with a single adder, using a shift and add approach to the multiply. This is feasible as long as the clock rate is high enough to support the desired channel count and fractional accuracy. The following equation describes the linear interpolation process, where $x$ is the input waveform, $y$ is the output sample, $n$ is the integer part of the phase accumulator, and $f$ is the fractional part of the phase accumulator:

$$y = x_n + f \times (x_{n+1} - x_n)$$

One can clearly see that when the fraction is zero, the output sample is equal to the input sample at the address indicated by the phase accumulator. As the fraction approaches 1.0, the output follows a straight line drawn between adjacent input samples. Figure 23.65 illustrates the result of linearly interpolating a sine wave. The quality is quite good if the frequency of the sine wave is low relative to the Nyquist, but the quality deteriorates significantly as the frequency approaches the Nyquist. The linear interpolator has a low-pass filtering effect that becomes noticeable above one-half the Nyquist frequency. In addition, the alias rejection is not very good for the images of signals above one-half the Nyquist frequency. Thus, linear interpolation affects the quality in both the frequency response and aliasing distortion for high frequencies.

#### *23.3.4.2.4 Multipoint Interpolation*

Multipoint interpolation can produce much better quality than linear interpolation in both frequency response and aliasing distortion. The ideal interpolator has a frequency response that is perfectly flat within the passband and attenuates all other frequencies to zero. Convolving the input waveform with a sinc function that runs from negative to positive infinite time produces such a frequency response. Unfortunately, we must work within the limits of finite time to build a real interpolator. In 1984, Gossett and Smith [7] showed an efficient way to use a finite-length, windowed sinc function as a finite-impulse response (FIR) filter for sample rate conversion over a wide range of pitches. The definition of the sinc function is $\sin t/t$.

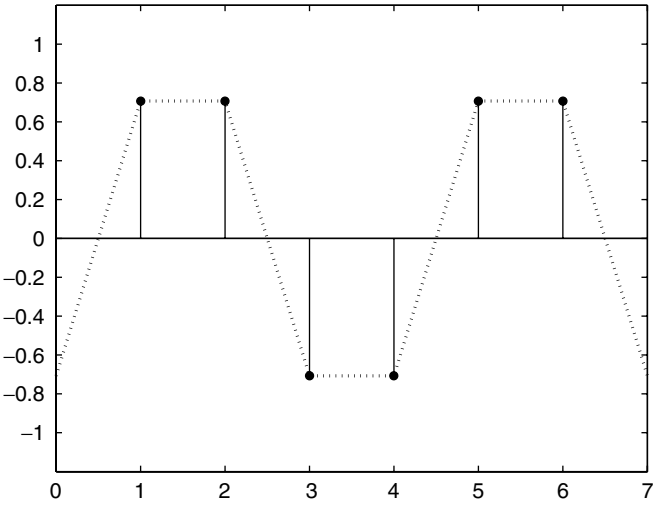



**FIGURE 23.65** Linear interpolation of simple time series.

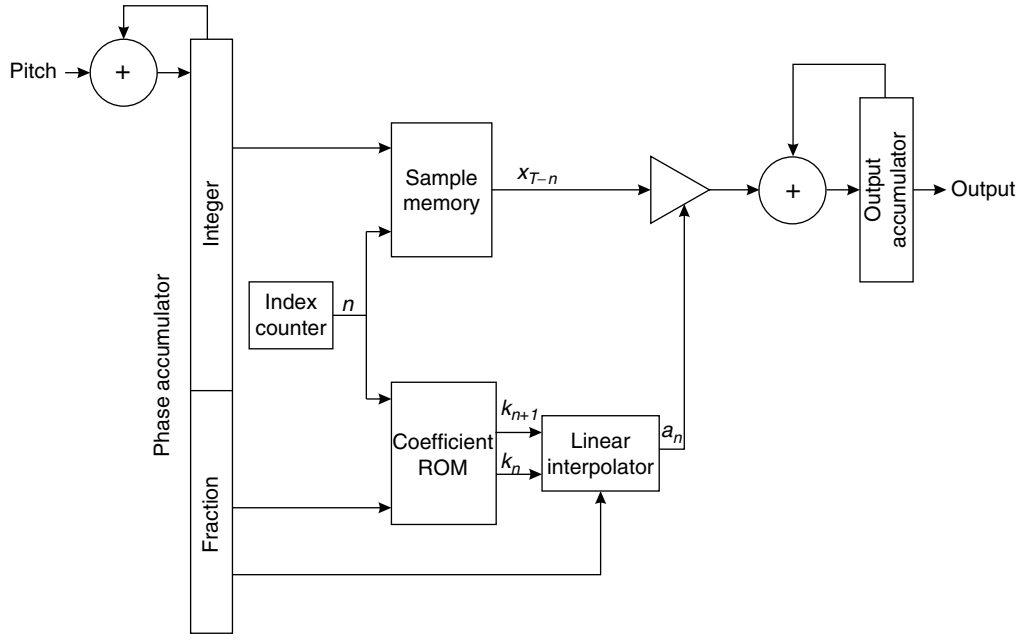



**FIGURE 23.66** Gossett–Smith interpolator block diagram.

The convolution equation is $\sum_{n=0}^{N-1} a_n x_{T-n}$, where $x$ is the input waveform and $a$ is a selected set of coefficients, possibly a windowed sinc function.

The hardware implementation of a Gossett–Smith sample rate converter consists of a read only memory (ROM) containing the filter coefficients, a linear interpolator to increase the resolution of the filter coefficient set, and a multiply–accumulate unit to perform the convolution. Figure 23.66 shows a block diagram of a typical Gossett–Smith interpolation system. Because the sinc function and other low-pass FIR filters are symmetric about their centers, it is only necessary to store half of the points in the ROM. Simple address mirroring makes the ROM appear to contain all the points.

Perceived quality is often more important than a measured quality. That is, it is more important to sound good to humans than to measure low distortion on laboratory instruments. Using a perceptual approach, one can design sample rate conversion filters that sound better [8]. For example, humans cannot hear above 20 kHz, yet a 48 kHz sample rate can represent frequencies up to 24 kHz. A filter that allows distortion within this guardband of 20 and 24 kHz can achieve better quality within the audible range of 20 Hz–20 kHz.

#### *23.3.4.2.5 Address Looping*

The phase accumulator of the sample rate converter must have some provision for looping back to a lower address. Clearly, it cannot continue to increment to infinity. The finite number of bits used to represent the integer part of the phase precludes that possibility. In addition, it is not useful simply to rely on binary wraparound of the address from its maximum value back to zero. This would imply all channels of sample rate conversion are reading from the same input waveform. At a minimum, the phase accumulator contains a loop address and loop size for each channel. When the value of the phase accumulator crosses the loop address, it loops back by the loop size and continues from the beginning of the loop. This enables both streaming audio and wavetable synthesis.

Many natural musical instrument sounds can be characterized by an attack phase, sustain phase, and release phase. The attack phase is often a primary cue to the listener as to the identity of the instrument. Usually, it consists of a rapidly changing and nonrepeating waveform. Conversely, the sustain phase is often a steady state that can be easily described by a repeating waveform. This is also true of the

release phase. During sustain and release phases, the phase accumulator can loop to create the repeating waveform, saving considerable memory. Besides, the length of the sustain phase is usually unknown because it is controlled by the length of time the musician presses the key.

When streaming audio, the software fills a circular buffer with a continuous waveform to play. The phase accumulator loops at the boundaries of the circular buffer and plays the stream. The software must be careful not to overwrite audio that the sample rate converter has not yet played.

#### 23.3.4.3 Envelopes and Modulation

It is often necessary to control various aspects of a sound, such as pitch, amplitude, and filter cutoff frequency with time-varying signals called envelopes. The audio system may use these envelopes to simulate the changes in sound that occur when a 3-D sound source moves, or as an integral part of the music synthesis process.

A typical music synthesizer envelope generator has four segments designated: attack, decay, sustain, and release (ADSR) as shown in Fig. 23.67. These four segments are a reasonable approximation of the amplitude envelopes of real musical instruments.

The first two segments are attack and decay, and are usually of a fixed duration. During these segments, the sound is changing rapidly, often containing transients and wideband noise corresponding to the initial strike of a drum or pluck of a string. The decay segment leads to the sustain segment, a variable duration, steady state corresponding to the portion of a note that is held for a length of time. The final segment, release, occurs after the musician releases the note.

Envelopes used for 3-D sound positioning do not have such a clearly defined set of segments. Instead, the 3-D positional audio is often interactive. It is not possible to predict the movements of the user in advance. For these applications, a fixed segment envelope generator may not be useful. A more useful method is to set a target value to which the hardware will smoothly ramp from the current value, enabling the software to be event driven.

In addition to the ADSR envelopes used for music synthesis and ramp-to-target envelopes used for 3-D positioning, a complete system requires a low-frequency oscillator (LFO). The system uses the LFO to create slowly modulating effects such as vibrato and tremolo.

The final control signal is usually a weighted sum of one or more ADSR envelopes and one or more LFOs. The scaling applied to the envelopes and LFOs are often time-varying signals as well. This enables, for example, vibrato to slowly increase during the sustain segment of a synthesized violin sound. It is important to note that the scaling and summation occurs in perceptual units, such as decibels and pitch cents, not physical units, such as voltage and hertz. This means that the result of the summation goes through a perceptual to physical units transform function before it is useful to the destination process. Example transform functions are $10^{x/20}$ for decibels and $2^{x/1200}$ for pitch cents.

#### 23.3.4.4 Filters

The most common filters in music synthesis are low-pass resonators. The frequency response of these filters is generally flat from 0 Hz up, with a characteristic resonance just below the low-pass cutoff frequency,

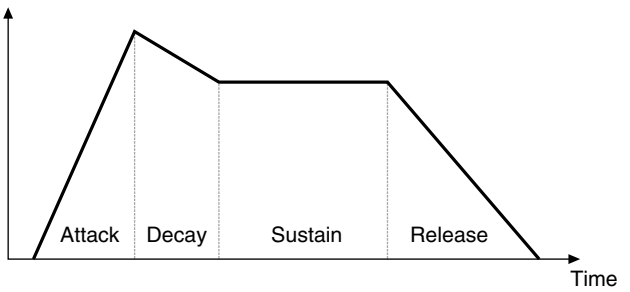



**FIGURE 23.67** ADSR envelope.

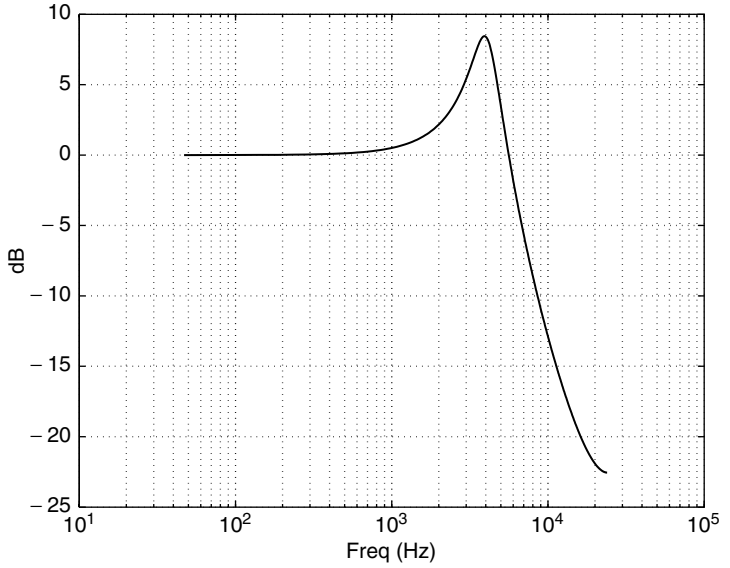



**FIGURE 23.68** Frequency response of low-pass resonant filter.

as shown in Fig. 23.68. It is most common to build these filters with an infinite impulse response (IIR) structure, such as that illustrated in Fig. 23.69. The software specifies the cutoff frequency and resonant gain of the filter. It is possible to sweep these parameters in real time, so it is important to ensure filter stability under time-varying conditions. Stability criteria are outside the scope of this discussion, but it is not difficult to drive IIR filters to instability when varying coefficients in the presence of an input signal.

Three-dimensional positional audio and environmental simulation also demands the use of filters; however, low-pass resonators are not the ideal choice. The head-related transfer function (HRTF) describes the filtering performed by the shape of the human head, earlobes, and ear canal [9]. In addition to loudness and interaural time delay, the brain uses cues provided by this filtering to determine the position of sound-emitting objects. The most common implementation of an HRTF uses a FIR structure such as the Gossett–Smith interpolator filters; however, both the impulse response and the usage of HRTF filters are much different from that of Gossett–Smith interpolator filters.

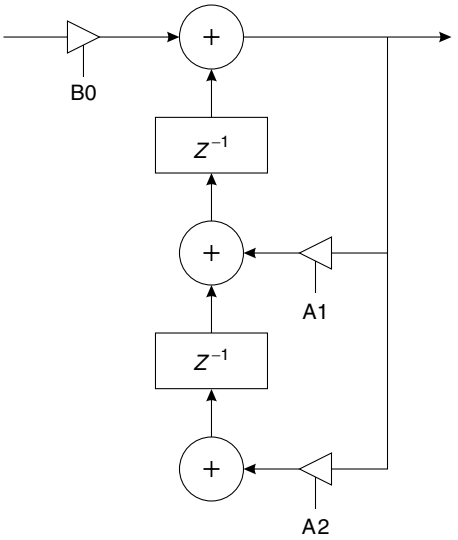



**FIGURE 23.69** Two-pole IIR filter block diagram.

Obstruction and occlusion are other filtering effects that occur when sound sources move in relation to other objects. For example, obstruction is the effect caused by an obstacle between the listener and a sound source in the same room. This applies a low-pass filtering effect on the direct sound, but not on the reverberation. In contrast, occlusion is the effect caused by a sound source located outside the same room as the listener. This applies a low-pass filtering effect on both the direct sound and the reverberation. Low-pass IIR filters are appropriate for these applications, although resonance is neither needed nor desired. The system can simulate these effects with the low-pass resonators used for music synthesis, but the typical −12 dB per octave attenuation slope is generally too steep. A gentle −3 or −6 dB per octave slope is more appropriate.

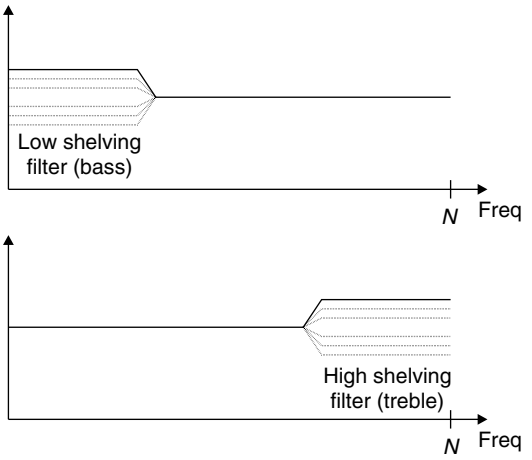



**FIGURE 23.70** Bass and treble tone control filter shapes.

The audio system of the PC also performs the role of a typical component stereo system, playing prerecorded music from CD or other sources. Thus, the software distributed with sound cards often includes equalization, such as tone controls and graphic equalizers. Simple tone controls, such as bass and treble, often use shelving filters as shown in Fig. 23.70. Graphic equalizers can use a bank of either bandpass filters or parametric equalizers into a summation matrix.

It is the designer's choice whether to implement the filter types required by the digital audio system of the PC in hardware or software. The system often includes a programmable digital signal processor (DSP), enabling a software implementation. As the required filter count increases, it often becomes more efficient to implement them in hardware, especially if multiple filter types can use the same filter structure. Moreover, while FIR and IIR filter structures are quite different, a clever designer may find opportunities to reuse the same basic arithmetic hardware for both. For example, both structures can use a multiply–accumulate arithmetic unit. If each structure requires only one-half the available bandwidth, they can share the same math unit by time-division multiplexing the inputs. Because arithmetic units are generally costly, techniques such as this can significantly reduce the system cost.

### 23.3.4.5 Effects

Music synthesis applications use effects, such as reverb, delay, and chorus as sweeteners. They are not required, but tend to make the music sound more pleasing. In contrast, 3-D positional and environmental simulation applications require delay and reverb to achieve realistic results. The fundamental unit of many digital audio effects is the digital delay line. A simple echo effect may use only one delay line, whereas a reverb effect may use 20 or more delay lines. Even modulation effects such as chorus and flange use delay lines. Digital delay lines require one memory location per sample period of delay, an address generator, and arithmetic units to scale the inputs and outputs of the delay line. Figure 23.71 illustrates a typical delay line implementing a repeating echo through the use of feedback.

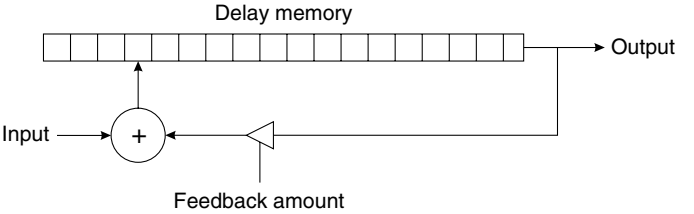



**FIGURE 23.71** Delay line with feedback implementing a repeating echo.

An obvious method of implementing a set of delay lines is to allocate memory buffers for each delay line and maintain circular address counters for each indicating the read and write locations. The delay time is equal to the difference between the read and write pointers, modulo buffer size. Maintaining circular address counters can easily use a large percentage of the total instruction bandwidth of a DSP. Many DSP implementations provide special instructions or self-maintaining address registers to reduce the load on the DSP.

To provide maximum flexibility in implementation of effects, most PC audio systems include a programmable DSP. The designer may choose to purchase an off-the-shelf DSP, either as a separate chip or as a core to integrate onto the same silicon as the rest of the solution. An alternative is to design a custom DSP to fit the particular needs of the system. The advantages of a custom design are the freedom to add or reduce features, easier integration, and often a lower cost. The disadvantages include possible increased time-to-market and the lack of standard development tools such as compilers.

#### 23.3.4.6 Digital Audio I/O

Eventually, the system must present the processed audio to the listener. A DAC outputs an analog voltage proportional to the value of a word of digital data written to it. The analog output voltage drives the input to an amplifier, and eventually the sound comes from a speaker. A DAC generally accepts serial digital data rather than parallel, so the processor must first perform a parallel to serial conversion. A DAC is usually stereo, so it accepts a time-multiplexed serial stream alternating between left and right channels. The serial protocols are usually synchronous, and come in a few varieties. The most common in use today are AC97, HD-Audio, and $I^2S$, all of which are easily available and inexpensive.

In addition to audio output, the system must be capable of recording audio from microphones and external line-level devices, such as CD players and tape decks. An ADC outputs digital data proportional to the magnitude of an analog voltage presented to its input. As in the case of a DAC, an ADC usually generates serial digital data consisting of time-multiplexed left and right channel data. The AC97 standard developed by Intel specifies a monolithic CODEC containing both a stereo DAC and a stereo ADC [10]. An AC97 implementation can sometimes be the most cost effective. The HD-Audio standard, also developed by Intel [11], provides more flexibility in the CODEC implementation. Generally, devices that use the $I^2S$ protocol are of a higher quality but do not include both a stereo ADC and a stereo DAC.

For digital transmission of audio between the computer and an external device, the S/PDIF protocol as specified in IEC-958 [12] is the connection of choice. It is a robust protocol intended for transmission over a 75 $\Omega$ coaxial cable. It uses Manchester encoding for the data, thus embedding a clock and making it insensitive to logical inversion. The ground is isolated, preventing hum and noise due to ground loops. Many consumer stereo components now have S/PDIF or its optical counterpart TOS-Link as integral connections.

#### 23.3.4.7 Emerging Applications

The basic digital audio system of a PC operates in stereo at 44.1 or 48 kHz sample rate with 16-bits of resolution. The current trend of audiophile systems is moving to a multichannel 96 kHz sample rate with 24-bits of resolution. Often, these audiophile trends trickle down to the mainstream systems as the cost comes down and demand rises. From a strict psychoacoustical viewpoint, there is little value in increasing the sample rate to 96 kHz, since the range of human hearing is generally restricted to 20 Hz–20 kHz. In addition, a 96 kHz waveform requires twice the storage and twice the computation of the equivalent waveform at 48 kHz; however, there is some benefit to processing audio at 96 kHz, primarily in the response of filters over the human hearing range. It is easy to trade channel count for the higher sample rate. A system that can process 128 channels at 48 kHz can only process 64 channels at 96 kHz. The designer should weigh the cost versus the benefit, but often the market drives the decision. If the market demands 96 kHz, the designer must deliver it.

There is, however, a very real benefit to using 24-bits of resolution. The maximum dynamic range of human hearing is around 130–140 dB. The 96 dB dynamic range of 16-bit resolution is insufficient to

cover this range. A 24-bit waveform has a dynamic range of 144 dB, more than sufficient to cover the range of human hearing. The design impact of 24-bits versus 16-bits is additional storage and larger arithmetic units, resulting in higher cost.

Likewise, there is a real benefit to multichannel audio beyond stereo. A real environment produces sounds from all directions instead of only two points as in a stereo speaker system. But the processor should still support stereo as a minimum baseline system, because the majority of audio systems are stereo. To achieve this, the processor must have more than two separate outputs and be capable of sending different audio to the multiple outputs.

Data rate reduction is another important trend in digital audio processing. The majority of the popular techniques in use, including Dolby Digital and MP3, process the signal in the frequency domain. They employ a perceptual model that attempts to determine the frequency components that are not perceivable by the human hearing system. By removing those components, the encoding process can reduce the amount of data required to represent the signal. For example, the MP3-encoding process achieves about a 10:1 compression ratio while maintaining a quality level, high enough to satisfy most people.

### 23.3.5 Conclusion

A digital audio processor for PCs consists of only a few components: a memory and host interface coupled to sample rate converters, filters, envelope generators, a mixer, and a programmable DSP. These components, as illustrated in Fig. 23.72, interact in various ways to become a variable-rate playback engine, a wavetable synthesizer, a 3-D positional audio processor, and an environmental audio simulator.

However, a digital audio system consists of both hardware and software. Given the wide range of possibilities, the designer has considerable freedom in choosing an implementation to meet particular market needs and price points. To meet the lowest price point, the designer chooses software implementations whenever possible. To achieve the highest performance, the designer chooses hardware; however, a hardware implementation may limit the flexibility of the end system. The task of partitioning the system into hardware and software components is one of the greatest hurdles to overcome. It requires a coordinated effort of strategic marketing with hardware and software engineering early in the design process. A proprietary chip is expensive to design and can take a long time from concept to production. The early involvement of strategic marketing and software engineering helps ensure the success of any new hardware design.

The customer purchases an audio system, not a chip. Thus, the designer must be aware of, yet look beyond the technical aspects of digital audio processors to create a system that provides the proper functionality at the right price.

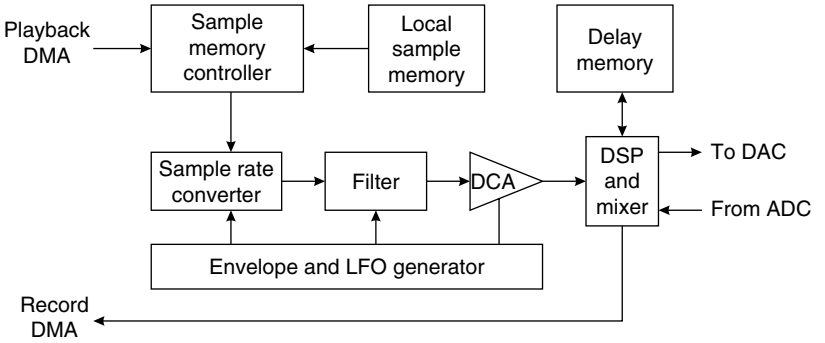



**FIGURE 23.72** Audio processor block diagram.

## References


1. Chowning, J., The synthesis of complex audio spectra by means of frequency modulation, *Journal of the Audio Engineering Society,* 21, 526, 1973.
2. The Complete MIDI 1.0 Detailed Specification, MIDI Manufacturers Association, Los Angeles, CA, 1996.
3. Rossing, T.D., *The Science of Sound,* Addison-Wesley, Reading, MA, 40, 1983.
4. *PCI Express Base Specification Revision 1.1*, PCI-SIG, Beaverton, OR, 2005.
5. Nyquist, H., Certain factors affecting telegraph speed, *Bell Systems Technical Journal,* 3, 324, 1924.
6. Shannon, C.E., A mathematical theory of communication, *Bell Systems Technical Journal,* 27, 379, 1948.
7. Smith, J.O. and Gossett, P., A flexible sampling-rate conversion method, *IEEE International Conference on Acoustics, Speech, and Signal Processing*, San Deigo, New York, 19.4.1, March 1984.
8. Rossum, D., Constraint based audio interpolators, *IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics*, Mohonk Mountain House, New Paltz, New York, 1993.
9. Zhang, M., Tan, K.C., and Er, M.H., Three-dimensional synthesis based on head-related transfer functions, *Journal of the Audio Engineering Society,* 46, 836, 1998.
10. *Audio Codec'97 Revision 2.2,* Intel Corporation, San Jose, CA, 2000.
11. *High Definition Audio Specification Revision 1.0*, Intel Corporation, San Jose, CA, 2004.
12. *IEC-958 Digital Audio Interface,* International Electrotechnical Commission, Geneva, Switzerland, 1989.


# 23.4 Modern Approximation Iterative Algorithms and Their Applications in Computer Engineering

*Sadiq M. Sait and Habib Youssef*

## 23.4.1 Introduction

This section discusses one class of combinatorial optimization algorithms: approximation iterative algorithms. We shall limit ourselves to four of these algorithms, which are, in order of their popularity among the engineering community: (1) simulated annealing (SA), (2) genetic algorithm (GA), (3) tabu search (TS), and (4) simulated evolution (SimE).

GA and SimE are evolutionary algorithms, a term used to refer to any probabilistic algorithm whose design is inspired by evolutionary mechanisms found in biological species. Evolutionary algorithms, SA and TS have been found very effective and robust in solving numerous problems from a wide range of application domains. Furthermore, they are even suitable for ill-posed problems where some of the parameters are not known beforehand. These properties are lacking in all traditional optimization techniques. The four algorithms share the following properties:

1. They are approximation algorithms, i.e., they do not guarantee finding an optimal solution. Actually, they are blind, in that they do not know when they reached an optimal solution. Therefore, they must be told when to stop.
2. They are neighborhood search algorithms, which start from one suboptimal solution (or a population of solutions) and perform a partial search of the solution space for better solutions.
3. They are all "general." They are not problem-specific and, practically, they can be tailored to solve any combinatorial optimization problem.
4. They all strive to exploit domain specific heuristic knowledge to bias the search toward "good" solution subspace. The quality of subspace searched depends to a large extent on the amount of heuristic knowledge used.
5. They are easy to implement. All that is required is to have a suitable solution representation, a cost function, and a mechanism to traverse the search space.
6. They have *hill climbing* property, i.e., they occasionally accept uphill (bad) moves.

The goal in this section is to briefly introduce these four powerful algorithms. It is organized into nine sections. In the next four subsections, an intuitive discussion of each of the four iterative algorithms is provided. The remaining sections briefly address convergence aspects of the heuristics, their parallel implementation, and examples of applications. The final subsection concludes the section with a comparison among the heuristics and a glimpse at the notion of hybrids. This section does not provide a full account of any of this important class of heuristics. For more details, readers should consult the numerous references cited in the body of this work.

## 23.4.2 Simulated Annealing

*Simulated annealing* (SA) is one of the most well-developed and widely used iterative techniques for solving optimization problems. It is a general adaptive heuristic and belongs to the class of nondeterministic algorithms [1]. It has been applied to several combinatorial optimization problems from various fields of science and engineering. The term *annealing* refers to heating a solid to a very high temperature (whereby the atoms gain enough energy to break the chemical bonds and become free to move), and then slowly cooling the molten material in a controlled manner until it crystallizes. By cooling the metal at a proper rate, atoms will have an increased chance to regain proper crystal structure with perfect lattices. During this annealing procedure, the free energy of the solid is minimized.

In the early 1980s, a correspondence between annealing and combinatorial optimization was established, first by Kirkpatrick, Gelatt and Vecchi [2] in 1983, and independently by Černy [3] in 1985. These scientists observed that a solution in combinatorial optimization is equivalent to a state in the physical system and the cost of the solution is analogous to the energy of that state. As a result of this analogy, they introduced a solution method in the field of combinatorial optimization. This method is thus based on the simulation of the physical annealing process, and hence the name *simulated annealing* [2,3].

Every combinatorial optimization problem may be discussed in terms of a *state space*. A *state* is simply a configuration of the combinatorial objects involved. For example, consider the problem of partitioning a graph of $2n$ nodes into two equal sized subgraphs such that the number of edges with vertices in both subgraphs is minimized. In this problem, any division of $2n$ nodes into two equal sized blocks is a configuration. A large number of such configurations exists. Only some of these correspond to global optima, i.e., states with optimum cost.

An iterative improvement scheme starts with some given state, and examines a *local neighborhood* of the state for better solutions. A local neighborhood of a state $S$, denoted by $N(S)$, is the set of all states which can be reached from $S$ by making a small change to $S$. For instance, if $S$ represents a two-way partition of a graph, the set of all partitions which are generated by swapping two nodes across the partition represents a local neighborhood. The iterative improvement algorithm moves from the current state to a state in the local neighborhood if the latter has a better cost. If all the local neighbors have larger costs, the algorithm is said to have *converged* to a *local optimum*. This is illustrated in Fig. 23.73. Here, the states are shown along the $x$-axis, and it is assumed that two consecutive states are local neighbors. It is further assumed that we are discussing a *minimization* problem. The cost curve is *nonconvex*, i.e., it has multiple minima. A greedy iterative improvement algorithm may start off with an initial solution such as **S** in Fig. 23.73, then slide along the curve and find a local minimum such as **L**. There is no way such an algorithm can find the global minimum **G** of Fig. 23.73, unless it "climbs the hill" at the local minimum L. In other words, an algorithm that occasionally accepts inferior solutions can escape from getting trapped in a local optimum. SA is such a hill-climbing algorithm.

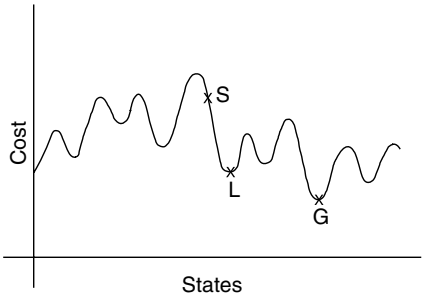



**FIGURE 23.73** Local vs. global optima.

During annealing, a metal is maintained at a certain temperature $T$ for a pre-computed amount of time, before

```
Algorithm Simulated_annealing(S0.T0.α.β.M. Maxtime):
  (*S0 is the initial solution *)
  (*BestS is the best solution *)
  (*T0 is the initial temperature *)
  (*α is the cooling rate *)
  (*β a constant *)
  (*Maxtime is the total allowed time for the annealing process *)
  (*M represents the time until the next parameter update *)
Begin
  T = T0:
  CurS = S0:
  Best S = CurS:/* Best S is the best solution seen so far */
  CurCost = Cost(CurS):
  BestCost = Cost(BestS):
  Time = 0:
  Repeat
    Call Metropolis(CurS. CurCost. BestS, BestCost. T. M):
    Time = Time + M:
    T = αT:
    M = βM
  Until (Time ≥ MaxTime):
  Return (BestS)
End. (* of Simulated_annealing*)
```

**FIGURE 23.74** Procedure for simulated annealing algorithm.

reducing the temperature in a controlled manner. The atoms have a greater degree of freedom to move at higher temperatures than at lower temperatures. *The movement of atoms is analogous to the generation of new neighborhood states in an optimization process.* In order to simulate the annealing process, much flexibility is allowed in neighborhood generation at higher "temperature," i.e., many "uphill" moves are permitted at higher temperatures. The temperature parameter is lowered gradually as the algorithm proceeds. As the temperature is lowered, fewer and fewer uphill moves are permitted. In fact, at absolute zero, the SA algorithm turns greedy, allowing only downhill moves.

The SA algorithm is shown in Fig. 23.74. The core of the algorithm is the *Metropolis* procedure, which simulates the annealing process at a given temperature $T$ (Fig. 23.75) [4]. The *Metropolis* procedure

```
Algorithm Metropolis (CurS. CurCost. BestS. BestCost. T. M):
Begin
  Repeat
    NewS = Neighbor(CurS): /* Return a neighbor from aleph(CurS) */
    NewCost = Cost(NewS):
    ΔCost = (NewCost–CurCost):
    If (Δ Cost < 0) Then
      CurS = NewS;
        If NewCost < BestCost Then
          BestS = NewS
        EndIf
    Else
      If (RANDOM < e^(−ΔCost/T)) Then
        CurS = NewS:
      EndIf
    EndIf
    M = M − 1
  Until (M = 0)
End. (* of Metropolis*)
```

**FIGURE 23.75** The Metropolis procedure.

receives as input the current temperature *T*, and the current solution *CurS*, which it improves through local search. Finally, *Metropolis* must also be provided with the value *M*, which is the amount of time for which annealing must be applied at temperature *T*. The procedure *Simulated_annealing* simply invokes *Metropolis* at decreasing temperatures. Temperature is initialized to a value $T_0$ at the beginning of the procedure and is reduced in a controlled manner (typically in a geometric progression); the parameter $\alpha$ is used to achieve this cooling. The amount of time spent in annealing at a temperature is gradually increased as temperature is lowered. This is done using the parameter $\beta > 1$. The variable *Time* keeps track of the time being expended in each call to the *Metropolis*. The annealing procedure halts when *Time* exceeds the allowed time.

The *Metropolis* procedure is shown in Fig. 23.75. It uses the procedure *Neighbor* to generate a local neighbor *NewS* of any given solution *S*. The function *Cost* returns the cost of a given solution *S*. If the cost of the new solution *NewS* is better than the cost of the current solution *CurS*, then the new solution is accepted, and we do so by setting *CurS* = *NewS*. If the cost of the new solution is better than the best solution (*BestS*) seen thus far, then *BestS* must be replaced by *NewS*. If the new solution has a higher cost in comparison to the original solution *CurS*, *Metropolis* will accept the new solution on a *probabilistic* basis. A random number is generated in the range 0 to 1. If this random number is smaller than $e^{-\Delta Cost/T}$, where $\Delta Cost$ is the change in costs, ($\Delta Cost = Cost(NewS) - Cost(CurS)$), and $T$ is the current temperature, the uphill solution is accepted. This criterion for accepting the new solution is known as the *Metropolis criterion*. The *Metropolis* procedure generates and examines *M* solutions.

The probability that an inferior solution is accepted by the *Metropolis* is given by $P(RANDOM < e^{-\Delta Cost/T})$. The random number generation is assumed to follow a *uniform distribution*. Remember that $\Delta Cost > 0$ because it is assumed that *NewS* is uphill from *CurS*. At very high temperatures (when $T \rightarrow \infty$), $e^{-\Delta Cost/T} \approx 1$, and, hence, the above probability approaches 1. When $T \rightarrow 0$, the probability $e^{-\Delta Cost/T}$ falls to 0.

In order to implement SA, a suitable cost function needs to be formulated for the problem being solved. In addition, as in the case of local search techniques, the existence of a neighborhood structure is assumed, and the *perturb* operation or *Neighbor* function needs to generate new states (neighborhood states) from current states. And finally, a control parameter is needed to play the role of temperature and a random number generator. The actions of SA are best illustrated with the help of an example. For the solution of the two-way partitioning problem using SA, please refer to [5].

A quality SA implementation requires the careful setting of a set of parameters that govern the convergence of the algorithm, namely (a) the initial value of temperature, (b) the number of iterations of the inner loop, (c) the rate of temperature decrease, and (d) the number of global iterations (the stopping criterion or the final value of temperature). This set of parameters is commonly referred as the "cooling schedule" [2,6,7]. It is customary to determine the cooling schedule by trial and error. However, some researchers have proposed cooling schedules that rely on some mathematical rigor. For a discussion on cooling schedule, and SA requirements the reader is referred to [8].

### 23.4.3 Genetic Algorithms

Genetic algorithm (GA), is a powerful, domain-independent search technique that was inspired by Darwinian theory. It emulates the natural process of evolution to perform an efficient and systematic search of the solution space to progress toward the optimum. It is an *adaptive* learning heuristic that is based on the theory of *natural selection* that assumes that individuals with certain characteristics are more able to survive, and hence pass their characteristics to their offspring. Several variations of the basic algorithm (modified to adapt to the problem at hand) exist. Subsequently, this set will be referred to as genetic algorithms (in plural).

GAs were invented by John Holland and his colleagues [9] in the early 1970s. Holland incorporated features of natural evolution to propose a *robust*, computationally simple, and yet powerful technique for solving difficult optimization problems. The structure that encodes how the organism is to be constructed is called a *chromosome*. One or more chromosomes may be associated with each member of the

population. The complete set of chromosomes is called a *genotype* and the resulting organism is called a *phenotype*. Similarly, the representation of a solution to the optimization problem in the form of an encoded string is termed as a *chromosome*. In most combinatorial optimization problems a single chromosome is generally sufficient to represent a solution, i.e., the genotype and the chromosome are the same. The symbols that make up a chromosome are known as *genes*. The different values a gene can take are called *alleles*.

The fitness value of an individual (genotype or a chromosome) is a *positive* number that is a measure of its goodness. When the chromosome represents a solution to the combinatorial optimization problem, the fitness value indicates the cost of the solution. In the case of a minimization problem, solutions with lower cost correspond to individuals that are more fit.

GAs operate on a *population* (or set) of *individuals* (or solutions) encoded as strings. These strings represent points in the search space. In each iteration, referred to as a generation, a new set of strings that represent solutions (called offspring) is created by crossing some of the strings of the current generation [10]. Occasionally, new characteristics are injected to add diversity. GAs combine information exchange along with *survival of the fittest* among individuals to conduct the search.

When employing GAs to solve a combinatorial optimization problem one has to find an efficient representation of the solution in the form of a chromosome. Associated with each chromosome is its *fitness value*. If we simulate the process of natural reproduction, combined with the biological principle of survival of the fittest, then, as each generation progresses, better and better individuals (solutions) with higher fitness values are expected to be produced.

Because GAs work on a population of solutions, an initial population constructor is required to generate a certain predefined number of solutions. The quality of the final solution produced by a genetic algorithm depends on the size of the population and how the initial population is constructed. The initial population generally comprises random solutions.

The population of chromosomes evolves from generation to the next through the use of two types of genetic operators: (1) unary operators such as mutation and *inversion,* which alter the genetic structure of a single chromosome, and (2) higher order operator, referred to as *crossover,* which consists of obtaining new individual by combining genetic material from two selected parent chromosomes. The resulting individuals produced when genetic operators are applied on the parents are termed as *offspring*. Then the new population is selected out of the individuals of the current population and its offspring.

The choice of parents for crossover from the set of individuals that comprise the population is probabilistic. In keeping with the ideas of natural selection, we assume that stronger individuals, i.e., those with higher fitness values, are more likely to mate than the weaker ones. One way to simulate this is to select parents with a probability that is directly proportional to their fitness values. Larger the fitness, the greater is chance of an individual being selected as one of the parents for crossover [10].

Several crossover operators have been proposed in the literature. Depending on the combinatorial optimization problem being solved some are more effective than others. One popular crossover that will also help illustrate the concept is the *simple crossover*. It performs the "cut-catenate" operation. It consists of choosing a *random* cut point and dividing each of the two chromosomes into two parts. The offspring is then generated by catenating the segment of one parent to the left of the cut point with the segment of the second parent to the right of the cut point.

*Mutation* ($\mu$) produces incremental random changes in the offspring by randomly changing allele values of some genes. In case of binary chromosomes it corresponds to changing single bit positions. It is not applied to all members of the population, but is applied probabilistically only to some. Mutation has the effect of perturbing a certain chromosome in order to introduce *new* characteristics not present in any element of the parent population. For example, in case of binary chromosomes, toggling some selected bits produces the desired effect.

*Inversion* is the third operator of GA and like mutation it also operates on a single chromosome. Its basic function is to laterally invert the order of alleles between two randomly chosen points on a chromosome.

```
Procedure (Genetic_Algorithm)
  M = Population size.                (*# Of possible solutions at any instance.*)
  N0 = Number of generations.         (*# Of iterations.*)
  N0 = Number of offsprings.          (*To be generated by crossover.*)
  Pp = Mutation probability.          (*Also called mutation rate Mx*)
  P ← Ξ(M)                            (*Construct initial population P.
                                      Ξ is population constructor.*)
  For j = 1 to M                      (*Evaluate fitnesses of all individuals.*)
    Evaluate f(P[j])                  (*Evaluate fitnesses of P.*)
  EndFor
  For i = 1 to N0
    For j = 1 to N0
(x,y) ← φ(P)                          (*Select two parents x and y from current population.*)
  offspring[j] ← χ (x,y)              (*Generate offsprings by crossover of parents x and y.*)
  Evaluate f(offspring[j])            (*Evaluate fitness of each offsprings.*)
  EndFor
  For j = 1 to N0                     (*With probability Pp apply mutation.*)
  mutated[j] ← μ(y)
    Evaluate f(mutated[j])
  EndFor
P ← Select(P, offsprings)             (*Select best M solutions from parents & offsprings.*)
  EndFor
  Return highest scoring configuration in P.
End
```

**FIGURE 23.76** Structure of a simple genetic algorithm.

A *generation* is an iteration of GA where individuals in the current population are selected for crossover and offspring are created. Due to the addition of offspring, the size of population increases. In order to keep the number of members in a population fixed, a constant number of individuals are selected from this set that consists of both the individuals of the initial population, and the generated offspring. If **M** is the size of the initial population and $N_0$ is the number of offspring created in each generation, then, before the beginning of next generation, **M** new parents from $\mathbf{M} + N_0$ individuals are selected. A greedy selection mechanism is to choose the best **M** individuals from the total of $\mathbf{M} + N_0$. The complete pseudo code of a simple GA is given in Fig. 23.76.

### 23.4.4 Tabu Search

The previous subsection discussed simulated annealing, which was inspired by the cooling of metals, and genetic algorithms, which imitate the biological phenomena of evolutionary reproduction. In this section, we present a more recent optimization method called tabu search (TS), which is based on selected concepts of artificial intelligence (AI).

Tabu search was introduced by Fred Glover [11–14] as a general iterative heuristic for solving combinatorial optimization problems. Initial ideas of the technique were also proposed by Hansen [15] in his *steepest ascent mildest descent* heuristic. TS is conceptually simple and elegant. It is a form of local neighborhood search. Each solution $S \in \Omega$ has an associated set of neighbors $N(S) \subseteq \Omega$. A solution $S' \in N(S)$ can be reached from $S$ by an operation called a *move* to $S'$. Normally, the neighborhood relation is assumed symmetric. That is, if $S'$ is a neighbor of $S$ then $S$ is a neighbor of $S'$. At each step, the local neighborhood of the current solution is explored and the best solution in that neighborhood is selected as the new current solution. Unlike local search that stops when no improved new solution is found in the current neighborhood, tabu search continues the search from the best solution in the neighborhood even if it is worse than the current solution. To prevent cycling, information pertaining to the most recently visited solutions are inserted in a list called *tabu list*. Moves to tabu solutions are not allowed. The tabu status of a solution is overridden when certain criteria (aspiration criteria) are satisfied. One example of an aspiration criterion is when the cost of the selected solution is better

than the best seen so far, which is an indication that the search is actually not cycling back, but rather moving to a new solution not encountered before.

Tabu search is a *metaheuristic,* which can be used not only to guide the search in complex solution spaces, but also to direct the operations of other heuristic procedures. It can be superimposed on any heuristic whose operations are characterized as performing a sequence of *moves* that lead the procedure from one trial solution to another. In addition to several other characteristics, the attractiveness of tabu search comes from its ability to escape local optima.

Tabu search differs from SA or GA, which are *memoryless,* and also from branch-and-bound, A* search, etc., which are rigid memory approaches. One of its features is its systematic use of *adaptive* (flexible) memory. It is based on very simple ideas with a clever combination of components, namely [16,17]:

1. A short-term memory component; this component is the core of the tabu search algorithm.
2. An intermediate-term memory component; this component is used for regionally **intensifying** the search.
3. A long-term memory component; this component is used for globally **diversifying** the search.

The central idea underlying tabu search is the exploitation of the above three memory components. Using the short-term memory, a *selective history* **H** of the states encountered is maintained to guide the search process. Neighborhood $N(S)$ is replaced by a modified neighborhood, which is a function of the history **H,** and is denoted by $N(\mathbf{H}, S)$. History determines which solutions may be reached by a move from $S$, since the next state $S'$ is selected from $N(\mathbf{H}, S)$. The short-term memory component is implemented through a set of *tabu* conditions and the associated *aspiration criterion.*

The major idea of the short-term memory component is to classify certain search directions as tabu (or *forbidden*). By doing so we avoid returning to previously visited solutions. Search is therefore forced away from recently visited solutions, with the help of a short-term memory (*tabu list* **T**). This memory contains *attributes* of some $k$ most recent moves. The size of the tabu list denoted by $k$ is the number of iterations for which a move containing that attribute is forbidden after it has been made. The tabu list can be visualized as a window on accepted moves as shown in Fig. 23.77. Moves that tend to undo previous moves within this window are forbidden. A flow chart illustrating the basic short-term memory tabu search algorithm is given in Fig. 23.78. An algorithmic description of a simple implementation of the tabu search is given in Fig. 23.79.

Intermediate-term and long-term memory processes are used to intensify and diversify the search, respectively, and have been found to be very effective in increasing both quality and efficiency [18,19].

The basic tabu search algorithm based on the short-term memory component is discussed first. Following this is a discussion on uses of intermediate and long-term memories.

Referring to Fig. 23.79, initially the current solution is the best solution. Copies of the current solution are perturbed with moves to get a set of new solutions. The best among these is selected and if it is not tabu then it becomes the current solution. If the move is tabu, its aspiration criterion is checked. If it passes the aspiration criterion, it becomes the current solution. If the move to the next solution is accepted, the move or some of its attributes are stored in the tabu list. Otherwise, moves are regenerated to get another set of new solutions. If the current solution is better than the best seen thus far, the best solution is updated. Whenever a move is accepted the iteration number is incremented. The procedure continues for a fixed number of iterations, or until some pre-specified stopping criterion is satisfied.

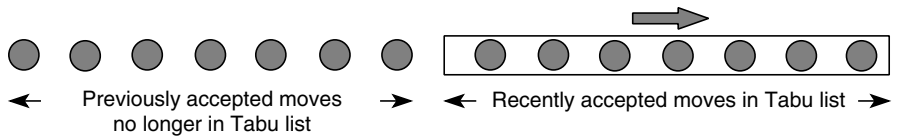



**FIGURE 23.77** The tabu list can be visualized as a window over accepted moves.

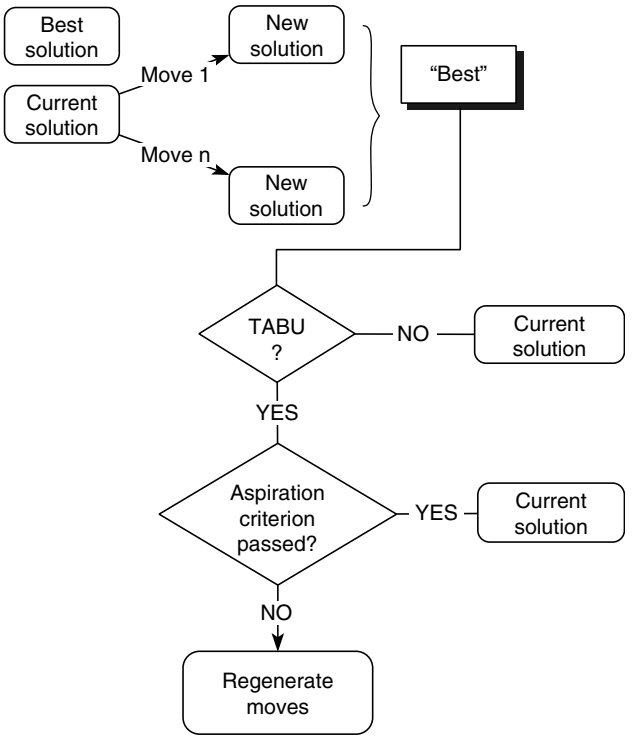



**FIGURE 23.78** Flow chart of the tabu search algorithm.

$\Omega$ : Set of feasible solutions.
$S$ : Current solution.
$S^*$ : Best admissible solution.
*Cost* : Objective function.
$\aleph(S)$ : Neighborhood of $S \in \Omega$.
**V***: Sample of neighborhood solutions.
**T** : Tabu list.
**AL**: Aspiration Level.

**Begin**
1. Start with an initial feasible solution $S \in \Omega$.
2. Initialize tabu lists and aspiration level.
3. **For** fixed number of iterations **Do**
4. Generate neighbor solutions $V^* \subset \aleph(S)$.
5. Find best $S^* \in \mathbf{V}^*$.
6. **If** move $S$ to $S^*$ is not in **T Then**
7. Accept move and update best solution.
8. Update tabu list and aspiration level.
9. Increment iteration number.
10. **Else**
11. **If** $Cost(S^*) <$ **AL Then**
12. Accept move and update best solution.
13. Update tabu list and aspiration level.
14. Increment iteration number.
15. **EndIf**
16. **EndIf**
17. **EndFor**

**End**.

**FIGURE 23.79** Algorithmic description of short-term tabu search (TS).

Tabu restrictions and aspiration criterion have a symmetric role. The order of checking for tabu status and aspiration criterion may be reversed, though most applications check if a move is tabu before checking for aspiration criterion. For more discussion on move attributes, types of tabu lists and the various tabu restrictions, the data structure to handle tabu-lists, and other aspiration criteria, the reader is referred to [8].

In many applications, the short-term memory component by itself has produced solutions superior to those found by alternative procedures, and usually the use of intermediate-term and long-term memory is bypassed; however, several studies have shown that intermediate and long-term memory components can improve solution quality and/or performance [19–22].

The basic role of the intermediate-term memory component is to *intensify* the search. By its incorporation, the search becomes more aggressive. As the name suggests, memory is used to intensify the search. Intermediate-term memory component operates as follows. A selected number $m \gg |\mathbf{T}|$ (recall that $|\mathbf{T}|$ is the size of tabu list) of best trial solutions generated during a particular period of search are chosen and their features are recorded and compared. These solutions may be $m$ consecutive best ones, or $m$ local optimal solutions reached during the search. Features common to most of these are then taken and new solutions that contain these features are sought. One way to accomplish this is to restrict/penalize moves that remove such attributes. For example, in the TSP problem with moderately dense graphs, the number of different edges that can be included into any tour is generally a fraction of the total available edges (Why?). After some number of initial iterations, the method can discard all edges not yet incorporated into some tour. The size of the problem and the time per iteration now become smaller. The search therefore can focus on possibilities that are likely to be attractive, and can also examine many more alternatives in a given span of time.

The goal of long-term memory component is to *diversify* the search. The principles involved here are just the *opposite* of those used by the intermediate-term memory function. Instead of more intensively focusing the search with regions that contain previously found good solutions, the function of this component is to drive the search process into new regions that are different from those examined thus far.

Diversification using long-term memory in tabu search can be accomplished by creating an evaluator whose task is to take the search to new starting points [11]. For example, in the traveling salesman problem (TSP), a simple form of long-term memory is to keep a count of the number of times each edge has appeared in the tours previously generated. Then, an evaluator can be used to penalize each edge on the basis of this count; thereby favoring the generation of *other, hopefully good* starting tours that tend to avoid those edges most commonly used in the past. This sort of approach is viewed as a **frequency**-based tabu criterion in contrast to the **recency**-based (tabu list) discussed earlier. Such a long-term strategy can be employed by means of a long-term tabu list (or any other appropriate data structure) that is periodically activated to employ tabu conditions of increased stringency, thereby forcing the search process into new territory [23].

It is easy to create and test the short-term memory component first, and then incorporate the intermediate/long components for additional refinements.

Let a matrix entry *Freq(i,j)* ($i$ and $j$ be movable or swappable elements) store the number of times swap $(i,j)$ was made to take the solution from current state $S$ to a new state $S^*$. We can then use this information to define a move evaluator $\varepsilon(\mathbf{H}, S)$, which is a function of both the cost of the solution, and the frequency of the swaps stored. Our objective is to diversify the search by giving more consideration to those swaps that have not been made yet, and to penalize those that frequently occurred, that is giving them less consideration [24]. Taking the above into consideration, the evaluation of the move can be expressed as follows:

$$\varepsilon(H,S^*) = \begin{cases} Cost(S^*) & Cost(S^*) \leq Cost(S) \\ Cost(S^*) + \alpha \times Freq(i,j) & Cost(S^*) > Cost(S) \end{cases}$$

$\alpha$ is a constant which depends on the range of the objective function values, the number of iterations, the span of history considered, etc. Its value ($\alpha'$s) is such that cost and frequency are appropriately balanced.

### 23.4.5 Simulated Evolution (SimE)

The simulated evolution algorithm (SimE) is a general search strategy for solving a variety of combinatorial optimization problems. The first paper describing SimE appeared in 1987 [25]. Other papers by the same authors followed [26–28].

SimE assumes that there exists a population $P$ of a set $M$ of $n$ (movable) elements. In addition, there is a cost function *Cost* that is used to associate with each assignment of movable element $m$ a cost $C_m$. The cost $C_m$ is used to compute the goodness (fitness) $g_m$ of element $m$, for each $m \in M$. Furthermore, Usually additional constraints must be satisfied by the population as a whole or by particular elements. A general outline of the SimE algorithm is given in Fig. 23.80.

SimE algorithm proceeds as follows. Initially, a population* is created at random from all populations satisfying the environmental constraints of the problem. The algorithm has one main loop consisting of three basic steps, *evaluation, selection*, and *allocation*. The three steps are executed in sequence until the population average *goodness* reaches a maximum value, or no noticeable improvement to the population *goodness* is observed after a number of iterations. Another possible stopping criterion could be to run the algorithm for a prefixed number of iterations (see Fig. 23.80). Some details of the steps of the SimE algorithm are presented in the next subsection.

#### 23.4.5.1 Evaluation

The *evaluation* step consists of evaluating the goodness of each individual $i$ of the population $P$. The *goodness* measure must be a single number expressible in the range [0,1]. *Goodness* is defined as follows:

$$g_i = \frac{O_i}{C_i} \tag{23.1}$$

where

$O_i$ is an estimate of the optimal cost of individual $i$

$C_i$ is the actual cost of $i$ in its current location

```
ALGORITHM Simulated_Evolution(M, L):
/* M: Set of movable elements; */
/* L: Set of locations: */
/* B: Selection bias: */
/* Stopping criteria and selection bias can be automatically adjusted; */
INITIALIZATION:
Repeat
  EVALUATION:
    ForEach m ∈ M Do g_m = O_m / C_m EndForEach:
  SELECTION:
    ForEach m ∈ M Do
      If Selection (m, B) Then P_s = P_s ∪ {m}
        Else P_r = P_r ∪ {m}
      EndIf;
    EndForEach;
  Sort the elements of P_s;
  ALLOCATION:
    ForEach m ∈ P_s Do Allocation(m) EndForEach:
Until Stopping-criteria are met;
Return (Best Solution);
End Simulated_Evolution.
```

**FIGURE 23.80** Simulated evolution algorithm.

*In SimE terminology, a population refers to a single solution. Individuals of the population are components of the solution; they are the movable elements.

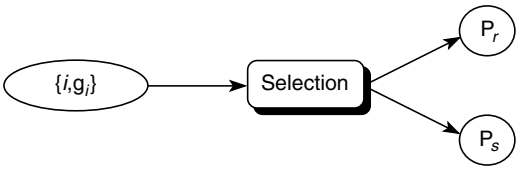



**FIGURE 23.81** Selection.

Equation 23.1 assumes a minimization problem (maximization of goodness). Notice that, according to the previous definition, the $O_i$'s **do not** change from generation to generation, and, therefore, are computed only once during the initialization step. Hence, only the $C_i$'s have to be recomputed at each call to the *Evaluation* function. Empirical evidence [29] shows that the accuracy of the estimation of $O_i$ is not very crucial to the successful application of SimE; however, the *goodness* measure must be strongly related to the target objective of the given problem.

### 23.4.5.2 Selection

The second step of the SimE algorithm is *selection*. Selection takes as input the population $P$ together with the estimated *goodness* of each individual and partitions $P$ into two disjoint sets, a selection set $P_s$ and a set $P_r$ of the remaining members of the population (see Fig. 23.81). Each member of the population is considered separately from all other individuals. The decision whether to assign individual $i$ to the set $P_s$ or set $P_r$ is based solely on its *goodness* $g_i$. The operator uses a selection function *Selection*, which takes as input $g_i$ and a parameter $B$, which is a *selection bias*. Values of $B$ are recommended to be in the range $[-0.2, 0.2]$. In many cases a value of $B = 0$ would be a reasonable choice.

The *selection* function returns *true* or *false*. The higher the *goodness* value of the element, the higher is its chance of staying in its current location, i.e., unaltered in the next generation. On the other hand, the lower the *goodness* value, the more likely the corresponding element will be selected for alteration (mutation) in the next generation (will be assigned to the selection set $P_s$). An individual with a high *fitness* (*goodness* close to one) still has a nonzero probability of being assigned to the selected set $P_s$. It is this element of nondeterminism that gives SimE the capability of escaping local minima.

For most problems, it is always beneficial to alter the elements of the population according to a deterministic order that is correlated with the objective function being optimized. Hence, in SimE, prior to the *allocation* step, the elements in the selection set $P_s$ are sorted. The sorting criterion is problem specific. Usually there are several criteria to choose from [8].

### 23.4.5.3 Allocation

*Allocation* is the SimE operator that has most impact on the quality of solution. *Allocation* takes as input the two sets $P_s$ and $P_r$ and generates a new population $P'$ that contains all the members of the previous population $P$, with the elements of $P_s$ mutated according to an *Allocation* function (see Fig. 23.82).

The choice of a suitable *allocation* function is problem specific. The decision of the *Allocation* strategy usually requires more ingenuity on the part of the designer than the *selection* scheme. The *allocation* function may be a nondeterministic function, which involves a choice among a number of possible mutations (moves) for each element of $P_s$. Usually, a number of *trial-mutations* are performed and rated with respect to their *goodnesses*. Based on the resulting goodnesses, a final configuration of the population $P'$ is decided. The goal of allocation is to favor improvements over the previous generation, without being too greedy.

The *allocation* operation is a complex form of genetic *mutation* that is one of the genetic operations thought to be responsible for the evolution of the various species in biological environments; however, there is no need for a *crossover* operation as in GA since only one parent is maintained in all generations; however, because *mutation* is the only mechanism used by SimE for inheritance and evolution, it must be more sophisticated than the one used in GA.

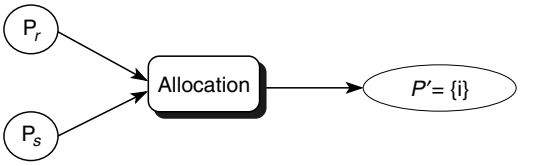



**FIGURE 23.82** Allocation.

*Allocation* alters (mutates) all the elements in the selected set $P_s$ one after the other in a predetermined order. For each individual $e_i$ of the selected set $P_s$, $W$ distinct trial alterations are attempted. The trial that leads to the best configuration (population) with respect to the objective being optimized is accepted and made permanent. The *goodness* of each individual element is also tightly coupled with the target objective, so the superior alterations are supposed to gradually improve the individual *goodnesses* as well. Hence, *allocation* allows the search to progressively evolve toward an optimal configuration where each individual is optimally located.

#### 23.4.5.4 Initialization Phase

This step precedes the iterative phase. In this step, the various parameters of the algorithm are set to their desired values, namely, the maximum number of iterations required to run the main loop, the selection bias $B$, and the number of trial alterations $W$ per individual. Furthermore, similar to any iterative algorithm, SimE requires that an initial solution be given. The convergence aspects of SimE are not affected by the quality of the initial solution; however, starting from a randomly generated solution usually increases the number of iterations required to converge to a near-optimal solution.

### 23.4.6 Convergence Aspects

One of the desirable properties that a stochastic iterative algorithm should possess is the convergence property, i.e., the guarantee of converging to one of the global optima if given enough time. The convergence aspects of the simulated annealing algorithm have been the subject of extensive studies. For a thorough discussion of simulated annealing convergence we refer the reader to [6,7,30].

For convergence properties of the GA heuristic based on Markovian analysis, the reader is referred to [31–37]. Fogel [38] provides a concise treatment of the main GA convergence results.

The tabu search algorithm as described in this article is known as *ordinary* or *deterministic tabu search*. Because of its deterministic nature, ordinary tabu search may never converge to a global optimum state. The incorporation of a nondeterministic element within tabu search allows the algorithm to lend itself to mathematical analysis similar to that developed for simulated annealing, making it possible to establish corresponding convergence properties. Tabu search with nondeterministic elements is called *probabilistic tabu search* [11,39]. Probabilistic tabu search has been shown to converge in the limit to a global optimum state. The proof is analogous to that of SA.

Proof of convergence of SimE can be found in [28,40]. For complete convergence analysis, the reader may refer to [8].

### 23.4.7 Parallelization/Acceleration

Due to their iterative and blind nature, the heuristics discussed in this section require large runtime, especially on large problems, and CPU-intensive cost functions. Substantial amount of work has been done to parallelize or design accelerators to run these time consuming heuristics. With respect to simulated annealing, which is inherently sequential, some ingenuity is required on the part of the designer to cleverly parallelize the annealing process. Several parallel implementations of SA have been reported in [6,41–47]. Hardware acceleration that consists of implementing time consuming parts in hardware is described in [48]. Parallel acceleration, where execution of the algorithm is partitioned on several concurrently running processors, is reported in [49–51]. Other approaches that have been applied to parallelize SA are found in [6,49,52,53]. The parallel accelerations follow two general strategies: (1) move acceleration, also called single-trial parallelism, and (2) parallel moves or multiple-trial parallelism.

The GA is highly parallel. The reported GA parallelization strategies fall into three general categories: the *island model* [54,55], the *stepping stone model* [56–59], and the *neighborhood model*, also called the cellular model [60,61].

Work on parallelization of the tabu search heuristic can be found in [8,62–66]. The heuristic has also been parallelized and executed on a network of workstations using PVM [67]. Techniques to accelerate the execution of SimE by implementing it on vector-processors [29] or on a network of workstations [68] are described in [8].

### 23.4.8 Applications

The first applications of SA were on placement [2]. Furthermore, the largest number of applications of SA was on digital design automation problems [5]. A popular package that uses SA for VLSI standard-cell placement and routing is the TimberWolf3.2 package [46]. In addition to placement, SA has been applied successfully to several other problems. These include classical problems such as the TSP [2], graph partitioning, matching problem, Steiner problems [69], linear arrangement [1], clustering problem [70], quadratic assignment [71], various scheduling problems [72,73], graph coloring [74], etc. In the area of engineering SA has been applied extensively to solve various hard VLSI physical design automation problems [5]. In addition, it has been applied with success in other areas such as topology design of computer networks [75], image processing [76], test pattern generation, code design, etc. A comprehensive list of bibliography of some of the above applications and some details of their implementation such as cost function formulation, move set design, parameters, etc., is available in [6,8,77–79].

In addition to their application to classical optimization problems such as the knapsack problem [80], TSP [81,82], Steiner tree problem [83], set covering problem [84], N-queens problem [85], clustering problem [86], graph partitioning [87], etc., GAs have also been applied to several engineering problems. Some examples of these applications include job shop and multiprocessor scheduling [81,88,89], discovery of maximal distance codes for data communications [90], bin-packing [91], design of telecommunication (mesh) networks [92], test sequence generation for digital system testing [93], VLSI design (cell placement [5,94–96], floorplanning [97], routing [98]), pattern matching [99], technology mapping [100], PCB assembly planning [101], and high-level synthesis of digital systems [102,103]. The books by Goldberg (1989) [10], Davis (1991) [104], recent conference proceedings on evolutionary computation, and on applications of genetic algorithms discuss in detail the various applications of GAs in science and engineering. These range from optimization of pipeline systems and medical imaging to applications such as robot trajectory generation and parametric design of aircraft [10,104].

TS has also been applied to solve combinatorial optimization problems appearing in various fields of science, engineering, and business. Results reported indicate superior performance to other previous techniques. Examples of some hard problems to which tabu search has been applied with success include graph partitioning [105], clustering [106], TSP [107], maximum independent set problem [108], graph coloring [109,110], maximum clique problem [111], and quadratic assignment problem [62,112] to name a few. In the area of engineering, tabu search has been applied to machine sequencing [113], scheduling [22,114–118], fuzzy clustering [119], multiprocessor scheduling [120], vehicle routing [121–123], general fixed charge problem [17], bin-packing [124], bandwidth packing [24], VLSI placement [125], circuit partitioning [126], global routing [127], high-level synthesis of digital systems [128,129], etc. A good summary of most recent applications of tabu search can be found in [8,18,130].

The SimE algorithm has also been used to solve a wide range of combinatorial optimization problems. Kling and Banerjee published their results with respect to SimE in design automation conferences [25,27] and journals [26,28]. This explains the fact that most published work on SimE has been originated by researchers in the area of design automation of VLSI circuits [40,131–134]. The first problem on which SimE was first applied is standard cell placement [25,28]. A number of papers describe SimE-based heuristics as applied to the routing of VLSI circuits [131,135–140]. SimE was also successfully applied in high-level synthesis [141–143]. Other reported SimE applications are in microcode compaction [144], automatic synthesis of gate matrix [134], and the synthesis of cellular architecture field programmable gate arrays (FPGAs) [145].

### 23.4.9 Conclusion

This section has introduced the reader to four effective heuristics that belong to the class of *general approximation iterative algorithms,* namely SA, GA, tabu search, and SimE. From the immense literature that is available it is evident that for a large variety of applications, in certain settings, these heuristics produce excellent results. All five algorithms are general iterative metaheuristics. A value of the objective function is used to compare results of consecutive iterations and a solution is selected based on its value.

All algorithms incorporate domain specific knowledge to dictate the search strategy. They also tolerate some element of nondeterminism that helps the search escape out of local minima. They all rely on the use of a suitable cost function, which provides feedback to the algorithm as the search progresses. The principle difference among these heuristics is how and where domain specific knowledge is used. For example, in SA such knowledge is mainly included in the cost function. Elements involved in a perturbation are selected randomly, and perturbations are accepted or rejected according to the *Metropolis* criterion, which is a function of the cost. The cooling schedule has also a major impact on the algorithm performance and must be carefully crafted to the problem domain as well as the particular problem instance.

For the two evolutionary algorithms discussed in the chapter, GA and SimE, domain specific knowledge is exploited in all phases. In the case of GA, the fitness of individual solutions incorporates domain specific knowledge. Selection for reproduction, the genetic operations, as well as generation of the new population also incorporate a great deal of heuristic knowledge about the problem domain. In SimE, each individual element of a solution is characterized by a goodness measure that is highly correlated with the objective function. The perturbation step (selection followed by allocation) affects mostly low goodness elements. Therefore, domain specific knowledge is included in every step of the SimE algorithm.

Tabu search is different from the above heuristics in that it has an explicit memory component. At each iteration the neighborhood of the current solution is partially explored, and a move is made to the best nontabu solution in that neighborhood. The neighborhood function as well as tabu list size and content are problem specific. The direction of the search is also influenced by the memory structures (whether intensification or diversification is used).

A classification of meta-heuristics proposed by Glover and Laguna [130] is based on three basic features: (1) the use of adaptive memory, where the letter $A$ is used if the meta-heuristic employs adaptive memory, and the letter $M$ is used if it is memoryless; (2) the kind of neighborhood exploration, where the letter $N$ is used if the meta-heuristic performs a systematic neighborhood search, and the letter $S$ is used if stochastic sampling is followed; and (3) the number of current solutions carried from one iteration to the next, where the digit 1 is used if the meta-heuristic maintains a single solution, and the letter $P$ is used if a parallel search is performed with a population of solutions of cardinality $P$. For example, according to this classification, GA is $M/S/P$, tabu search is $A/N/1$, SA is $M/S/1$, and SimE is also $M/S/1$.

It is also possible to make hybrids of these algorithms. The basic idea of hybridization is to enhance the strengths and compensate for the weaknesses of two or more complementary approaches. For the details about the hybridization the readers are referred to [8].

In this section, it has not been the authors' intention to demonstrate the superiority of one algorithm over the other. Actually it would be unwise to rank such algorithms. Each one of them has its own merits. Recently, an interesting theoretical study has been reported by Wolpert and Macready in which they proved a number of theorems stating that the average performance of any pair of iterative (deterministic or nondeterministic) algorithms across all problems is identical. That is, if an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the remaining set of problems [146]; however, it should be noted that the reported theorems assume that the algorithms do not include domain specific knowledge of the problems being solved. Obviously, it would be expected that a well-engineered algorithm would exhibit superior performance to that of a poorly engineered one.

## Acknowledgment

The authors acknowledge King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia, for all support. This work is carried out under university-funded project number COE/ITERATE/221. Special thanks to Junaid Asim Khan and Salman Khan for their tremendous assistance and help in the preparation of this manuscript.

## References

1. S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. *International Journal of Computer Aided VLSI Design,* 1(1):1–23, 1989.
2. S. Kirkpatrick, Jr., C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science,* 220 (4598): 498–516, May 1983.
3. V. Černy. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Application,* 45(1):41–51, January 1985.
4. N. Metropolis et al. Equation of state calculations by fast computing machines. *Journal of Chem. Physics,* 21:1087–1092, 1953.
5. Sadiq M. Sait and Habib Youssef. *VLSI Design Automation: Theory and Practice.* McGraw-Hill, Europe (also co-published by IEEE Press), 1995.
6. Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* John Wiley & Sons, New York, 1989.
7. R.H.J.M. Otten and L.P.P.P. van Ginneken. *The Annealing Algorithm.* Kluwer Academic Publishers, Boston, MA, 1989.
8. Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems.* IEEE Computer Society Press, 1999.
9. J.H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.
10. D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.
11. F. Glover. Tabu search—Part I. *ORSA Journal on Computing,* 1(3):190–206, 1989.
12. F. Glover. Tabu search—Part II. *ORSA Journal on Computing,* 2(1):4–32, 1990.
13. F. Glover, E. Taillard, and D. de Werra. A user's guide to Tabu search. *Annals of Operations Research,* 41:3–28, 1993.
14. F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, Boston, MA, 1997.
15. P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. *Congress on Numerical Methods in Combinatorial Optimization,* 1986.
16. F. Glover. Artificial intelligence, heuristic frameworks and Tabu search. *Managerial and Decision Economics,* 11:365–375, 1990.
17. Minghe Sun and P.G. McKeown. Tabu search applied to the general fixed charge problem. *Annals of Operations Research,* 41:405–420, 1993.
18. F. Glover. Tabu search and adaptive memory programming-advances, applications and challenges. *Technical Report, College of Business, University of Colorado at Boulder,* 1996.
19. F. Dammeyer and Stefan VoB. Dynamic Tabu list management using the reverse elimination method. *Annals of Operations Research,* 41:31–46, 1993.
20. M. Malek, M. Guruswamy, M. Pandya, and H. Owens. Serial and parallel Simulated Annealing and Tabu search algorithms for the traveling salesman problem. *Annals of Operations Research,* 21:59–84, 1989.
21. J. Ryan, editor. *Heuristics for Combinatorial Optimization,* June 1989.
22. O. Icmeil and S. Selcuk Erenguc. A Tabu search procedure for the resource constrained project scheduling problem with discounted cash flows. *Computers & Operations Research,* 21(8):841–853, 1994.

23. J.P. Kelly, M. Laguna, and F. Glover. A study of diversification strategies for the quadratic assignment problem. *Computers & Operations Research,* 21(8):885–893, 1994.
24. M. Laguna and F. Glover. Bandwidth Packing; A Tabu search approach. *Management Science,* 39(4):492–500, 1993.
25. Ralph Michael Kling and Prithviraj Banerjee. ESP: A new standard cell placement package using Simulated Evolution. *Proceedings of 24th Design Automation Conference,* pp. 60–66, 1987.
26. R.M. Kling and P. Banerjee. ESP: Placement by Simulated Evolution. *IEEE Transactions on Computer-Aided Design,* 8(3):245–255, March 1989.
27. R.M. Kling and P. Banerjee. Optimization by Simulated Evolution with applications to standard-cell placement. *Proceedings of 27th Design Automation Conference,* pp. 20–25, 1990.
28. R.M. Kling and P. Banerjee. Empirical and theoretical studies of the Simulated Evolution method applied to standard-cell placement. *IEEE Transactions on Computer-Aided Design,* 10(10): 1303–1315, October 1991.
29. R.M. Kling. *Optimization by Simulated Evolution and its Application to cell placement.* Ph.D. Thesis, University of Illinois, Urbana, 1990.
30. E.H.L. Aarts and P.J.N. Van Laarhoven. Statistical cooling: a general approach to combinatorial optimization problem. *Philips Journal of Research,* 40(4):193–226, January 1985.
31. D.E. Goldberg and P. Segrest. Finite Markov chain analysis of Genetic algorithms. *Genetic Algorithms and Their Applications: Proceedings of 2nd International Conference on GAs,* pp. 1–8, 1987.
32. A.E. Nix and M.D. Vose. Modeling Genetic algorithms with Markov chains. *Annals of Mathematics and Artificial Intelligence,* pp. 79–88, 1993.
33. A.H. Eiben, E.H.L. Aarts, and K.M. Van Hee. Global convergence of Genetic algorithms: A Markov chain analysis. In H.P. Schwefel and Männer, editors, *Parallel Problem Solving from Nature,* pp. 4–12. Springer-Verlag, Berlin, 1990.
34. T.E. Davis and J.C. Principe. A Simulated Annealing like convergence theory for the simple Genetic algorithm. *Proceedings of the 4th International Conference on Genetic Algorithm,* pp. 174–181, 1991.
35. T.E. Davis and J.C. Principe. A Markov chain framework for the simple Genetic algorithm. *Proceedings of the 4th International Conference on Genetic Algorithm,* 13:269–288, 1993.
36. S.W. Mahfoud. Finite Markov chain models of an alternative selection strategy for the Genetic algorithm. *Complex systems,* 7:155–170, 1993.
37. G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks,* 5:1:96–101, 1994.
38. D.B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press, 1995.
39. U. Faigle and W. Kern. Some convergence results for probabilistic Tabu search. *ORSA Journal on Computing,* 4(1):32–37, Winter 1992.
40. C.Y. Mao and Y.H. Hu. Analysis of convergence properties of Stochastic Evolution algorithm. *IEEE Transactions on Computer Aided Design,* 15(7):826–831, July 1996.
41. M.D. Huang, F. Romeo, and A.L. Sangiovanni-Vincentelli. An efficient general cooling schedule for Simulated Annealing. In *IEEE International Conference on Computer-Aided Design,* pp. 381–384, 1986.
42. H. Szu and R. Hartley. Fast Simulated Annealing. *Physics Letters, A,* 122:157–162, 1987.
43. J.W. Greene and K.J. Supowit. Simulated Annealing without rejected moves. *IEEE Tansactions on Computer-Aided Design,* 5:221–228, 1986.
44. P.J.M. Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications.* Reidel, Dordrecht, 1987.
45. F. Catthoor, H. DeMan, and J. Vandewalle. Samurai: A general and efficient Simulated-Annealing schedule with fully adaptive annealing parameters. *Integration,* 6:147–178, 1988.
46. C. Sechen and A.L. Sangiovanni-Vincentelli. Timberwolf3.2: A new standard-cell placement and global routing package. *Proceedings of 23rd Design Automation Conference,* pp. 432–439, 1986.

47. L.K. Grover. A new Simulated Annealing algorithm for standard-cell placement. In *IEEE International Conference of Computer Aided Design*, pp. 378–380, 1986.
48. A. Iosupovici, C. King, and M. Breuer. A module interchange placement machine. *Proceedings of 20th Design Automation Conference,* pp. 171–174, 1983.
49. Saul A. Kravitz and Rob A. Rutenbar. Placement by Simulated Annealing on a multiprocessor. *IEEE Transactions on Computer Aided Design,* CAD-6(4):534–549, July 1987.
50. F. Darema-Rogers, S. Kirkpatrick, and V.A. Norton. Parallel algorithms for chip placement by simulated annealing. *IBM Journal of Research and Development,* 31:391–402, May 1987.
51. F. Darema, S. Kirkpatrick, and V.A. Norton. Parallel techniques for chip placement by Simulated Annealing on shared memory systems. *Proceedings of International Conference on Computer Design: VLSI in Computers & Processors, ICCD-87,* pp. 87–90, 1987.
52. M.D. Durand. Accuracy vs. Speed in placement. *IEEE Design & Test of Computers,* pp. 8–34, June 1989.
53. Xin Yao. Global optimization by evolutionary algorithms. In *Proceedings of IEEE International Symposium on parallel algorithms architecture synthesis,* pp. 282–291, 1997.
54. T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed Genetic algorithm. In *Parallel problem solving from nature,* 1991.
55. M. Tanese. Distributed Genetic Algorithms. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pp. 434–439. Morgan-Kaufmann, San Maeto, CA, 1989.
56. M. Gorges-Schleuter. Explicit parallelism of Genetic algorithms through population structures. In H.P. Schwefel and R. Männer, editors, *Problem Solving from Nature,* pp. 150–159. Springer-Verlag, New York, 1991.
57. N. Eldredge and S.J. Gould. *Punctuated Equilibrea: An alternative to phyletic gradualism. Models of Paleobiology,* T.J.M. Schopf, Ed. San Fransisco: CA, Freeman. Cooper and Co., 1972.
58. N. Eldredge. *Time Frames.* New York: Simon and Schuster, 1985.
59. J.P. Cohoon, S.U. Hegde, W.N. Martin, and D.S. Richards. Distributed Genetic algorithms for the floorplan design problem. *IEEE Transactions on Computer-Aided Design,* CAD-10:483–492, April 1991.
60. V. Scott Gordon and Darrell Whitley. A machine-independent analysis of parallel Genetic algorithms. *Complex Systems,* 8:181–214, 1994.
61. M. Gorges-Schleuter. ASPARAGOS-An asynchronous parallel Genetic optimization strategy. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms and their Applications,* pp. 422–427. Morgan-Kaufmann, San Maeto, CA, 1989.
62. E. Taillard. Robust Tabu search for the quadratic assignment problem. *Parallel Computing,* 17:443–455, 1991.
63. E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research,* 417:65–74, 1990.
64. Bruno-Laurent Garica, Jean-Yves Potvin, and Jean-Marc Rousseau. A parallel implementation of the Tabu search heuristic for vehicle routing problems with time window constraints. *Computers & Operations Research,* 21(9):1025–1033, November 1994.
65. I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro. Improving search by incorporating evolution principles in parallel Tabu search. In *Proc. of the first IEEE Conference on Evolutionary Computation-ICEC '94,* pp. 823–828, June 1994.
66. E. Taillard. Parallel iterative search methods for the vehicle routing problem. *Networks,* 23:661–673, 1993.
67. Sadiq M. Sait, Habib Youssef, H. Barada, and Ahmed Al-Yamani. A parallel Tabu search algorithm for VLSI standard-cell placement. *Proceedings of IEEE International Symposium on Circuits and Systems,* May 2000.
68. Ralph Michael Kling and Prithviraj Banerjee. Concurrent ESP: A placement algorithm for execution on distributed processors. *Proceedings of the IEEE International Conference on Computer-Aided Design,* pp. 354–357, 1987.

69. K.A. Dowsland. Hill climbing, Simulated Annealing, and the Steiner problem in graphs. *Eng. Opt.,* 17:91–107, 1991.
70. S. Selim and K.S. Al-Sultan. A Simulated Annealing algorithm for the clustering problem. *Pattern Recognition,* 24(10):1003–1008, 1991.
71. D.T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research,* 46:93–100, 1990.
72. I.H. Osman and C.N. Potts. Simulated Annealing for permutation flow-shop annealing. *OMEGA,* 17:551–557, 1989.
73. F.A. Ogbu and D.K. Smith. The application of the Simulated Annealing algorithm to the solution of the $n/m/c_{\max}$ flowshop problem. *Computers & Operations Research,* 17:243–253, 1990.
74. M. Chams, A. Hertz, and D. de Werra. Some experiments with Simulated Annealing for coloring graphs. *European Journal of Operational Research,* 32:260–266, 1987.
75. C. Ersoy and S.S. Panwar. Topological design of interconnected LAN/MAN networks. *IEEE Journal on Selected Areas in Communications,* 11(8):1172–1182, 1993.
76. S. Geman and D. Geman. Stochastic relaxation, Gibbs distribution, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* PAMI-6:721–741, 1984.
77. K.A. Dowsland. Simulated Annealing. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Optimization Problems.* McGraw-Hill, Europe, 1995.
78. N.E. Collins, R.W. Eglese, and B.L. Golden. Simulated annealing: An annotated bibliography. *AJMMS,* 8:209–307, 1988.
79. R.W. Eglese. Simulated Annealing: A tool for operational research. *European Journal of Operational Research,* 46:271–281, 1990.
80. R. Spillman. Solving large Knapsack problems with a Genetic algorithm. In *International Conference on Systems, Man and Cybernetics,* pp. 632–637, 1995.
81. D. Whitley, T. Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesmen: the genetic edge recombination operator. In *Proceedings of the 3rd International Conference on Genetic Algorithms and their Applications,* pp. 133–140, 1989.
82. H. Tamaki et al. A comparison study of Genetic codings for the traveling salesman problem. In *Proceedings of the 1st IEEE Conference on Evolutionary Computation,* pp. 1–6, 1994.
83. J. Hesser, R. Manner, and O. Stucky. Optimization of Steiner trees using Genetic algorithms. In *ICGA '89,* pp. 231–236, 1989.
84. K.S. Al-Sultan, M.F. Hussain, and J.S. Nizami. A Genetic algorithm for the set covering problem. *Journal of the Operational Research Society,* 47:702–709, 1996.
85. A. Homaifar, J. Turner, and Samia Ali. The N-queens problem and Genetic algorithms. In *IEEE Proceedings of Southeastcon '92,* pp. 262–267, April 1992.
86. K.S. Al-Sultan and M. Maroof Khan. Computational experience on four algorithms for the hard clustering problem. *Pattern Recognition Letters,* 17:295–308, 1996.
87. H. Pirkul and E. Rolland. New heuristic solution procedures for uniform graph partitioning problem: Extensions and evaluation. *Computers & Operations Research,* 21(8):895–907, October 1994.
88. E.S.H. Hou, N. Ansari, and R. Hong. Genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems,* 5(2):113–120, February 1994.
89. M.S.T. Benten and Sadiq M. Sait. Genetic scheduling of task graphs. *International Journal of Electronics,* 77(4):401–415, 1994.
90. K. Dontas and K. De Jong. Discovery of maximal distance codes using Genetic algorithms. *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence,* pp. 805–811, 1990.
91. E. Falkenauer and A. Delchambre. A genetic algorithm for bin packing and line balancing. *Proceedings of International Conference on Robotics and Automation,* pp. 1186–1192, May 1992.
92. King-Tim Ko et al. Using genetic algorithms to design mesh networks. *Computer,* pp. 56–60, 1997.

93. E.M. Rudnick, J.H. Patel, G.S. Greenstein, and T.M. Niermann. Sequential circuit test generation in a Genetic algorithm framework. In *Proceedings of the 31st Design Automation Conference,* pp. 698–704, 1994.
94. J.P. Cohoon and W.D. Paris. Genetic placement. *IEEE Transactions on Computer-Aided Design,* CAD-6:956–964, November 1987.
95. K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys,* 23(2):143–220, June 1991.
96. K. Shahookar and P. Mazumder. A genetic approach to standard cell placement using meta-genetic parameter optimization. *IEEE Transactions on Computer-Aided Design,* 9(5):500–511, May 1990.
97. Sadiq M. Sait et al. Timing influenced general-cell genetic foorplanner. In *ASP-DAC '95: Asia and South-Pacific Design Automation Conference,* pp. 135–140, 1995.
98. H.I. Han et al. GenRouter: A genetic algorithm for channel routing problems. In *Proceeding of TENCON 95, IEEE Region 10 International Conference on Microelectronics and VLSI,* pp. 151–154, November 1995.
99. N. Ansari, M.-H. Chen, and E.S.H. Hou. Point pattern matching by genetic algorithm. In *16th Annual Conference on IEEE Industrial electronics,* pp. 1233–1238, 1990.
100. V. Kommu and I. Pomeranz. GAFPGA: Genetic algorithms for FPGA technology mapping. In *Proceeding of EURO-DAC '93: IEEE European Design Automation Conference,* pp. 300–305, September 1993.
101. M.C. Leu, H. Wong, and Z. Ji. Genetic algorithm for solving printed circuit board assembly planning problems. *Proceedings of Japan-USA Symposium on Flexible Automation,* pp. 1579–1586, July 1992.
102. S. Ali, Sadiq M. Sait, and M.S.T. Benten. GSA: Scheduling and allocation using Genetic algorithms. In *Proceedings of EURODAC '94: IEEE European Design Automation Conference,* pp. 84–89, September 1994.
103. C.P. Ravikumar and V. Saxena. TOGAPS: A testability oriented genetic algorithm for pipeline synthesis. *VLSI Design,* 5(1):77–87, 1996.
104. L. Davis, editor. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, NY, 1991.
105. A. Lim and Yeow-Meng Chee. Graph partitioning using Tabu search. In *1991 IEEE International Symposium on Circuits and Systems,* pp. 1164–1167, 1991.
106. K.S. Al-Sultan. A Tabu search approach to the clustering problem. *Pattern Recognition,* 28(9): 1443–1451, 1995.
107. M. Malek, M. Heap, R. Kapur, and A. Mourad. A fault tolerant implementation of the traveling salesman problem. *Research Report, Department of EE and Computer Engineering, The University of Texas-Austin,* May 1989.
108. C. Friden, A. Hertz, and D. de Werra. TABARIS: An exact algorithm based on Tabu search for finding a maximum independent set in a graph. *Computers & Operations Research,* 19(1–4):81–91, 1990.
109. N. Dubois and D. de Werra. EPCOT: An efficient procedure for coloring optimally with Tabu search. *Computers Math Application,* 25(10/11):35–45, 1993.
110. A. Hertz and D. de Werra. Using Tabu search techniques for graph coloring. *Computing,* 39:345–351, 1987.
111. M. Gendreau, P. Soriano, and L. Salvail. Solving the maximum clique problem using a Tabu search approach. *Annals of Operations Research,* 41:385–404, 1993.
112. J. Skorin-Kapov. Tabu search applied to the quadratic assignment problem. *ORSA Journal on Computing,* 2(1):33–45, 1990.
113. C.R. Reeves. Improving the efficiency of Tabu search for machine sequencing problems. *Journal of Operational Research Society,* 44:375–382, 1993.
114. E.L. Mooney and R.L. Rardin. Tabu search for a class of scheduling problems. *Annals of Operations Research,* 41:253–278, 1993.

115. M. Dell'Amico and M. Trubian. Applying Tabu search to the job-shop scheduling problem. *Annals of Operations Research,* 41:231–252, 1993.
116. M. Widmer and A. Hertz. A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research,* 41:186–193, 1989.
117. J.W. Barnes and J.B. Chambers. Solving the job shop scheduling problem with Tabu search. *IIE Transactions,* 27:257–263, 1995.
118. M. Widmer. Job shop scheduling with tooling constraints: A Tabu search approach. *Journal of Operational Research Society,* 42(1):75–82, 1991.
119. K.S. Al-Sultan and C.A. Fedjki. A Tabu search based algorithm for the fuzzy clustering problem. *Pattern Recognition,* 1998.
120. R. Hubscher and F. Glover. Applying Tabu search with influential diversification to multiprocessor scheduling. *Computers & Operations Research,* 21(8):877–884, 1994.
121. I.H. Osman. Metastrategy Simulated Annealing and Tabu search algorithms for the vehicle routing problem. *Annals of Operations Research,* 41:421–451, 1993.
122. F. Semet and E. Taillard. Solving real-life vehicle routing problems efficiently using Tabu search. *Annals of Operations Research,* 41:469–488, 1993.
123. J. Renaud, G. Laporte, and F.F. Boctor. A Tabu search heuristic for the multi-depot vehicle routing problem. *Computers Ops Research,* 23:229–235, 1996.
124. F. Glover and R. Hubscher. Binpacking with a Tabu search. *Technical Report, Graduate School of Business Administration, University of Colorado at Boulder,* 1991.
125. L. Song and A. Vannelli. VLSI placement using Tabu search. *Microelectronics Journal,* 17(5): 437–445, 1992.
126. S. Areibi and A. Vannelli. Circuit partitioning using a Tabu search approach. In *1993 IEEE International Symposium on Circuits and Systems,* pp. 1643–1646, 1993.
127. Habib Youssef and Sadiq M. Sait. Timing driven global router for standard cell design. *International Journal of Computer Systems Science and Engineering,* 1998.
128. Sadiq M. Sait, S. Ali, and M.S.T. Benten. Scheduling and allocation in high-level synthesis using stochastic techniques. *Microelectronics Journal,* 27(8):693–712, October 1996.
129. S. Amellal and B. Kaminska. Functional synthesis of digital systems with TASS. *IEEE Transactions on Computer-Aided Design,* 13(5):537–552, May 1994.
130. F. Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems.* McGraw-Hill, Europe, 1995.
131. Y.L. Lin, Y.C. Hsu, and F.H.S. Tsai. SILK: A Simulated Evolution router. *IEEE Transactions on Computer-Aided Design,* 8(10):1108–1114, October 1989.
132. A. Ly and Jack T. Mowchenko. Applying Simulated Evolution to high level-synthesis. *IEEE Transactions on Computer-Aided Design,* 12(3):389–409, March 1993.
133. C.Y. Mao and Y.H. Hu. SEGMA: A Simulated Evolution gate matrix layout algorithm. *VLSI Design,* 2(3):241–257, 1994.
134. C.Y. Mao. *Simulated Evolution Algorithms for Gate Matrix layouts.* Ph.D. Thesis, University of Wisconsin, Madison, 1994.
135. Ching-Dong Chen, Yuh-Sheng Lee, A.C.-H. Wu, and Youn-Long Lin. Tracer-FPGA: A router for RAM-based FPGA's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 14(3):371–374, March 1995.
136. T. Koide, S. Wakabayashi, and N. Yoshida an integrated approach to pin assignment and global routing for VLSI building-block layout. In *1993 European Conference on Design Automation with the European Event in ASIC Design,* pp. 24–28, Loss Alamitos, CA, USA, Feb 1993. IEEE Computer Society Press.
137. Yirng-An Chen, Youn-Long Lin, and Yu-Chin Hsu. A new global router for ASIC design based on Simulated Evolution. In *Proceedings of the 33rd IEEE Midwest Symposium on Circuits and Systems,* pp. 261–265, New York, NY, USA, May 1989.

138. Youn-Long Lin, Yu-Chin Hsu, and Fur-Shing Tsai. A detailed router based on Simulated Evolution. In *Proceedings of the 33rd Midwest Symposium on Circuits and Systems,* pp. 38–41, New York, NY, USA, Nov 1988. IEEE Computer Soc. Press.
139. Yuh-Sheng Lee and A.C.-H Wu. A performance and routability driven router for FPGAS considering path delays. *ANSI/IEEE Std 802.lb-1995,* pp. 557–561, March 1995.
140. Yung-Ching Hsich, Chi-Yi Hwang, Youn-Long, and Yu-Chin Hsu. LIB: A CMOS cell compiler. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 10(8):994–1005, Aug 1991.
141. T.A. Ly. and J.T. Mowchenko. Applying Simulated Evolution to scheduling in high level synthesis. In *Proceedings of the 33rd Midwest Symposium on Circuits and Systems,* vol. 1, pp. 172–175, New York, NY, USA, Aug. 1990, IEEE Computer Soc. Press.
142. Yau-Hwang Kuo and Shaw-Pyng Lo. Automated synthesis of asynchronous pipelines. In *Custom Integrated Circuits Conference, Proceedings of the IEEE 1992,* vol. 2, pp. 685–688, New York, NY, USA, May 1992.
143. Yau-Hwang Kuo and Shaw-Pyng Lo. Partitioning and scheduling of asynchronous pipelines. In *Computer Systems and Software Engineering, CompEuro 1992 Proceedings,* pp. 574–579, Loss Alamitos, CA, USA, May 1992. IEEE Computer Soc. Press.
144. I. Ahmad, M.K. Dhodhi, and K.A Saleh. An evolutionary-based technique for local microcode compaction. In *Proceedings of ASP DAC '95/CHDL '95/VLSI with EDA Technofair,* pp. 729–734, Tokyo, Japan, Sept. 1995. Nihon Gakkai Jimu Senta.
145. A.K. Dasari, N. Song, and M. Chrzanowska-Jeske. Layout-driven factorization and fitting for cellular-architecture FPGAS. In *Proceedings of IEEE, NORTHCON '93 Electrical and Electronics Convention,* pp. 106–111, New York, NY, USA, October 1993.
146. D.H. Wolpert and W.G. Macready. No Free Lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation,* 1(1):67–82, April 1997.

## 23.5 Parallelization of Iterative Heuristics

*Sadiq M. Sait, Habib Youssef, and Mohammad Faheemuddin*

### 23.5.1 Introduction

Iterative heuristics such as simulated annealing (SA), genetic algorithms (GAs), and tabu search (TS) are stochastic optimization algorithms that have found applications in a myriad of complex problems in science and engineering. The Section 23.4 discussed these algorithms in detail, and elaborated on their various characteristics. However, with increasing domain complexity and sizes, in spite of their robust nature and capability, these algorithms can have very high runtimes. Although there are acceleration strategies for these heuristics, these are often just parameter tweaks and are at best problem-specific and often nonscalable.

Parallelization of these algorithms to achieve reduced runtime as well as possibly find better solutions has increasingly attracted attention over the years. With the ever-decreasing cost–performance ratio of generic computer systems, cluster environments have become a norm in both academia and industry. Also as network technology keeps chipping away at the latency constraint, these adistributed computing environments offer very promising and competing alternatives to expensive multiprocessor machines.

In this chapter, we document parallelization strategies for different popular iterative heuristics, namely SA, GAs, TS, and simulated evolution (SimE). A detailed description of these heuristics is available in the book by Sait and Youssef [1]. A broad classification is presented for the different parallel models followed by an enumeration and description of relevant strategies. The focus is on parallel approaches for cluster environments. This chapter concludes with further directions in contemporary research.

### 23.5.2 Parallelization Issues

Distributed computing today offers extensive opportunities in collectively utilizing computing power for performance gains. The concept behind parallelization is often to allocate fairly independent sections of the algorithm to individual processors, collect the results, do the needful, and continue onward with the next iteration. However, this approach, though simple and straightforward does not scale well—Amdahl's law is soon to set in—the speedup achievable is limited to the nonparallelizable, serial fraction of the algorithm. This is especially aggravated by the inherently sequential nature of various heuristics, such as SA, TS, and SimE.

Parallelization strategies can be broadly classified into the following three approaches:

1. Low-level parallelization (type 1): This is also known as move acceleration, in this approach the computation-intensive operations within a single iteration are distributed among nodes. Such approaches seek to divide the workload for each iteration across multiple processors, and as a consequence, leave the algorithm characteristics unaffected.
2. Domain decomposition (type 2): In this approach, the problem state space is divided and assigned to different processors. Also known as multiple-trials parallelism, this strategy can involve either partitioning the single solution across available processors or distributing the search space by assigning processors sets of moves or perturbations. In both these cases, all nodes work with the same single copy of the solution. This usually implies a conspicuous departure from the functionality and characteristics of the serial algorithm.
3. Multithreaded or parallel search (type 3): Parallelism here is implemented as multiple concurrent exploration of the solution space using search threads with various degrees of synchronization or information exchange. Often modeled as multiple-Markov chains (MMC), these methods allow for increasing the variety of the search threads particularly by having different types of searches—same method with different parameter settings or even different meta-heuristics—proceeding concurrently. The distinguishing feature of this approach is that the independent processors work on their own individual solutions and can periodically communicate to cooperatively navigate the search space.

Of these various approaches, move acceleration is suitable only to tightly coupled, multiprocessor environments, rather than cluster computing. Even on the former, Amdahl's law restricts achievable scalability. As such, the discussion here is restricted to domain decomposition strategies and parallel search models. Such cooperative parallel models are generally applicable with minor variations to almost all heuristics in general, providing impressive performance gains.

### 23.5.3 Simulated Annealing

Simulated annealing is arguably the pioneering success story of iterative heuristics and their application in combinatorial optimization problems. Proposed in the early 1980s, SA simulates the effect of a heat bath on the structure of metals [2,3]. In the metallurgical annealing process, heated metals are cooled at a controlled rate, thereby transitioning from a higher energy state to a lower one over a period. A proper, controlled cooling schedule allows the metal atoms to achieve perfect crystal lattices. This process was first mathematically studied by Metropolis et al. in 1953 who established criteria to simulate how thermodynamic systems change from one energy level to another. Lower energy states are assumed to be always accepted, whereas acceptance of higher energy states is probabilistic. In simulation parlance, this is referred to as the Metropolis acceptance criterion which defines this probability given by the following expression:

$$\text{Prob(accept)} = e^{-\left(\frac{\Delta E}{K_B T}\right)} \tag{23.2}$$

where

$K_B$ is the Boltzman constant

$T$ indicates the temperature

In SA, every iteration at a certain temperature, within the Metropolis loop comprises the following steps:

1. Perturb the current solution to create a new solution
2. Compute the difference in the cost between the current and new solutions
3. Decide whether to accept or reject the new solution

#### 23.5.3.1 Domain Decomposition Strategies

This multiple-trials parallelization approach of SA is very general and can be tailored to any particular problem instance. In this strategy, several trials (moves) are generated and evaluated in parallel, where each trial is executed by a single processor. The processors are forced to concurrently search for an acceptable solution in the neighborhood of the same current solution. To ensure that all processors are always working with the same current solution configuration, one has to force them to communicate and synchronize their actions whenever at least one of the trials is successful (accepted move).

Figure 23.83 is a possible parallel SA algorithm following this multiple-trials parallelization approach. Here, it is assumed that one master processor is ordering the concurrent execution and evaluation of $p$ trials, where $p$ is the number of processors. These new solutions are returned to the master, which arbitrates between them. In case of no success, the master then orders the parallel evaluation of $p$ new trials; otherwise, it selects the best new current solution, and updates the state of all processors. This process repeats until the defined termination criterion is reached. Also at the end of each $p$ new trials, the master processor checks to see whether equilibrium has been reached at current temperature. If so, the algorithm parameters are updated.

An examination of the algorithm in Fig. 23.83 reveals an evident overhead in communication, where the synchronous nature forces communication after each trial. However, since the current solution will get updated only when a processor makes a successful trial, the various nodes should be allowed to proceed asynchronously till one of them achieves an acceptable move. Therefore, one can markedly improve the parallel algorithm of Fig. 23.83 by making the following change. Synchronization is forced only when one of the processors performs a successful trial. In this new variation, communication is minimal. Furthermore, it is a more efficient parallelization since no processor is forced to remain idle waiting for other processors with more elaborate trials to finish.

```
Algorithm Parallel_SA;
  (*S0 is the initial solution *)
Begin
  Initialize parameters;
  Best S = S0;
  Cur S = S0;
    Repeat
      Repeat
      Communicate CurS to all processors;
      ParFor each processor i
        Perturb(CurS, NewSi);
        Ai = Accept (CurS, NewSi) (* Ai is true if NewSi is accepted *)
      EndParFor
      If Success Then
        (* Success = (∨_{i=1}^{p} Ai = True) *)
        Select(NewS);
        If Cost(NewS) < Cost(BestS) Then BestS = NewS;
      EndIf
    Until Time to update parameters;
  Until Time to stop;
  Output Best solution found
End. (*Parallel_SA*)
```

**FIGURE 23.83** General parallel simulated annealing (SA) algorithm where synchronization is forced after each trial.

Both variations of this parallel algorithm can be implemented to run on a multicomputer or a multiprocessor machine. The assumed parallel model is a multiple instruction and single data (MISD) or a multiple instruction and multiple data (MIMD) machine. For both algorithms, it is assumed that each processor must be able to set a common variable to *true* whenever it accepts a move; then the solution accepted by the processor is communicated to a master processor, which will force all other processors to halt and to properly update the current solution. Here, there are two possibilities. If the processors do not halt immediately but rather are allowed to complete the trials that were in progress when the request to stop was received then there could be more than one solution accepted, and therefore, the master processor has to arbitrate between them, select the best, and pass a copy to each processor. The other possibility is when a processor is supposed to abort whatever activity is in progress, as soon as it receives a request to stop. In that case, the first solution accepted by any of the processors would be the new solution of all the processors.

A valid concern here is the behavior of this parallel model with respect to the SA temperature parameter. In the early regime (high temperature), SA behaves close to a random search algorithm, where almost every move is accepted. This means that for the multiple-trials approach, the speedup will be low (almost 1) at high temperatures because the processors will be forced to communicate after each trial. On the other hand, as the temperature is lowered, less and less moves are accepted, reducing by the same token the need for communication, thus allowing the $p$ processors to concurrently be working most of the time. Therefore, in the cold regime, the speedup will be approaching the number of processors.

#### 23.5.3.2 Multithreaded Parallel Search Strategies

In this class of strategies, each processor runs its own self-contained, independent annealing algorithm on its own solutions, with periodic exchange of information to collectively guide the search process. Each of these search threads can either be synchronous or asynchronous. In the former, all processors periodically stop work at a defined time, and communicate information at once. In the asynchronous approach, there are no such process barriers, and all nodes are free to communicate at their own discretion. We document a recently reported adaptive variation of such an asynchronous multiple Markov chains (AMMC) method here.

The basic AMMC approach is shown in the Fig. 23.84, where each slave process runs its own annealing algorithm. The assigned master processor is excluded from the main computational workload and instead manages information exchange between the slaves. After each Metropolis loop, the slave returns its best-achieved cost to the master, which then compares this value against the global best reached so far. If the former is better, the slave is instructed to send the entire solution, and the associated global values are updated. Otherwise, the master sends its copy of the global best solution and associated cost to the slave, which replaces its current relevant values and continues with the next iteration.

The adaptivity mechanism deals with a dynamic value of $M$—the number of perturbations within a single-Metropolis routine, i.e., the number of moves allowed before a temperature update. Initially, during the high temperature region, where annealing approaches a random walk, $M$ is kept very low, and is incrementally increased allowing more thorough exploration of the neighboring search space at lower, stable temperatures. This strategy was empirically found to give significantly improved results over static AMMC approaches.

### 23.5.4 Genetic Algorithms

Genetic algorithms are a powerful domain-independent, robust search technique inspired by the Darwinian theory of evolution. Invented in the early 1970s by John Holland and his colleagues [4], GAs emulate the process of natural evolution whereby high fitness individuals survive and mate thus passing on their characteristics to offsprings. This adaptive algorithm works with a population of solutions called chromosomes, which are encoded as strings. Each of these solutions represent a point

```
Algorithm Parallel_Simulated_Annealing(S0, T0, α, β, M, Maxtime, my_rank, p)
Notation
  (* S0 is the initial solution. *)
  (* BestS is the best solution. *)
  (* T0 is the initial temperature. *)
  (* α is the cooling rate. *)
  (* M is the time until next parameter update. *)
  (* Maxtime is the total allowed time for the annealing process. *)
  (* my_rank of current process; 0 for master, !0 for slaves. *)
  (* p is the total number of running processes. *)
Begin
  T = TO;
  CurS = SO; // only master has the initial Solution
  BestS = CurS;
  CurCost = Cost(CurS);
  BestCost = Cost(BestS);
  Time = 0;
  If (my_rank = = 0) // i.e. Master process
    Broadcast(CurS);
  Endif
  If (my_rank != 0) // i.e. Slave process
    Repeat
      Call Metropolis(CurS, CurCost, BestS, BestCost, T, M);
      Time = Time + M;
      T = α T;
      M = β M;
      Send_to_Master(BestCost);
      Receive_frm-Master(verdict);
      If (verdict = = 1)
        Send_to_Master (BestS);
      Else
        Receive_frm_Master(BestS);
      EndIf
    Until (Time ≥ Maxtime);
  EndIf
  If (my_rank = = 0) // i.e. Master process
    Repeat
      Receive_frm_Slave(BestCost);
      Send_to_Slave(verdict);
      If (verdict = = 1)
        Receive_frm_Slave(BestS);
      Else
        Send_to_Slave (BestS);
      EndIf
    Until (All Slaves are done);
    Return(BestS);
  EndIf
End. (*Parallel_Simulated_Annealing*)
```

**FIGURE 23.84** Procedure for parallel simulated annealing (SA) using asynchronous MMC.

in the solution space, and in each iteration, referred to as a generation, a new set of strings that represent solutions (called offsprings) is created by crossing some of the strings of the current generation [5]. Occasionally, new characteristics are injected to add diversity. In this manner, GAs combine information exchange along with survival of the fittest among individuals to conduct their search for the optimum solution.

In GAs, a number of initial solutions are generated as string-based chromosomes. An equal number of offsprings are generated by selecting parent chromosomes, two at a time, and implementing a crossover

mechanism that copies substrings of solutions from both parents into the offspring. The generated solutions are evaluated and a selection operator decides the solutions that are passed on to the next generation as parents.

#### 23.5.4.1 Domain Decomposition Strategies

As GAs work with a population of solutions, they lend themselves to straightforward workload division strategies. Such move-acceleration strategies distribute the population among processors in every generation for fitness calculation and possibly even recombination.* The calculated fitness values are then collected at the assigned master, which then applies the GA operators and moves to the next generation. In such approaches, also referred to as global parallel models, the algorithm characteristics are left undisturbed as the decision processes such as population selection and mutation are done by a single processor. However, such strategies have predictable and limited scalability [6]. A more popular approach is the multi-deme parallel model, where subpopulations on independent nodes communicate and collectively navigate the search space. These come under the type-3 or multithreaded parallelization strategies.

#### 23.5.4.2 Multithreaded Parallel Search Strategies

Multiple-population GAs provide a more sophisticated parallelization strategy wherein several subpopulations evolve independently on individual processors and exchange individuals periodically. This exchange of solutions is called migration and is a core aspect of this parallel model. Multi-population GAs are known with different names. They are referred to as multi-deme parallel GAs (drawing on the analogy of natural evolution), distributed GAs (as they are often implemented on distributed parallel architectures), and coarse-grained GAs (since the computation to communication ratio is usually high). This model of parallel GAs is very popular, but also the most difficult to understand because of the effect of migration and a large number of influential parameters.

The first systematic study of parallel GAs with multiple populations was Grosso's work in the 1980s [7]. The objective was to simulate the interaction of several parallel subcomponents of an evolving population. The population was divided into five demes, each of exchanging individuals with the other after a fixed interval. The effect of migration on the search process and population convergence was documented, and the findings were later on bolstered by the further work of Grefenstette et al. [8]. From these studies, it was seen that favorable traits spread faster when the population is divided into smaller demes. However, when the demes were isolated, the rapid rise in fitness stops at a lower fitness value than with the complete population; i.e., the quality of the solution found after convergence was worse. This is expected as the quality of solution is heavily dependent on the initial population size. However, the controlled movement of solutions between these subpopulations called migration can significantly alter this behavior. The migration policy associated with a multi-deme GA can be a function of the following parameters:

1. Migration frequency (MF): This is defined as the timescale between the movement of individual solutions between demes. MF is also referred to as epoch length, and the interval between migrations is called an epoch.
2. Migration rate (MR): This is the number of individual solutions that will migrate from one deme to another per epoch. MF and rate are closely related, and can be either static; i.e., a fixed number of solutions moving between populations after a fixed number of generations or dynamic, wherein both parameters are controlled by the rate of convergence in individual subpopulations.
3. Migrant absorption: This parameter defines how migrants are absorbed into destination demes. Although the usual policy for identifying migrant solutions in the source demes is to select from

*Genetic operators such as crossover, selection, and mutation are often trivial in terms of runtime compared to the more computation intensive fitness calculations. As such, the runtime gain by distributing these operators may not be justifiable with the associated communication latencies.

the best, the absorption policy within the destination deme can vary. Common models advocate replacing worst solutions, replacing random solutions or roulette wheel-based probabilistic replacement.

4. Communication topology: This defines the connectivity between individual demes. Again, this can be static, wherein movement of solutions is predefined by the network topology or dynamic, wherein movement of migrants to and from demes depends on their population diversity, average fitness, and convergence.

The tuning of these parameters has often been on an intuitive basis, with different values reported for distinct problems. In the case of MR and MF, most models adopt a synchronous approach, triggering movement of solutions at periodic intervals. However, an alternative policy is to be asynchronous, wherein demes communicate only when near convergence [9,10]. The purpose of this model is most often to prevent premature convergence by restoring diversity into the demes. It is found that there is a certain critical MR and MF that allows the communicating subpopulations to achieve solutions of almost the same quality as panmictic populations. Lower values of these parameters would not allow proper mixing of solutions, thus achieving the same as isolated demes. Higher values of migration and frequency may show no improvement, thus wasting time and resources or worse, may actually hinder population diversity among processors.

Communication topology is an important factor controlling the spread of good solutions among different demes. It defines the direction of movement of migrant solutions from one deme to another. Although a dense topology would encourage rapid spread of good solutions throughout all demes, it might also prevent each subpopulation from following a separate evolutionary path. A more sparse connectivity will achieve certain amount of isolation between demes, thus allowing different solutions to be found and exploring different areas of the search space. Another possible classification of topologies distinguishes them into static, where the communication is already defined, and dynamic, wherein migrant destination is decided by factors such as average fitness or population diversity of candidate destination demes [10].

An extensive study of the effect of these parameters, especially of the influence of migration and population sizes was done by Cantú Paz. In spite of its complexities, this multi-deme approach shown in Fig. 23.85 has often been favored over simplistic data distribution as in the earlier model. The initial population constructor on the master (root) processor creates the initial population, which is then distributed to all nonroot processors. Following this, all nodes, including the root execute the serial GA on their allocated population for a predefined number of iterations called the MF. Then each node sends a certain number of its best solutions to the root. The number of solutions sent is controlled by the MR parameter. The root determines the MR best solutions from the collective MR * ($N$) solutions and broadcasts it to all processors. These migrants if not already present on the processors, are then absorbed into the existing population by weeding out and replacing the weakest solutions. Each processor then continues with the serial GA for another MF number of generations. Every interval between migrations, i.e., the length of time defined by MF number of generations is called an epoch. The stopping criterion is a predefined number of epochs.

It is important to note that the migrant absorption policy dictates the replacement of worst solutions with incoming migrants only if the migrants already do not exist within the population. Also, logically this model could represent a fully connected topology of nonhierarchical processing elements, which cooperate to determine the best MR solutions among themselves and absorb these into their existing populations.

### 23.5.5 Tabu Search

Conceptually, TS is an elegant combinatorial optimization method, which belongs to the class of local search techniques. It enhances the performance of a local search method by using memory structures, to control navigation of the search space. It uses a local or neighborhood search procedure to iteratively move from a solution $x$ to a solution $x'$ in the neighborhood of $x$, until some stopping criterion has been

```
ALGORITHM Multi – Deme_Parallel_GA
NOTATION
RANK: ROOT = Root Processor designated by Rank = 0
RANK: NON – ROOT = All other Processors designated by rank > 0
RANK: ANY = All processors, including Root
MF = Migration Frequency
MR = Migration Rate
N = Number of Processors
Epoch = Instances of Migration
EPOCH_MAX = Maximum Number of Migrations Stopping Criteria
Begin
(Multi – Deme_Parallel_GA)

FOR RANK:ROOT
Initial Population Constructor
Distribute Initial Population
ENDFOR RANK:ROOT

FOR RANK:ANY
  Receive Allocated Population
ENDFOR RANK:ANY

LOOP-A
    FOR RANK:ANY
    LOOP-B
      Serial GA on Allocated Population:
        Choice of Parents
        Crossover and Offspring Generation
        Fitness Calculation
        New Population Selection
      END LOOP-B IF [Num_Iterations > = MF]
      Send MR Best Solutions and Costs to ROOT
    ENDFOR RANK:ANY

  FOR RANK:ROOT
    Collect the best MR*N solutions
    Determine best MR distinct solutions
    Broadcast MR solutions
  ENDFOR RANK:0

  FOR RANK:ANY
    Receive MR Best Solutions
    IF [Received Migrants not present in existing Population]
    Replace Worst Solutions with Received Solutions
    ENDIF
  ENDFOR RANK:ANY

END LOOP-A IF [Epoch > = EPOCH_MAX]

FOR RANK:0
    Return Best solution.
ENDFOR RANK:0

End (Multi – Deme_Parallel_GA)
```

**FIGURE 23.85** Structure of the multi-deme parallel GA.

satisfied. To explore regions of the search space that would be left uncovered by the local search procedure and—by doing this—escape local optimality, TS modifies the neighborhood structure of each solution as the search progresses. The solutions admitted to $N^*(x)$, i.e., the new neighborhood, are determined through the use of special memory structures. The search now progresses by iteratively moving from one solution to another in $N^*(x)$.

One of the common types of short-term memory structures to determine which solutions comprise the neighborhood is the use of a tabu list. In its simplest form, a tabu list contains solutions that have been visited in the recent past (less than $n$ moves ago, where $n$ is the tabu tenure). Therefore, solutions in the tabu list are excluded from $N^*(x)$. However, tabu lists containing attributes are much more effective, although they raise a new problem. With forbidding an attribute, probabilistically more than one solution might be matched and declared tabu. Some of these solutions that must now be avoided might be of excellent quality and may not yet have been visited. To overcome this problem, aspiration criteria are introduced that allow the overriding of the tabu state of a solution and thus including it in the allowed set. A commonly used aspiration criterion is to allow solutions that are better than the currently best known solution.

#### 23.5.5.1 Parallel Tabu Search Taxonomy

Of the various heuristics covered, TS stands out uniquely being the only algorithm that employs memory systems to control navigation of the search space. However, with increasing problem sizes TS shows rapidly increasing runtimes, and as such can benefit from intelligent parallelization efforts.

Parallel TS has drawn the attention of many researchers, especially in comparison with similar acceleration strategies applied to other heuristics. Unlike Gas, however, the TS algorithm with its structure and process flow is highly sequential. The first reported studies were published in the early 1990s [11,12,13]. Crainic et al. [14] classified the different parallel TS heuristics based on taxonomy along three dimensions as enumerated below:

- The first dimension is **control cardinality**, where the algorithm is either 1-control, where one processor executes the search and distributes tasks to other processors or $p$-control, where each processor is responsible for its own search and communicates with other processors.
- The second dimension is **control and communication type**, where the algorithm can either follow a synchronous rigid or knowledge synchronization (KS) approach, or it can be asynchronous collegial (C), or knowledge collegial (KC). In a synchronous operation mode, the processes are forced to establish communication and exchange information at specific, explicitly defined points. In an asynchronous operation mode, the processes can independently decide on communication depending on the global characteristics of good solutions, the search strategy, and the possible content of that communication.
- The third dimension is **search differentiation** where the algorithm can be SPSS (single point single strategy), SPDS (single point different strategies), MPSS (multiple point single strategy), or MPDS (multiple point different strategies).

In addition to this type of classification, a more general category based on processor communication is also used. This divides various approaches as either synchronous or asynchronous. In the former, various processors working with the same solution communicate in a synchronous manner, where the managing processor orchestrates the activities of all others. In asynchronous strategies, each processor communicates independently of other nodes using either the master–slave or peer-to-peer model.

#### 23.5.5.2 Synchronous Parallel Tabu Search

In this approach, the master is primarily in charge of controlling movement in the search process, while the slaves are used for executing their assigned workload. Depending on the variants of this strategy, slave processors may start with either the same or different initial solution. After searching its allocated

```
Algorithm MasterProcess;
Begin
  Initialize parameters and data structures;
  S0 = Initial solution;
  BestS = S0;
  CurS = S0; /* Current solution */
  Send CurS to all slave processes;
  While not-time-to-stop
    Begin
      Wait for best moves from all slaves;
      Select the best move subject to tabu restrictions;
      Send the selected move to all slaves;
    End
  Force all slaves to stop;
  Return (BestS) /* of slave running on same machine */
End. /* MasterProcess */
```

**FIGURE 23.86** Synchronous parallel tabu search (TS): the master process.

```
Algorithm SlaveProcess;
Begin
  Initialize parameters and data structures;
  Wait for initial solution S0 from master process;
  BestS = S0;
  CurS = S0; /* Current solution */
  Repeat
    Wait for selected move from the master;
    Perform the move;
    Update tabu_list;
    Update BestS and CurS;
    Try all moves in partial neighborhood;
    Select best move and send it to the master;
  Until stop;
End. /* SlaveProcess */
```

**FIGURE 23.87** Synchronous parallel tabu search: the slave process.

part of the current neighborhood, each slave process reports its best move back to the master. The master process selects the best among these, subject to tabu conditions. If the stopping criteria are met then the search stops, otherwise the master determines a new set of moves and distributes them among the slaves, which continue with the search.

A more detailed view of this approach, from both the perspectives of the master and slave processors, is given in Figs. 23.86 and 23.87, respectively.

#### 23.5.5.3 Asynchronous Parallel Tabu Search

In this approach, each processor explores a subset of the neighborhood of its current solution. Each of these is competing with its neighbors (its adjacent processors) in finding a superior solution. When the stopping criteria are met, every processor reports its best solution. The general outline of this parallelization approach is given in Fig. 23.88. Similar asynchronous parallel TS implementations for the traveling salesman and quadratic assignment problems have been reported in Ref. [11].

Acceleration of TS through parallelization has been proved to be an effective strategy in numerous areas. Continuing with the literature on parallel TS, we cover its various applications and classify these according to Crainic's taxonomy.

Malek et al. [15] compared the performance of serial and parallel implementations of SA and TS for traveling salesman problem (TSP). The reported experiments were performed on a 10 processor Sequent

**Algorithm** AsynchronousParallelTabuSearch;
**Begin**
1. Construct initial solution and initialize parameters;
2. Explores own neighborhood;
3. Select best move subject to tabu restrictions;
4. Update tabu list;
5. Exchange current best solution with neighbors;
6. Update current solution based on received neighbor solutions;
7. **If** *time-to-stop* **Then Return** best solution;
8. **Goto** step 2

**End**

**FIGURE 23.88** Pseudocode for asynchronous parallel tabu search (TS) algorithm.

Balance 8000 computer. The authors reported that the parallel version of TS outperforms not only its sequential counterpart but also produced comparable or better results than the serial and parallel version of SA. Their strategy is synchronous following a 1-control, KS, SPDS approach.

To solve the flow shop sequencing problem, Taillard [12] employed a parallel implementation of the TS algorithm using a search space decomposition strategy. It is a 1-control, RS, SPSS algorithm. Battiti and Tecchiolli [16] used TS to solve the quadratic assignment problem (QAP) with hashing procedures. The scheme used is *p*-control, RS, MPSS. The authors report that the parallelization strategy is efficient and the average success time decreases with an increase in the number of processors.

Taillard [17] used parallel TS for vehicle routing problem. The parallelization is based on domain decomposition strategy, which is *p*-control, KS, MPSS. It was implemented on a Silicon Graphics 4D/35 workstation with four processors. Another effort by Taillard [18] to apply parallel TS to quadratic assignment problem follows a 1-control, RS, SPSS strategy. A ring of 10 transputers were used, but no implementation details were given. Chakrapani and Skorin-Kapov [19] also used parallel TS to solve QAP, which is 1-control, RS, SPSS. The search is performed sequentially, while the move evaluation is done in parallel. The implementation is specifically designed for connection machines CM-2: a massively parallel SIMD machine. The authors report that the best known solutions were obtained in a lesser number of iterations. Furthermore, they were able to determine good suboptimal solutions to bigger problems in reasonable time.

Another effort to parallelize TS for TSP by Fiechter [20] used the *p*-control, KS, MPSS strategy. Intensification and diversification steps were implemented in the synchronous version. The algorithm was implemented on a network of transputers arranged in a ring structure. The authors report near-optimal solutions to large problems and almost linear speedups. TS was also applied for the vehicle routing problem by Garica et al. [13] also using search space decomposition strategy. It was a 1-control, RS, SPSS algorithm. The authors reported a noticeable improvement in solution quality over one of the best constructive algorithms for vehicle routing problem, with substantial reduction in runtime.

To improve parallel TS using evolutionary principles, the algorithm presented by De Falco et al. [11] used multi-search thread strategy for the traveling salesman and quadratic assignment problems. It is a *p*-control, C, MPSS algorithm. The results reported indicate a marked improvement in solution quality as well as convergence speedup. Parallel TS for the 0–1 multidimensional knapsack problem was demonstrated by Nair and Freville [21], who used a multi-search threads strategy in a *p*-control, RS, MPSS algorithm. Taillard's extensive work with TS continued where he applied a *p*-control, RS, MPSS strategy to the job sequencing problem [22]. Several parallelization ideas that focused on speeding up computations related to neighborhood evaluation did not yield good results, either because the communication overtook computation, or the available computing platform (a ring of transputers, and a 2-processor Cray) were not suitable.

Crainic et al. [14], the authors who put forth the taxonomy of classification of TS, presented several of the strategies for both synchronous and asynchronous TS for multi-commodity location–allocation problems with balancing requirements. It was implemented on a heterogenous network of 16 SUN Sparc

workstations. The results show that the average gap improved in most of the cases, when the number of processors increased. Mori and Hayashim [23] used parallel TS algorithm for voltage and reactive power control in power systems. Of the two schemes, one of them used the domain decomposition strategy, whereas the other scheme followed a multi-search threads strategy. The first one is 1-control, RS, SPSS algorithm and the second one is *p*-control, RS, MPDS algorithm.

A more recent work by Yamani et al. [24] parallelized TS for VLSI cell placement on heterogenous cluster of workstations, using PVM. The algorithm was parallelized on two levels simultaneously. The higher parallelization level can be classified as *p*-control whereas the lower level was 1-control. The synchronization strategy was RS, and MPSS search differentiation strategy was used for both the levels. The authors reported obtaining proportional speedup in most of the cases.

## 23.5.6 Simulated Evolution

SimE is a powerful general iterative heuristic for solving combinatorial optimization problems. It starts from an initial assignment, and then, following an evolution-based approach, it seeks to reach better assignments from one generation to the next. It is stochastic because the selection of the components of a solution to be changed is done according to a stochastic rule. Already well-located components have a high probability to remain where they are. The probabilistic feature gives SimE its hill-climbing property.

SimE assumes that there exists a population $P$ of a set $M$ of $n$ (movable) elements. There is a cost function *cost* that is used to associate with each assignment of movable element $m$ a cost $C_m$. The cost $C_m$ is used to compute the goodness $g_m$ of element $m$, for each $m \in M$. This goodness value is closely related to the overall target fitness value of the solution.

SimE algorithm proceeds as follows. Initially, a population* is created at random from all populations satisfying the environmental constraints of the problem. The algorithm has one main loop consisting of three basic steps: evaluation, selection, and allocation. The three steps are executed in sequence until the population average goodness reaches a maximum value, or no noticeable improvement to the population goodness is observed after a number of iterations.

Parallelization of SimE has not attracted much attention from practitioners, with very few reported schemes mostly by the inventors of the algorithm themselves [25].

### 23.5.6.1 Domain Decomposition

Classified under the type-II scheme, this approach in SimE involves the partitioning of a complete solution into smaller domains to be optimized in parallel. This implies concurrent execution of all its operators, including allocation. Hence the search behavior of this approach would differ from that of the serial algorithm. Such a parallelization strategy was reported by Kling and Banerjee [26] for VLSI cell placement, where alternating sets of rows are distributed among processors in every iteration. With each processor limited to applying the SimE steps of evaluation, selection, and allocation on its assigned set of rows, this alternating distribution scheme would allow each cell to move to any position in the placement within two iterations. A variation of this scheme was reported much later, which proposed random assignment of rows to processors [27].

Figures 23.89 and 23.90 give the outlines of this parallelization strategy from both the master and slave perspectives. The slaves communicate back their assigned rows, modified by their respective allocation scheme to the master after every iteration. The master reconstructs the complete solution, computes overall fitness, and reassigns the rows for the next iteration.

This model is capable of achieving significant speedups for large problem sizes where such a row distribution scheme would provide a fair work distribution. However, the overall fitness values achieved by this method may vary from the serial values reached. In some cases, such as multi-objective VLSI

*In SimE terminology, a population refers to a single solution. Individuals of the population are components of the solution; they are the movable elements.

```
ALGORITHM TypeII_Parallel_SimE_Master_Process
NOTATION
  (* k_s: Set of row indices for each process s. *)
  (* Φ: The complete current solution. *)
INITIALIZATIONS;
  Read_User_Input_Parameters
  Read_Input_Files
  Construct_Initial_Placement
Begin
  Repeat
    ForEach s ∈ m Generate_Row_Indices k_s EndForEach;
  (* For each slave process. *)
    ParFor
      Slave_ Process (Φ, k_s)
  (* Broadcast cur. placement and row-indices. *)
    EndParFor
    ParFor
      Receive_Partial_Placement_Rows
    EndParFor
    Construct_Complete_Solution
  Until (Stopping Criteria is Satisfied)
  Return Best_Solution.
End. (*Master_Process*)
```

**FIGURE 23.89** Outline of master process for type II parallel SimE algorithm.

```
ALGORITHM Typell_Parallel_SimE_Slave_Process (Φ, k_s)
NOTATION
  (* B is the bias value. *)
  (* Φ^s are the rows assigned to slave s. *)
  (* m_i is module i in Φ^s. *)
  (* g_i is the goodness of m_i. *)
Begin
  Receive Placement_And_Indices
  EVALUATION:
    ForEach m_i ∈ Φ^s evaluate g_i EndForEach;
  SELECTION:
    ForEach m_i ∈ Φ^s DO
      Begin
        If Random > Min (g_i + B, 1)
        Then
          Begin
            S^s = S^s ∪ m_i; Remove m_i from Φ^s
          End
        End
      Sort the elements of S^s
  ALLOCATION:
      ForEach m_i ∈ S^s Do
        Begin
          Allocate(m_i, Φ_i^s. *)
        (* Allocate m_i in local partial solution rows Φ_i^s. *)
        End
      Send_Partial_Placement_Rows
      End. (*Slave_Process*)
```

**FIGURE 23.90** Outline of slave process for type II parallel SimE algorithm.

design, a loss in solution quality has been reported along with reduced runtimes with increasing number of processors [28].

#### 23.5.6.2 Multithreaded Parallel Search

As part of the type-III approach, this scheme implements parallel, independent search threads executed concurrently by each processor which may communicate periodically to exchange information and collectively navigate the search space. Such models have been reported with excellent results for other optimization algorithms such as SA and GAs.

Figure 23.91 demonstrates an example of such a parallel model, which is very similar to the AMMC strategy discussed earlier for SA. After a fixed number of user-defined iterations, a slave returns back its

```
ALGORITHM TypeIII_Parallel_SimE_Process
NOTATION
  (* Count is the current retry value. *)
Begin
  INITIALIZATIONS:
  Read_User_Input_Parameters
  Read_Input_Files
  Construct_Initial_Placement
Repeat
    EVALUATION:
      ForEach m_i ∈ Φ evaluate g_i;
    SELECTION:
      ForEach m_i ∈ Φ DO
        Begin
          If Random > Min(g_i + B, 1)
          Then
          Begin
            S = S ∪ m_i; Remove m_i from Φ
          End
      End
  Sort the elements of S
  ALLOCATION:
  ForEach m_i ∈ S Do
    Begin
    (* Allocate m_i in Φ_i. *)
    Allocate(m_i, Φ_i)
    End
  Calculate_Costs;
  If Cur_Cost > Best_Cost
  Then
    Begin
    Inform_Master;
    Count = 0
    End
  Else
    Count = Count + 1
  EndIf
  If Count > Retry_Threshold
  Then
    Begin
      If Cost_master < Cost_cur
        Then Get_New_Placement
      End
  Until (Stopping Criteria is Satisfied)
  End.
```

**FIGURE 23.91** Structure of the type III parallel SimE algorithm.

fitness value to the master to compare against the global best received thus far. This central processor then either provides a better solution, or directs the slave to provide the complete solution if the latter has higher fitness.

In spite of the success of Markov chain models with annealing and GAs, this scheme fails to perform with SimE [28]. The reason behind this is the nature of the heuristic and its intelligence. The primary concept behind this approach is to force each thread to explore a nonoverlapping part of the search space near the global best solution reached so far. However, with the selection operator heavily dependent on element goodness, and the deterministic allocation process, achieving such random behavior on individual processors is highly unlikely.

### 23.5.7 Conclusion

The increasing computational power of generic PCs today represents a fantastic opportunity for cluster systems wherein high performance computing environments can be assembled from off-the-shelf hardware. With growing standardization of parallel communication and computation libraries, out-of-the-box clustering solutions, and inexpensive, low latency networks, these computing platforms provide excellent avenues for accelerating performance and devising highly efficient algorithms.

In this chapter, we focused on four popular heuristics, which have been extensively used for optimization in numerous areas—SA, GAs, TS, and SimE. Different parallel strategies were discussed under the context of domain partitioning and multithreaded parallel search methods.

## Acknowledgment

The authors express their gratitude to the King Fahd University of Petroleum & Minerals for support under the project code # COE/CELLPLACE/263.

## References

1. Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. IEEE Computer Society Press, CA, December 1999.
2. Kirkpatrick S. Jr., Gelatt C., and Vecchi M. Optimization by simulated annealing. *Science*, 220 (4598):498–516, 1983.
3. Cerny V. Thermodynamical approach to the traveling salesman problem: An efficient simulated algorithm. *Journal of Optimization Theory Application*, 1(1):41–45, 1985.
4. Holland J.H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Harbor, MI, 1975.
5. Goldberg D.E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.
6. Cantú-Paz E. Designing efficient master–slave parallel genetic algorithms. *Genetic programming 1998: Proceedings of the Third Annual Conference*, 455 pp, 1998.
7. Grosso O.B. *Computer simulations of genetic adaptation: Parallel subcomponent interaction in a multilocus model*. Ph.D. Thesis, The University of Michigan, Michigan, MI, 1985.
8. Pettey C.C., Leuze M., and Grefenstette J.J. A parallel genetic algorithm. *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 155–161, 1987.
9. Braun H.C. On solving traveling salesman problems with genetic algorithms. H.-P. Schwefel and R. Männer, (Eds.), *Parallel Problem Solving by Nature*, Berlin, Germany, Springer-Verlag, 1991, Vol. 496, Lecture Notes in Computer Science, pp. 129–133.
10. Munetomo M., Takai Y., and Sato Y. An efficient migration scheme for subpopulation-based asynchronously parallel genetic algorithms. *Proceedings of the Fifth International Conference on Genetic Algorithms*, p. 649, 1993.

11. De Falco I., Del Balio R., Tarantino E., and Vaccaro R. Improving search by incorporating evolution principles in parallel tabu search. *Proceedings of the First IEEE Conference on Evolutionary Computation (ICEC'94)*, pp. 823–828, June 1994.
12. Taillard E. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 417:65–74, 1990.
13. Garica B.-L., Potvin J.-Y., and Rousseau J.-M. A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers and Operations Research*, 21 (9):1025–1033, 1994.
14. Crainic T.G., Toulouse M., and Gendreau M. Towards a taxonomy of parallel tabu search heuristics. *INFORMS Journal of Computing*, 9(1):61–72, 1997.
15. Malek M., Guruswamy M., Pandya M., and Owens H. Serial and parallel simulated annealing and tabu search algorithm for the travelling salesman problem. *Annals of Operation Research*, 21:59–84, 1989.
16. Battiti R. and Tecchiolli G. Parallel biased search for combinatorial optimization: Genetic algorithms and tabu. *Microprocessors and Microsystems*, 16:351–367, 1992.
17. Taillard E. Parallel iterative search methods for the vehicle routing problem. *Networks*, 23:661–673, 1993.
18. Taillard E. Robust tabu search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.
19. Chakrapani J. and Skorin-Kapov J. Massively parallel tabu search for quadratic assignment problem. *Annals of Operation Research*, 41:327–341, 1993.
20. Fiechter C.N.A parallel tabu search algorithm for large travelling salesman problems. *Discrete Applied Mathematics*, 51:243–267, 1994.
21. Nair S. and Freville A. A parallel tabu search algorithm for the 0–1 multidimensional knapsack problem. *The Eleventh International Parallel Processing Symposium*, April 1997.
22. Taillard E. Parallel tabu search techniques for the job sequencing problem. *ORSA: Journal on Computing*, 6(2):108–117, 1994.
23. Mori H. and Hayashim T. New parallel tabu search for voltage and reactive power control in power systems. *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems (ISCAS'98)*, pp. 431–434, May 1998.
24. Ahmad Al-Yamani, Sadiq M. Sait, Habib Youssef, and Hassan Barada. Parallelizing tabu search on a cluster of heterogenous workstations. *Journal of Heuristics*, 8:277–304, May 2002.
25. Kling R.M. and Banerjee P. Concurrent ESP: A placement algorithm for execution on distributed processors. *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 354–357, 1987.
26. Kling R.M. and Banerjee P. ESP: Placement by simulated evolution. *IEEE Transaction on Computer-Aided Design*, 3(8):245–255, 1989.
27. Sadiq M. Sait, Mustafa I. Ali, and Ali M. Zaidi. Multiobjective VLSI cell placement using distributed simulated evolution. *International Symposium on Circuits and Systems*, ISCAS, 6:6226–6229, May 2005.
28. Sadiq M. Sait, Mustafa I. Ali, and Ali M. Zaidi. Evaluating parallel simulated evolution strategies for VLSI cell placement. *The Ninth International Workshop on Nature Inspired Distributed Computing (NIDISC'06)*, Rhodes Island, Greece, April 2006.

# 24

# Internet Architectures




Borko Furht
*Florida Atlantic University*


## 24.1 Introduction

### 24.1.1 Computing Models for Internet-Based Architectures

The increasingly competitive global marketplace puts pressure on companies to create and deliver their products faster, with high quality and greater performance. To get the new products and technologies to consumers is through a new industry called application service providers (ASPs). Similar to Internet service providers, that linked businesses and consumers up to the Internet, ASPs lease software applications to businesses and consumers via the Internet. These applications range from word processing programs to payroll management software, document management systems, and many others. The major challenge is to develop an efficient Internet-based architecture, which will efficiently provide access to these software applications over the Internet.

Application architectures have traditionally followed software development architectures. The software development architectures can be classified into:

- Traditional desktop computing model
- Client-server computing model
- Network computing model
- Server-based computing model

*Traditional desktop computing model* assumes that the whole application is on the client and the application is executed locally. The client must be a "fat" client.

*Client-server computing model* assumes that clients are powerful and processing is centered around local execution on clients. Computer resources were split between a server and one or several clients. This architecture allowed for larger, more scalable, applications to be brought to a larger number of clients; however, the key for this architecture was to successfully partition the complexity of overall application and determine correctly which part should reside on the server and which part should run on the client. As more and more functionality migrated to the client, it became harder for applications to be maintained and updated.

*Network computing model,* supported by Sun, Oracle, Netscape, IBM, and Apple, assumes that software applications are dynamically downloaded from the network into the client for execution by the client. This architecture requires that the clients are fat.

*Server-based computing model*, supported by Citrix, assumes that business applications reside on the servers and can be accessed by users without requiring them to be downloaded to the client. The client can be either "thin" or "fat."

### 24.1.2 Server-Based Computing Model

The fundamental three elements of the server-based (or host-based) computing model are [1]:

- Multi-user operating system
- Efficient computing technology
- Centralized application and client management

*Multi-user operating system* allows multiple concurrent users to run applications in separate, protected sessions on a single server.

*Efficient computing technology* separates the application from its user interface, so only simple user's commands, received through keystrokes, mouse clicks, and screen updates, are sent via the network. As a result, application performance does not depend on network bandwidth.

*Centralized application and client management* allows efficient solution of application management, access, performance, and security.

A server-based computing model is very efficient for enterprise-wide application deployment, including cross-platform computing, Web computing, remote computing, thin-client device computing, and branch-office computing, as illustrated in Fig. 24.1 [1].

## 24.2 Evolution of Internet-Based Application Service Architectures

Similar to software development architectures, applications service architectures have emerged from the traditional client-server architectures to three-tier and multitier architectures.

The first generation of Internet-based application service architecture was based on delivery of information via public Web sites. This technology, sometimes referred to as the "first wave" Internet [2] employs the Web to present the information to the user and then allows the user to give some relevant information back. The primary focus of this architectural model is mass distribution of public information over the Internet. This architecture, which focuses on accessing information, consists of three levels (or three tiers)—presentation level, content level, and data and service level, as shown in Fig. 24.2 [2].

At the presentation level, there is the client system, which is used to view Web page information. The client contains both presentation and application logic components. At the content level, there is a Web server that provides interactive view of information from a relational database. Finally, at the data and service level, there is a relational database system, which provides data for the Web server. This architecture is also called three-tier architecture consisting of client tier, Web server tier, and database tier.

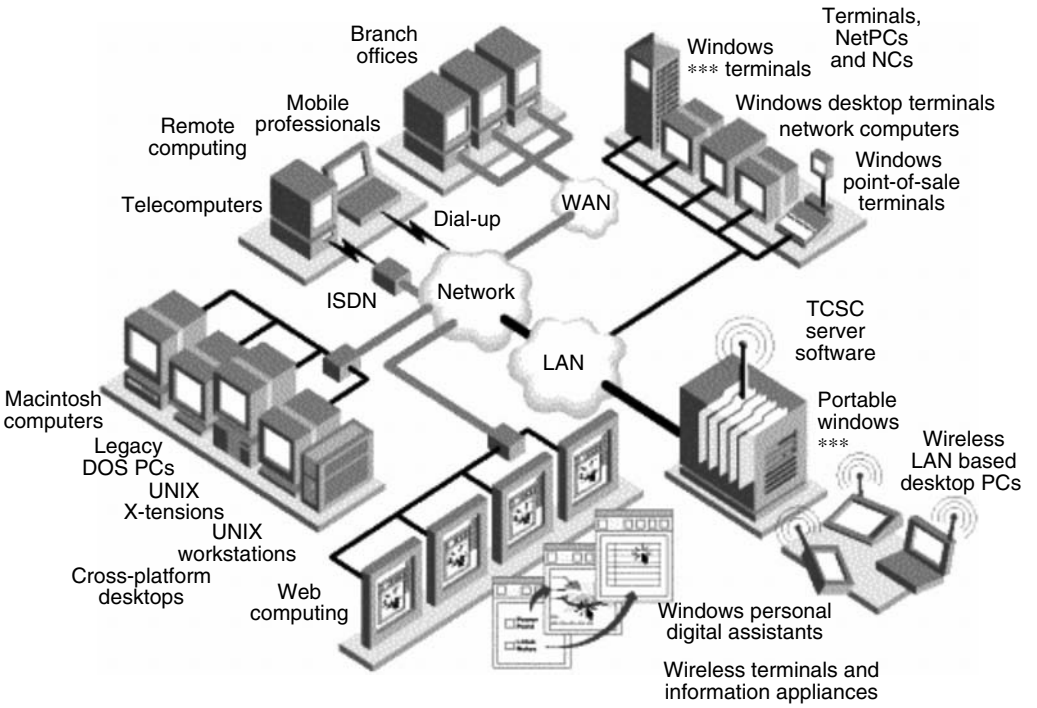



**FIGURE 24.1** Server-based computing models can be used for enterprise-wide application deployment.

With the advancements of the Internet, the Web, and related technologies (such as Java and HTML), as well as acceptance of standard communication protocols (such as TCP/IP and HTTP), a new architecture has emerged. In this architecture, sometimes referred as to the "second wave" Internet [2] or network-based application architecture [3], focus is on highly targeted, private distribution of software services over Intranets and Extranets. In this architecture, the Web page is not only the agent for providing information but also offers a variety of application services to speed up business transactions and offer additional services. This architecture consists of *n*-tiers and offers maximum functionality and flexibility in a heterogeneous Web-based environment. An example of four-tier architecture is shown in Fig. 24.3.

At the presentation level, the client views Web pages for information as well as for a variety of application services. At the second, content level, the Web server provides an interactive view of information and supports client-initiated transactions. At the third, application level, there is an application server, which is used to find requested data and services, makes them available for viewing, and carries out transactions. At the fourth, data and service level, there is a variety of data and services accessible by the application server. This architecture, also called multitier architecture, consists of client tier, Web server tier, application server tier, and database tier.

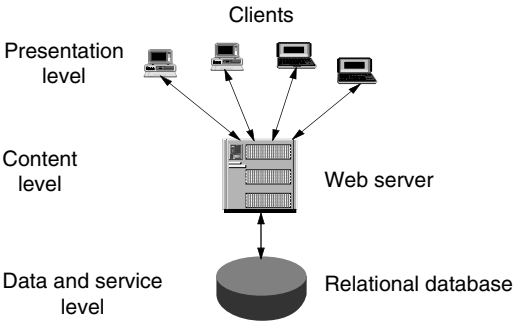



**FIGURE 24.2** The three-tier architecture for application service providers (ASPs) is focused on accessing information.

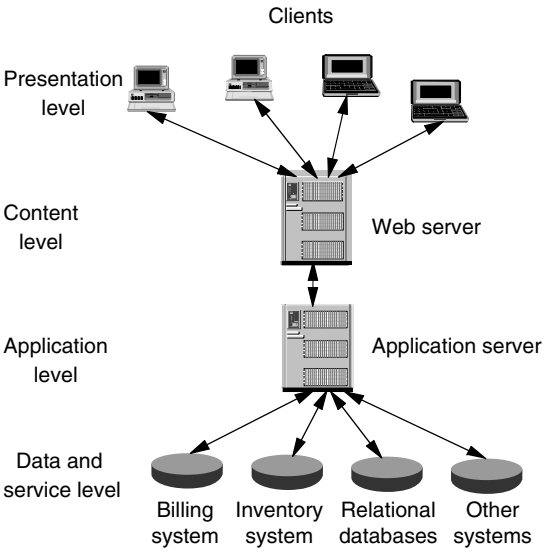



**FIGURE 24.3** The multitier Internet-based architecture for ASPs is focused on accessing application services.

Two-tier Internet architecture is typically limited for systems with a small number of users, a single database, and nonsecure network environments.

## 24.3 Application Server

In the second generation of Internet architectures, the focus has shifted to access to business services rather than to information only. The main component of the system is an application server, which searches for services and data—this is done in the background without involving the user.

The main challenges in developing the first generation of Internet architectures and application services were related to user interfaces and cross-platform interoperability. In developing the second generation of Internet architectures, the main challenge for service developers is to deliver their services seamlessly via the Internet, which in turn requires innovations in many areas. The following challenges need to be addressed in developing the second generation of Internet architectures:

- **Standards**. Many standards are used for developing Web pages, which causes difficulties for developers.
- **Increased programming complexity**. The implementation of business services on the Internet is a very complex programming task.
- **Network performance**. Business applications over Intranets and Extranets require very reliable and high-performance networks.
- **Security**. Business applications on the Internet require a very high level of security.
- **Web access to legacy applications**. As mentioned earlier, the new Internet architectures are focused on accessing various business applications rather than just information.
- **Database connection support across Web-based requestors**. Users should be able to access a variety of databases connected to the application server.

The majority of these functions, sometimes called *middleware*, are implemented in application servers that provide support for developing and deploying business applications located on the server or partitioned across client and server.

Application server offers support for developing and deploying business logic that may be located on the server or, more often, partitioned across client and server. Running business applications on the server provides many benefits [4].

### 24.3.1 Key Technologies for Application Servers

Key technologies for developing contemporary application servers include:

- Java programming language and environment
- JavaBeans—the Java-based component technology, which allows the development of new applications more rapidly and economically
- ActiveX—the competing technology to JavaBeans, which is Windows platform-dependent and language-independent
- Java Database Connectivity (JDBC)—the Java SQL that provides cross-platform database access for Java programs
- Java servlets—small Java routines that service HTTP requests and dynamically generate HTML
- Common object request broker architecture (CORBA)—provides a standard architecture for distributed computing and interoperability on the Internet

Java application servers have recently emerged as an efficient solution, with many features, for the application server tier. A Java application server:

- Makes it easy to develop and deploy distributed Java applications.
- Provides scalability, so hundreds to thousands of cooperative servers can be accessed from ten of thousands clients. Therefore, Java must be fully multithreaded and have no architectural bottlenecks that prevent scaling.
- Provides an integrated management environment for comprehensive view of application resources (e.g., Java Beans, objects, events, etc.), network resources (databases), system resources (ACLs, threads, sockets, etc.), and diagnostic information.
- Provides transaction semantics to protect the integrity of corporate data even as it is accessed by distributed business components.
- Provides secure communications.

CORBA and JavaBeans are open standards for component software development and deployment that allow writing small code objects that can be reused in multiple applications and updated quickly. They also allow developers to expose legacy system data and functionality as services available over the Web, and therefore most application servers are based on these technologies.

For example, the CORBA architecture makes it possible to find and use services over the Internet. Similarly, Enterprise JavaBeans is a standard server component model for Java application servers that provides services to network-enable applications, so that they may be easily deployed on Intranets, Extranets, and the Internet [5].

CORBA provides universal connectivity in broadly distributed environments as well as cross-platform interoperation of both infrastructures and applications. The object Web model based on CORBA and other standards is shown in Fig. 24.4 [7].

CORBA currently provides many services including naming, security, transactions, and persistence, as illustrated in Fig. 24.5 [7].

## 24.4 Implementations of Internet Architectures

In this section, four popular Internet architectures developed by Sun, Netscape, IBM, and Microsoft are presented.

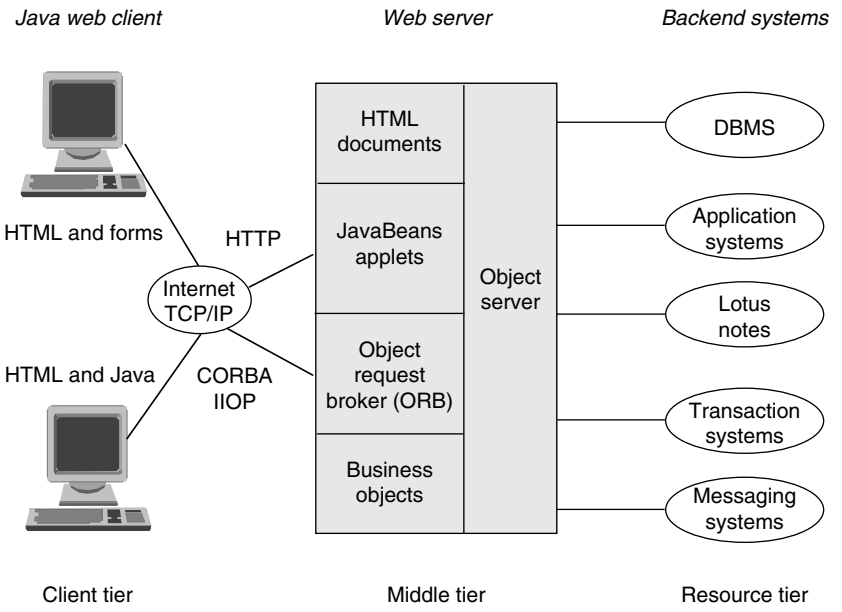



**FIGURE 24.4** The object Web model based on CORBA and other standards provides universal connectivity in distributed environments.

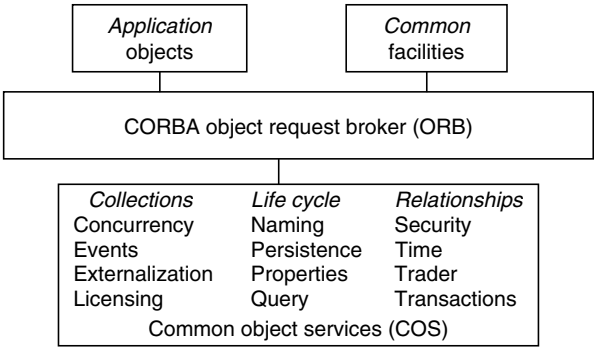



**FIGURE 24.5** CORBA provides a standard for interoperability that includes many services required by object applications.

## 24.4.1 Sun's Architecture

Initially, Sun Microsystems defined, in Fall 1996, Java-based application development architecture, which consisted of three tiers: the client tier that provided user interface, the middle tier for business logic and database access, and the database tier, as illustrated in Fig. 24.6 [8].

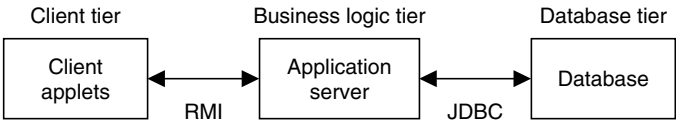



**FIGURE 24.6** Sun's Java-based three-tier architecture for ASP.

Sun selected Java language for the client tier, which provided more sophisticated GUI capabilities than HTML implementation. Client applets did not perform significant business logic functions in order to keep clients as thin as possible. Java technology was also used for the middle tier and the middle tier servers were implemented as stand-alone Java applications.

Because both client and middle tiers are implemented using Java, client middle tier communication was performed using remote method invocation (RMI), where the middle tier servers created the necessary RMI objects and made them available to clients via the RMI object registry. The middle tier communicated with the database via the JDBC API. This architecture is based on client-server computing model, in which client resides on the user's desktop, and the middle and database tiers reside on one or more of five data centers around the company.

Recently, Sun has developed an enhanced multitier architecture, which includes an additional tier—the WebTop server tier, as shown in Fig. 24.7 [8].

In the three-tier architecture (Fig. 24.6), applets were dynamically downloaded at runtime to the users' locations from an application server. For remote locations and modem connections with constrained bandwidth, applet download time was a few minutes, which was unacceptable.

Another issue related to three-tier architecture was the access to network resources such as files and printers. Java prohibits applets from accessing any local or network resources. In addition, Java does not allow communications with any machine other than the one from which the applet was downloaded. As a result of these limitations, file access occurred at the middle tier. This meant that information might be sent from the client to the middle tier and then back to a file server near the client.

Introducing a new tier, WebTop server tier, has resolved the issues related to the three-tier architecture. The WebTop server runs the Java Web server and is located near the users it serves. This server is used as a cache for applets and static application data, so the first problem was resolved. The server also supports services that access network resources such as user files and printers, which are typically located near the users. Finally, the WebTop server is used to find the services that users need.

In the architecture in Fig. 24.7 the client is thin and typically includes a graphical user interface written as an applet that runs from a Web browser. The application server tier provides access to data and implements business logic and data validation. The application server is responsible for all database transaction handling.

For the communication between the client and WebTop server tier and between the WebTop server and the application server tier, HTTP and RMI are used. Communication between application servers and databases is performed via JDBC.

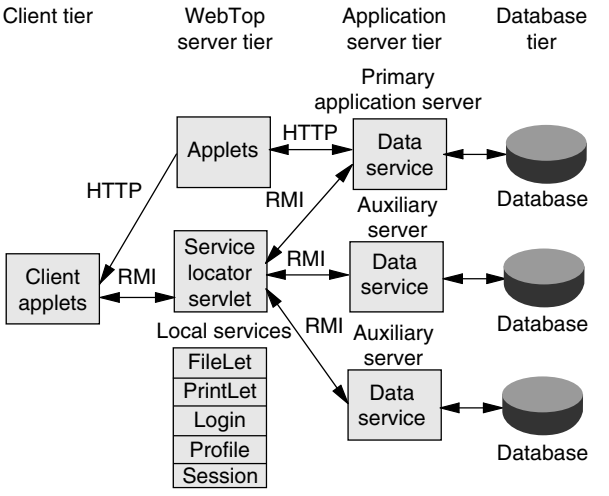



**FIGURE 24.7** Sun's Java-based multitier architecture for ASP.

One of the main benefits of the multitier architecture is that it increases application scalability and performance by enabling clients to be connected concurrently. In a client-server model clients are directly connected to databases, while in a multitier architecture only application servers connect directly to databases. In this way, the application server can process multiple requests from many clients through a pool of preallocated database connections, thus reducing the database server load. Load on the application server tier can be balanced by using multiple application servers.

Another benefit of the multitier architecture is that it supports thinner clients, because most of the logic runs in the application server and database tiers. Thus, broad range of client platforms can run the applications.

### 24.4.2 Netscape's Architecture

Similar to Sun's architecture, Netscape recently developed multitier architecture for application development and distributed computing, which is based on the separation of presentation logic from application logic, as illustrated in Fig. 24.8 [2].

In Netscape's multitier architecture, the client tier is typically based on an open-standard browser such as Netscape Navigator. The presentation logic and GUI is built using HTML pages that include Java applets. At the content level, a Web server primarily uses HTTP. It provides base-level information and content services as well as simple database information access via Java, JavaScript, and other high-level CGI scripting languages such as Perl.

The application server uses CORBA and JavaBeans components or objects. Transaction services enable access to relational databases and other legacy systems.

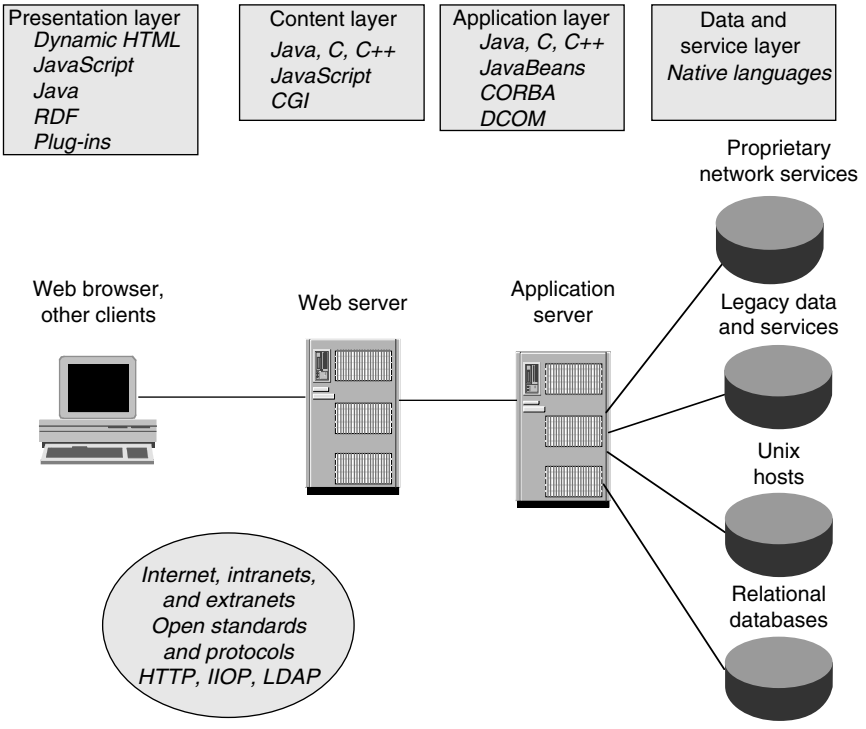



**FIGURE 24.8** Netscape's multitier architecture for ASP.

The first three levels in the multitier architecture in Fig. 24.8 are provided by Netscape technologies and products, while the last levels—back-end services and other legacy systems—are accessed through standard Internet interfaces.

### 24.4.3 IBM's Architecture

IBM has developed the Component Broker, which is Internet middleware for distributed objects [7]. Component Broker is a software system that allows developers to build, run, and manage Web-enabled business objects, components, and applications. Component Broker consists of:

- Tools for building distributed and business objects, and applications
- A runtime that provides a distributed-object infrastructure on the middle tier
- A system management functions for the distributed object runtime and its resources

Component Broker architecture, shown in Fig. 24.9, accepts inputs from any clients (Java or C++) transported via Internet InterORB Protocol, and ActiveX transported via a bridge. The object server consists of components that provide control, services, context, and connection resources.

The Component Broker receives client requests through the CORBA-compliant object request broker (ORB). Object services are supplied through the CORBA common object services (COS). These services provide object transaction services, database services, system services, and object management functions, as illustrated in Fig. 24.9.

Application adapters connect Component Broker object applications with existing software systems and applications.

### 24.4.4 Microsoft's Architecture

Microsoft Internet architecture is a component-based architecture based on Windows DNA [14]. The heart of Windows DNA is the component object model (COM) that allows developers to build applications from binary software components at any tier of the application architecture. These components provide support for packaging, partitioning, and distributing application functionality [14]. Distributed COM (DCOM) enables communications between COM components that reside on different machines. DCOM is a competing model for distributed object computing to CORBA, described in Section 24.3.

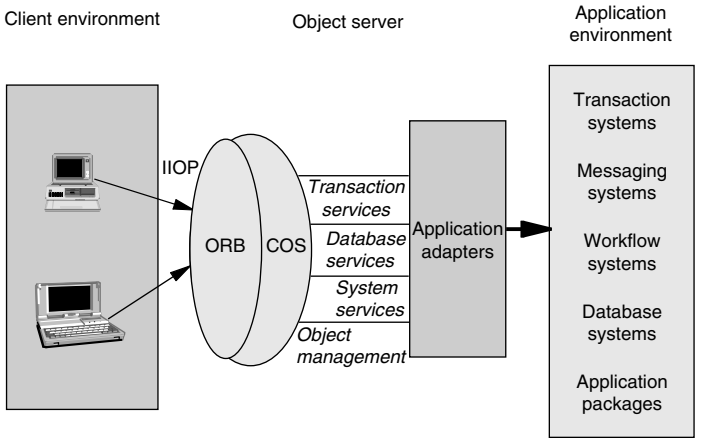



**FIGURE 24.9** Architecture of IBM's Component Broker at the middleware tier.

## 24.5 A Contemporary Architecture for Application Service Providers

In this section, ASP computing architecture using server-based computing model and the related ASP application architecture is presented.

### 24.5.1 ASP Computing Architecture

Our computing architecture for application service providers is based on the server-based computing model, described in the Section 24.1.1. As we indicated earlier, in server-based computing all applications and data are managed, supported, and executed on the server. This architecture provides the following benefits:

- Single-point management
- Predictable ownership costs
- High reliability
- Bandwidth-independent performance
- Universal application access
- Use of thousands of off-the-shelf applications
- Low-cost and fast application development
- Use of open standards
- Graphical and rich user interface
- Wide choice of client devices

The proposed server-based architecture uses two technologies developed by Citrix:

- Independent computing architecture (ICA)
- Windows-based terminal (WBT)

*Independent computing architecture* is a Windows presentation services protocol that turns any client device (thin or fat) into the thin client. The ICA consists of three components: server software, client software, and network protocol.

On the server, ICA separates applications from the user interface, while on the client users see and work with applications' interface. The application logic executes on the server. The ICA protocol transports keystrokes, mouse clicks, and screen updates over standard protocols requiring less than 20 kbps of network bandwidth.

*A Windows-based terminal* is a thin-client hardware device that connects to Citrix server-based system software. The WBT does not require downloading of the operating system or applications and there is no local processing of applications at the client, as in the case of other thin clients such as network computers or NetPCs. A WBT has the following features:

- An embedded operating system such as DOS, Windows CE, or any real-time operating system
- ICA protocol to transport keystrokes, mouse clicks, and screen updates between the client and the server
- Absolute (100%) execution of application logic on the server
- No local execution of application at the client device

The proposed architecture also allows consumers and business to access software applications from their Internet browsers. This is provided using Citrix's software *Charlotte*. In addition, software component *Vertigo* allows more interactive applications on the Web. This software allows customized Web pages such as electronic trading accounts to be updated automatically without hitting the refresh button on the computer.

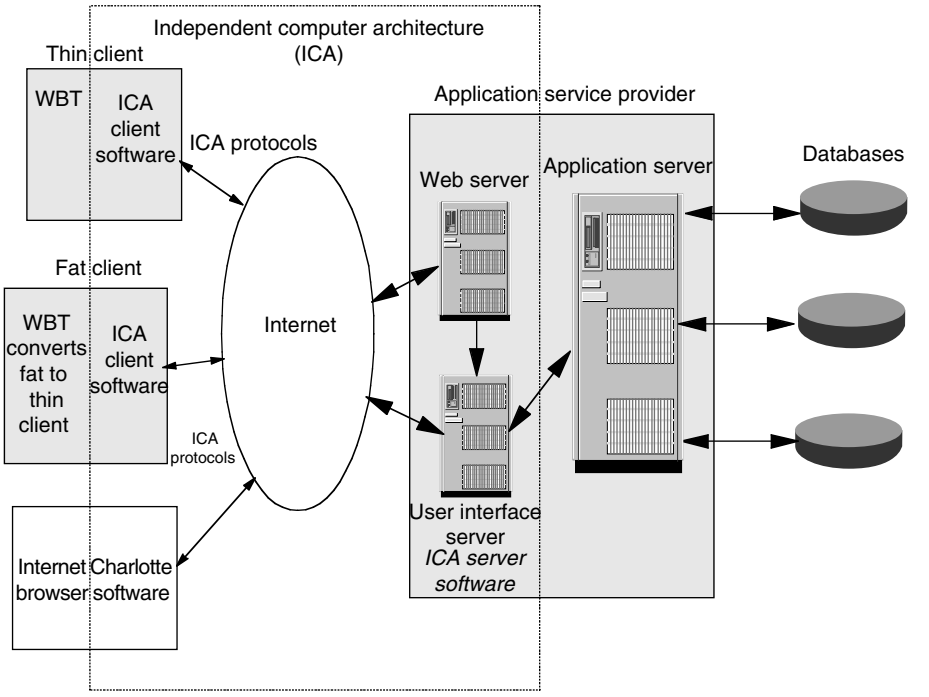



**FIGURE 24.10** The proposed architecture for ASP uses server-based model. All applications are executed at the server or cluster of servers.

The proposed architecture for ASP using server-based model and Citrix technologies is shown in Fig. 24.10.

The proposed architecture is platform independent and allows non-Windows and specialized ICA devises to run Windows applications residing and executing on application server farm. Application server farm is a group of application servers that are linked together as a single system to provide centralized administration and scalability.

The architecture in Fig. 24.10 allows ASPs to rapidly develop and deploy applications across complex computing environments. It also provides application access to all users, regardless of the their location, type of client device, or form of network connectivity. The architecture can be applied to any type of client hardware, and therefore requires no change in client hardware. The system significantly reduces requirements for network bandwidth compared to other architectures. Finally, the proposed architecture reduces the total cost of application, as analyzed in Section 24.6.

## 24.5.2 ASP Application Architecture

To take maximum advantage of ASP computing architecture, a new breed of applications needs to be developed. The key drivers of new distributed application architecture is a need for wide spectrum of thin clients, bandwidth usage optimization, application multi-identity shared back-end computing, reliable data flow management, security, legacy application integration, and long list of service operation requirements. The diagram shown in Fig. 24.11 can depict a desired architecture of an ASP application.

### 24.5.2.1 Client Software

ASP application client software is in general very different from types of client software provided as part of traditional client-server applications available on the market today. To support ASP business model,

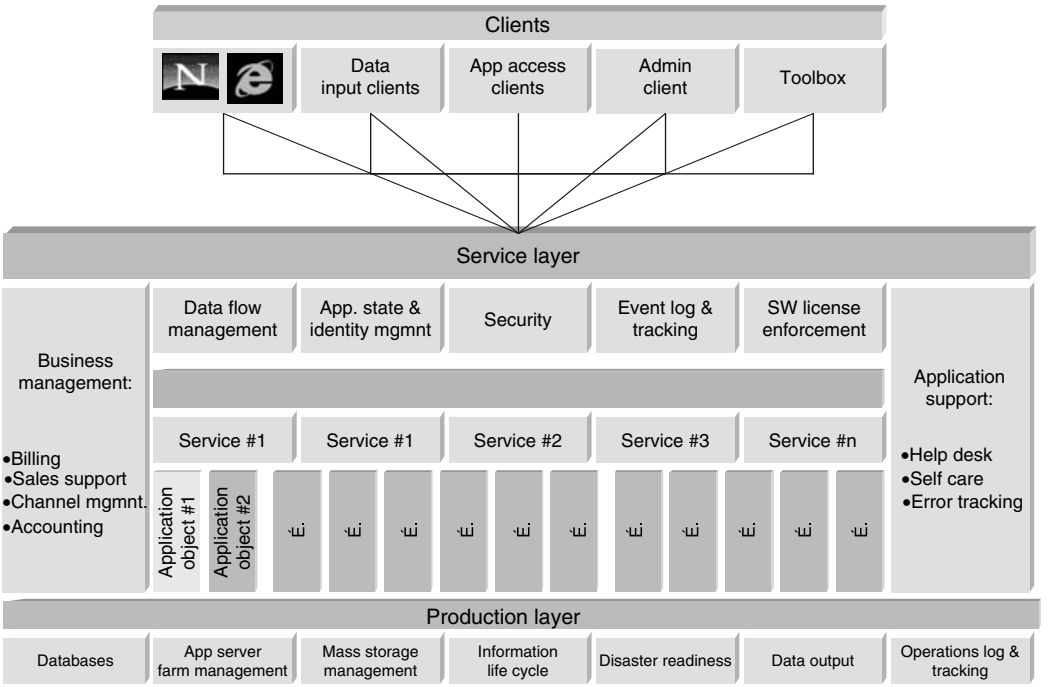



**FIGURE 24.11** Architecture of an ASP application.

client software must be "thin," i.e., requiring minimum computing power, installation and support effort, minimum communication bandwidth, and minimum version upgrade. Highly distributed nature of ASP service requires from client software ability to support versatile data inputs, highest level of user's security, and ability to support multiple communication protocols.

#### *24.5.2.1.1 Data Input*

ASP service architecture is in essence remote computing architecture, which requires capabilities to generate and import application data into the remote application. Data can be generated as part of specialized batch program or as by-product of third party software. Data input clients can be stand alone or integrated within other clients or legacy applications. Multistep data flow requires advanced information security, tracking, reporting, and above all ability to restore data in case that any stage system failure results in data loss. Data input clients may or may not be thin. The footprints of these clients are primarily defined by local functionality necessary to create the data at optimum cost.

#### *24.5.2.1.2 Application Access*

Application access clients are characterized by limited local computation capability and remote command capability to the server side application concentrated at service back end. These clients are the ones that should be as generic and as thin as possible. The smaller and simpler the client, the lesser the operational cost at the front end. The ideal application access client is plain Web browser. However, browser access is limited to very low level of functionality provided by HTML protocol. Function rich application computing requires specialized client software or plug-ins providing access to remote application at the back end.

#### *24.5.2.1.3 Toolbox*

To bridge the existing legacy applications with ASP service, an ASP application software requires a comprehensive set of APIs or application enabling tools providing the system integration capabilities and customizations.

#### *24.5.2.1.4 Administration*

This client should provide the end user with the ability to completely control its own application. Desired functions are: adding new users, setting up security profiles, managing application specific variables, usage tracking and reporting, and billing presentments and reporting.

#### *24.5.2.1.5 Security*

Client software security capability must include ability to authenticate users on the front end and to create virtual private channel of communication with the service back end.

### 24.5.2.2 Service Layer

Server side application is characterized by concentration of all computing and data intensive processes at back end, application multi-identity, sophisticated data flow management, and by its ability to integrate with business management, application support, and service production components. The ultimate goal of such application engineering is to create the fastest computing environment, economy of scale through all customers' sharing of common computing and data management infrastructure, and maximum operational readiness.

#### *24.5.2.2.1 Application Layer*

At the core of service layer is the application layer of software providing actual computing application packaged as specific service, for example: Service #1. This service application can be either stand-alone application or user interface into integrated solution based on several other independent third-party applications.

#### *24.5.2.2.2 Data Flow Management*

Data generated through data input clients is managed by data flow management software. One can consider this software component as a data switch capable of accepting data input, decompressing and decoding data, identifying the owner of data and target data base, importing data in the target data base, cashing and mirroring data at each stage for disaster readiness reasons, and creating logs for data input tracking and reporting.

#### *24.5.2.2.3 Application State and Identity Management*

An ASP provider will have many different applications for many different customers simultaneously. Also, each individual application will have many different users requiring different application setup and profile. Application state and identity management software acts as an application switch identifying individual users and applications and then assigning the appropriate user's profile. Therefore, ASP applications must support multiple identity capability. Ability to share the same computing and data management resources between many different users and applications is essential for reliable service delivery and economy of scale.

#### *24.5.2.2.4 Business Management*

The ASP application should also integrate into business management software enabling automatic account creation and usage data feed into billing solution.

#### *24.5.2.2.5 Application Support*

The ASP application should also integrate with application support solution that consists from customer self support site.

## 24.6 Evaluation of Various Architectures

Analysts and IT professionals have developed numerous models for estimating the total cost of IT services, sometimes called "total cost of ownership (TCO)." In the past, these models had the hardware-centric view because they analyzed the costs of owning and maintaining desktop computer hardware. In the age of the Internet, Web-based computing, and E-commerce, applications must be accessible across a wide variety of connectivity options, from low-speed, dial-up connections to wireless, WAN, and

Internet connections. A contemporary cost analysis should consider the total cost of application ownership (TCA), rather than the total cost associated with specific computing devices. The Tolly Group has developed a model for comparing the TCA of different computing models, discussed earlier [9]. We present and discuss their results in this section.

In order to determine the cost of application deployment, four computing models introduced in Section 24.1 can be analyzed from the following points of views:

- Physical location of the application
- Execution location of the application
- Physical location of data
- Location of the user and means of connectivity

The cost of complexity of deploying and managing an application strongly depends on physical location of the application. The cost of application distribution, installation, and managing of updates must be considered.

The choice of where an application is executed determines the hardware, network, and connectivity costs. An application can run on the server, on the client, or in a distributed server-client environment. In some cases, the application must be downloaded from a server to a client, which has an impact on performance and productivity.

The location of stored data determines the speed at which information is available. It also has an impact on the cost related to protecting and backing up critical corporate data.

The location of the user and the means of connectivity also have an impact on the cost and complexity of deploying an application.

Table 24.1 summarizes the application deployment characteristics for four computing models introduced in Section 24.1 [9].

Tolly Group has analyzed and calculated the total cost of application ownership for a medium-size enterprise of 2500 users, with 175 mobile users working on the road. The calculated costs were divided into (a) Initial (first-year) cost (which includes hardware, software, network infrastructure, and user training) and (b) annual recurring costs (which includes technical support and application maintenance). The results of analysis are presented in Fig. 24.12.

Traditional desktop computing approach requires relatively high initial cost for hardware, software, network infrastructure, and training ($14,000) as well as very high annual recurring costs for technical support and application maintenance ($11,000 annually).

Client-server and network computing approaches require slightly higher initial investment ($16,000) in order to replace existing client hardware; however, annual recurring costs are reduced ($9,500). This model becomes less expensive than the traditional desktop model from the third year forward.

The server-based approach gives the best TCA both in terms of initial costs and annual recurring costs ($6,000 and $2,600, respectively). The reason for it is that this model allows any type of client to access any application across any type of connection. This model also provides single point for the deployment and management of applications.

**TABLE 24.1** Computing Models and Application Deployment Characteristics

| | Application Location | Application Execution | Data Location | User Access | Network Requirements |
|---|---|---|---|---|---|
| Traditional desktop | Client | Client | Client | Local | None |
| Client-server | Client and server | Client and server | Client and server | Lan, WAN, Internet | High bandwidth |
| Network-based | Server | Client and server | Server or client | LAN, WAN, Internet | High bandwidth |
| Server-based | Server | Server | Server | LAN, WAN, Internet | Low bandwidth |

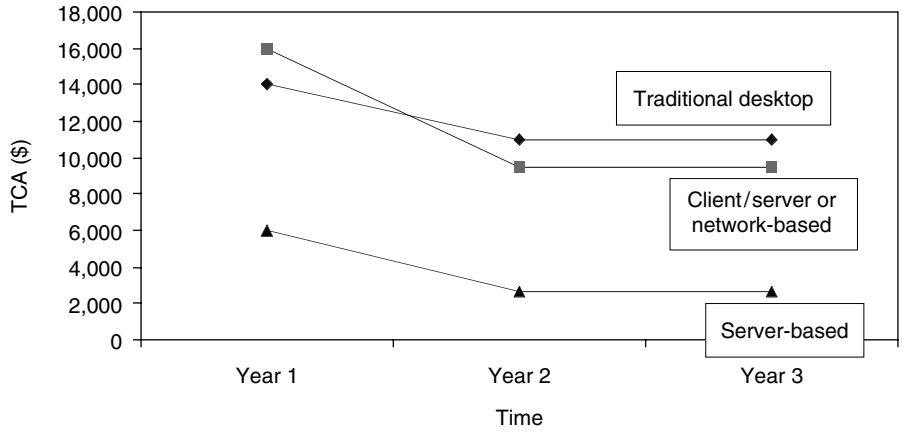



**FIGURE 24.12** Analysis of total cost of application (TCA) for various computing approaches [9].

In summary, the server-based model, which was applied in our architecture, is the most efficient and cost-effective solution to application deployment and management.

## 24.7 Conclusions

This chapter presented and evaluated contemporary multitier Internet architectures, which are well suited for distributed applications on the Internet including ASPs. The chapter also evaluated several computing models for Internet-based architectures and proposed a server-based computing model, which has a number of advantages over the other models.

### References

1. Citrix Systems, "Server-based computing," white paper, www.citrix.com, 1999.
2. P. Dreyfus, "The second wave: Netscape on usability in the services-based Internet," *IEEE Internet Computing,* Vol. 2, No. 2, March/April 1998, pp. 36–40.
3. Sun Microsystems, "Software development for the Web-enabled enterprise," white paper, 1999.
4. BEA WebLogic, "What is a Java application server?" weblogic.beasys.com, 1999.
5. A. Thomas, "Selecting enterprise JavaBeans technology," WebLogic, Inc., Boston, MA, July 1998.
6. R. Orfali, D. Harkey, and J. Edwards, *Instant CORBA*, John Wiley & Sons, New York, 1997.
7. C. McFall, "An object infrastructure for internet middleware: IBM on component broker," *IEEE Internet Computing*, Vol. 2, No. 2, March/April 1998, pp. 46–51.
8. Gupta, C. Ferris, Y. Wilson, and K. Venkatassubramanian, "Implementing Java computing: Sun on architecture and application development," *IEEE Internet Computing*, Vol. 2, No. 2, March/April 1998, pp. 60–64.
9. The Tolly Group, "Total cost of application ownership," Manasquan, NJ, white paper No. 199503, June 1999.
10. J.B. Eichler, R.Y. Roberts, K.W. Evans, and A.L. Carter, "The Internet: redefining traditional business and giving rise to new ones," Report, Stephens, Inc., Little Rock, AR, May 1999.
11. D. Rosenberg, "Bringing Java to the enterprise: Oracle on its Java server strategy," *IEEE Internet Computing*, Vol. 2, No. 2, March/April 1998, pp. 52–59.
12. M. Benda, "Internet architecture: its evolution from an industry perspective," *IEEE Internet Computing*, Vol. 2, No. 2, March/April 1998, pp. 32–35.
13. Sun Microsystems, "Enterprise JavaBeans technology: server component model for the Java platform," white paper, java.sun.com, 1999.

14. G.R. Voth, C. Kindel, and J. Fujioka, "Distributed application development for three-tier architectures: microsoft on Windows DNA," *IEEE Internet Computing*, Vol. 2, No. 2, March/April 1998, pp. 41–45.
15. C.J. Woodard and S. Dietzen, "Beyond the distributed object decision: using components and Java application servers as a platform for enterprise information systems," *Distributed Computing,* 1998.
16. G. Pour and J. Xu, "Developing 3-tier Web-based enterprise applications: integrating CORBA with JavaBeans and Java servlets," in *Proceedings of the 3rd International Conference on Internet and Multimedia Systems and Applications,* Nassau, Bahamas, October 1999.
17. L. Downes and Chunka Mui, "Unleashing the Killer App," Harvard Business School Press, Boston, MA, 1998.

# 25

# Microelectronics for Home Entertainment




Yoshiaki Hagiwara
*Sony Corporation*


## 25.1 Introduction

The history of home entertainment consumer electronics begins in May 7, 1946, with the founding of Tokyo Tsushin Kogyou (Tokyo Telecommunication Engineering) by Masaru Ibuka (36) and Akio Morita (25) in Tokyo, Japan. Had these two bright young men not met and combined their considerable resolve and talents, the home electronics business would not have accelerated so much as we see it today, and our semiconductor business efforts would have been aimed only for military purposes for a while.

In the Founding Prospectus, Ibuka eloquently stated his dreams for the company. Morita, together with the company's first directors headed by Kazuo Iwama, led employees to realize these goals. Throughout their work, the young force was inspired by the free and dynamic atmosphere of the "ideal" factory they were striving to create. From the onset, Ibuka, Morita, and Iwama endeavoured to develop unique and exciting products that fulfil their customers' dream.

Iwama was 35 when he visited Western Electric to study transistors in January 1954. Iwama was the first engineer in Japan who understood the concept of "electron fog" in the bipolar transistor device physics.

He worked as the leader of the bipolar transistor development project to realize the epoch-making portable bipolar transistor radio TR-55 introduced to the home entertainment electronics market in August 1955.

Seven years had passed since the invention of the bipolar transistor in Bell Lab in December 1947. I was only seven years old and had no idea about how a transistor works at that time.

I was a junior undergraduate at CalTech in Pasadena, California, in 1969 when I learned how the bipolar transistor and MOSFET work with the classical textbook by Grove. My class instructor was Prof. James McCaldin who was known as the co-inventor of basic planar passivation technology in modern MOS transistor fabrications.

In the summer of 1971, I visited Sony Atsugi plant right after I received a B.S. from CalTech and worked as a reliability engineer in Bipolar IC production line for Sony's Trinitron color TV sets.

In the fall of 1971, I returned to CalTech to pursue further my graduate work and learned how to design MOS LSIs from Professor Carver Mead. My Ph.D. thesis was about the buried channel CCD imagers, which can be applied to low light intensity solid state imagers. Prof. T.C. McGill was my Ph.D. thesis advisor.

After defending my Ph.D., in February 1975, I joined Sony at the Central Research Center in Yokohama, Japan, and engaged in further research on high-performance CCD imagers project headed by Iwama who was the pioneer engineer in the early bipolar technology development effort in Sony.

My first patent filed in Sony in November 1975 was about a simple pnp-substructure used as the light sensing device for imagers. The sensor structure is now called the HAD sensor in Sony's current video cameras and digital still cameras.

Sony put most of its engineering sources in CCD imagers and camera systems in 1970s. We engineers had to design signal processing and camera control chips by ourselves. Those experiences were useful to apply to other MOS LSI design applications, which made possible the current home entertainment LSI chip sets such as digital cameras, home robots, and games.

In this chapter, some basic semiconductor device concepts are first reviewed briefly. They are about the concept of "electron fog," the bipolar and MOSFET device model, the buried channel CCD imager structure, and the pnp-substructure which is used as the light sensing device, which is now universally adopted in most of high performance solid state imagers. Then, some general discussions on the product specifications and performance aspects of the home entertainment consumer LSI chip sets such as for digital cameras, home robotics, and games are presented in detail.

## 25.2 Basic Semiconductor Device Concepts

In this section, some introductory comments on the basic semiconductor device concepts are explained. They are strongly related to the microelectronics of the present home entertainment LSI chips.

### 25.2.1 Concept of Electron Fog

Figure 25.1 shows the electron fog in metal and semiconductor. Electrons in metal are depicted in this picture as the moisture above the water surface in the container, while the electrons in the

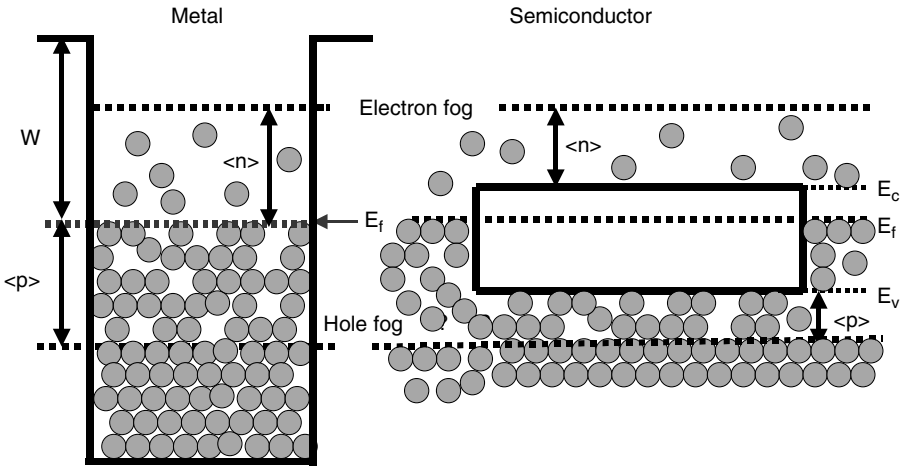



**FIGURE 25.1** Electron fog model in metal and semiconductor.

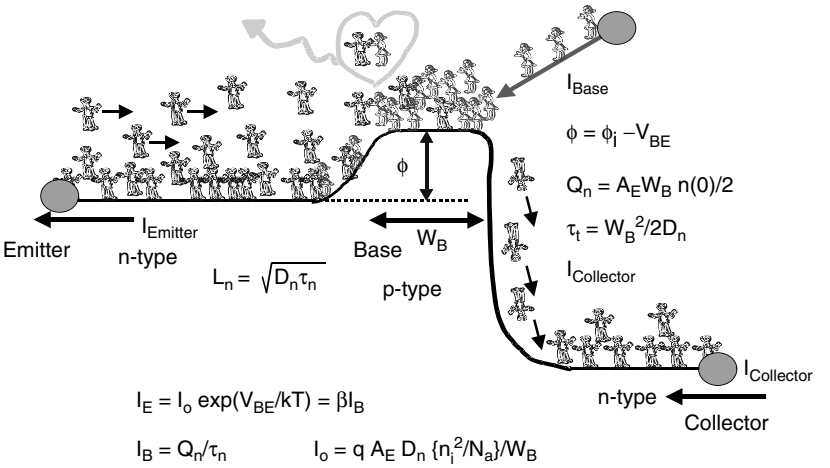



**FIGURE 25.2** Bipolar transistor action.

semiconductor are depicted as the moisture on the top of a floating box in water. If the box is heavy, the water surface is very close to the top of the box and there is a lot of moisture.

This corresponds to the n-type semiconductor band diagram. If the box is relatively light, only a small bottom portion of the box is submerged into the water and the top of the box can be quite dry, and there will a lot of bubbles (holes) under the bottom of the box. This corresponds to the p-type semiconductor.

Applying these p- and n-type semiconductor box models, a diode behavior model can be constructed and the diode rectifying characteristics can be explained.

## 25.2.2 Bipolar Transistor Device Model

Figure 25.2 shows energetic boys (electron fog in the emitter region) trying to climb a hill (base region) to catch the girls on the hill (hole fog, which is the majority carrier in the base region). Some of the boys can luckily catch girls on the hill, recombine, become happy and disappear as light or heat energy. But the hill width is very short and most of the boys will not have enough time to catch girls and fall down the cliff (the base-collector depletion region). The poor boys are now collected deep down the cliff in the collector region.

In the time interval $\Delta t$, $I_E \Delta t$ boys are jumping to the hill to catch girls on the hill. Some boys are lucky enough to catch girls on the hill. The number of girls caught by the energetic boys in $\Delta t$ is $I_B \Delta t$, which is proportional to the number of the average boys on the hill $Q_n$. The girls are supplied as the base current $I_B$. Other salient physical parameters normally used in the bipolar transistor device modeling are also given in the figure.

## 25.2.3 MOSFET Model

Figure 25.3 shows a MOSFET structure. If you see how the electron fog moves from the left source n+ region to the right n+ region through the $Si-SiO_2$ surface under the MOS gate, one can see that it is also considered as an electron transportation along an npn-structure. In this case, however, the potential in the p-region is controlled by the gate voltage isolated by the thin oxide.

The figure shows the electron fog moving from the source to the region under the gate at the onset of strong inversion at the $Si-SiO_2$ surface. At this point the electron fog density at the channel is equal to the density of the majority "hole fog" in the p-type Si substrate, and the gate voltage at this point is defined to be the threshold voltage $V_{th}$ of the MOSFET.

Figure 25.4 shows water flowing from the right source region to the left drain region through the water gate. The depth of the channel $V_{ch}$ is given as $(V_g - V_{th})$, where $V_g$ is the applied gate voltage

$$V_{th}(V_{BB},V_S) = V_{FB} + \{B + V_S - V_{BB}\} + \gamma\sqrt{\{B + V_S - V_{BB}\}}$$

$$V_{FB} = V_{BB} - (kT/q)\ \ln\{N_c\ N_A/n_i^2\} - \{\chi_{si} - \phi_m\}/q + Q_{SS}/C_{OX}$$

$$K = q\ \varepsilon_{Si}\ N_A/C^2_{OX} \qquad \gamma = \sqrt{2K} \qquad B = 2(kT/q)\ \ln(N_A/n_i)$$

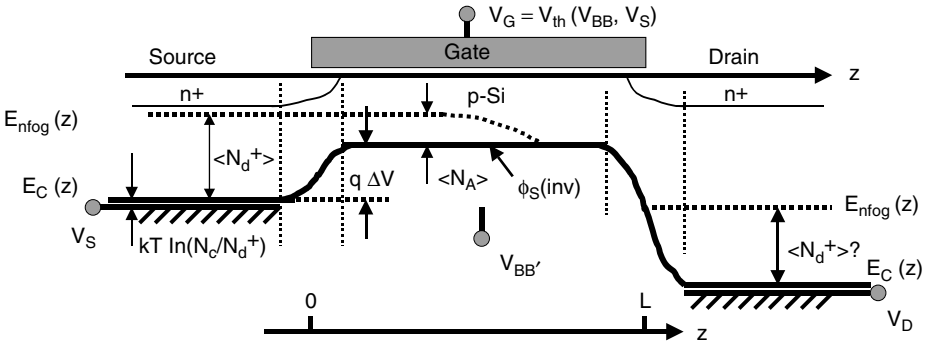
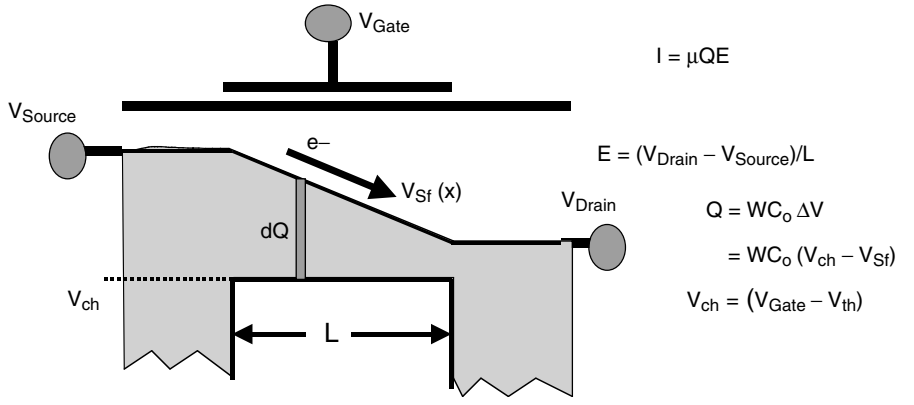



**FIGURE 25.3** MOSFET at Onset $V_G = V_{th}$.




$$I = \mu QE$$

$$E = (V_{Drain} - V_{Source})/L$$

$$Q = WC_o\,\Delta V$$

$$= WC_o\,(V_{ch} - V_{Sf})$$

$$V_{ch} = (V_{Gate} - V_{th})$$

$$\langle V_{Sf}\rangle = (V_{Drain} + V_{Source})/2$$

$$Q = WC_o\{V_{ch} - \langle V_{Sf}\rangle\}$$

$$= WC_o\ \{V_{Gate} - V_{th} - (V_{Drain} + V_{Source})/2\}$$

$$I = (W/2L)\ \mu C_o\{2(V_G - V_{th}) - (V_{Drain} + V_{Source})\}\ (V_{Drain} - V_{Source})$$

$$I_{Sat} = (W/2L)\ \mu C_o(V_{Drain} - V_{Source})^2 \text{ for } V_G > (V_{Drain} + V_{th})$$

**FIGURE 25.4** MOSFET I-V characteristics.

which induces the channel depth $V_{ch} = (V_g - V_{th})$. The amount of the water flow $I$ is proportional to the mobility $\mu$, the water amount $Q$ under the gate and the electric field $E$, i.e., $I = \mu QE$ can be written in this rough approximation.

In the first approximation, take $E = (V_d - V_s)/L$, where $V_d$, $V_s$, and $L$ are the drain voltage, the source voltage, and the gate channel length. The total charge can be approximated as $Q = WC_o\Delta V$, where $W$ and $C_o$ are the channel width and the oxide capacitance of the actual corresponding MOSFET transistor, respectively. Now, $\Delta V$ corresponds to the voltage difference between the average water surface $(V_d + V_s)/2$ and the channel potential $V_{ch} = (V_g - V_{th})$.

That is, $\Delta V = (V_d + V_s)/2 - V_{ch}$. Hence, since $Q = WC_o\Delta V$, the equivalent amount $Q$ of the water (or charge) under the gate is given as $Q = WC_o[(V_d + V_s)/2 - V_{ch}]$, where $V_{ch} = (V_g - V_{th})$, $E = (V_d - V_s)/L$. Now if these relationships are put into the original equation $I = \mu QE$, this leads, without going through the calculations normally done in the classical gradual channel approximation, finally to the classical MOS I-V equation:

$$\begin{aligned} I &= (W/2L)\mu C_o[V_d + V_s - 2V_{ch}](V_d - V_s) \\ &= (W/2L)\mu C_o[V_d + V_s - 2(V_g - V_{th})](V_d - V_s) \end{aligned}$$

## 25.2.4 Buried Channel CCD Structure

Figure 25.5 shows the physical structure and the potential profile of a buried channel CCD. The signal charge is the electron fog in the lightly doped n-region at the surface. As you can see, these signal charges are isolated from the direct contact to the Si−$SiO_2$ interface and do not suffer the charge trapping. This structure gives a good CCD charge transfer efficiency of more than 99.9999% along the buried channel CCD shift register in the direction of this chapter. At very high light, excess charge can be drained into the substrate by lowering the well voltage $V_{well}$ or making the substrate voltage very deep and inducing the punch-through mode in the n-p-n(sub) structure.

High-density and high-performance, solid-state imagers became available applying this structure as the scanning system. The surface n-layer is completely depleted when there is no signal charge. It is dynamically operated.

It is considered as one extended application of dynamic MOS device operations. The most well-known dynamic operation of a MOS device application is the DRAM data storage operation.

## 25.2.5 HAD Sensor, a pnp-Substructure

The floating diode structure for image sensing unit was well known in early 1970s. The author simply proposed to use a pnp-substructure instead for the imaging element. Figure 25.6 shows the proposed structure.

It is a simple pnp bipolar transistor structure itself with a very lightly doped base region, operated in the strong cut-off mode with the base majority charge completely depleted.

It is the first practical application of the bipolar transistor in dynamic operation mode, which turned out to be the best structure and way to convert photons to electrons for imaging including the current

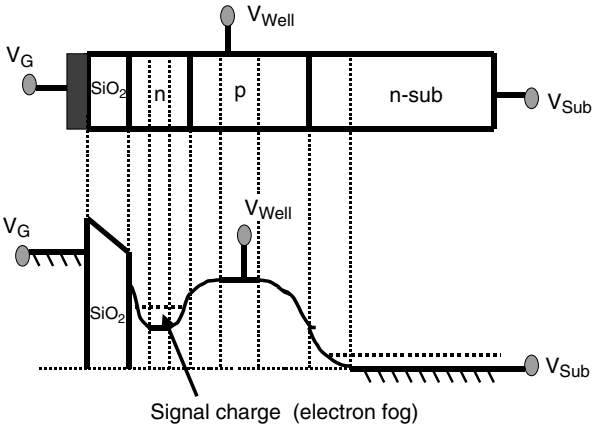



**FIGURE 25.5** Buried channel CCD structure.

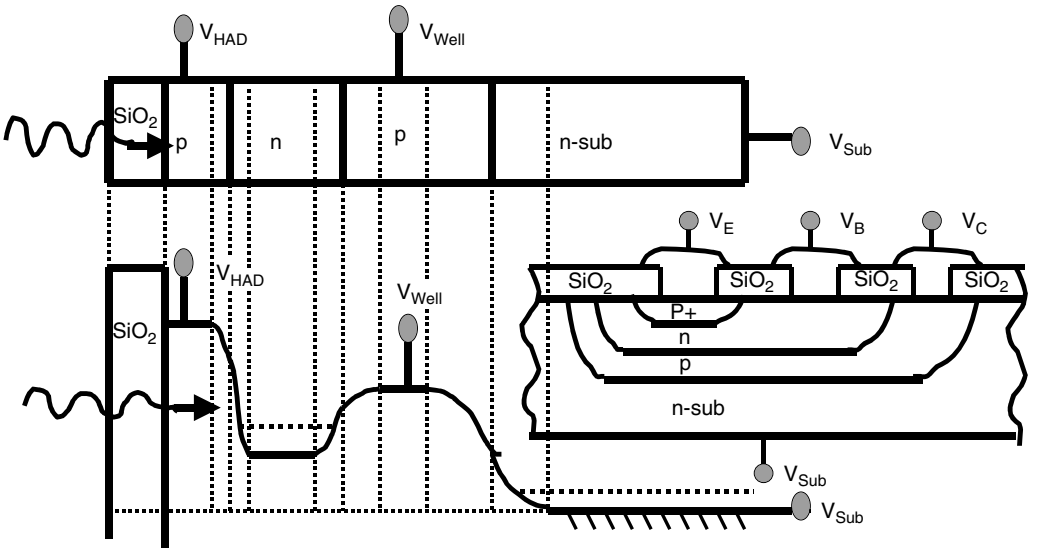



**FIGURE 25.6** A typical PNP bip Tr structure in the early 1970s, and a proposed application as an image-sensing element in 1975.

MOS imagers applications. The sensor structure is now called the HAD sensor in Sony's current video cameras and digital still cameras.

## 25.3 LSI Chips for Home Entertainment

### 25.3.1 Digital Still Camera

The picture in the Fig. 25.7 shows a 2/3 in. 190 K pixel IT CCD imager, ICX016/XC-37, which the author designed when he was still a young CCD design engineer in early 1981. This model became the model of the world's first consumer CCD video camera for mass production in 1983.

The goal now is to become "Imaging Device No. 1!" Many applications of CCD and LCD are used, as seen in Fig. 25.8.

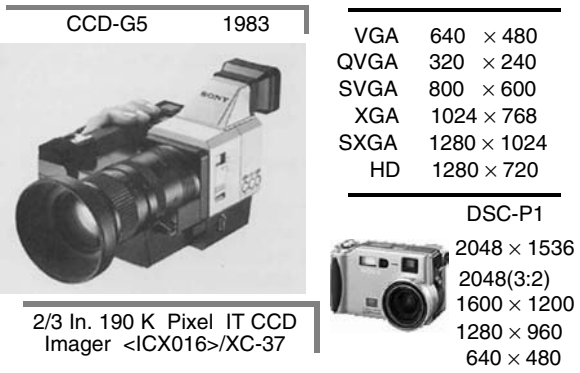



**FIGURE 25.7** The world's first consumer CCD video camera for mass production 1983.

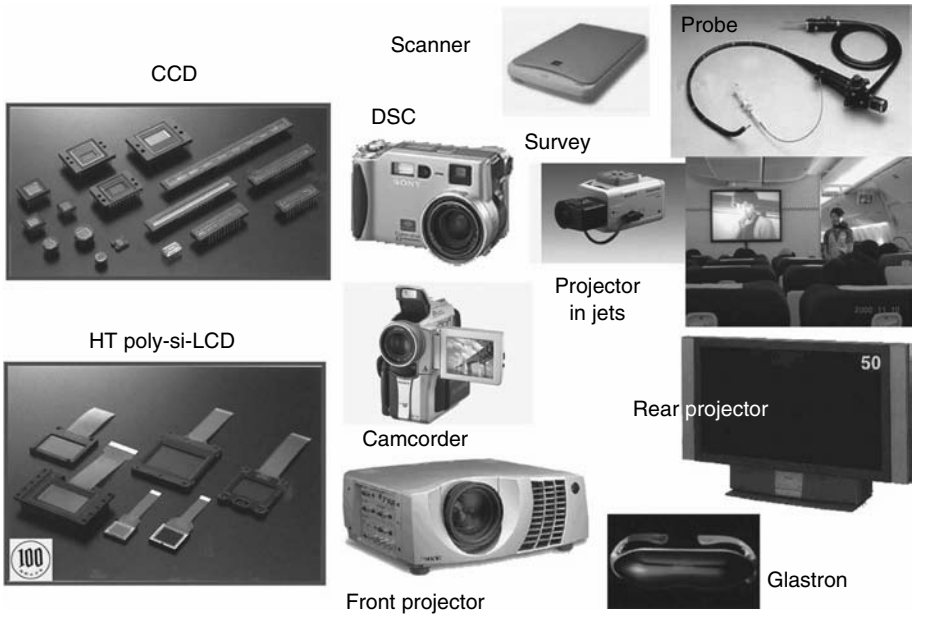



**FIGURE 25.8** Applications of CCD and LCD.

## 25.3.2 AIBO, a Home Entertainment Robot

This subsection reviews the most popular product, the entertainment robot AIBO shown in Fig. 25.9. When you buy a brand new AIBO, it is like a baby, so it does not have any knowledge. It has a certain intelligence level that is preprogrammed. You can play with the AIBO and gradually your AIBO will recognize your gestures and voices. AIBO will remember the wonderful time you spent together with it.

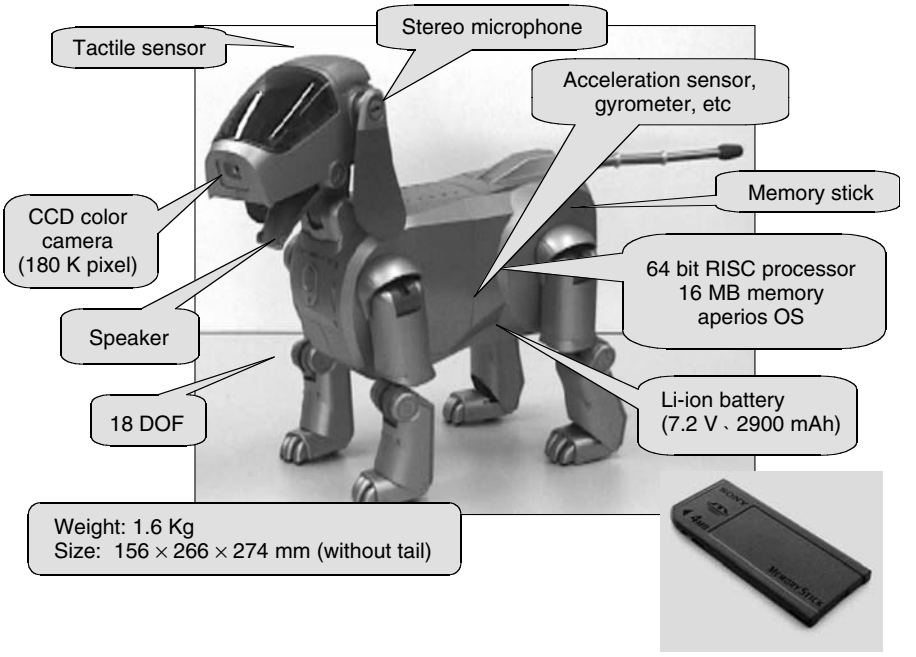



**FIGURE 25.9** AIBO model ERS-110.

Actually the experience and knowledge AIBO accumulates during these memorable moments are stored in a chewing gum size NVRAM called a memory stick shown in Fig. 25.9.

This memory stick can be also used in other products such as PCs, digital audios, and DSCs. Unfortunately it is not used in PS and PS2 for generation compatibility as of now. But in one form or another, there is a definite need NVRAMs in PS, DSC, digital audio, PC, and the future home entertainment robots.

The twenty-first century will become an era of autonomous robots, which are partners of human beings. Autonomous robot will help and support people in the future. AIBO is designed to be the first product model of robot entertainment systems. The main application of this robot is a pet-style robot, which must be lifelike in appearance.

Although AIBO is not a nursing robot, the development of AIBO is the first step of the era of autonomous robots in the twenty-first century.

The following are some works done in the Digital Creation Laboratory at Sony. Most of the works were actually done by the pioneering engineers, Mr. Fujita, Mr. Kageyama, Mr. Kitano, and Mr. Sabe.

The epoch-making debut of AIBO, model ERS-110 in 1999, had the following features:

First of all, it has a CCD color camera with 180 K pixels. Of course, it does not have a mechanical shutter. It does not have any eyelid. It has an audio sensor called microphones, a pair of them for stereo audio pick-up. It also has an acceleration sensor, gyrometer, and also a tactile sensor. So, if you pat it on the head gently, it will show some happy gesture. If you strike it on the head, it will interpret it as your sermon. The moving joints have 18 degrees-of-freedom in total.

Before introducing this first AIBO model, ERS-110, the basic research period lasted about five years. Now we have the second generation AIBO model, ERS-210 and also another type of robot, Sony Dream Robot, SDR-3, as seen in Fig. 25.10.

The second generation AIBO model, ERS-210, has the following features:

Joint DOF: neck: 3, mouth: 1, ear: 2, legs: $3 \times 4$, tail: 2, total: 20
Sensors: color CMOS image
sensor (1100 K pixel)
Microphone $\times$ 2
Infrared sensor
Acceleration sensor $\times$ 3

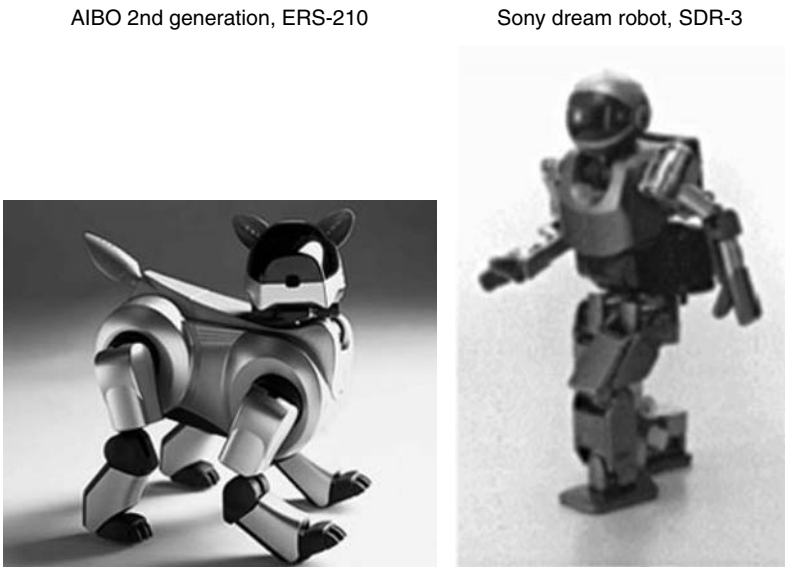



**FIGURE 25.10** New AIBO models: ERS-210 and SDR-3.

Tactile sensor × 7
CPU: 64 bit RISC processor (192 MHz)
Memory: 32 MB DRAM
OS, Architecture: Aperios, OPEN-R1.1
IF: PCMCIA, memory stick

The model SDR-3 has the following features:

Joint DOF: neck: 2, body: 2, arms: 4 × 2, legs: 6 × 2, total: 24
Sensors: color CCD camera
1800 K pixel, microphone × 2
Infrared sensor, acceleration sensor × 2 gyrometer × 2, tactile sensor × 8
CPU: 64 bit RISC processor × 2
Memory: 32 MB DRAM × 2
OS, Architecture: Aperios, OPEN-R

It weighs 5.0 kg and its size is 500 × 220 × 140 mm.

It has an OPEN-R architecture. It is made of configurable physical components (CPCs). The CPU in the head recognizes the robot configuration automatically. The components are built for plug & play or hot plug-in use. The relevant information in each segment is memorized in each CPC.

Each CPS may have a different function such as behavior planning, motion detection, color detection, walking, and camera module. Each CPS is also provided the corresponding object oriented programming and software component.

With this OPEN-R architecture, the body can be decomposed or assembled anyway for plug & play or hot plug-in use. The diagram in Fig. 25.11 shows the details of the logical hardware block diagrams, which contain DMAC : FBK: CDT: IPE and HUB

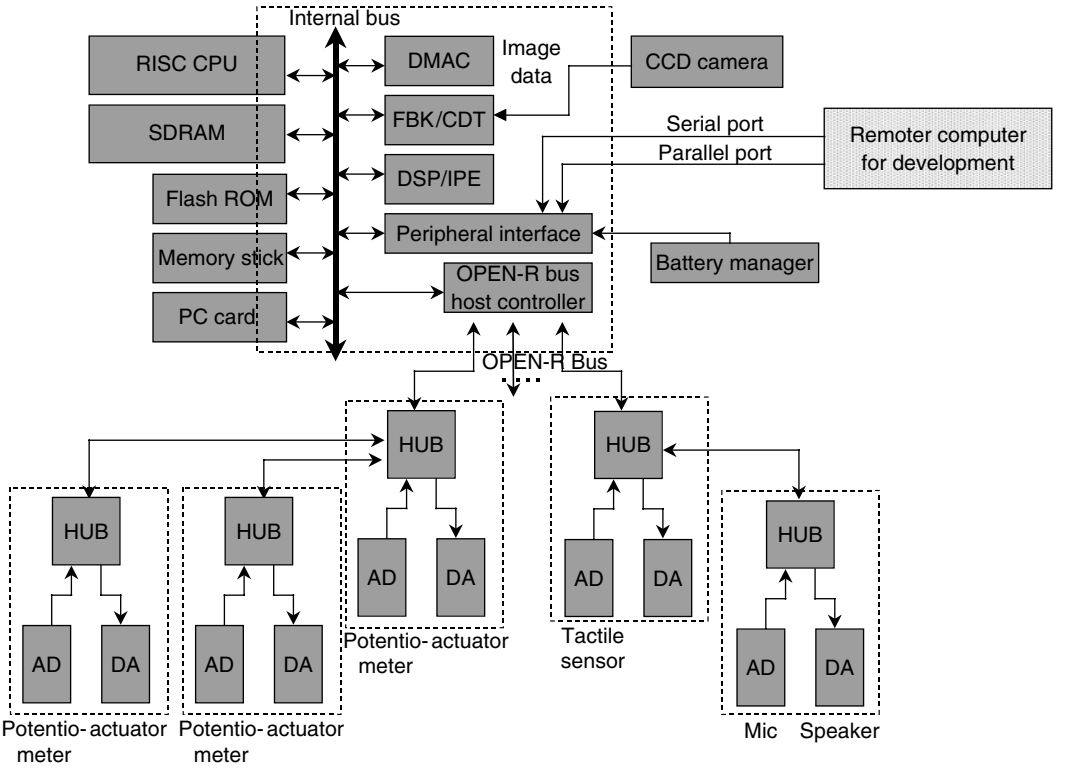



**FIGURE 25.11** Logical hardware block diagram.

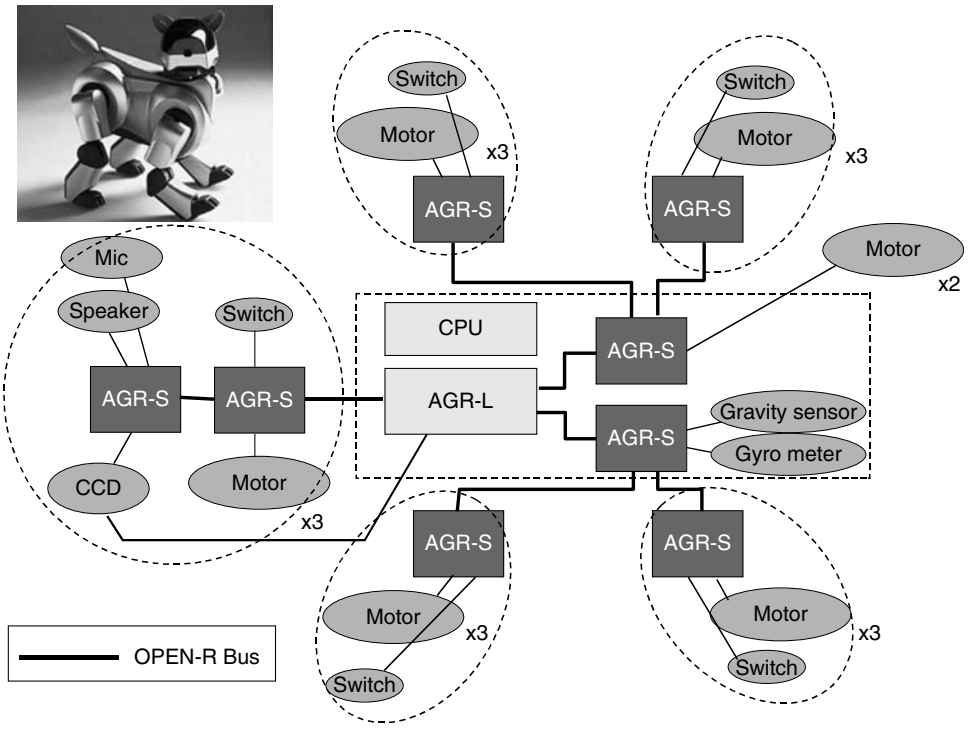



**FIGURE 25.12** Topology of ERS-110.

In the following two figures, Figs. 25.12 and 25.13, the topology of model ERS-110 and Model SDR-3x are shown, respectively.

At the same time, it is very important to have a powerful software platform that covers the top semantic layer to the deep bottom of the device driver objects codings. Careful design considerations are very important to make the middleware software components.

### 25.3.3 Memory Stick

AIBO, VAIO PC, and other audio and video products now use memory sticks as digital data recording media.

In July 1997, Sony had a technical announcement. The following year, in January 1998, the VAIO center was inaugurated. On July 1998, Sony had a product announcement. The 4 Mbyte and 8 Mbyte memory sticks were on sale in September 1998. In February 1999, Sony announced Magic Gate, that is, memory sticks with copyright protection feature. Figure 25.14 shows the form comparison. The memory stick is unique in its chewing gum-like shape and it is much taller in length than other media. The difference in appearance of memory stick from other media is clear in size and features.

Figure 25.15 shows the internal structure. It is fool proof. It features a simple 10-pin connection and it is impossible to touch the terminals directly.

The shape was designed intentionally to make exchanging of media easy, without having to actually see them, and to guide the direction for easy and correct insertion. Much contrivance is made in the design.

In order to decrease the number of connector pins for ensuring reliability of the connectors, serial interface was adopted instead of parallel interface used in conventional memory cards. As a result, connector pins were reduced to 10. And as the structure is such that these pins do not touch the terminal directly, extremely high reliability is ensured. The length is same as AA size battery of 50 mm for further deployment to portable appliances. The width is 21.5 mm and the thickness is 2.8 mm.

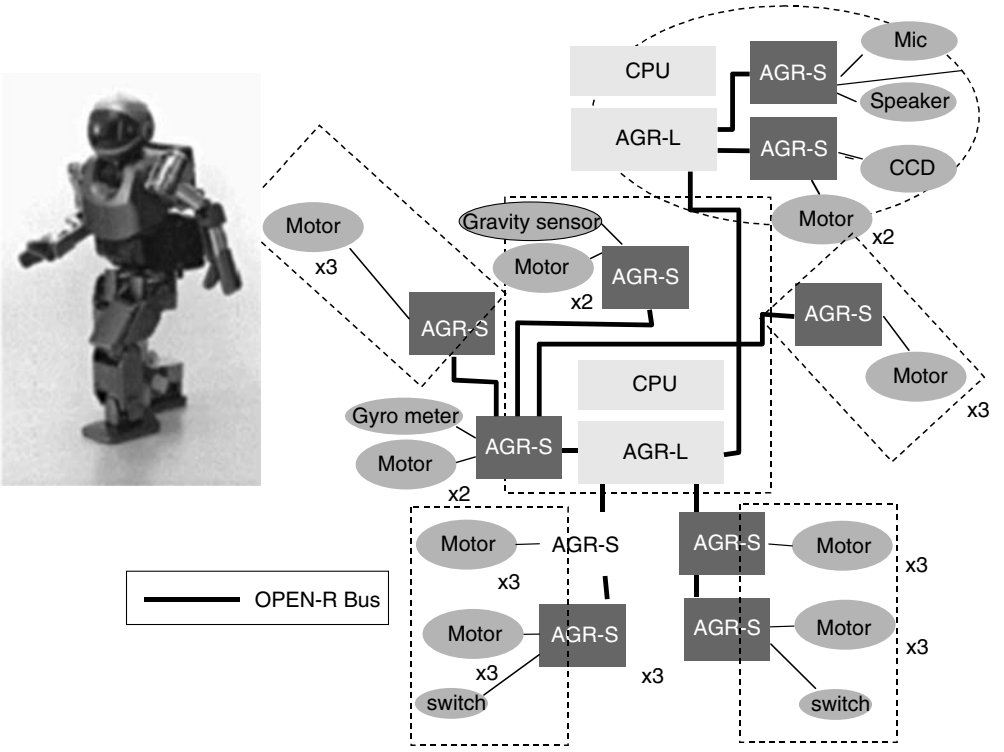



**FIGURE 25.13** Topology of SDR-3x.

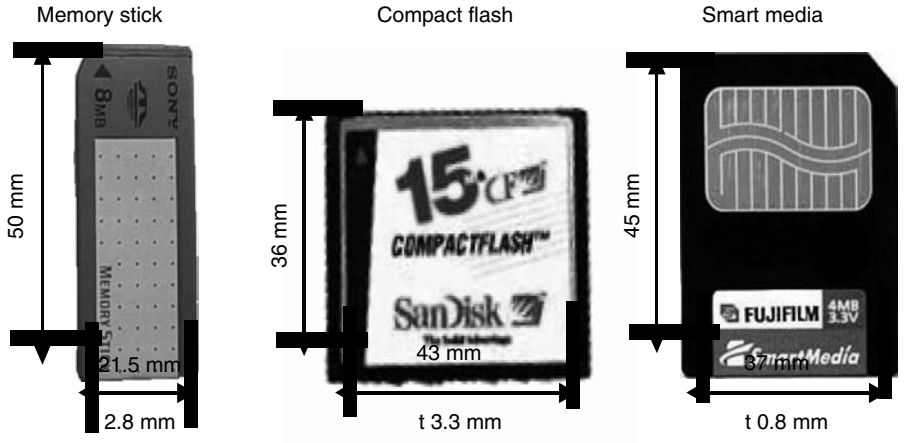



**FIGURE 25.14** Form comparison.

The memory stick consists of Flash EEPROM and a controller, controlling multiple Flash EEPROM, flexible to their variations, and capable of correcting errors unique to different Flash EEPROMs used. The memory stick converts parallel to/from serial data with the controller designed in compliance with the serial interface protocol; any kind of existing or future Flash EEPROM can be used for the memory stick. The function load on the controller chip is not excessive, and its cost can be kept to a minimum.

It is light and the shape makes it easy to carry around and to handle. Also, the write-protection switch enables easy protection of variable data.

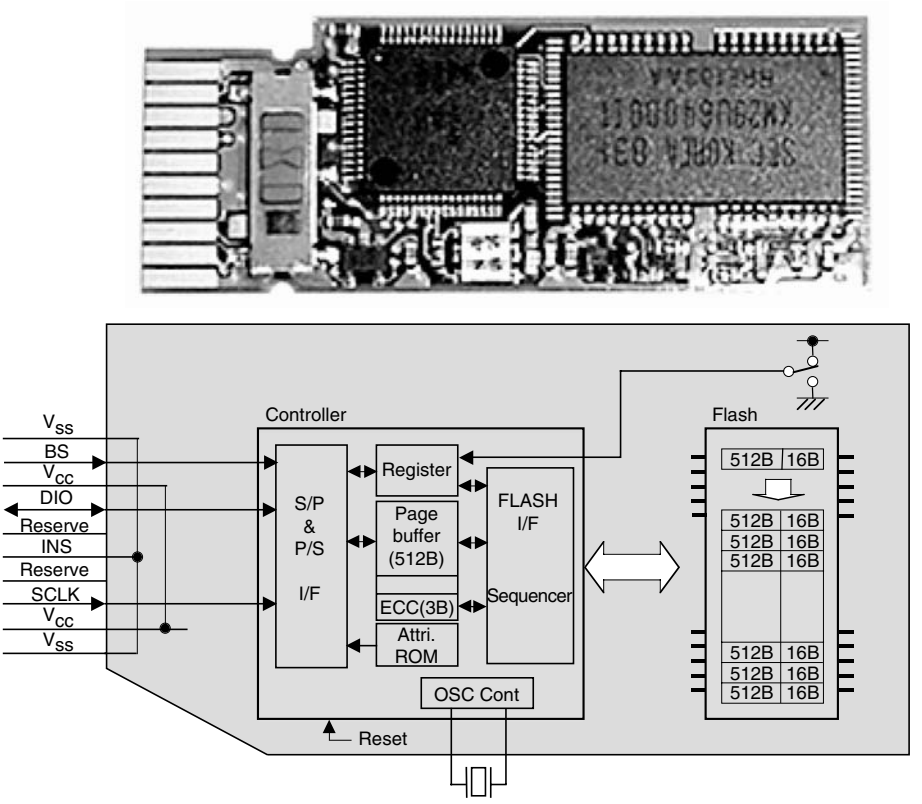



**FIGURE 25.15** Internal structure.

For still-image format, DCF standardized by JEIDA is applied. DCF stands for design rule for camera file system and JEIDA stands for Japan Electronic Industry Development Association. For voice format, ITU-T Recommendation G.726 ADPCM is adopted. The format is regulated for applications that convert voice data to text data by inserting a memory stick to a PC.

The memory stick can handle multiple applications such as still image, moving image, voice, and music on the same media. In order to do this, formats of respective application and directory management must be stipulated to realize compatibility among appliances. Thus, simply by specifying the "control information" format, one can have a new form of enjoyment through connecting AV appliances and the PC. This format, which links data handed in AV appliances, enables relating multiple AV applications. For example, voice recorded on IC recorder can be dubbed on to a still image file recorded by a digital still camera.

Presently, the music world is going from analog to digital, and the copyright protection issue is becoming serious along with the wide use of the Internet. The memory stick can provide a solution to this problem by introducing "Magic Gates (MG)," a new technology.

Open MG means (1) allowing music download through multiple electronic music distribution platforms, (2) enabling playback of music files and extracting CD on PCs (OpenMG Jukebox), (3) transferring contents securely from PCs to portable devices.

Figure 25.16 shows the stack technology applied to the memory stick with four stacked chips.

## 25.3.4 PlayStation 2

PlayStation 2 was originally aimed at the fusion of graphics, audio/video, and PC. The chipset includes a 128-bit CPU called "Emotion Engine" with 300 MHz clock frequency with direct Rambus DRAM of

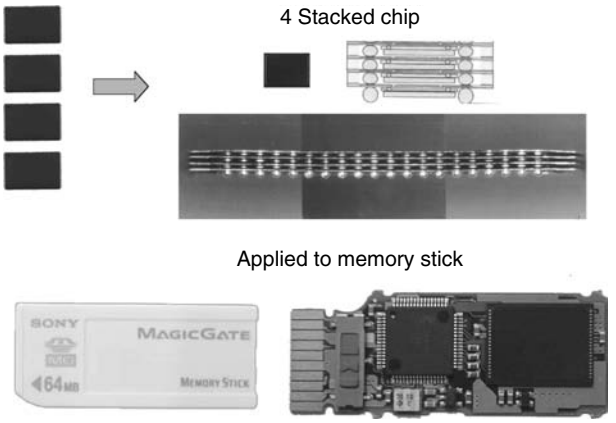



**FIGURE 25.16** Stack technology.

32 Mbyte main memory. The chipset also includes a graphic synthesizer chip with 150 MHz clock frequency. It has 4 MB video RAM as an embedded cache.

As SPUs, the chipset also has an I/O processor for X24 speed CR-ROM drive and X4 speed DVD-ROM. Figure 25.17 shows PlayStation 2 (SCPH-10000) system block diagram.

PlayStation 2, which Sony Computer Entertainment, Inc., released in March 2000, integrates games, music, and movies into a new dimension. It is designed to become the boarding gate for computer entertainment. PlayStation 2 uses an ultra-fast computer and 3-D graphics technology to allow the creation of video expressions that were not previously possible.

Although it supports DVD, the latest media, it also features backward compatibility with PlayStation CD-ROM so that users can enjoy the several thousand titles of PlayStation software. PlayStation 2 is designed as a new generation computer entertainment system that incorporates a wide range of future possibilities. The table shows the performance specifications of the graphic synthesizer chip, CXD2934.

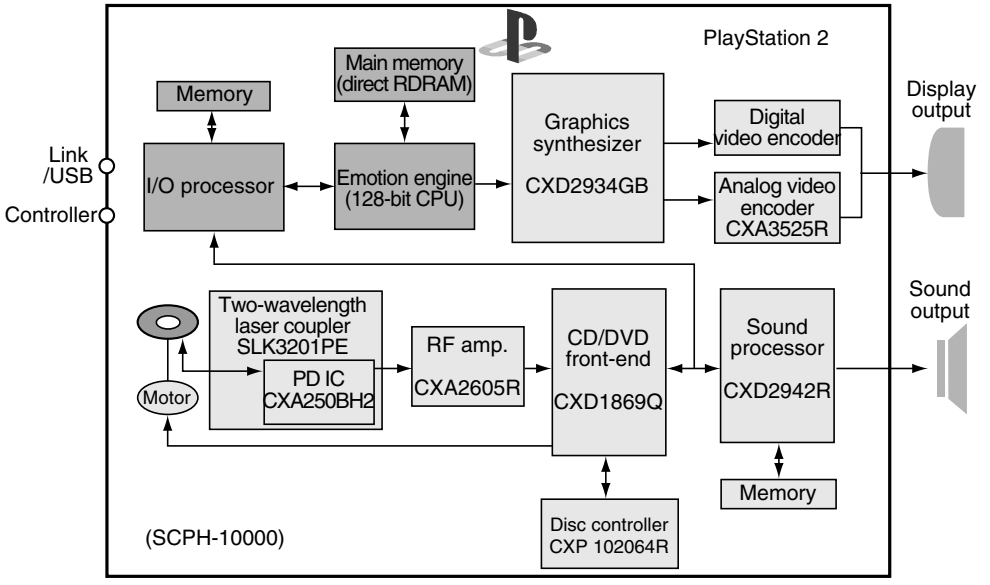



**FIGURE 25.17** PSX2 system block diagram.

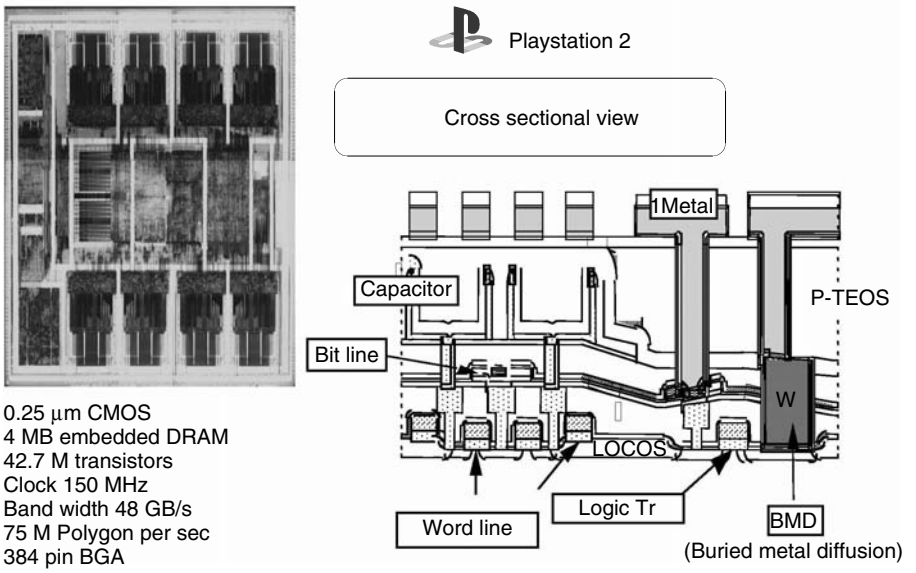



**FIGURE 25.18** 4 MB EmDRAM for PSX2.

| | |
|---|---|
| Clock Frequency | 150 MHz |
| Number of pixel engines | 16 parallel processors |
| Hybrid DRAM capacity | 4 MB@150 MHz |
| Total memory bandwidth | 48 GB/s |
| Maximum number of display colors | 2560 bits |
| Z buffer | 32 bits (RGBA: 8-bit each) |
| Process | |
| Technology | 0.25 $\mu$m |
| Total number of transistors | 43 M Tr'sPackage |
| 384-pin BGA image output formats | NTSC/PAL, D-TV, VESA (upto 1280 $\times$ 1024 dots) |

In addition to the 128-bit CPU Emotion Engine™ and I/O processor, Playstation 2 adopts several advanced technologies. The graphics synthesizer graphic engine, CXD2934GB, takes full advantage of embedded DRAM system LSI technology. Figure 25.18 shows the chip photograph of Sony's 0.25 $\mu$m CMOS 4 MB embedded DRAM, which has 42.7 M Trs. The clock rate is 150 MHz, with 48 GB/s bandwidth. It can draw 75 M polygons per second. It has 384 pin in BGA. Its cross-sectional view is also shown here.

The semiconductor's optical integrated device technology contributes significantly to miniaturization and high reliability in the optical pickups, SLK3201PE, a two-wavelength laser coupler chip. PlayStation 2 also adopts the optical disc system chip solution which has a solid track record, CXD2942R, a sound processor chip, and has earned the trust of the optical disc system market.

It also includes CXD1869 (CD/DVD signal processor LSI), CXP102064R (disk controller), CXA2605R (Cd/DVD RD matrix amplifier), and CXA3525R (analog video encoder).

The first commercial product for use in consumer products were the 0.5 $\mu$m LSI chips for 8-mm camcorders in 1995. Then, Sony had 0.35 $\mu$m LSI chips for MD products with low voltage operation of 2.0 V. Now, the 0.25 $\mu$m PlayStation 2 graphics synthesizer has eDRAM with 48 GB/s bandwidth. Figure 25.19 shows the EmDRAM history.

Sony Em-DRAM has a high-band performance of 76.8 GB/s. See Fig. 25.20. In the following three figures, Figs. 25.21 through 25.23, the memory cell size trend, some details of our embedded DRAM history, and the vertical critical dimensions between 0.25 and 0.18 $\mu$m EmDRAM process are shown, respectively.

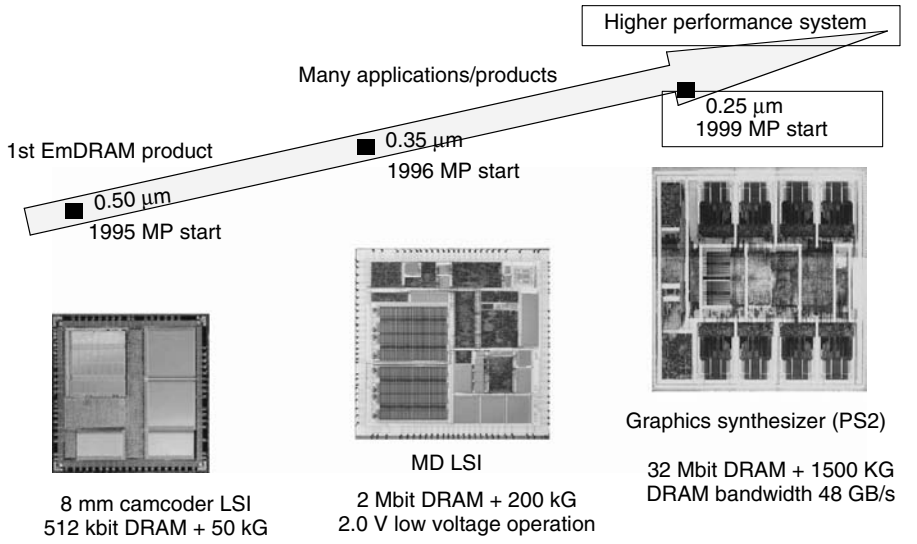



**FIGURE 25.19** Embedded DRAM history.

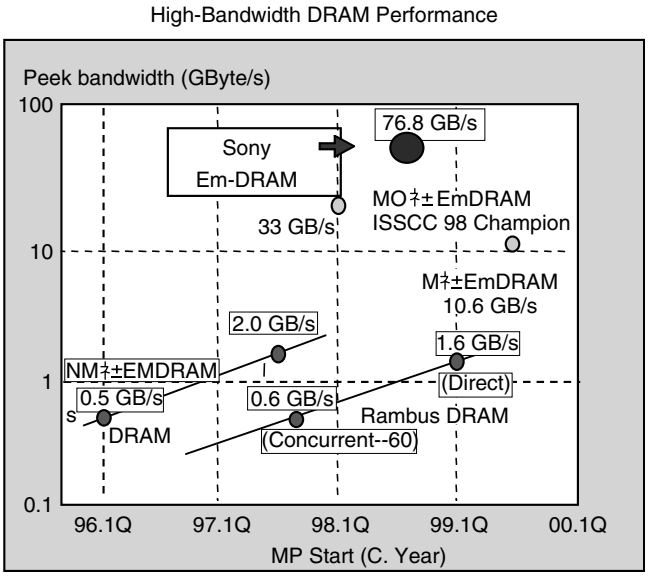



**FIGURE 25.20** Performance of embedded DRAM.

Now, a few words on the feature and critical issues of 130 nm Emb-DRAM LSI process. The most advanced design rule to achieve high performance Tr –

Enhance resolution, and
refine OPC system (speed, accuracy)
Large variation in duty cycles
Reduce isolation—dense bias
High global step-> Enlarge D.O.F
High aspect hole process
Enhance etching durability

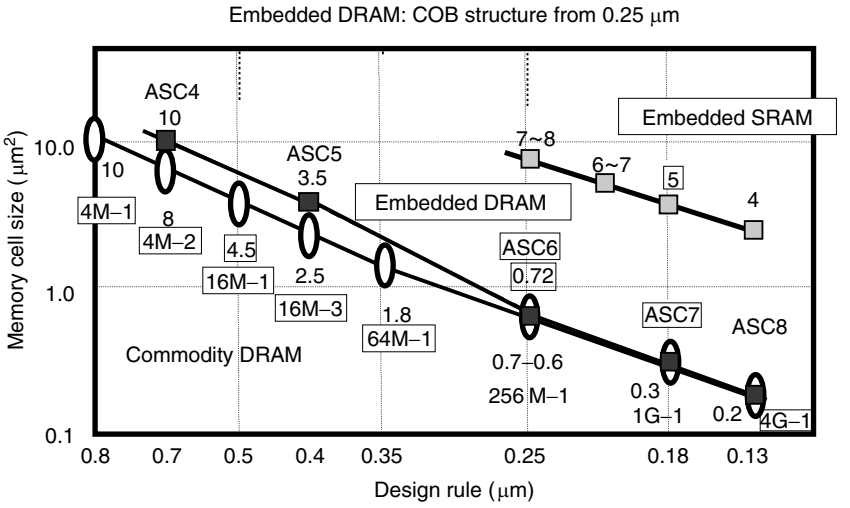



**FIGURE 25.21** CMOS memory cell size.

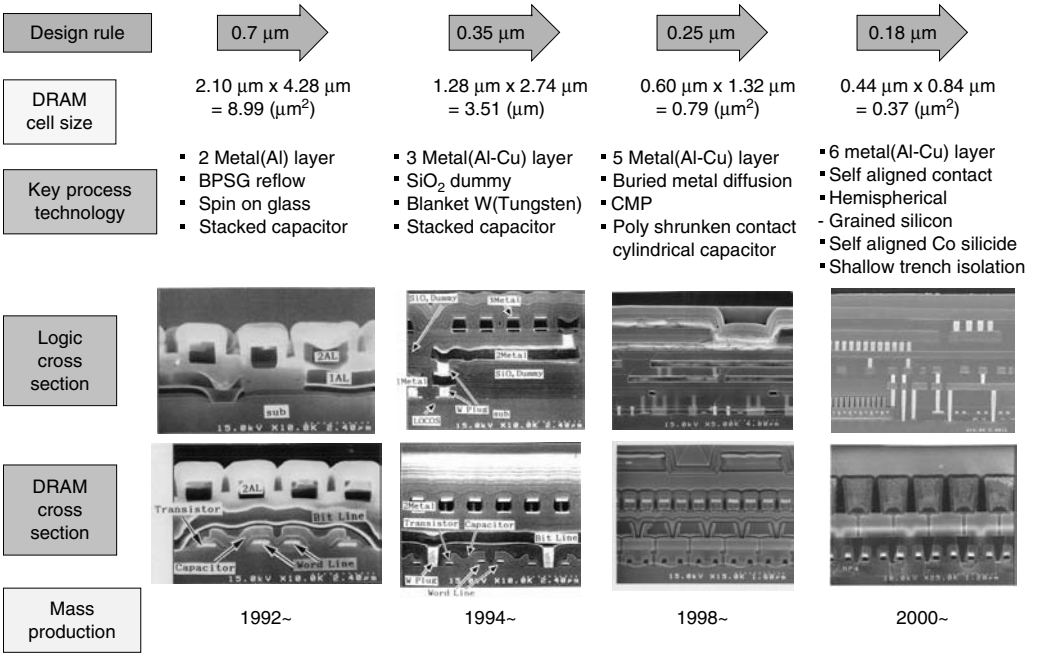



**FIGURE 25.22** Embedded DRAM history.

OPC = optical proximity correction
DOF = depth of focus

In the 0.18 $\mu$m EmDRAM process, the optical proximity correction (OPC) technology and the phase-shift mask technology (PSM) were very important. See Figs. 25.24 and 25.25. Many high-performance manufacturing and measurement automatic machines, such as those shown in Fig. 25.26, are necessary.

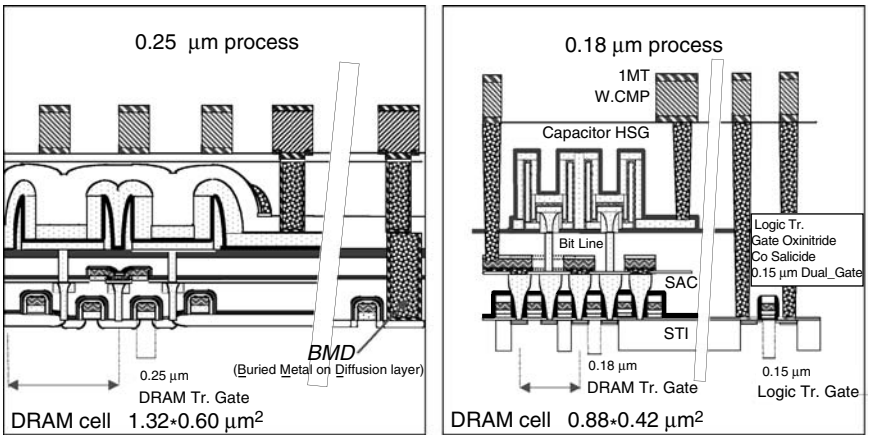



**FIGURE 25.23** Em-DRAM process technology.

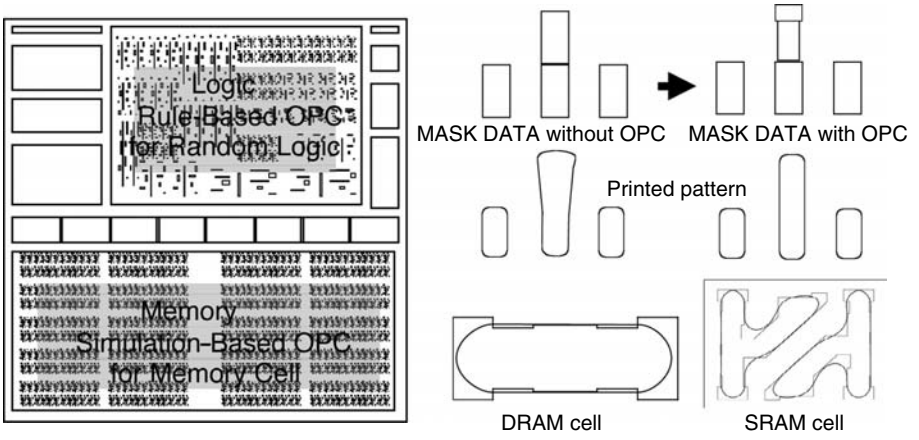



**FIGURE 25.24** Optical proximity correction.

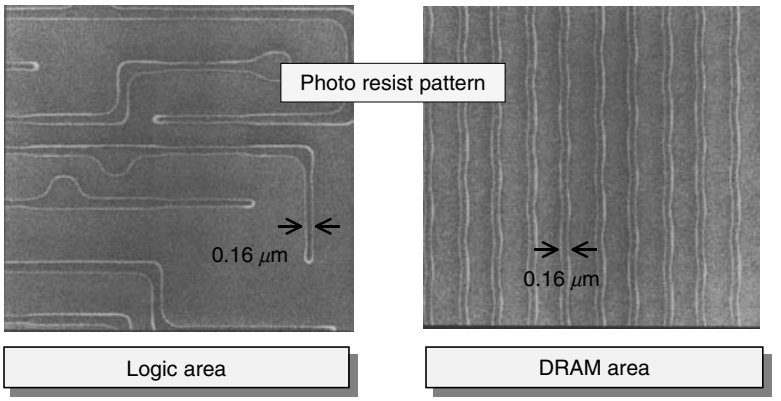



**FIGURE 25.25** Phase-shift mask (PSM) technology.

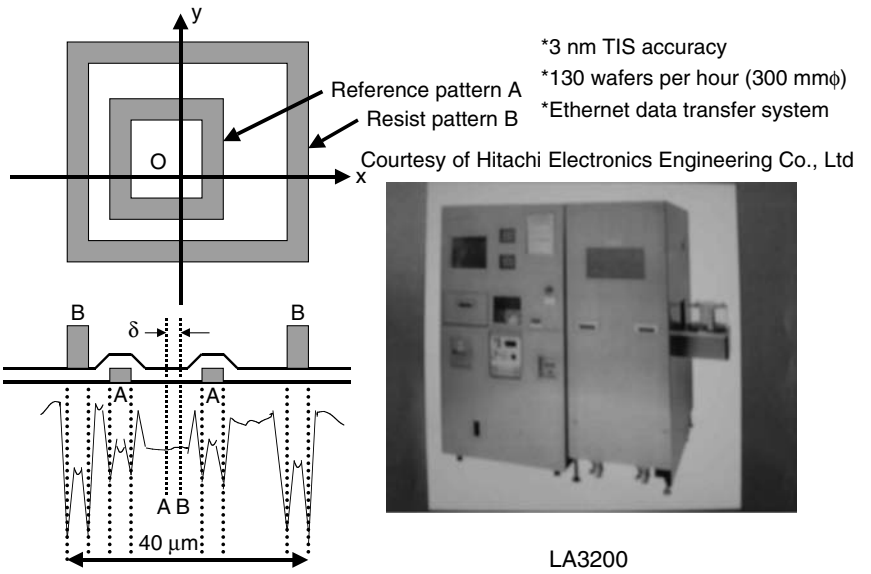



**FIGURE 25.26** Overlay accuracy measurement system.

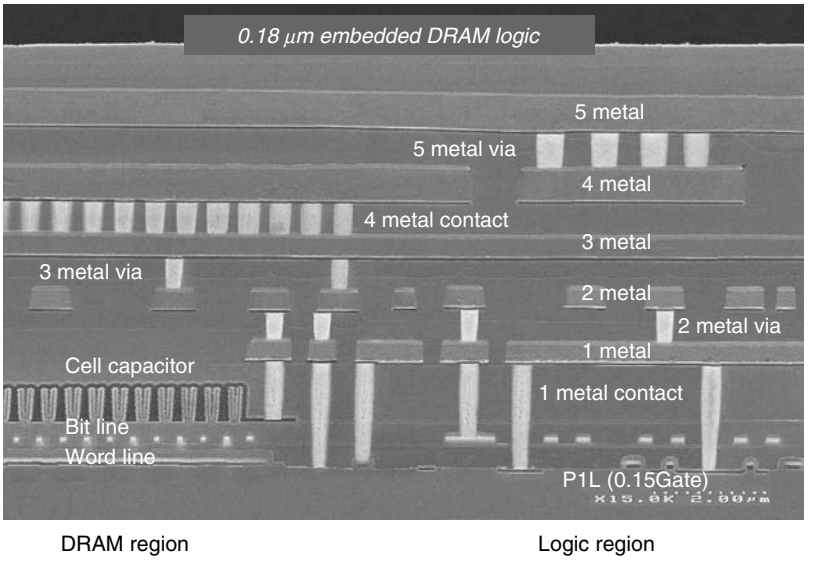



**FIGURE 25.27** Cross-sectional view.

Figure 25.27 shows the cross-sectional view of 0.18 $\mu$m EmDRAM, which was realized by utilizing all these technologies and high-performance machines.

Now some comments on key factors: technology extention such as optical extention and full flat process technology. KrF lithograpy optical extention features high NA, ultra-resolution, thin photo resist, and the OPC technology. Wirings are fully planarized interlayers of Cu/Dual Damascene. The EmDRAM features a fully planarized capacitor with the global step-less DRAM/logic structure by self-align process.

## 25.4 Conclusion

Some introductory comments on the basic semiconductor device concepts were given. They are strongly related to the microelectronics of the present home entertainment LSI chips. The chapter covered in detail some product specifications and performance aspects of the home entertainment LSI chip sets, such as those used in digital cameras, home robotics, and games. Cost of EmDRAM and its solutions by using EmDRAM are strongly related with new market creation such as PSX2. The EmDRAM technology for PS2/computer and some other future home entertainment electronics gadgets has a potential to be the technology driver in the years to come.

### References

1. Yoshiaki Hagiwara, "Solid State Device Lecture Series Aph/E183 at CalTech" in 1998–1999, http:www.ssdp.Caltech.edu/apheel83/.
2. Yoshiaki Hagiwara, "Measurement technology for home entertainment LSI chips," Presentation at the Tutorial Session in ICMTS2001, Kobe, Japan, March 19–22, 2001.
3. M. Fujita and H. Kitano: "{{D}evelopment of and {A}utonomous {Q}uadruped {R}obot for {R}obot {E}ntertainment}," *Autonomous Robots,* vol. 5, pp. 7–8, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1998.
4. Kohtaro Sabe, "Architecture of entertainment robot-development of AIBO –," *IEEE Computer Element MESA Workshop 2001,* Mesa, Arizona, Jan. 14–17, 2001.
5. JP 1215101 (a Japanese Patent #58-46905), Nov. 10, 1975 by Yoshiaki Hagiwara.

# 26

# Mobile and Wireless Computing




John F. Alexander
Raymond Barrett
*University of North Florida*

Babak Daneshrad
*University of California*

Samiha Mourad
Garret Okamoto
*Santa Clara University*

Mohammad Ilyas
*Florida Atlantic University*

Abdul H. Sadka
*University of Surrey*

Giovanni Seni
*Motorola Human Interface Labs*

Jayashree Subrahmonia
*IBM Thomas J. Watson Research Center*

Larry Yaeger
*Indiana University*

Ingrid Verbauwhede
*Katholieke Universiteit Leuven and UCLA*

# 26.1 Bluetooth—A Cable Replacement and More

*John F. Alexander and Raymond Barrett*

## 26.1.1 What is Bluetooth?

Anyone who has spent the time and effort to connect a desktop computer to its peripheral devices, network connections, and power source knows the challenges involved, despite the use of color-coded connectors, idiot proof icon identification, clear illustrations, and step-by-step instructions. As computing becomes more and more portable, the problems are compounded in the laptop computer case, and the palmtop device case, let alone the cell phone case, where cabling solutions are next to impossible. The challenges associated with cabling a computer are tough enough for purposes of establishing the "correct" configuration, but are nearly unmanageable if the configuration must be dismantled each time a portable device is carried about in its portable mode.

Similar to a knight in shining armor, along comes Bluetooth; offering instant connectivity, intelligent service identification, software driven system configuration, and a myriad of other advantages associated with replacing cabling with an RF link. All of this good stuff is provided for a target price of $5 per termination, a cost that is substantially lower than the cost of most cables with a single pair of terminations. This miracle of modern communication technology is achieved with a 2.4-GHz frequency hopping trans-ceiver and a collection of communications protocols. At least, that is the promise. The participants who include such industrial giants as IBM, Motorola, Ericsson, Toshiba, Nokia, and over a thousand other consortium participants provide credibility for the promise.

There has been considerable interest in the press over the past few years in the evolution of the open Bluetooth® [1] specification for short-range wireless networking [2]. Bluetooth is one of many modern technological "open" specifications that are publicly available. The dream is to support Bluetooth short-range wireless communications (10–100 m) any where in the world. The 2.4 GHz frequency spectrum was selected for Bluetooth primarily because of its globally available free license. As we entered the twenty-first century there were already more than 1800 members of Bluetooth special interest group (SIG) [3]. Its reasonably high data rate (1 Mb/s gross data rate) and advanced error correction make it a serious consideration that is irresistible for hundreds of companies in a very diverse group of industries, all interested in ad hoc wireless data and voice linkages.

The Bluetooth specification utilizes a frequency hopping spread spectrum algorithm for the hardware and specifies rapid frequency hopping of 1600 times per second. As one might conclude 2.4 GHz digital radio transceivers that support this type of high frequency communication are quite complex, however, the hardware design and implementation is just the tip of the iceberg in understanding Bluetooth. The goal of this chapter is to provide the reader with a thorough overview of Bluetooth. An overview is detailed in the standard, but the Bluetooth specifications alone are thousands of pages.

Some of the proposed and existing Bluetooth usage models are the cordless computer, the ultimate headset, the three-in-one phone, the interactive conference (file transfer), the Internet bridge, the speaking laptop, the automatic synchronizer, the instant postcard, ad hoc networking, and hidden computing.

## 26.1.2 Competing Infrared Technology

First, a brief digression will be taken into infrared wireless communication. With the advent of the personal digital assistant (PDA), it was obvious for the need of a low cost, low power means of wireless communication between user's devices and peripherals. At an Apple Newton users group one could see hundreds of enthusiasts "beaming" business cards back and forth. As other vendors came out with PDA each had its own proprietary infrared communication scheme. Eventually one "standard" method of communication between users applications came about as an outgrowth of the work of the Infrared Data Association. This specification became known as IrDA [4]. An international organization creates and promotes interoperable, low cost infrared data interconnection standards that support a walk-up,

point-to-point user model. The standards support a broad range of appliances, computing, and communications devices.

Several reasons exist for mentioning the IrDA. First, many of the companies involved in the Bluetooth effort are members of the IrDA and have many products, which support IrDA protocols. Thus, much of the learning time in developing and attempting to implement a workable open standard for ad hoc short range wireless communication is *in house*. Also the IrDA has been one of the many well thought out high technology products that never gained much user acceptance. Many of the members of the Bluetooth SIG were anxious not to make the same mistake but to gain a way to profit from all the hard work invested in IrDA.

The proposed solution seemed simple. Just include more or less the entire IrDA software protocol stack in Bluetooth. Thus, the many already developed but seldom-used "beaming" applications out there could readily use Bluetooth RF connectivity. Whether this was a good idea, only time can tell. But it is important in understanding the Bluetooth specification because it is so heavily influenced by millions of hours of corporate IrDA experience and frustrations.

### 26.1.3 Secure Data Link

Providing a secure data link is a fundamental goal for the Bluetooth SIG. One could envision the horror of walking through an airport with your new proprietary proposal on your laptop and having the competition wirelessly link to your machine and steal a copy. Without good security Bluetooth could never gain wide acceptance in virtually all cell phones, laptops, PDAs, and automobiles that the drafters envisioned.

Secure and nonsecure modes of operation are designed into the Bluetooth specification. Simple security is provided via authentication, link keys, and PIN codes, similar to bank ATM machines. The relatively high frequency hopping at 1600 hops/sec adds significantly to the security of the wireless link. Several levels of encryption are available if desired. In some cases, this can be problematic in that the level of encryption allowed for data and voice varies between countries and within countries over time. The Bluetooth system provides a very secure environment, eavesdropping is difficult. Bluetooth probably will be shown to be more secure than landline data transmission [5].

### 26.1.4 Master and Slave Roles

The Bluetooth system provides a simple network, called a piconet, nominally 10 m in radius. This is the 1-mW power mode (0 dbm). There is also a 10-mW mode allowed, which probably could reach a 100 m in ideal cases, but it may not become widely implemented. One should think of a Bluetooth piconet as a 10 m personal bubble providing a moderately fast and secure peer-to-peer network. The specification permits any Bluetooth device to be either a master or a slave. At the baseband level, once two devices establish connection, one has to be a master and the other a slave. The master is responsible for establishing and communicating the frequency-hopping pattern based on the Bluetooth device address and the phase for the sequence based on its clock [6].

Up to seven active slaves are allowed all of which must hop in unison with the master. The Bluetooth specification allows for the direct addressing of up to 255 total slave units, but all but seven of the slaves must be in a "parked" mode. The master–slave configuration is necessary at the low protocol levels to control the complex details of the frequency hopping, however, at higher levels, the communication protocol is a peer-to peer and the connection established looks like point-to-point. The protocol supports several modes, which include active, sniff & hold, and park. Active uses the most power. While the master unit is in sniff mode, it conserves power by periodically becoming active. Additionally, the slave is in a hold mode but wakes up periodically based on timing from the master to "see" if any data is ready for it. While a slave is in park mode it consumes the least power, but the slave still maintains synchronization with the master.

A more complex Bluetooth communication topology is the scatternet. In one of the simpler scatternet schemes there are two masters with a common slave device active in two piconets. In another variation on the scatternet, one device is a slave in one piconet and the master in another. Using this scatternet

idea some have speculated that an entire wireless network could be formed by having many piconets, each with one common node. Theoretically, this would work, but the rest of the original Bluetooth specification is not designed for making Bluetooth a wireless LAN. It is likely the newer SIG work group on personal area networking will be interested in expanding the definition and capability of Bluetooth scatternet capability. Currently there is lots of interest in forming location aware ad hoc wireless networks [7]. NASA has already approached this author for ideas for use of Bluetooth for ad hoc small area networks in space missions. The appeal of a wireless link made up of five dollar, very small, low-power, self-configuring, parts capable of connecting various sensors is irresistible for complex space missions where power and payload weight is at a premium.

### 26.1.5 Bluetooth SIG Working Groups

To understand the Bluetooth specification it is important to understand how the very large Bluetooth SIG is organized. The actual work in producing the various specifications is done by the various SIG working groups. Given that the Bluetooth specification is thousands of pages of detailed technical documentation, it is not practical to just sit down and read the specification sheet. Briefly, five major groups compose the SIG including the air interface group, the software group, the interoperability group, the legal group, and the marketing group [3].

The software group contains three working subgroups primarily responsible for the Bluetooth protocol stack. These are the lower Transport Protocol Group, the Middleware Protocol Group, and the Application Group. The protocol stack follows the international origination of standardization (ISO) seven-layer reference model for open system interconnection [8].

### 26.1.6 The Transport Protocol Group

The Transport Protocol Group includes ISO layers one and two, which are the Bluetooth radio, the link controller baseband, the link manager, the logical link controller and application protocol (L2CAP) layer, and the host controller interface. Collectively this set of protocol groups form a virtual pipe to move voice and data from one Bluetooth device to another. Audio applications bypass all of the higher level layers to move voice from one user to another [6].

The L2CAP layer prevents higher level layers from having to deal with any of the complexity of the frequency hopping Bluetooth radio and its complex control or special packets used over the Bluetooth air radio interface. The responsibility of the L2CAP layer is to coordinate and maintain the desired level of service requested and coordinate new incoming traffic. The L2CAP layer's concern is with asynchronous information (ACL packet) transmission [6]. This layer does not know about the details of the Bluetooth air interface such as master, slave, polling, frequency hopping, and such. Its job is to support the higher layer protocol multiplexing so multiple applications can establish connectivity over the same Bluetooth link simultaneously [9].

Device authentication is based on an interactive transaction from the link manager. When an unknown Bluetooth device request connectivity, the device requested ask the requester to send back a 16 byte random number key, which is similar to the familiar bank ATM PIN code procedure. Once a device is authenticated it is necessary for the device to store the authentication codes so this process can be automatic in future connections. Link encryption up to 128 bytes is supported and is controlled by desirability and governing legal issues of the area. Encryption applies only to the data payload and is symmetric.

Power management of connected devices is also handled at this level. In sniff mode the slave must wake up and listen at the beginning of each even-numbered slot to see if the master intends to transmit [6]. In hold mode the slave is suspended for a specified time. The API for hold mode puts the master in charge but provisions are available to negotiate the time. In Park mode, the slave dissociates itself from the piconet while still maintaining synchronization of the hopping sequence. Before going in to park mode the master informs the slave of a low-bandwidth beacon channel the master can use to wake the parked slave if there not already seven active slaves.

Paging schemes allow for a more repaid reconnection of Bluetooth devices. For example, paging is used in the event a master and a slave need to switch rolls to solve some problem such as forming some sort of local area network. Support for handling paging is optional in the Bluetooth specification. Another role of the link managers is to exchange information about each other to make passing data back and forth more efficient.

### 26.1.7 The Bluetooth Transceiver

The Bluetooth systems operate in the industrial and scientific (ISM) 2.4 GHz band. This band is available license free on a global basis and is set a side for wireless data communications. In the United States the Federal Communication Commission (FCC) sets up rules for transmitters operating in the ISM band under section 15.247 of the Code of Federal Regulations. The frequency allocated is from 2,400 MHz to 2,483.5 MHz. The Bluetooth transceiver operates over 79 channels each of which is one megahertz wide. At least 75 of the 79 frequencies hoped to must be pseduo-random. Bluetooth uses all 79 channels and hops at a rate of 1600 hopes per second.

### 26.1.8 The Middleware Protocol Group

The Middleware Protocol Group includes ISO layers three and six, which are made up of the RFCOMM protocol, the service discovery protocol (SDP), IrDA interoperability protocols, IrDA, and Bluetooth wireless protocol, and the audio and telephony control protocol. Fitting Bluetooth into the ISO model is really up to the developer. If you want to make it fit it makes sense, but there is lots of strange baggage imbedded protocols in Bluetooth that makes this difficult to see. First, we have already seen the voice communication connect down at the L2CAP layer. Now we are faced with how the toss in multiplexed serial port emulation, IrDA interoperability, and a bunch of protocols from telephony world. No wonder the standard goes on for thousands of pages and hundreds of companies around the world are struggling with comparability testing of various Bluetooth devices designed from this very complex specification.

### 26.1.9 The Application Protocol Group

The Application Protocol Group includes ISO layer seven. This grouping contains the most extensive variety of special-purpose profiles all of which rely on the six lower levels for service. These include the generic profiles, the serial and object exchange profile, the telephony profiles, and the networking profiles.

The generic profiles includes the generic access profile and the service discovery application profile. The serial and object exchange profile contains the serial port profile, the generic object exchange profile, the object push profile, the file transfer profile, the synchronization profile, the networking profiles, the dial-up networking profile, the LAN access profile, the fax profile, the telephony profiles, the cordless telephony profile, the intercom profile, the headset profile, and the cordless telephony profile. Most of these applications profiles are self-explanatory and are only of detailed interest to the software developer when developing a specific application using the appropriate profile. This is not to say that they are not important, but they provide very detailed application programmer interfaces (API) [15].

The possible Bluetooth applications keep expanding. This stimulates interest in expanding the array of application profiles in the Bluetooth specification. Several of the newer application profiles are the car profile, a richer audio/video profile, and a local positioning profile.

### 26.1.10 Bluetooth Development Kits

Given the obvious complexity of the Bluetooth hardware and software applications, having access to good development kits is essential to speed the implementation of the specification. The first inexpensive development kit to become widely available to universities was Ericsson's Bluetooth Application and Training Toolkit. This is a first generation Bluetooth kit that demonstrates important Bluetooth features

and has a well defined, but extensive proprietary API in C++. Application development is possible, but is time-consuming and tedious requiring knowledge of C++ to learn a vast API. Newer kits, specifically for development, are more efficient.

Cambridge Silicon Radio (CSR) Bluetooth silicon radio has been very well publicized in the Bluetooth development and features "all CMOS" one chip solution. The CSR development kit includes software for CRR "BlueCore™" [11] IC with on-chip Bluetooth™ protocol and a PC development environment. Tools for embedded "1-chip" products are provided. Bluetooth BlueCore-to-host serial protocol and integrated Bluetooth protocol: BlueStack™. An innovative feature is that BlueCore devices enable users to configure the level of BlueStack that loads at boot time using software switches. SCR clams that running the full Bluetooth protocol locally on a BlueCore device significantly reduces the load on the host embedded processor, delivering major advantages to users of there Bluetooth system on a chip solution [12].

Many other development tools can be found currently at the http://www.bluetooth.com/product/dev_tools/development.asp. The above two are referenced because they have been around for a year or so and the authors have direct experience with them [2].

### 26.1.11 Interoperability

There is a conflict with IEEE 802.11 Wireless Network Specification, which uses a direct sequence spread spectrum approach in the same frequency band. The direct sequence modulation is incompatible with the frequency hopping approach employed in Bluetooth. It is unlikely that an elegant interoperability solution can be found, without duplication of the entire hardware solutions for each; however, some early ad hoc reports in the trade press seem to point to the interoperability between 802.11 and Bluetooth to be minor [13,14].

Both operate in the 2.4 GHz ISM band, and both are a form of spread spectrum, but the 802.11 is direct sequence modulated spread spectrum and allows more power. Bluetooth is frequency hopping and low power.

### 26.1.12 Bluetooth Hardware Implementation Issues

First, for Bluetooth to achieve the stated goals for widespread usage at low cost, there are severe hardware constraint issues to be addressed. Second, the environment into which Bluetooth is likely to be deployed is rapidly changing. Finally, the business models for adoption of Bluetooth technology are also impacted.

Broadly speaking, there are two classes of hardware implementation for Bluetooth, one employs discrete multiple chips to produce a solution, and a second in which Bluetooth becomes an embedded intellectual property (IP) block in a system-on-a-chip (SoC) product.

For the short run, the multiple chip strategy provides an effective implementation directed at prototype and assembly-level products. The strategy is effective during the initial period of development for Bluetooth, while the specification is still evolving and the product volumes are still low; however, a strong case can be made that the high volumes, low cost, and evolving environment make an IP block approach inevitable if Bluetooth is to enjoy wide acceptance.

In addressing the environmental issues, the most widely dispersed communications product today is the cell phone, with its service variants. The Internet connectivity of cell phones is soon to surpass the Internet connectivity of the desktop computer. The consequence of the cell phone driving the environment for all information processing connectivity under its constraints of low power, tight packaging, high volume, and low cost manufacturing forces examination of IP blocks for SoC solutions.

Bluetooth is highly attractive for cell phone products as a wire replacement, enabling many of the existing profiles from a cell phone, as well as providing expansion for future applications. The desktop computer embraces Bluetooth also as a wire replacement, but has a history of services supported by cabling. The cell phone cannot support many services with cabling, and in contrast to the desktop, service extensions for the raw communication capability of 3G and 4G cell phones must be addressed by wireless solutions.

Once Bluetooth IP block solutions exist, the market forces will drive high-volume products toward either embedded or single-chip Bluetooth implementations.

Technological hurdles must be overcome in the road toward Bluetooth IP block solutions. Presently, the RF front-end solutions for bluetooth are nearly all implemented in bipolar IC technology, implying at least a BiCMOS IC, which is widely recognized as too high cost for high-volume SoC products. As the lithography becomes available for denser CMOS IC products, the deep submicron devices provide the density and speed increases to support SoC solutions in the digital arena, and also improve the frequency response of the analog circuitry, enabling the possibility of future all-CMOS implementation.

In addition, communications system problems must be solved in order to ensure the feasibility of an all-CMOS implementation. For example, one of the more popular architectures for a modern communications receiver is the zero-IF (ZIF) approach. Unfortunately, the ZIF approach usually converts the RF energy immediately to baseband without significant amplification, which places the very small signal in the range of 1/f noise of the semiconductor devices employed. Typically, the only devices with substantially low enough noise are bipolar devices, which are to be avoided for system level considerations. Alternative architectures that are suitable include variants of super heterodyne architectures that usually require tuned amplifiers, which are also seldom suitable for integration. One approach that seems to meet all the requirements is one variant of the super heterodyne architecture known as low-IF, that places the energy high enough in the spectrum to avoid noise considerations, but low enough to be addressed by DSP processing to achieve the requisite filtering.

Regardless of the particular architecture chosen, the rapid channel switching involved in the frequency-hopping scheme necessitates frequency synthesis for local oscillator functions. There is considerable design challenge in developing a fully integrated voltage-controlled-oscillator (VCO) for use in a synthesizer that slews rapidly and still maintains low phase noise.

To compound the above issues, true IP block portability implies a level of process independence that is not currently enjoyed by any of the available architectures. Portability issues are likely to be addressed by intelligence in the CAD tools that are used to customize the IP blocks to the target process through process migration and shrink paths.

## References

1. Bluetooth is a trademark owned by Telefonaktiebolagent L M Ericsson, Sweden and licensed to the promoters and adopters of the Bluetooth Special Interest Group.
2. http://www.bluetooth.com/developer/specification/specification.asp Bluetooth Specification v1.1 core and v1.1 profiles.
3. http://www.bluetooth.com/sig/sig/sig.asp Bluetooth Special Interest Group (SIG).
4. Infrared Data Association IrDA http://www.irda.org.
5. Bray, J. and Sturman, C.F., Bluetooth, *Connectivity with our Cables,* Prentice-Hall, Englewood Cliffs, NJ, 2001.
6. Miller, B.A. and Bisdikian, C., *Bluetooth Revealed,* Prentice-Hall, Englewood Cliffs, NJ, 2001.
7. Tseng, Y., Wu, S., Liao, W., and Chao, C., Location Awareness in Ad Hoc Wireless Mobile Networks. [June 2001], *IEEE Computer,* 46,52.
8. International Origination of Standardization Information processing systems–Open Systems Interconnection–Connection Oriented Transport Protocol Specification, International Standard number 8825, ISO, Switzerland, May 1987.
9. Held, G., *Data Over Wireless Networks,* McGraw-Hill, New York, 2001.
10. http://www.comtec.sigma.se/Ericsson's Bluetooth Application and Training Tool Kit 200.
11. BlueCore™" and BlueStack™" are registered trademarks of Cambridge Silicon Radio, Cambridge, England, 2001.
12. http://www.csr.com/software.htm, Development software for BlueCore™ ICs Cambridge Silicon Radio, Cambridge, England, 2001.

13. Dornan, A., Wireless Ethernet: Neither Bitten or Blue. *Network Magazine,* May 2001.
14. Merritt, R., Conflicts Between Bluetooth and Wireless LANs Called Minor. *EE Times,* February 2001.
15. Muller, N.J., *Bluetooth Demystified,* McGraw-Hill, New York 2000.

## 26.2 Signal Processing ASIC Requirements for High-Speed Wireless Data Communications

*Babak Daneshrad*

### 26.2.1 Introduction

To date, the role of application specific integrated circuits (ASICs) in wireless communication systems has been rather limited. Almost all of the signal processing demands of second generation cellular systems such as GSM and IS-136 (US TDMA) can be met with the current generation of general purpose DSP chips (e.g., TI TMS320, Analog Device's ADSP 21xx, or Lucent's DSP16xx families). The use of ASICs in wireless data communications has been limited to wireless LAN systems such as Lucent's WaveLAN and the front end, chip-rate processing needs of DSSS-CDMA based systems such as IS-95 (US CDMA).

Several major factors are redirecting the industry's attention towards ASICs for the realization of highly complex and power efficient wireless communications equipment. First, the move toward third generation (3-G) cellular systems capable of delivering data rates of up to 384 kbps in outdoor macro-cellular environments (an order of magnitude higher than the present second generation systems) and 2 Mbps in indoor micro-cellular environments. Second, the emergence of high-speed wireless data communications, whether in the form of high-speed wireless LANs [1] or in the form of broadband fixed access networks [2]. A third, but somewhat more subtle factor is the increased appeal of software radios. Radios that can be programmed to transmit and receive different waveforms and thus enable multi-mode and multi-standard operation. Although ASICs are by nature not programmable, they are parameterizable. In other words, ASIC designers targeting wireless applications must develop their architectures in such a way as to provide the user with features such as variable symbol rates and carrier frequency, as well as the ability to shut off parts of the circuit that may be unused under benign channel conditions. For DSSS systems the ASICs should provide sufficient flexibility to accommodate programmability of the chip-rate, spreading factor, and the spreading code to be used.

The next subsection further explores these elements and identify key signal processing tasks that are suited for ASIC implementation. Section 26.2.3 will present signal processing algorithms and ASIC architectures for the realization of these blocks. This section ends with Section 26.2.4.

### 26.2.2 Emerging High-Speed Wireless Systems

#### 26.2.2.1 Third Generation (3-G) Cellular Networks

Second generation cellular systems such as IS-136, GSM, and IS-95 have mainly focused on providing digital voice services and low-speed data traffic. With the growing popularity of the Internet and the need for multimedia networking, standardization bodies throughout the world are looking at the evolution of current systems to support high-speed data and multimedia services. The technology of choice for all such 3-G systems is wideband code division multiple access (W-CDMA) based on direct sequence spread spectrum (DSSS) techniques [3,4]. The targeted chipping rate for these systems is 3.84 Mcps for the European UTRA standardization work, and a multiple of 1.2288 Mcps for the CDMA-2000 proposal.

In addition to providing higher data rates, which come about in part due to the increased bandwidth utilization of 3-G systems, a second and equally important aim of these systems is to increase the capacity of a cell (number of simultaneous calls supported by a cell). To this end, all the current

proposals call for the use of sophisticated receivers utilizing multi-user detection and possibly smart antenna technologies.

In order to better appreciate the signal processing requirements of these receiver units, consider the block diagrams presented in Fig. 26.1. Figure 26.1a depicts the transmitter of a DSSS system, along with a candidate successive interference canceller (SIC) shown in Fig. 26.1b [5]. The details of the rake receiver are shown in Fig. 26.1c.

The tremendous processing requirements of this architecture will become evident by considering a modest system operating at a chip rate of say, 4 Mcps using a 32-tap shaping filter, four rake fingers per user, four complex correlators per rake finger and 10 users in the cell, the number of operations (real multiply-adds) needed for a 5-stage SIC is upwards of 14 billion operations per second or giga-operations per second (GOPS). This amount of processing can easily overwhelm even the latest generation of general-purpose processors such as the TI TMS320C6x which delivers 1.6 giga-instructions per seconds (GIPS), but only 400 mega multiply-add operations per second [6]. At an anticipated power dissipation of 850 mW per processor, the overall power consumption of a SIC circuit based on such units will be quite large.

It is also worth noting that many operands are in the SIC or other MUD receiver that require only a few number of bits (i.e., multiplication with a 1-bit PN code sequence). This fact can be exploited in a dedicated ASIC datapath architecture but not in a general-purpose software programmable architecture.

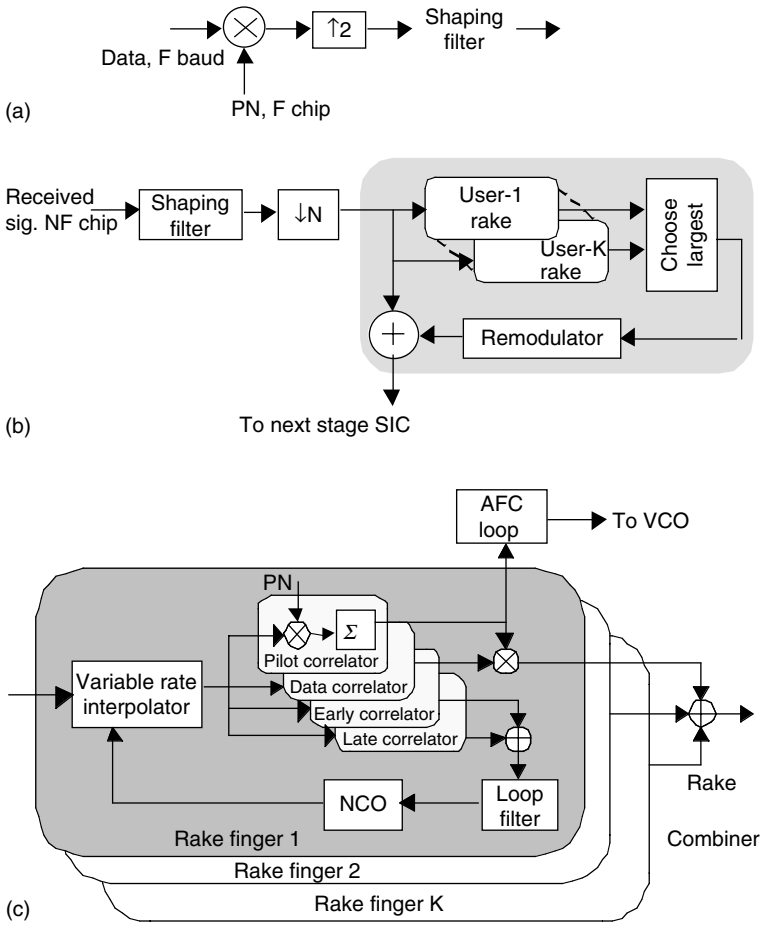



**FIGURE 26.1** Block diagram of (a) generic DSSS transmitter, (b) successive interference canceller for multiuser detection, and (c) rake receiver for a system with parallel pilot channel (i.e., IS-95).

### 26.2.2.2 Broadband Wireless Networks

Emerging broadband fixed wireless access systems provide high-speed connectivity between a cellular base station and a home or office building at data rates of a few Mbps to a few tens of Mbps. On the other hand, standardization activities that are currently targeting high-speed wireless mico-cellular (wireless LAN) systems are looking at delivering 10–20 Mbps over the air data rates in the near future, with higher rates projected in the long term.

It is generally accepted that in order to achieve such high data rates, beam switching or beamforming techniques must be integrated into the development of the nodes. In addition, single carrier systems must include adaptive equalization to overcome time varying channel impairments, while multicarrier systems based on OFDM will require a large number of subcarriers [7]. The signal processing requirements for such high data rate systems could easily mount into the tens of GOPS range, thus necessitating the development of ASICs.

Furthermore, the flexibility of digital implementation compared to an analog implementation of the down-conversion path makes a digital IF architecture more appealing. Figure 26.2 depicts the detailed block diagram of a single carrier high-speed wireless communication receiver complete with adaptive beamforming, adaptive equalization, and variable symbol rates. The flexibility offered by such an architecture can meet the demands of different systems requiring different levels of performance.

In this architecture, the direct digital frequency synthesizer (DDFS) serves three roles. First, it enables down-conversion of any carrier frequency up to half the sampling frequency of the analog-to-digital converter. Second, it can replace or complement a VCO for the purposes of carrier recovery, and third it can easily generate different phases needed by the beamforming circuit.

The variable rate decimator block is a key element in variable symbol rate systems where it is desired to maintain the same exact analog filtering, but yet accommodate user defined symbol rates. This is particularly important in wireless systems where a predefined data rate is difficult to guarantee due to statistical channel variations such as fading and shadowing. In such scenarios, the user can simply back-off on the symbol rate and provide connectivity albeit at a lower data rate.

The flexible decimation architecture depicted in Fig. 26.2 consists of two stages. The first is a course decimator block, which can decimate the signal by $2^N$ for $N = 0, 1, 2, \ldots, M$. This section is realized

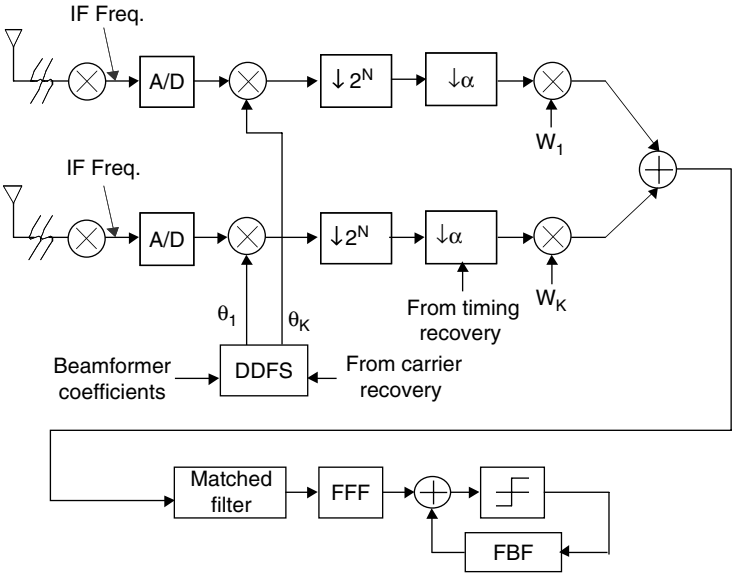



**FIGURE 26.2** Block diagram of an all-digital receiver for a single carrier system (i.e., QAM) featuring digital IF sampling, beamforming, variable symbol rate, adaptive equalization, all digital timing, and carrier recovery loops.

using a cascade of $N$ decimate by two stages. The second part of the decimator is a variable rate interpolator block, which can change the sampling rate by any value in the range of 2–4. Not only can this block be used to change the sampling frequency of the signal, but it is the vital element in the realization of an all digital timing recovery loop.

The matched filter is typically a fixed coefficient fixed impulse response (FIR) filter. This block is followed by a decision feedback equalizer (DFE) that helps mitigate the effects of intersymbol interference (ISI) caused by the multipath nature of the channel. The DFE is made up of two adaptive FIR filters referred to as the feedforward filter (FFF) and the feedback filter (FBF).

The amount of processing (in terms of real multiply-adds per second) needed to realize these blocks can easily run into several GOPS. As an example, a baseband QAM receiver consisting of a 30-tap matched filter, a 10-tap FFF and a 5-tap FBF adapted using the least mean squares (LMS) algorithm, and running at 10 Mbaud requires close to 2.5 GOPS of processing. Once the processing needs of the DDFS, variable rate filters, and the beamforming network are also factored in, the processing requirements can easily reach 7–8 GOPS.

## 26.2.3 VLSI Architectures for Signal Processing Blocks

### 26.2.3.1 Fixed Coefficient Filters

The most intuitive means of implementing a FIR filter is to use the direct form implementation presented in Fig. 26.3a [12]. Applying the transposition theorem to this filter we get the transposed structure shown in Fig. 26.3b. The two structures are identical in terms of I/O, however, the transposed form is ideal for high speed filtering operations since the critical path for an $N$ tap filter is always one multiplier delay plus one adder delay. The critical path of the direct form, however, is one multiplier delay plus $N$-1 adder delays. The fact is that the symbol rate for most wireless communication systems is a few tens of megahertz, whereas a typical multiplier in today's CMOS process technologies can easily reach speeds of 80–100 MHz. It is thus desirable to use the hybrid architecture shown in Fig. 26.3c where each multiplier

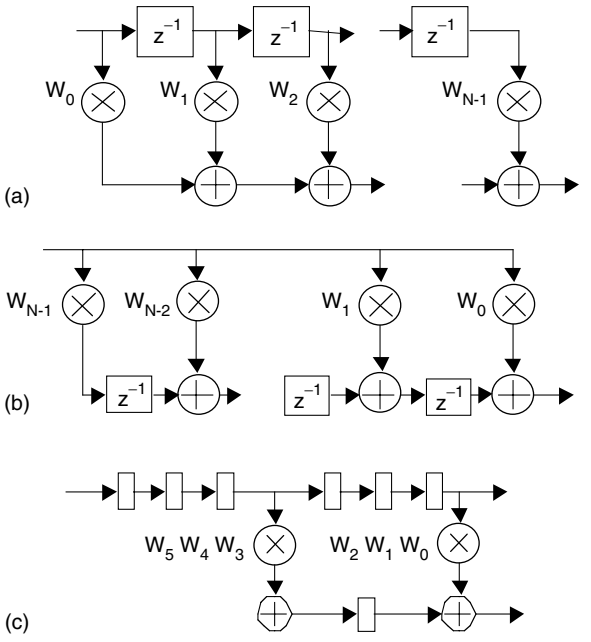



**FIGURE 26.3** Alternative FIR filter structures: (a) direct form FIR structure, (b) transposed form FIR structure, and (c) hybrid FIR structure.

accumulator is time-shared between several taps (three in this case) resulting in a more compact circuit for lower symbol rates.

The implementation of fixed coefficient FIR filters can be further simplified by moving away from the use of 2's complement number notation, and using a signed-digit number system in which each digit can take on one of three values $\{-1, 0, 1\}$. In general there are multiple signed-digit representations for the same number and a canonic signed-digit (CSD) representation can be defined for which no two nonzero digits are adjacent [8]. The added flexibility of signed-digit numbers allows us to realize the same coefficient using fewer nonzero coefficients than would be possible with a simple 2's complement representation. Using an optimization program, it is possible to design an FIR filter using CSD filters with as few as three or four nonzero digits for each coefficient. This could help significantly reduce the complexity of fixed coefficient multipliers since the number of partial products generated is directly proportional to the number of nonzero digits in the multiplier.

### 26.2.3.2 Direct Digital Frequency Synthesizer (DDFS)

Given an input frequency word $W$, a DDFS will produce a frequency proportional to $W$. The most common techniques for realizing a DDFS consist of first accumulating the frequency word $W$ in a phase accumulator and then producing the sine and cosine of the phase accumulator value using a table lookup or a coordinate rotation (CORDIC) algorithm. These two approaches are depicted in Fig. 26.4. The two metrics for measuring the performance of a DDFS are the minimum frequency resolution $\Delta$f and the spurious free dynamic range (SFDR). The frequency resolution can be improved by increasing the wordlength used in the accumulator, while the SFDR is affected by the wordlengths in both the accumulator as well as the sine/cosine generation block.

One of the main challenges in the development of the table lookup DDFS has been to limit the size of the sine/cosine table. This has been accomplished through two steps [9]. First, by exploiting the symmetry of the sine and cosine functions it is only necessary to store $\beta$ of the period of a sine wave and derive the remainder of the period through manipulation of the saved portion. Second, the number of bits per entry can be reduced by dividing the sine table between a coarse ROM and a fine ROM with the final result obtained after simple post-processing of the values. Combining these two techniques can result in the reduction of the sine tables by an order of magnitude or better.

In the CORDIC algorithm, Fig. 26.4, sine and cosine of the argument are calculated using a cascade of stages, each of which rotates its input complex vector by $\delta/2^k$ ($\delta = \pi/2$) if the $k$th bit of $W$ is 0 and $-\delta/2^k$ if the bit is a 1. Thus each stage performs the following matrix operation:

$$\begin{bmatrix} x_{\text{out}} \\ y_{\text{out}} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{\text{in}} \\ y_{\text{in}} \end{bmatrix} = \cos\theta \begin{bmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{bmatrix} \begin{bmatrix} x_{\text{in}} \\ y_{\text{in}} \end{bmatrix}$$

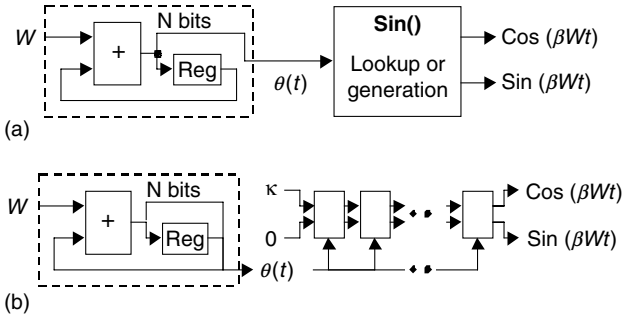



**FIGURE 26.4** Two most common DDFS architectures: (a) table lookup and (b) coordinate rotation (CORDIC).

In [10] a simplification of the CORDIC DDFS is presented in which for small $\theta$, $\tan(\theta)$ is simply approximated by $\theta$. In [11] a different modification to the CORDIC architecture is proposed that will facilitate low-power operation in cases where a sustained frequency is to be generated. This is achieved by calculating the necessary angle of rotation for each sampling clock period, and dedicating a single rotation stage in a feedback configuration to contiually rotate the phasor through the desired angle.

#### 26.2.3.3 Decimate/Interpolate Filters

Variable rate interpolation and decimation filters play a very important role in the development of highly flexible and self contained all-digital receivers. As previously mentioned, they are the critical element of all digital timing recovery loops as well as systems capable of operating at a host of user defined symbol rates. Additionally, digital resampling allows the ASIC designer to ensure that the clock frequency at all portions of the circuit are the minimum that they need to be to properly represent the signals. This could have significant impact on the size and power consumption of the resulting ASIC since power scales with the clock frequency and the square of the supply voltage. Thus, for a given circuit with a critical path of say $\tau$ seconds, if the data rate into the block is lowered by a factor $K$, then the frequency dependent portion of the dissipated power is also scaled by the same factor; however, additional power savings can be achieved by noting that the block now has $K\tau$ seconds to complete its task. Because the speed of a digital circuit is proportional to the supply voltage, we can reduce the supply voltage and still ensure that the circuit meets the speed constraints.

Given the coefficients of an FIR decimation or interpolation filter, the structure of choice for the realization of a decimate by $D$ or an interpolate by $D$ filter is the polyphase structure [12] shown in Fig. 26.5. The attractiveness of this structure is in the fact that the filter is always operated at the lower sampling frequency.

In many cases it is desirable to resample the signal by a power of $2^N$. In which case $N$ decimate (interpolate) by two stages can be cascaded one after the other. Each decimator will consist of a halfband filter followed by a decimator. The halfband filter could be realized using the polyphase structure to simplify its implementation. Moreover, these filters are typically very small consisting of anywhere from 7 to 15 taps depending on the specified stopband attenuation and the size of the transition band. Their implementation can be simplified by exploiting the fact that close to half of the coefficients are zero and the remainder are symmetric about the main tap due to the linear phase characteristics of the halfband filter. Finally, since these are fixed-coefficient filters, they can be realized using CS$D$ coefficients [13].

It is interesting to note that for the special case of a decimate (interpolate) by $2N$, it is possible to reuse the same hardware element and simply recirculate the data through it. In this architecture, the filter is run at the highest data sampling rate. The first pass through the filter will use up $1/2$ of its computational resources, the second pass will use up $1/4$ of the resources, and so on [14]. Although conceptually attractive, the clock generation circuit for such an architecture is quite critical and complex and this approach looses its appeal for recirculating factors greater than 3 or 4.

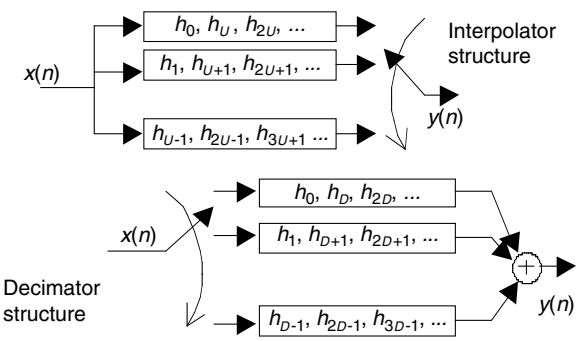



**FIGURE 26.5** Polyphase filter structures for interpolation and decimation.

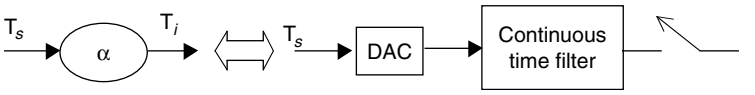



**FIGURE 26.6** Variable rate interpolation.

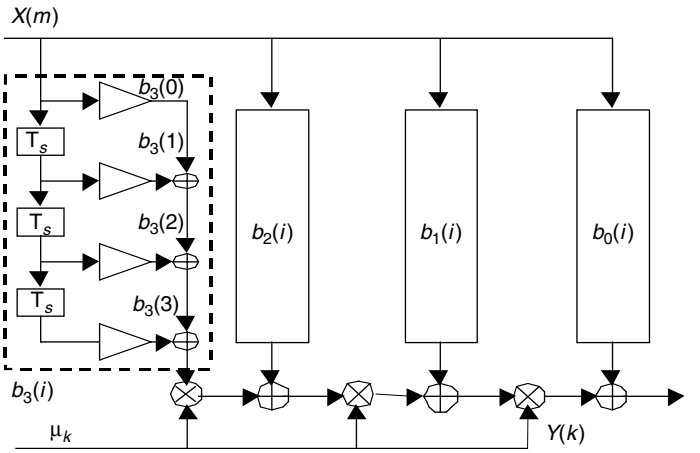



**FIGURE 26.7** Farrow structure.

In cases where the oversampling ratio is large (e.g., narrowband signal) an alternative approach using a cascaded integrator-comb (CIC) structure can be used to implement a multiplierless decimator. The interested reader is referred to [15] for a brief overview of a CIC ASIC.

The continuously variable decimator block shown in Fig. 26.2 can resample the input signal by any factor $\alpha$ in the range of 2–4. The operation of this block is equivalent to that shown in Fig. 26.6, where the input data $x(n)$, originally sampled at $1/T_s$ is resampled to produce an output sequence $y(n)$ sampled at $1/T_i$. The entire operation is performed digitally.

To better understand the operation of this block, let us define the variable $\mu_k$, to be the time difference between the output sample $y(k)$ and the most recent input sample $x_1$. The job of the variable rate interpolator is to weight the adjacent input samples $(\ldots, x_0, x_1, \ldots)$, based on the ratio $\mu_k/T_s$, and add the weighted input samples to obtain the value of the output sample, $y(k)$. Mathematically, a number of

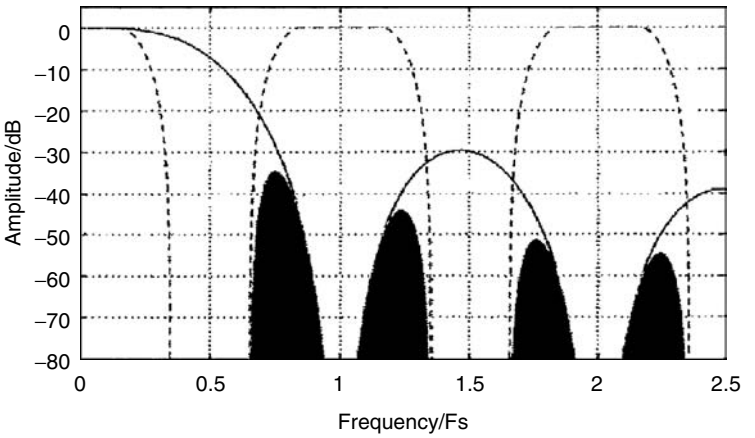



**FIGURE 26.8** Frequency response of polynomial-based interpolator [18].

interpolation schemes can perform the desired operation; however, many of them, such as sinc-based interpolation, require excessive computational resources for a practical hardware implementation. For real-time calculation, Erup et al. [16] found polynomial-based interpolation to yield satisfactory results while minimizing the hardware complexity. In this approach, the weights of the input samples are given as polynomials in the variable $\mu_k$ and can be easily implemented in hardware using the Farrow structure [17] shown in Fig. 26.7. In this sructure, all the filter coefficients are fixed and polynomials in $\mu_k$ are realized by nesting the multipliers as shown in Fig. 26.7.

The signal contained in the imageband will cause aliasing after resampling; however, proper choice of the coefficients in the Farrow structure can help optimize the frequency response of the interpolator for a particular application. An alternative method to determine the filter coefficients is outlined in (see Fig. 26.8) [18].

### 26.2.4 Conclusions

Section 26.2 reviewed trends in the wireless communications industry towards high speed data communications in both the macrocellular and the microcellular environments. The implication of these trends on the underlying digital circuits will move designers towards dedicated circuits and ASICs to meet these demands. As such the paper outlined the major signal processing tasks that these ASICs will have to implement.

## References

1. K. Pahlavan, et al., "Wideband local access: wireless LAN and wireless ATM," *IEEE Commun. Mag.,* pp. 34–40, Nov. 1997.
2. J. Mikkonen, et al., "Emerging wireless broadband networks," *IEEE Commun. Mag.,* vol. 36, no. 2, pp. 112–17, Feb. 1998.
3. E. Dahlman, Bjorn Gudmundson, M. Nilsson, and J. Skold, "UMTS/IMT-2000 based on wideband CDMA," *IEEE Commun. Mag.,* pp. 70–80, Sept. 1998.
4. Y. Furuya, "W-CDMA: an approach toward next generation mobile radio system, IMT-2000," *Proc. IEEE GaAs IC Symposium,* pp. 3–6, Oct. 1997.
5. A. Duel-Hallen, J. Holtzman, and Z. Zvonar, "Multiuser detection for CDMA systems," *IEEE Pers. Commun. Mag.,* pp. 46–58, April 1995.
6. http://www.ti.com/sc/docs/dsps/products.htm.
7. B. Daneshrad, et al., "Performance and implementation of clustered OFDM for wireless communications," ACM MONET special issue on PCS, vol. 2, no. 4, pp. 305–14, 1997.
8. H. Samueli, "An improved search algorithm for the design of multiplierless FIR filters with powers-of-two coefficients," *IEEE TCAS,* vol. 36, no. 7, pp. 1044–1047, July 1989.
9. H.T. Nicholas, III and H. Samueli, "A 150-MHz direct digital frequency synthesizer in 1.25 $\mu$m CMOS with $-90$ dBc spurious performance," *IEEE JSSC,* vol. 25, no. 12, pp. 1959–969, Dec. 1991.
10. A. Madisetti, A. Kwentus, and A.N. Willson, Jr., "A sine/cosine direct digital frequency synthesizer using an angle rotation algorithm," *Proc. IEEE ISSCC '95*, pp. 262–63.
11. E. Grayver and B. Daneshrad, "Direct digital frequency synthesis using a modified CORDIC," *IEEE ISCAS,* June 1998.
12. J.G. Proakis and D.G. Manolakis, Introduction to Digital Signal Processing, Macmillan, London, 1988.
13. J. Laskowsky and H. Samueli, "A 150-MHz 43-tap halfband FIR digital filter in 1.2-$\mu$m CMOS generated by silicon compiler." *Proc. IEEE CICC '92*, pp. 11.4/1–4, May 1992.
14. T.J. Lin and H. Samueli, "A VLSI architecture for a universal high-speed multirate FIR digital filter with selectable power-of-two decimation/interpolation ratios," *Proc. ICASSP '91*, pp. 1813–816, May 1991.

15. A. Kwentus, O. Lee, and A. Willson, Jr., "A 250 Msample/sec programmable cascaded integrator-comb decimation filter," *VLSI Signal Processing, IX, IEEE*, New York, pp. 231–40, 1996.
16. L. Erup, F.M. Gardner, and R.A. Harris, "Interpolation in digital modems. II. Implementation and performance," *IEEE Trans. on Commun.*, vol. 41, no. 6, pp. 998–1008, June 1993.
17. C.W. Farrow, "A continuously variable digital delay element," *Proc. ISCAS '88*, pp. 2641–645, June 1988.
18. J. Vesma and T. Saramaki, "Interpolation filters with arbitrary frequency response for all-digital receivers," *IEEE ISCAS '96*, pp. 568–71, May 1996.

## 26.3 Communication System-on-a-Chip

*Samiha Mourad and Garret Okamoto*

### 26.3.1 Introduction

Communication traffic worldwide is exploding: wired and wireless, data, voice, and video. This traffic is doubling every 100 days and it is anticipated that there will be a million people online by 2005. Today, more people are actually using mobile phones than are surfing the Internet. This unprecedented growth has been encouraged by the deployment of digital subscriber lines (DSL) and cable modems, which telephone companies have provided promptly and at a relatively low price. Virtual corporations have been created because of the availability and dependability of communication products such as laptops, mobile phones and pagers, which all support mobile employees. For example, vending machines may contact the suppliers when the merchandise level is low so that suppliers remotely vary the prices of the merchandise according to supply and demand.

With such proliferation in communication products and the need for a high volume, high speed transfer of data, new standards such as ATM and ITU-T are being developed. In addition, a vast body of knowledge, central to problems arising in the design and planning of communication systems, has been published; however, in fabricating products to meet these needs, the industry has continually attempted to use new design approaches that have not been fully researched or documented.

Communication devices need to be of small size and low power dissipation for portability and need to be operated at very high speed. Any of these devices, as other digital products, may consist of a single integrated circuit (IC) or more likely many ICs mounted on a printed circuit broad (PCB). Although the new technology (small feature size) has resulted in higher speed ICs, the transfer of data from one IC to another still creates a bottleneck of information. The I/O pads, with their increasing inductance, cause supply surges that compromise signal integrity. As an alternative to PCB design, another design approach known as multichip module (MCM) consists of placing more than one chip in the same package. The connections between modules have a large capacitive load that slows down communication among all of the modules. In the late 1990s, a new paradigm design called system-on-a-chip (SoC) has been successfully used to integrate the components of an entire system on one chip. This is in contrast to the traditional design where the components are implemented in separate ICs and then assembled on a PCB.

Section 26.3 describes the new design paradigm of a SoC and its beneficial attributes are outlined. The remainder of the paper concentrates on communication devices and Section 26.3.3 emphasizes the need for these systems. Descriptions of communication SoCs and projections on their characteristics are given in Section 26.3.4. Latency, an important attribute, is the subject of Section 26.3.5; and Section 26.3.6 describes the integration of these systems with analog parts in MCM.

### 26.3.2 System-on-a-Chip (SoC)

The shift toward very deep submicron technology has encouraged IC designers to increase the complexity of their designs to the extent that an entire system is now implemented on a single chip. To increase the design productivity and decrease time-to-market, reuse of previously designed modules

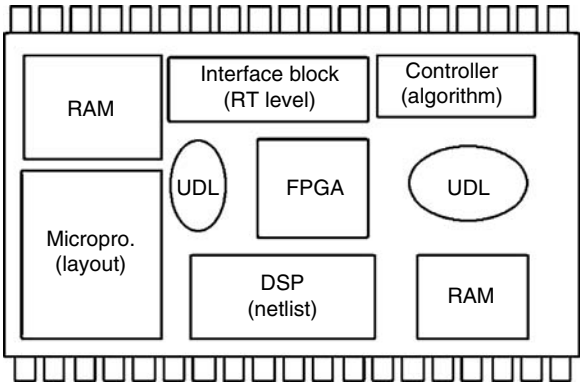



**FIGURE 26.9** A system-on-a-chip (SoC).

is becoming common practice in SoC design; however, the reuse approach is not limited to in-house designs. It is extended to modules that have been designed by others as well. Such modules are referred to as *embedded cores*. This design approach has encouraged the founding of several companies that specialize in providing embedded cores to service multiple customers. It is predicted that in the near future, cores, of which 40% to 60% will be from external sources (Smith 1997), will populate 90% of a chip. Except for a very few, individual companies do not have the wide range of expertise that can match the spectrum of design types in demand today.

Core-based design, justified by the need to decrease time-to-market, has created a host of challenging problems for the design and testing community. First, there are legal issues for the core provider and the user, regarding the intellectual property (IP). Second, there are problems with integrating and verifying a mix of proprietary and external cores that are more involved than simply integrating ICs on a PCB.

A typical SoC configuration is shown in Fig. 26.9. It consists of several *cores* that are also referred to as *modules, blocks*, or *macros*. Often, these terms are used interchangeably. These cores may be DSP, RAM modules, or controllers. This same image of an SoC may be perceived as a PCB with the cores being the ICs mounted on it.

It also resembles standard cells laid on the floor of an IC. In the latter case, the blocks are of elementary gates of the same layout height. That is, they are all ICs in the PCB case or all standard cells in the IC case. For an SoC, they may consist of a several types, as described below. A UDL is a user defined logic that is basically equivalent to glue logic in microprocessors.

Cores are classified in three categories: hard, firm, and soft (Gupta 1997). *Hard cores* are optimized for area and performance and they are mapped into a specific technology and possibly a specific foundry. They are provided as layout files that cannot be modified by the users. *Soft cores*, on the other hand, may be available as HDL technology-independent files. From a design point of view, the layout of a soft core is flexible, although some guidelines may be necessary for good performance. The flexibility allows optimization to the desired levels of performance or area. *Firm cores* are usually provided as technology-dependent netlists using library cells whose size, aspect ratio, and pin location can be changed to meet the customer needs. Table 26.1 summarizes the attributes of reusable cores. The table indicates a clear trade-off between design flexibility on one hand, and predictability and hence time-to-market performance complexity on the other. Soft cores are easily embedded in a design. The ASIC designers have complete control over the implementation of this core, but it is the designer's job to optimize it for area, test, or power performance.

Hard cores are very appropriate for time critical applications, whereas soft cores are candidates for frequent customization. The relationship between flexibility and predictability is illustrated in Fig. 26.10. The cores can also be classified from a testing perspective. For example, there is typically no way to test a hard core unless the supplier provides a test set for this core, whereas a test set for the soft core needs to

**TABLE 26.1** Categorizing Reusable Cores

| Type | Flexibility | Design Flow | Representation | Libraries | Process Technology | Portability |
|---|---|---|---|---|---|---|
| Soft | Very flexible<br>Unpredictable | System design | Behavioral | Not applicable | Independent | Unlimited |
| | | RTL design | RTL | | | |
| Firm | Flexible | Floor planning | RTL, blocks<br>Netlist | Reference | Generic | Library mapping |
| | | Placement | | Footprint, timing model | | |
| Hard | Inflexible<br>Predictable | Routing<br>Verification | Polygon data | Process specific library and design rules | Fixed | Process mapping |

*Source*: Hunt 1996.

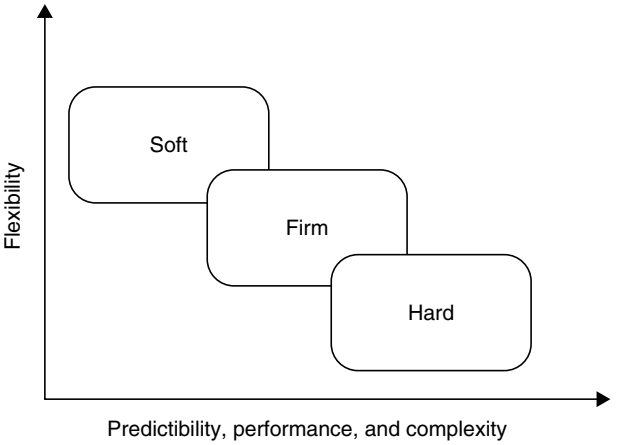



**FIGURE 26.10** Trade-offs among types of cores (Hunt 1996).

be created if not provided by the core provider. This makes hard cores more demanding when developing a test strategy for the chip. For example, it would be difficult to transport through hard cores a test for an adjacent block that may be another core or a UDL component. In some special cases, the problem may be alleviated if the core includes well described testability functions.

#### 26.3.2.1 Design and Test Flow

An integrated design and test process is highly recommended. This approach cannot be more appropriate than it is for core-based systems. Conceptually, the SoC paradigm is analogous to the integration of several ICs on a PCB, but there is a fundamental difference. Whereas in a PCB the different ICs have been designed, verified, fabricated, and tested independently from the board, fabrication and testing of an SoC are done only after integration of the different cores. This fact implies that even if the cores are accompanied by a test set, incorporation of the test sets is not that simple and must be considered while integrating the system. In other words, reuse of design does not translate to easy reuse of the test set. What makes this task even more difficult is that the system may include different cores that have different test strategies. Also, the cores may cover a wide range of functions as well as a diverse range of technologies, and they may be described using different HDL languages, such as Verilog, VHDL, and Hardware C to GDSII.

The basic design flow applies to SoC design in the sense that the entire system needs to be entered, debugged, modified for testability, validated, and mapped to a technology; but all of this has to be done in an *integrated framework*. Before starting the design process, an overall strategy needs to be chartered to

facilitate the integration. In this respect, the specification phase is enlarged and a test strategy is included. This move toward more design on the system level and less time on the logic level.

The design must first be partitioned. Then decisions must be made on such questions as:

- Which partition can be instantiated by an existing core?
- Should a core be supplied by a vendor or done in-house?
- What type of core should be used?
- What is the integration process to facilitate verification and testing?

Because of the wide spectrum of core choices and the diversity of design approaches, SoC design requires a *meta-methodology*. That is, a methodology that can streamline the demands of all other methodologies used to design and test the reusable blocks as well as their integration with user defined logic. To optimize on the core-based design, an industry group deemed it necessary to establish a common set of specifications. This group, known as the virtual socket interface alliance (VSIA), was announced formally in September 1996. Its intent is to establish standards that facilitate communication between core creators and users, the SoC designers (IEEE 1999a).

An example of using multiple cores is the IBM-designed PowerPC product line, based on the PowerPC 40X chip series (Rincon 1997). The PowerPC micro-controller consisted of a hard core and several soft cores. For timing critical components such as the CPU, a hard core was selected, while soft cores were used for peripheral functions such as the DMA controller, external bus interface unit (EBIU), timers, and serial port unit (SPU). The EBIU may be substituted by, say, a hard core from Rambus.

A change in the simulation and synthesis processes is required for embedded cores due primarily to the need to protect the intellectual property of the core provider. Firm cores may be encrypted in such a manner as to respond to the simulator without being readable by humans. For synthesis, the core is instantiated in the design. In the case of a soft core, sometimes the parameters are scaled to meet the design constraints. To preserve the core performance, the vendor may include an environment option to prevent the synthesis program from changing some parts of the design. This will protect the core during optimization, but the designer may remove such an option and make some changes in the design. A hard or a firm core is treated as a black box from the library and goes through the synthesis process untouched.

#### 26.3.2.2 Advantages of SoCs

The overall size of the end product is reduced because manufacturers can put the major system functions on a single chip, as opposed to putting them on several chips. This reduces the total number of chips needed for the end product. For the same reason, the power consumption is reduced.

SoC products provide faster chip speeds due to the integration of the components/functions into one chip. Many applications such as high-speed communication devices (VoIP, MoIP, wireless LAN, 3G cellular phones) require chip speeds that may be unattainable with separate IC products. This is primarily due to the physical limitations of moving data from one chip to another, through bonding pads, wires, buses, etc. Integrating chip components/functions into one chip eliminates the need to physically move data from one chip to another, thereby producing faster chip speeds. Another important advantage of SoCs is the reuse of previously designed circuits, thereby reducing the design process time. This consequently translates into shorter time-to-market. In addition to decreasing time-to-market, it is very important to decrease the cost of packaging and testing, which are constantly increasing with the finer technology features. Instead of testing several chips and the PCB on which they are assembled, testing time is reduced to only one IC. SoCs are, however, very complex and standards are now being developed to facilitate their testing (IEEE 1995b). In the remainder of this paper, we focus on communication systems that we will refer to as comm. SoC or simply SoC.

### 26.3.3 Need for Communication Systems

Public switched telephone networks (PSTN) are becoming congested due to increasing Internet traffic as shown in Fig. 26.11. This drives the development of broadband access technology and high-speed optical

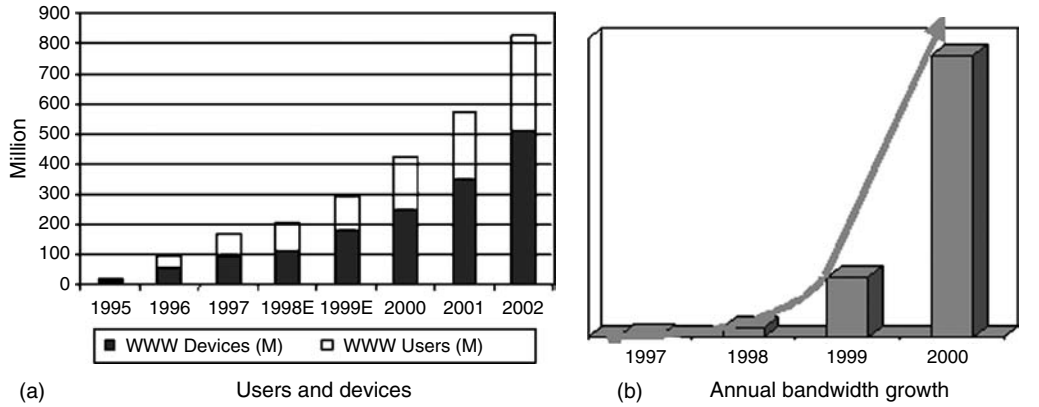



**FIGURE 26.11** Internet growth.

networks. Another important factor is the convergence of voice, data, and video. As a consequence, there is a need for low and uniform latency devices for real time traffic. In addition, Internet service providers (ISP) and corporate Intranet are needed for voice and data IP gateways. Mobile users drive the development of wireless and satellite devices. In addition, there is an increasing demand for routers/switches, DSL modems, etc.

All needs mentioned above require smaller size and faster communication devices. Telephone calls that used to last an average of three minutes now exceed an hour or more when connected to the Internet. This has resulted in increasing the demand on DSL that transmit data over Internet protocols (IP) such as voice-over-IP (VoIP), mobile-over-IP (MoIP), and wireless requires speeds that may be unattainable with separate IC products. Examples of products:

1. 2G and 3G wireless devices (CDMA2000, WCDMA), etc.
2. DSL modems
3. Infrastructure, carrier, and enterprise circuit, packet switched and VoIP devices
4. Satellite modems
5. Cable modems and HFC routing devices
6. De/MUX for data stream on optical network
7. Web browsers (WAP) or short messaging systems (I-mode)
8. LAN telephony
9. ATM systems
10. Enterprise, edge network and media-over-IP switches and high-speed routers
11. Wireless LAN (IEEE 802.11 IEEE 802.11a, and IEEE 802.11b)
12. Bluetooth

Maybe the most important example of an emerging wireless communication standard is Bluetooth. This is a wireless personal area network (PAN) technology from the Bluetooth special interest group (SIG), founded in 1998 by Ericsson, IBM, Intel, Nokia, 3Com, Lucent, Microsoft, Motorola, and Toshiba. Bluetooth is an open standard for short-range transmission of digital voice and data between mobile devices (cellular phones, PDAs, laptops) and desktop devices. Bluetooth may provide a common standard to enable PDAs, laptop and desktop computers, cellular phones, thermostats, and virtually every other home and business electronic device to communicate with each other. Manufacturers will rely on SoC advances to help reach the target of $5 added cost to a consumer appliance by 2001. A study by Merrill Lynch projected that Bluetooth semiconductor revenue will reach $3.4 billion in 2005, with

Bluetooth included in 1.7 billion devices that year, and the Bluetooth SIG estimated that the technology would be a standard feature in 100 million mobile phones by the end of 2001.

### 26.3.4 Communication SoCs

The exponential growth of the Internet and the bandwidth shown in Fig. 26.11, indicate that more communication products are geared towards this technology, which requires a communication mode different than that used in traditional switching telephony. For example, in a PSTN, circuit switching is used and requires a dedicated physical circuit through the network during the life of a telephone session. In Internet and ATM technology, however, packet switching is used. Packet switching is a connectionless technology, where a message is broken into several small packets to be sent to a destination. The packet header contains the destination and source address, plus a sequence number so that the message can be reassembled.

There is a paradigm shift in digital communication motivated by the evolution of Internet as mission critical service that demands migration from circuit switch to packet switch. The older paradigm supported the data traffic part of the telephone networks. Whereas the new paradigm support the convergence of voice, data, and video and require a new class of media-over-IP systems voice traffic as part of the data network, thus requiring communication SoC for VoIP.

Most communication SoC consists of few components that are clustered around a central processing unit (CPU), which controls some or all of the following: (1) Packet processing, (2) Programmable DSP for data and signaling algorithm/protocol implementation, (3) I/O for interface with voice and data network such as ATM, PCI, Ethernet, H100/110, (4) memory system for intermediate storage of voice and data streams, (5) hardwired DSP or accelerators for Codec and multi level mod/demod to increase system throughput, and (6) MPEG cores for media-over-IP MoIP processing (Fig. 26.12).

Communication SoCs are actually a mix of software and hardware. Some of the circuits contain hardwired algorithms for code processing, but the software can be stored on the chip for protocols that process data. Figure 26.13 shows the software for a typical VoIP. This include several layers of software and IP such as:

1. *Telephony signaling*: Network interface protocol, which contains address translation and parsing and protocols such as H-3xx, media gateway control protocol (MGCP), and real time conferencing protocol (RTCP).
2. *Voice processing*: includes voice-coding unit using G.xx protocol, voice activation detection (VAD), comfort noise generation (CNG), which is used in fax-to-fax communication.
3. *User interface*: provides system services to the user such as key pad and display drivers and user procedures.

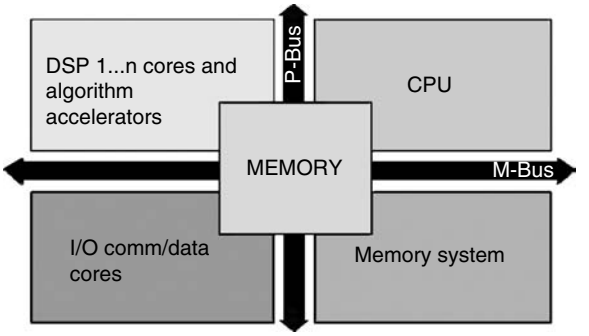



**FIGURE 26.12** Components of a communication SoC.

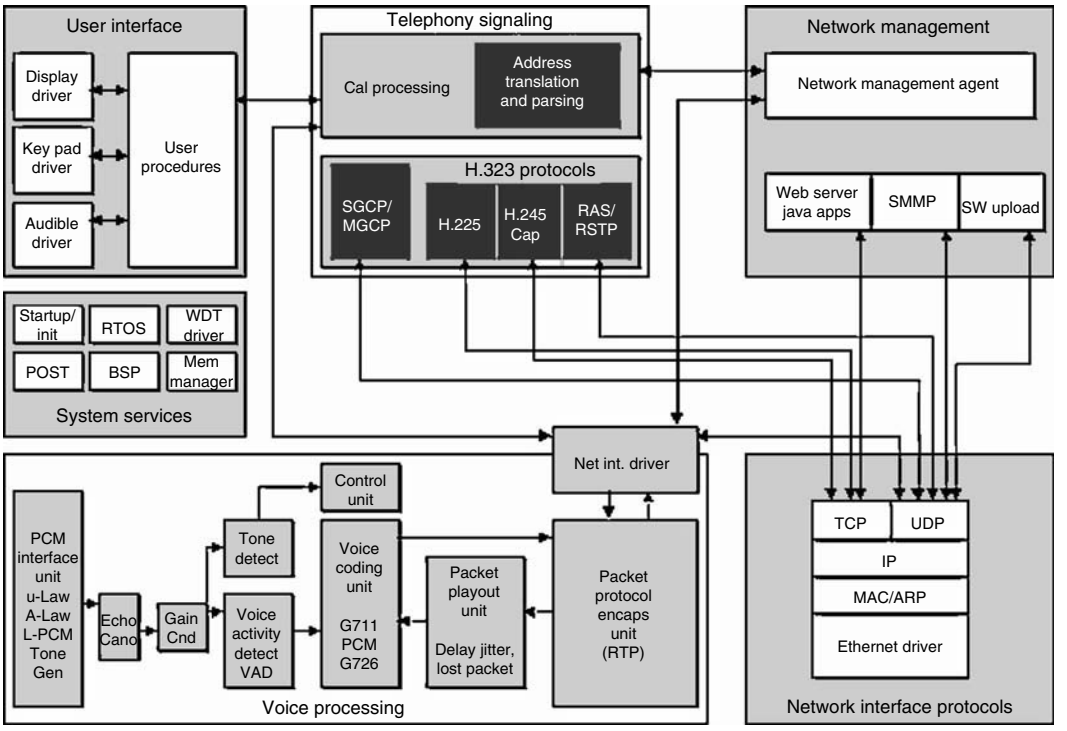



**FIGURE 26.13** Software for VoIP SoC.

4. *Network management*: software upload and handling Java applets.
5. *Network Interface Protocols*: such as transmission control protocol (TCP), user datagram protocol (UDP), which is a TCP/IP, and Ethernet driver.

Other software and protocol may also be included such as packet processing and network management protocols, call control/signaling protocols/fax and modem tone detection, echo canceller, VAD, CNG, read to order systems (RTOS), and other software components for MoIP systems. Communication SoCs that accomplish the above tasks are expected to grow in size as projected in Fig. 26.14. The number of gates per chip will increase from one million in 1999 to 7 M in 2003. A major component in a communication SoC is the embedded memory banks, which is also expected to increase from 1 to 16 Mbit. The type of memory used will change from static RAMs (SRAM) to enhanced dynamic RAMs (EDRAM), which are much more compact.

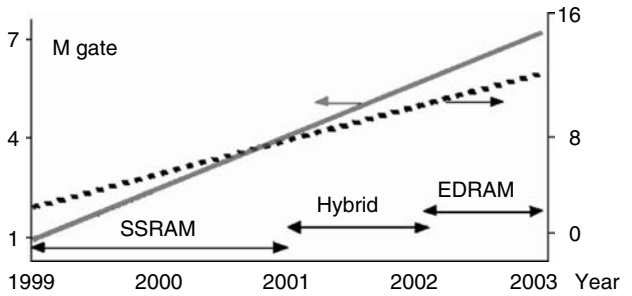



**FIGURE 26.14** Communications SoCs: Density and memory size.

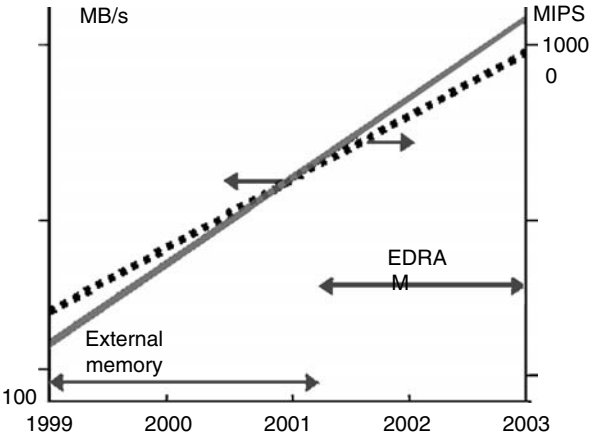



**FIGURE 26.15** Communications SoCs processing power and memory handwidth.

The processing power of these SoCs is also expected to increase as illustrated in Fig. 26.15. The processing power is measured in million instructions per second (MIPs). It is predicted to grow from 100 to 1000 MIPs (dashed line) from 1999 to 2003. In same time period, the memory bandwidth (solid line) will increase from 100 to 1000 Mbits. The growth of the number of DSP processors by SoC is shown in Fig. 26.16a. With all of this growth, it is interesting that the price of SoCs is estimated to decrease according to the trend shown in Fig. 26.16b.

Several predictions were given to the bandwidth of communication chips. Two of these predictions are shown in Fig. 26.17. One assumes that the bandwidth will triple each year in the next 25 years as illustrated by the solid line (George Dilder-Telecosm). The other shows that the growth will be at the rate of 8–16 times a year [SUN Microsystems]. In the 1990s, Bill Gates claimed that "we will have infinite bandwidth in a decade of time (Gates 1994)."

### 26.3.5 System Latency

Latency is defined as the delay experienced a certain processing stage. The latency trends in Fig. 26.18 refer to the time taken to map the voice data into a packet to be transmitted. Three main types of latency are usually identified:

- Frame/packetization Delay
- Media processing delay/complexity of the system
- Bridging delay, e.g., used for conferencing or multi SoC system

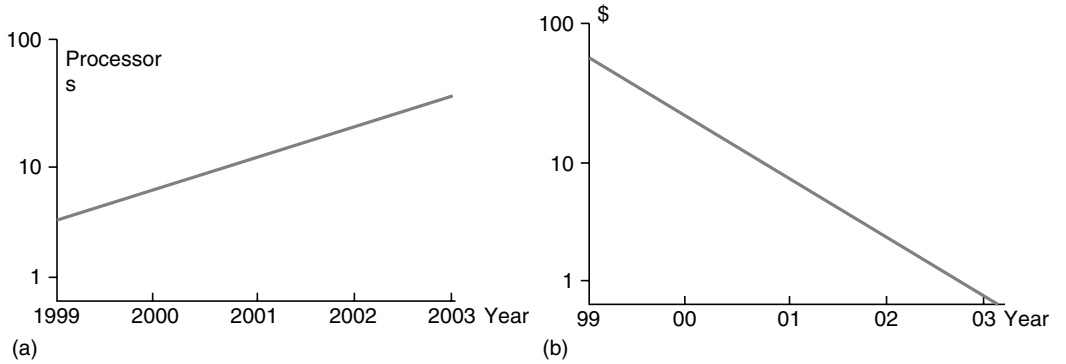



**FIGURE 26.16** (a) Number of DSP processors per SoC. (b) Price per functional VoIP channel.

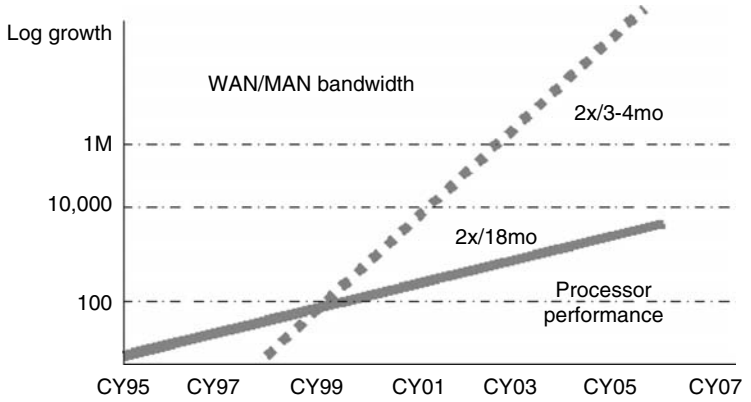



**FIGURE 26.17** Bandwidth trends.

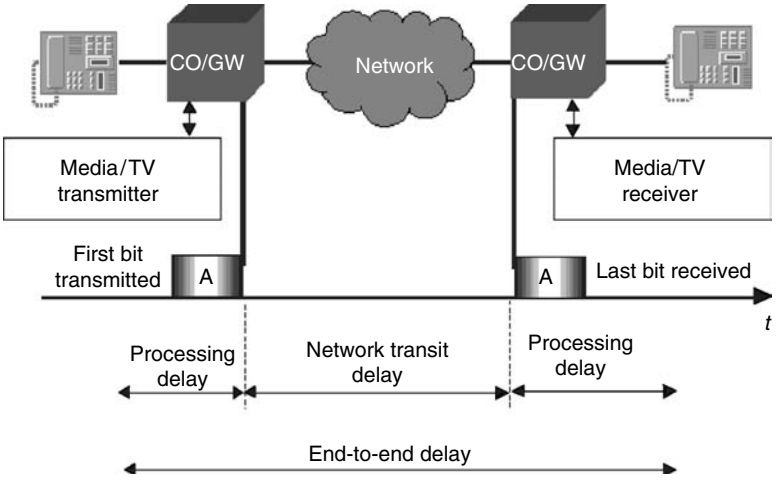



**FIGURE 26.18** System latency.

These delays may occur at different times in the life of the data in the communication system. A simplified communication system is shown in Fig 26.18. It starts with the sender transmitting data through the network to a receiver at the other end. The total system latency is known as the end-to-end delay. It consists of the time taken to send the first bit of a packet to the time it takes to receive the last bit in the stream of data, i.e.,

- Delay in processing the data at the sending end
- Transit delay within the network
- Delay in processing the data at the receiving end

With the use of SoC, latency has been reduced and this reduction is projected to continue as the technology feature is getting finer. The trend is illustrated in Fig. 26.19. Several SoCs may themselves be integrated in one multichip module (MCM) as will be discussed next.

### 26.3.6 Communication MCMs

Digital communication SoCs are usually connected to external analog functions and I/O as depicted in Fig. 26.20. In order to optimize the interface between digital SoCs and analog functions, it is beneficial to

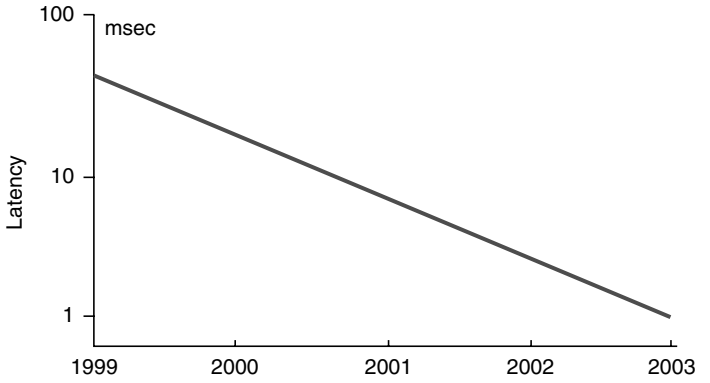



**FIGURE 26.19** Latency for voice to packet in communication SoCs.

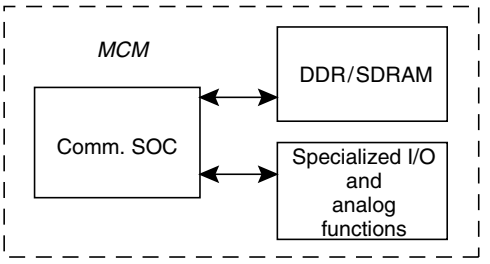



**FIGURE 26.20** Communication MCMs.

integrate both designs in a MCM. The simplest definition for an MCM is a single electronic package containing more than one IC (Doanne 1993). An MCM then combines high performance ICs with a custom-designed common substrate structure that provides mechanical support for the chips and multiple layers of conductors to interconnect them. Such an arrangement takes advantage of the performance of the ICs because the interconnect length is much shorter.

Multichip modules are not new, they preceded SoC. They have several advantages because they improve the maximum external memory bandwidth achieved, reduce size and weight of the product, increase the operating speed, and decrease power dissipation of the system; however, they are limited by wiring capacitance to frequencies below 150 MHz, e.g., Sony's HandyCam. Thus, they are limited by slower memory in comparison with the massive parallel processing power of an SoC with embedded memory.

MCM wide bus pin out is restricted by cost and yield in comparison with an SoC that provides high throughput data processing with wide 256–1024 bit on chip data bus. System configurability is harder to achieve in MCM than in SoC that are software configurable.

Analog and digital functions are separately optimized in MCM while in SoC many analog functions are optimized and their yield improved by using on chip integrated DSP algorithms. Multiple communication SoCs and analog functions can be packaged on a single MCM. The advantage of MCMs is even more pronounced when the package is enhanced. For example, flip-chips may be used or even more advanced package.

The interconnections between the various SoCs and the memory chips are the major paths for crosstalk and other types of signal distortion. Reducing the routing length of the connection will help to increase the operation speed. This can be achieved with a chip-on-chip (CoC) module. The metal redistribution layers were fabricated on the top of the processor and the two memory chips, while the original bond pads still remained for the wire bonding to the substrate. The memory chips can be mounted on the top of the processor using flip-chip technology. Redistribution layers have been used to replace the bond wires and traces on the substrate to provide the interconnections between memory chip

and processor. Since Know Good Die memory chips are usually used, testing only requires open/short test between the processor and memory chips. No burn-in and extensive memory tests are required, so the connection to the package ball can be removed as a new test program is implemented with the open/short test of the memory interface through other IO paths of the VGA processor.

### 26.3.7 Summary

The broadband access, infrastructure, carrier, and enterprise Communication SoCs will demand higher MIPS, integration, and memory bandwidth. They will also demand lower latency, power dissipation, and cost/channel or function. Comm. SoC utilizes programmable DSP, hardwired DSP accelerators, and I/O to implement Comm. protocols and systems in a highly integrated form. Higher memory access frequency, DSP interface speeds, and specialized analog functions will demand the integration of Comm. SoCs on Comm. MCM.

## References

Batista, Elisa, "Bluetooth Promises and Hurdles," Wired News, June 2000.

Doanne, D.A. and P.D. Franzon, Eds. (1993), *Multichip Module Technologies and Alternatives: The Basics,* Van Nostrand Reinhold, New York.

Gehring and Koutroubinas, "Designing cableless devices with the bluetooth specification," Communication Systems Design, February 2000.

Gupta, R.K. and Y. Zorian (1997), "Introduction to core—based system design," *IEEE Des. Test Comput.*, Vol. 14, No. 4, pp. 15–25.

Hunt, M. and J.A. Rowson (1996), "Blocking in a system on a chip," *IEEE Spectrum,* Vol. 36, No. 11, pp. 35–41.

IEEE (1999a), P1450 Web site *http//grouper.ieee.org/groups/1450/*.

IEEE (1999b), P1500 Web site *http//grouper.ieee.org/groups/1500/*.

Mourad, S. and Y. Zorian, Principles of testing electronic systems, Wiley, 2000.

Mourad, S. and B. Greene (2000), "Scan-path based testing of system on a chip," *Proc. IEEE International Conference of Electronics, Circuits and Systems,* Cyprus, pp. 1081–1084.

Murray, B.T. and J.P. Hayes (1996), "Testing ICs: getting to the core of the problem," *IEEE Computer,* Vol. 29, No. 11, pp. 32–38.

Okamoto, G, *Smart Antenna Systems and Wireless LANs,* Kluwer Academic Publishers, Boston, MA, 1999.

Okamoto, G., S.-S. Jeng, and G. Xu, "Evaluation of timing synchronization algorithms for the smart wireless LAN system," *Proceedings of the IEEE VTC '99 Conference,* May 1999, pp. 2014–2018.

Okamoto, G. and C.-W. Chen, "Capacity improvement of smart antenna systems via the maximum SINR beam forming algorithm," *Proceedings of the ICSPAT 2000 Conference,* October 2000.

Okamoto, G., et al., "An improved algorithm for dynamic slot assignment for the SWL system," *Proceedings of the Asilomar 2000 Conference,* Pacific Grove, CA, October 2000.

Smith, G. (1997), "Test and system level integration," *IEEE Des. Test Comput.*, Vol. 14, No. 4.

Varma, P. and S. Bhatia (1997), "A structured test reuse methodology for core-based system chip," *Proc. IEEE International Test Conference,* pp. 294–302.

Zorian, Y. (1993), "A distributed BIST control scheme for complex VLSI devices," *Proc. 11th IEEE VLSI Test Symposium,* pp. 6–11.

Zorian, Y. (1997), "Test requirements for embedded core-based systems and IEEE P-1500," *Proc. IEEE International Test Conference,* pp. 191–199.

Zorian, Y., et al. (1998), "Testing embedded-core based system chips," *Proc. IEEE International Test Conference,* pp. 135–149.

VSI (1998), VSI Alliance Web site *http:/www.vsi.org/*. http://www.digianswer.com/bluetooth/.

## 26.4 Communications and Computer Networks

*Mohammad Ilyas*

The field of communications and computer networks deals with efficient and reliable transfer of information from one point to another. The need to exchange information is not new but the techniques employed to achieve information exchange have been steadily improving. During the past few decades, these techniques have experienced an unprecedented and innovative growth. Several factors have been and continue to be responsible for this growth. The Internet is the most visible product of this growth and it has impacted the life of each and every one. Section 26.4 describes salient features and operational details of communications and computer networks.

The contents of Section 26.4 are organized in several subsections. Section 26.4.1 describes a brief history of the field of communications. Section 26.4.2 deals with the introduction of communication and computer networks. Section 26.4.3 describes operational details of computer networks. Section 26.4.4 discusses resource allocation mechanisms. Section 26.4.5 briefly describes the challenges and issues in communication and computer networks that are still to be overcome. Section 26.4.6 summarizes the article.

### 26.4.1 A Brief History

Exchange of information (communications) between two or more entities has been a necessity since the existence of human life. It started with some form and shape of human voice that one entity can create and other(s) can listen and interpret. Over a period of several centuries, these voices evolved into languages. As the population of the world grew, more and more languages were born. For a long time, languages were used for face-to-face communications. If there were ever a need to convey some information (a message) over a distance, someone would be briefed and sent to deliver the message to a distant site. Gradually, additional methods were developed to represent and exchange the information. These methods included symbols, shapes, and eventually alphabets. This development facilitated information recording and use of nonvocal means for exchanging information. Hence, preservation, dissemination, sharing, and communication of knowledge became easier.

Until about 150 years ago, all communication was via wireless means and included smoke signals, beating of drums, and use of reflective surfaces for reflecting light signals (optical wireless). Efficiency of these techniques was heavily influenced by the environmental conditions. For instance, smoke signals were not very effective in windy conditions. In any case, as we will note later, some of the techniques that were in use centuries ago for conveying information over a distance, were similar to the techniques that we currently use. The only difference is that the implementation of those techniques is exceedingly more sophisticated now than it was centuries ago.

As the technological progress continued and electronic devices started appearing on the surface, the field of communication also started making use of the innovative technologies. Alphabets were translated into their electronic representations so that information may be electronically transmitted. Morse code was developed for telegraphic exchange of information. Further developments led to the use of telephone. It is important to note that in earlier days of technological masterpieces, users would go to a common site where one could send a telegraphic message over a distance or could have a telephonic conversation with a person at a remote location. This was a classic example of resource sharing. Of course, human help was needed to establish a connection with remote sites.

As the benefits of the advances in communication technologies were being harvested, the electronic computers were also emerging and making the news. The earlier computers were not only expensive and less reliable, they were also huge in size. For instance, the computers that used vacuum tubes, were of the size of a large room and used roughly about 10,000 vacuum tubes. These computers would stop working if a vacuum tube had burnt, and the tube would need to be replaced by using a ladder. On the average, those computers would function for a few minutes, before another vacuum tube's replacement was

necessary. A few minutes of computer time was not enough to execute a large computer program. With the advent of transistors, computers not only became smaller in size, less expensive, but also more reliable. These aspects of computers resulted in their widespread applications. With the development of personal computers, there is hardly any side of our lives that has not been impacted by the use of computers. The field of communications is no exception and the use of computers has escalated our communication capabilities to new heights.

## 26.4.2 Introduction

Communication of information from one point to another in an efficient and reliable manner has always been a necessity. A typical communication system consists of the following components as shown in Fig. 26.21:

- Source that generates or has the information to be transported
- Transmitter that prepares the information for transportation
- Transmission medium that carries the information from one end to the other
- Receiver that receives the information and prepares it for delivering to the receiver
- Destination that takes the information from receiver and utilizes it as necessary

The information can be generated in analog or in digital form. Analog information is represented as a continuous signal that varies smoothly in time. As one speaks in a microphone, an analog voice signal is generated. Digital information is represented by a signal that stays at some fixed level for some duration of time followed by a change to another fixed level. A computer works with digital information that has two levels (binary digital signals). Figure 26.22 shows an example of analog and digital signals. Transmission of information can also be in analog or in digital form. Therefore, we have the following four possibilities in a communication system [21]:

- Analog information transmitted as an analog signal
- Analog information transmitted as a digital signal
- Digital information transmitted as an analog signal
- Digital information transmitted as a digital signal


**FIGURE 26.21** A typical communication system.

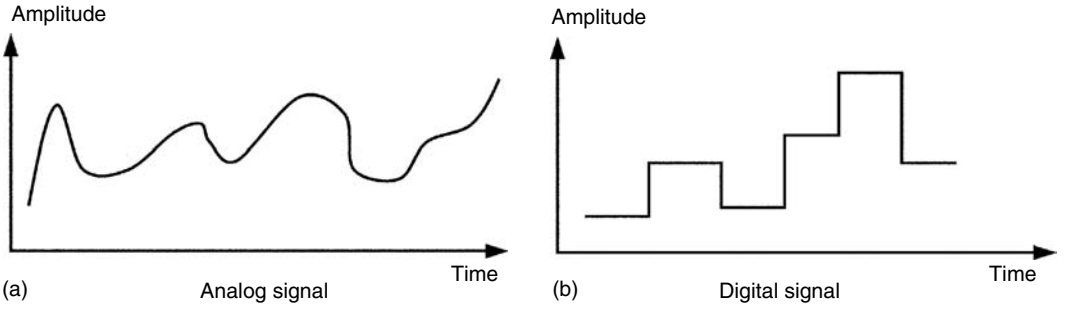



**FIGURE 26.22** Typical analog and digital signals.

There may not be a choice regarding the form (analog or digital) of information being generated by a device. For instance, a voice signal as one speaks, a video signal as generated by a camera, a speed signal generated by a moving vehicle, and an altitude signal generated by the equipment in a plane will always be analog in nature; however, there is a choice regarding the form (analog or digital) of information being transmitted over a transmission medium. Transmitted information could be analog or digital in nature and information can be easily converted from one form to another.

Each of these possibilities has its pros and cons. When a signal carrying information is transmitted, it looses its energy and strength and gathers some interference (noise) as it propagates away from the transmitter. If energy of signal is not boosted at some intermediate point, it may attenuate beyond recognition before it reaches its intended destination. That will certainly be a wasted effort. In order to boost energy and strength of a signal, it must be amplified (in case of analog signals) and rebuild (in case of digital signals). When an analog signals is amplified, the noise also becomes amplified and that certainly lowers expectations about receiving the signal at its destination in its original (or close to it) form. On the other hand, digital signals can be processed and reconstructed at any intermediate point and, therefore, the noise can essentially be filtered out. Moreover, transmission of information in digital form has many other advantages including processing of information for error detection and correction, applying encryption and decryption techniques to sensitive information, and many more. Thus, digital information transmission technology has become the dominant technology in the field communications [9,18].

As indicated earlier, communication technology has experienced phenomenal growth over the past several decades. The following two factors have always played a critical role in shaping the future of communications [20]:

- Severity of user needs to exchange information
- State of the technology related to communications

Historically, inventions have always been triggered by the severity of needs. It has been very true for the field of communications as well. In addition, there is always an urge and curiosity to make things happen faster. When electricity was discovered and people (scattered around the globe) wanted to exchange information over longer distances and in less time, telegraph was invented. Morse code was developed with shorter sequences (of dots and dashes) for more frequent alphabets. That resulted in transmission of message in a shorter duration of time. Presence of electricity, and capability of wires to carry information over longer distances, led to the development of devices that converted human voice into electrical signal, and thus led to the development of telephone systems. Behind this invention was also a need/desire to establish full-duplex (two-way simultaneous) communication in human voice. As the use of telephone became widespread, there was a need for a telephone user to be connected to any other user, and that led to the development of switching offices. In the early days, the switching offices were operated manually. As the state of the technology improved, the manual switching offices were replaced by automatic switching offices. Each telephone user was assigned a telephone number for identification purposes and a user able to dial the number for the purpose of establishing a connection with the called party. As the computer technology improved and the computers became easier to afford and smaller in size, they found countless uses including their use in communications. The computers not only replaced the automatic (electromechanical) switching offices, they were also employed in many other aspects of communication systems. Examples include conversion of information from analog to digital and vice versa, processing of information for error detection and/or correction, compression of information, and encryption/decryption of information, etc.

As computers became more powerful, there were many other applications that surfaced. The most visible application was the amount of information that users started sharing among themselves. The volume of information being exchanged among users has been growing exponentially over the last three decades. As users needed to exchange such a mammoth amount of information, new techniques were invented to facilitate the process. There was not only a need for users to exchange information with

others in an asynchronous fashion, there was also need for computers to exchange information among themselves. The information being exchanged in this fashion has different characteristics than the information being exchanged through the telephone systems. This need led to the interconnection of computers with each other and that is what is called computer networks.

### 26.4.3 Computer Networks

Computer networks is an interconnection of computers. The interconnection forms a facility that provides reliable and efficient means of communication among users and other devices. User communication in computer networks is assisted by computers, and the facility also provides communication among computers. Computer networks are also referred to as computer communication networks. Interconnection among computers may be via wired or wireless transmission medium [5,6,10,13,18].

There are two broad categories of computer networks:

- Wide area networks
- Local/metropolitan area networks

Wide area computer networks, as the name suggests, span a wider geographical area and essentially have a global scope. On the other hand, local/metropolitan area networks span a limited distance. Local area networks are generally confined to an industrial building or an academic institution. Metropolitan area networks also have limited geographical scope but it is relatively larger than that of the local area networks [19]. Typical wide and local/metropolitan area networks are shown in Fig. 26.23.

Once a user is connected to a computer network, it can communicate with any other user that is also connected to the network at some point. It is not required that a user must be connected directly to another user for communicating. In fact, in wide area networks, two communicating users will rarely be directly connected with each other. This implies that the users will be sharing the transmission links for exchanging their information. This is one of the most important aspects of computer networks. Sharing of resources improves utilization of the resources and is, of course, cost-effective as well. In addition to sharing the transmission links, the users will also share the processing power of the computers at the switching nodes, buffering capacity to store the information at the switching nodes, and any other resources that are connected to the computer network. A user that is connected to a computer network at any switching node will have immediate access to all the resources (databases, research articles, surveys, and much more) that are connected to the network as well. Of course, access to specific information may be restricted and a user may require appropriate authorization to access the information.

The information from one user to another may need to pass through several switching nodes and transmission links before reaching its destination. This implies that a user may have many options available to select one out of many sequences of transmission links and switching nodes to exchange its information. That adds to the reliability of information exchange process. If one path is not available, not feasible or is not functional, some other path may be used. In addition, for better and effective sharing of resources among several users, it is not appropriate to let any user exchange a large quantity of information at a time; however, it is not uncommon that some users may have a large quantity of information to exchange. In that case, the information is broken into smaller units known as packets of information. Each packet is sent towards destination as a separate entity and all packets are assembled together at the destination side to re-create the original piece of information [2].

Due to resource sharing environment, users may not be able to exchange their information at any time they wish to because the resources (switching nodes, transmission links) may be busy serving other users. In that case, some users may have to wait for some time before they begin their communication. Designers of computer networks should design the network so that the total delay (including wait time) is as small as possible and that the total amount of information successfully exchanged (throughput) is as large as possible.

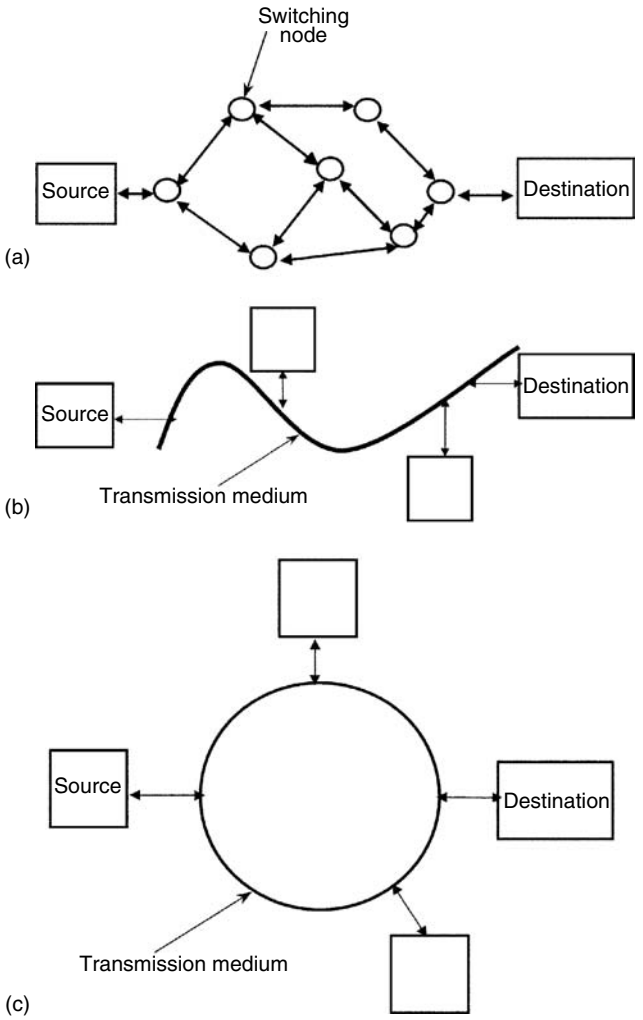



**FIGURE 26.23** (a) A typical wide area computer communication network. (b) A typical local/metropolitan area communication bus network. (c) A typical local/metropolitan area communication ring network.

As can be noted, many aspects must be addressed for enabling networks to transport users' information from one point to another. The major aspects are listed below:

- Addressing mechanism to identify users
- Addressing mechanism for information packets to identify their source and destination
- Establishing a connection between sender and receiver and maintaining it
- Choosing a path or a route (sequence of switching nodes and transmission links) to carry the information from a sender to a receiver
- Implementing a selected route or path
- Checking information packets for errors and recovering from errors
- Encryption and decryption of information
- Controlling the flow of information so that shared resources are not over taxed
- Informing the sender that the information has been successfully delivered to the intended destination (acknowledgment)

- Billing for the use of resources
- Making sure that different computers that are running different applications and operating systems, can exchange information
- Preparing information appropriately for transmission over a given transmission medium

This is not an exhaustive list of items that need to be addressed in computer networks. In any case, all such issues are addressed by very systematic and detailed procedures. The procedures are called communication protocols. The protocols are implemented at the switching nodes by a combination of hardware and software. It is not advisable to implement all these features in one module of hardware or software because that will become very difficult to manage. It is a standard practice that these features be divided in different smaller modules and then modules be interfaced together to collectively provide implementation of these features. International Standards Organization (ISO) has suggested dividing these features into seven distinct modules called layers. The proposed model is referred to as Open System Interconnection (OSI) reference model. The seven layers proposed in the OSI reference model are [2]:

- Application layer
- Presentation layer
- Session layer
- Transport layer
- Network layer
- Data link layer
- Physical layer

Physical layer deals with the transmission of information on the transmission medium. Data link layer handles the information on a single link. Network layer deals with the path or route of information from the switching node where source is connected to the switching node where receiver is connected. It also monitors end-to-end information flow. The remaining four layers reside with the user equipment. Transport layer deals with the information exchange from source to the sender. Session layer handles establishment of session between source and the receiver and maintains it. Presentation layer deals with the form in which information is presented to the lower layer. Encryption/decryption of information can also be performed at this layer. Application layer deals with the application that generates the information at the source side and what happens to it when it is delivered at the receiver side.

As the information begins from the application layer at the sender side, it is processed at every layer according to the specific protocols implemented at that layer. Each layer processes the information and appends a header and/or a trailer with the information before passing it on to the next layer. The headers and trailers appended by various layers contribute to the overhead and are necessary for transportation of the information. Finally, at the physical layer, the bits of information packets are converted to an appropriate signal and transmitted over the transmission medium. At the destination side, the physical layer receives the information packets from the transmission medium and prepares them for passing these to the next higher layer. As a packet is processed by the protocol layers at the destination side, its headers and trailers are stripped off before it is passed to the next layer. By the time information reaches the application layer, it should be in the same form as it was transmitted by the source.

Once a user is ready to send information to another user, he or she has two options. He or she can establish a communication with the destination prior to exchanging information or he can just give the information to the network node and let the network deliver the information to its destination. If communication is established prior to exchanging the information, the process is referred to as connection-oriented service and is implemented by using virtual circuit connections. On the other hand, if no communication is established prior to sending the information, the process is called connectionless service. This is implemented by using datagram environment. In connection-oriented

(virtual circuit) service, all packets between two users travel over the same path through a computer network and hence arrive at their destination in the same order as they were sent by the source. In connectionless service, however, each packet finds its own path through the network while traveling towards its destination. Each packet will therefore experience different delay and the packets may arrive at their destination out of sequence. In that case, destination will be required to put all the packets in proper sequence before assembling them [2,10,13].

As in all resource sharing systems, allocation of resources in computer networks requires a careful attention. The main idea is that the resources should be shared among users of a computer network as fairly as possible. At the same, it is desired to maintain the network performance as close to its optimal level as possible. The fairness definition, however, varies from one individual to another and depends upon how one is associated with a computer networks. Although fairness of resource sharing is being evaluated, two performance parameters—delay and throughput—for computer networks are considered. The delay is the duration of time from the moment information is submitted by a user for transmission to the moment it is successfully delivered to its destination. The throughput is amount of information successfully delivered to its intended destination per unit time. Due to the resource sharing environment in computer networks, these two performance parameters are contradictory. It is desired to have the delay as small as possible and the throughput as large as possible. For increasing throughput, a computer network must handle increased information traffic, but the increased level of information traffic also causes higher buffer occupancy at the switching nodes and hence, more waiting time for information packets. This results in an increase in delay. On the other hand, if information traffic is reduced to reduce the delay, that will adversely affect the throughput. A reasonable compromise between throughput and delay is necessary for a satisfactory operation of a computer network [10,11].

#### 26.4.3.1 Wide Area Computer Networks

A wide area network consists of switching nodes and transmission links as shown in Fig. 26.23a. Layout of switching nodes and transmission links is based on the traffic patterns and expected volume of traffic flow from one site to another site. Switching nodes provide the users access to a computer network and implement communication protocols. When a user is ready to transmit its information, the switching node, to which the user is connected to, will establish a connection if a connection-oriented service has been opted. Otherwise, the information will be transmitted in a connectionless environment. In either case, switching nodes play a key role in determining path of the information flow according to some well-established routing criteria. The criteria include performance (delay and throughput) objectives among other factors based on user needs. For keeping the network traffic within a reasonable range, some traffic flow control mechanisms are necessary. In late 1960s and early 1970s, when data rates of transmission media used in computer networks were low (a few thousands of bits per second), these mechanisms were fairly simple. A common method used for controlling traffic over a transmission link or a path was an understanding that sender will continue sending information until the receiver sends a request to stop. The information flow will resume as soon as the receiver sends another request to resume transmission. Basically the receiver side had the final say in controlling the flow of information over a link or a path. As the data rates of transmission media started increasing, this method was not deemed efficient. To control the flow of information in relatively faster transmission media, a sliding window scheme was used. According to this scheme, sender will continuously send information packet but no more than a certain limit. Once the limit has reached, the sender will stop sending the information packets and will wait for the acknowledgement of the packets that have been transmitted. As soon as an acknowledgement is received, the sender may send another packet. This method ensures that there are no more than a certain specific number of packets in transit from sender to receiver at any given time. Again the receiver has the control over the amount of information that sender can transmit. These techniques for controlling the information traffic are referred to as reactive or feedback based techniques because the decision to transmit or not to transmit is based on the current traffic conditions.

The reactive techniques are acceptable in low to moderate data rates of transmission media. As the data rates increase from kilobits per second to megabits and gigabits per second, the situation changes. Over the past several years, the data rates have increased manifold. Optical fibers provide enormously high data rates. Size of the computer networks has also experienced tremendous increase. The amount of traffic flowing through these networks has been increasing exponentially. Given that, the traffic control techniques used in earlier networks are not quite effective anymore [11,12,22]. One more factor that has added to the complexity of the situation is that users are now exchanging different types of information through the same network. Consider the example of Internet. The geographical scope of Internet is essentially global. Extensive use of optical fiber as transmission media provides very high data rates for exchanging information. In addition, users are using Internet for exchanging any type of information that they come across, including voice, video, data, etc. All these factors have essentially necessitated use of modified approach for traffic management in computer networks. The main factor leading to this change is that the information packets are moving so fast through the computer networks that any feedback-based (or reactive) control will be too slow to be of any use. Therefore, some preventive mechanisms have been developed to maintain the information traffic inside a computer network to a comfortable level. Such techniques are implemented at the sender side by ensuring that only as much information traffic is allowed to enter the network as can be comfortably handled by the networks [1,20,22]. Based on the users' needs and state of the technology, providing faster communications for different types of services (voice, video, data, and others) in the same computer network in an integrated and unified manner, has become a necessity. These computer networks are referred to as broadband integrated services digital networks (BISDNs). Broadband ISDNs provide end-to-end digital connectivity and users can access any type of communication service from a single point of access. Asynchronous transfer mode (ATM) is expected to be used as a transfer mechanism in broadband ISDNs. ATM is essentially a fast packet switching technique where information is transmitted in the form of small fixed-size packets called cells. Each cell is 53 bytes long and includes a header of 5 bytes. The information is primarily transported using connection-oriented (virtual circuit) environment [3,4,8,12,17].

Another aspect of wide area networks is the processing speed of switching nodes. As the data rates of transmission media increases, it is essential to have faster processing capability at the switching nodes. Otherwise, switching nodes become bottlenecks and faster transmission media cannot be fully utilized. When transmission media consists of optical fibers, the incoming information at a switching node is converted from optical form to electronic form so that it may be processed and appropriately switched to an outgoing link. Before it is transmitted, the information is again converted from electronic form to optical form. This slows down the information transfer process and increases the delay. To remedy this situation, research is being conducted to develop large optical switches to be used as switching nodes. Optical switches will not require conversion of information from optical to electronic and vice versa at the switching nodes; however, these switches must also possess the capability of optical processing of information. When reasonable sized optical switches become available, use of optical fiber as transmission media together with optical switches will lead to all-optical computer and communication networks. Information packets will not need to be stored for processing at the switching nodes and that will certainly improve the delay performance. In addition, wavelength division multiplexing techniques are rendering use of optical transmission media to its fullest capacity [14].

#### 26.4.3.2 Local and Metropolitan Area Networks

A local area network has a limited geographical scope (no more than a few kilometers) and is generally limited to a building or an organization. It uses a single transmission medium and all users are connected to the same medium at various points. The transmission medium may be open-ended (bus) as shown in Fig. 26.23b or it may be in the form of a loop (ring) as shown in Fig. 26.23c. Metropolitan area networks also have a single transmission medium that is shared by all the users connected to the network, but the medium spans a relatively larger geographical area, upto 150 km.

They also use a transmission medium with relatively higher data rates. Local and metropolitan area networks also use a layered implementation of communication protocols as needed in wide area networks; however, these protocols are relatively simpler because of simple topology, no switching nodes, and limited distance between the senders and the receivers. All users share the same transmission medium to exchange their information. Obviously, if two or more users transmit their information at the same time, the information from different users will interfere with each other and will cause a collision. In such cases, the information of all users involved in a collision will be destroyed and will need to be retransmitted. Therefore, there must be some well-defined procedures so that all users may share the same transmission medium in a civilized manner and have successful exchange of information. These procedures are called medium access control (MAC) protocols.

There are two broad categories of MAC protocols:

- Controlled access protocols
- Contention-based access protocols

In controlled access MAC protocols, users take turns in transmitting their information and only one user is allowed to transmit information at a time. When one user has finished its transmission, the next user begins transmission. The control could be centralized or distributed. No information collisions occur and, hence, no information is lost due to two or more users transmitting their information at the same time. Example of controlled access MAC protocols include token-passing bus and token-passing ring local area networks. In both of these examples, a token (a small control packet) circulates among the stations. A station that has the token is allowed to transmit information, and other stations wait until they receive the token [19].

In contention-based MAC protocols, users do not take turns in transmitting their information. When a users becomes ready, it makes its own decision to transmit and also faces a risk of becoming involved in a collision with another stations who also decides to transmit at about the same time. If no collision occurs, the information may be successfully delivered to its destination. On the other hand, if a collision occurs, the information from all users involved in a collision will need to be retransmitted. An example of contention-based MAC protocols is carrier sense multiple access with collision detection (CSMA/CD) which is used in Ethernet. In CSMA/CD, a user senses the shared transmission medium prior to transmitting its information. If the medium is sensed as busy (someone is already transmitting the information), the user will refrain from transmitting its information; however, if the medium is sensed as free, the user transmits its information. Intuitively, this MAC protocol should be able to avoid collisions, but collisions still do take place. The reason is that transmissions travel along the transmission medium at a finite speed. If one user senses the medium at one point and finds it free, it does not mean that another user located at another point of the medium has not already begun its transmission. This is referred to as the effect of the finite propagation delay of electromagnetic signal along the transmission medium. This is the single most important parameter that causes deterioration of performance in contention-based local area networks [11,19].

Design of local area networks has also been significantly impacted by the availability of transmission media with higher data rates. As the data rate of a transmission medium increases, the effects of propagation delay becomes even more visible. In higher speed local area networks such as Gigabit Ethernet, and 100-BASE-FX, the medium access protocols are designed such that to reduce the effects of propagation delay. If special attention is not given to the effects of propagation delay, the performance of high-speed local area networks becomes very poor [15,19].

Metropolitan area networks essentially deal with the same issues as local area networks. These networks are generally used as backbones for interconnecting different local area networks together. These are high-speed networks and span a relatively larger geographical area. MAC protocols for sharing the same transmission media are based on controlled access. Two most common examples of metropolitan area networks are fiber distributed data interface (FDDI) and distributed queue dual bus (DQDB). In FDDI, the transmission medium is in the form of two rings, whereas DQDB uses two

buses. FDDI rings carry information in one but opposite directions and this arrangement improves reliability of communication. In DQDB, two buses also carry information in one but opposite directions. The MAC protocol for FDDI is based on token passing and supports voice and data communication among its users. DQDB uses a reservation-based access mechanism and also supports voice and data communication among its users [19].

#### 26.4.3.3 Wireless and Mobile Communication Networks

Communication without being physically tied-up to wires has always been of interest and mobile and wireless communication networks promises that. The last few years have witnessed unprecedented growth in wireless communication networks. Significant advancements have been made in the technologies that support wireless communication environment and there is much more to come in the future. The devices used for wireless communication require certain features that wired communication devices may not necessarily need. These features include low power consumption, light weight, and worldwide communication ability.

In wireless and mobile communication networks, the access to a communication network is wireless so that the end users remain free to move. The rest of the communication path could be wired, wireless, or combination of those. In general, a mobile user, while communicating, has a wireless connection with a fixed communication facility and rest of the communication path remains wired. The range of wireless communication is always limited and therefore range of user mobility is also limited. To overcome this limitation, cellular communication environment has been devised. In a cellular communication environment, geographical region is divided into smaller regions called cells, thus the name cellular. Each cell has a fixed communication device that serves all mobile devices within that cell. However, as a mobile device, while in active communication, moves out of one cell and into another cell, service of that connection is transferred from one cell to another. This is called handoff process [7,16].

The cellular arrangement has many attractive features. As the cell size is small, the mobile devices do not need very high transmitting power to communicate. This leads to smaller devices that consume less power. In addition, it is well known that the frequency spectrum that can be used for wireless communication is limited and can therefore only support a small number of wireless communication connections at a time. Dividing communication region into cells allows use of the same frequency in different cells as long as they are sufficiently apart to avoid interference. This increases the number of mobile devices that can be supported. Advances in digital signal processing algorithms and faster electronics have led to very powerful, smaller, elegant, and versatile mobile communication devices. These devices have tremendous mobile communication abilities including wireless Internet access, wireless e-mail and news items, and wireless video (through limited) communication on handheld devices. Wireless telephones are already available and operate in different communication environments across the continents. The day is not far when a single communication number will be assigned to every newborn and will stay with that person irrespective of his/her location.

Another field that is emerging rapidly is the field if ad hoc wireless communication networks. These networks are of a temporary nature and are established for a certain need and for a certain duration. There is no elaborate setup needed to establish these networks. As a few mobile communication devices come in one another's proximity, they can establish a communication network among themselves. Typical situations where ad hoc wireless networks can be used are classroom environment, corporate meetings, conferences, disaster recovery situations, etc. Once the need for networking is satisfied, the ad hoc networking setup disappears.

### 26.4.4 Resource Allocation Techniques

As discussed earlier, computer networks are resource sharing systems. Users share the common resources as transmission media, processing power and buffering capacity at the switching nodes, and other resources that are part of the networks. A key to successful operation of computer networks is a fair and

efficient allocation of resources among its users. Historically, there have been two approaches to allocation of resources to users in computer networks:

- Static allocation of resources
- Dynamic allocation of resources

Static allocation of resources means that a desired quantity of resources is allocated to each user and they may use it whenever they need. If they do not use their allocated resources, no one else can. On the other hand, dynamic allocation of resources means that a desired quantity of resources is allocated to users on the basis of their demands and for the duration of their need. Once the need is satisfied, the allocation is retrieved. In that case, someone else can use these resources if needed. Static allocation results in wastage of resources, but does not incur the overhead associated with dynamic allocation. Which technique should be used in a given a situation is subject to the famous concept of supply and demand. If resources are abundant and demand is not too high, it may be better to have static allocation of resources; however, when the resources are scarce and demand is high, dynamic allocation is almost a necessity to avoid the wastage of resources.

Historically, communication and computer networks have dealt with both the situations. Earlier communication environments used dynamic allocation of resources when users will walk to public call office to make a telephone call or send a telegraphic message. After a few years, static allocation of resources was adopted, when users were allocated their own dedicated communication channels and these were not shared among others. In late 1960s, the era of computer networks dawned with dynamic allocation of resources and all communication and computer networks have continued with this tradition to date. With the advent of optical fiber, it was felt that the transmission resources are abundant and can satisfy any demand at any time. Many researchers and manufacturers held the opinion in favor of going back to the static allocation of resources, but a decision to continue with dynamic resource allocation approach was made and that is here to stay for many years to come [10].

### 26.4.5 Challenges and Issues

Many challenges and issues are related to communications and computer networks that are still to be overcome. Only the most important ones will be described in this subsection.

High data rates provided by optical fibers and high-speed processing available at the switching nodes has resulted in lower delay for transferring information from one point to another. However, the propagation delay (the time for a signal to propagate from one end to another) has essentially remained unchanged. This delay depends only on the distance and not on the data rate or the type of the transmission medium. This issue is referred to as latency versus delay issue [11]. In this situation traditional feedback-based reactive traffic management techniques become ineffective. New preventive techniques for effective traffic management and control are essential for achieving the full potential of these communication and computer networks [22].

Integration of different services in the same networks has also posed new challenges. Each type of sexrvice has its own requirements for achieving a desired level of quality of service (QoS). Within the networks any attempt to satisfy QoS for a particular service will jeopardize the QoS requirements for other service. Therefore, any attempt to achieve a desired level of quality of service must be uniformly applied to the traffic inside a communication and computer network and should not be intended for any specific service or user. That is another challenge that needs to be carefully addressed and its solutions achieved [13].

Maintaining security and integrity of information is another continuing challenge. The threat of sensitive information passively or actively falling into unauthorized hands is very real. In addition, proactive and unauthorized attempts to gain access to secure databases are also very real. These issues need to be resolved to gain the confidence of consumers so that they may use the innovations in communications and computer networking technologies to their fullest [13].

### 26.4.6 Summary and Conclusions

Section 26.4 discussed the fundamentals of communications and computer networks and the latest developments related to these fields. Communications and computer networks have witnessed tremendous growth and sophisticated improvements over the last several decades.

Computer networks are essentially resource sharing systems in which users share the transmission media and the switching nodes. These are used for exchanging information among users that are not necessarily connected directly. Transmission rates of transmission media have increased manifold and the processing power of the switching nodes (which are essentially computers) has also been multiplied. The emerging computer networks are supporting communication of different types of services in an integrated fashion. All types of information, irrespective of its type and source, is being transported in the form of packets (e.g., ATM cells). Resources are being allocated to users on a dynamic basis for better utilization. Wireless communication networks are emerging to provide worldwide connectivity and exchange of information at any time.

These developments have also posed some challenges. Effective traffic management techniques, meeting QoS requirements, and information security are the major challenges that need to be surmounted in order to win the confidence of users.

## References

1. Bae, J., and Suda, T., Survey of traffic control schemes and protocols in ATM networks, *Proceedings of the IEEE,* Vol. 79, No.2, February 1991, pp. 170–189.
2. Beyda, W., *Data Communications from Basics to Broadband*, Third Edition, 2000.
3. Black, U., *ATM: Foundation for Broadband Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
4. Black, U., *Emerging Communications Technologies*, Second Edition, Prentice-Hall, Englewood Cliffs, NJ, 1997.
5. Chou, C., "Computer networks in communication survey research," *IEEE Transactions on Professional Communication,* Vol. 40, No. 3, September 1997, pp. 197–208.
6. Comer, D., *Computer Networks and Internets*, Prentice-Hall, Englewood Cliffs, NJ, 1999.
7. Goodman, D., *Wireless Personal Communication Systems*, Addison-Wesley, Reading, MA, 1999.
8. Goralski, W., *Introduction to ATM Networking*, McGraw-Hill, New York, 1995.
9. Freeman, R., *Fundamentals of Telecommunications*, John Wiley & Sons, New York, 1999.
10. Ilyas, M., and Mouftah, H.T., "Performance evaluation of computer communication networks," *IEEE Communications Magazine*, Vol. 23, No. 4, April 1985, pp. 18–29.
11. Kleinrock, L., "The latency/bandwidth tradeoff in gigabit networks," *IEEE Communications Magazine,* Vol. 30, No. 4, April 1992, pp. 36–40.
12. Kleinrock, L., "ISDN-The path to broadband networks," *Proceedings of the IEEE,* Vol. 79, No. 2, February 1991, pp. 112–117.
13. Leon-Garcia, A., and Widjaja, I., *Communication Networks, Fundamental Concepts and Key Architectures*, McGraw Hill, New York, 2000.
14. Mukherjee, B., *Optical Communication Networks*, McGraw-Hill, New York, 1997.
15. Partridge, C., *Gigabit Networking*, Addison-Wesley, Reading, MA, 1994.
16. Rappaport, T., *Wireless Communications*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
17. Schwartz, M., *Broadband Integrated Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
18. Shay, W., *Understanding Communications and Networks*, Second Edition, PWS, 1999.
19. Stallings, W., *Local and Metropolitan Area Networks*, Sixth Edition, Prentice-Hall, Englewood Cliffs, NJ, 2000.
20. Stallings, W., *ISDN and Broadband ISDN with Frame Relay and ATM*, Fourth Edition, Prentice-Hall, Englewood Cliffs, NJ, 1999.

21. Stallings, W., *High-Speed Networks, TCP/IP and ATM Design Principles*, Prentice-Hall, Englewood Cliffs, NJ, 1998.
22. Yuan, X., "A study of ATM multiplexing and threshold-based connection admission control in connection-oriented packet networks," Doctoral Dissertation, Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Florida 33431, August 2000.

## 26.5 Video over Mobile Networks

*Abdul H. Sadka*

### 26.5.1 Introduction

Due to the growing need for the use of digital video information in multimedia communications especially in mobile environments, research efforts have been focusing on developing standard algorithms for the compression and transport of video signals over these networking platforms. Digital video signals, by nature, require a huge amount of bandwidth for storage and transmission. A 6-second monochrome video clip of QCIF ($176 \times 144$) resolution and a frame rate of 30 Hz requires over 742 kbytes of raw video data for its digital representation where each pixel has an 8-bit luminance (intensity) value. When this digital signal is intended for storage or remote transmission, the occupied bandwidth becomes too large to be accommodated and thus compression becomes necessary for the efficient processing of the video content. Therefore, in order to transmit video data over communication channels of limited bandwidth, some kind of compression must be applied before transmission.

Video compression technology has witnessed a noticeable evolution over the last decade as research efforts have revolved around the development of efficient techniques for the compression of still images and discrete raw video sequences. This evolution has then progressed into improved coding algorithms that are capable of handling both errors and varying bandwidth availability of contemporary communication media. The contemporary standard video coding algorithms provide both optimal coding efficiency and error resilience potential. Current research activity is focused on the technologies associated with the provision of video services over the future mobile networks at user-acceptable quality and with minimal cost requirements. Section 26.5 discusses the basic techniques employed by video coding technology, and the associated most prominent error resilience mechanisms used to ensure an optimal trade-off between the coding efficiency and quality of service of standard video coding algorithms. This section also sheds the light on the algorithmic concepts underlying these technologies and provides a thorough presentation of the capabilities of contemporary mobile access networks, such as general packet radio service (GPRS), to accommodate the transmission of compressed video streams at various network conditions and application scenarios.

### 26.5.2 Evolution of Standard Image/Video Compression Algorithms

The expanding interest in mobile multimedia communications and the concurrently expanding growth of data traffic requirements have led to a tremendous amount of research work during a period of over 15 years for developing efficient image and video compression algorithms. Both International Telecommunications Union (ITU) and International Organization for Standardization (ISO) have released a number of standards for still image and video coding algorithms that employ the discrete cosine transforms (DCT) and the Macroblock (MB) structure of an image to suppress the temporal and spatial redundancies incorporated in a sequence of images. These standardized algorithms aimed at establishing an optimal trade-off between the coding efficiency and the perceptual quality of the reconstructed signal. After the release of the first still-image coding standard, namely JPEG [1], CCITT recommended the standardisation of the first video compression algorithm for low-bit rate communications at $p \times 64$ kbit/s over ISDN, namely ITU-T H.261 [2] in 1990. In post 1990s, intensive work has been carried out to develop improved versions of the aforementioned ITU standard, and this has

culminated in a number of video coding standards, namely MPEG-1 [3] for audiovisual data storage (1.5-2 Mbit/s) on CD-ROM, MPEG-2 [4] (or ITU-T H.262) for HDTV applications (4–9 Mbit/s), ITU-T H.263 [5] for very low bit rate (<64 kbit/s) communications over PSTN networks, and then the first content-based, object-oriented audiovisual compression algorithm, namely MPEG-4 [6], for multimedia communications over mobile networks in 1998. Recent standardization work resulted in recommending annexes to ITU-T H.263 standard, namely H.263+ [7] and H.263++ [8] for improved coding efficiency, bit rate scaleability, and error resilience performance. ITU-T is currently considering the standardization of H.26L, a new video compression algorithm expected to outperform H.263 at very low bit rate applications. Despite this remarkable evolution of digital video coding technology, the common feature for all the released standards so far is that they all employ the same algorithmic concepts and build on them for further improvement in both quality and coding efficiency. In this chapter section, the fundamental techniques that constitute the core of today's video coders are presented.

## 26.5.3 Digital Representation of Raw Video Data

A video signal is a sequence of still images. When played at a high enough rate, the sequence of images (mostly referred to as video frames) gives the impression of an animated video scene. Video frames are captured by a camcorder at a certain sampling rate and processed as a sequence of still pictures correlated by motion dependencies. When adjacent frames are strongly correlated, smaller redundancy is found in the video signal if only the difference between successive frames is encoded. The process of exploiting temporal redundancies between adjacent frames by subtracting the prediction image (sometimes referred to as the motion compensated image) from the original input image and then coding the resulting residual is called INTER frame coding. If no motion prediction was employed in encoding a video frame and only spatial redundancies were exploited to compress a video frame, then the frame is said to be INTRA coded.

Each video frame is a two-dimensional matrix of pixels, each of which is represented by a luminance (intensity) component and two chrominance (color) components *Y, U,* and *V,* respectively. In block-based video coders, each frame is divided into groups of blocks (GOB). Each GOB is divided into a number of MBs (macroblock). A MB relates to 16 pixels by 16 lines of luminance *Y* and the spatially corresponding 8 pixels by 8 lines of chrominance *U* and *V.* A MB consists of four *Y*-blocks and two spatially corresponding color difference blocks. Figure 26.24 depicts the hierarchical layering structure of a video frame of Quadrature Common Intermediate Format (QCIF) resolution, i.e., 176 pixels by 144 lines.

## 26.5.4 Basic Concepts of Block-Based Video Coding Algorithms

Despite their differences, the video coding standards have the same core structure. They all adopt the MB structure as described in the previous section and consist of the same major building blocks. The standard video coding algorithms employ one of the two coding modes, INTRA or INTER. A typical block diagram of a block-based transform video coder is depicted in Fig. 26.25.

### 26.5.4.1 Discrete Cosine Transforms (DCT)

The 64 coefficients of an $8 \times 8$ block of data are passed through a DCT transformer. DCT extracts the spatial redundancies of the video block by gathering the biggest portion of its energy in the low frequency components that are located in the top left corner of the block. The transfer function of a two-dimensional DCT transformer employed in a block-based video coder is given in Eq. 26.1 below:

$$F(u,v) = \frac{1}{4} C(u)C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos\left[\pi(2x+1)\frac{u}{16}\right] \cos\left[\pi(2y+1)\frac{v}{16}\right] \tag{26.1}$$

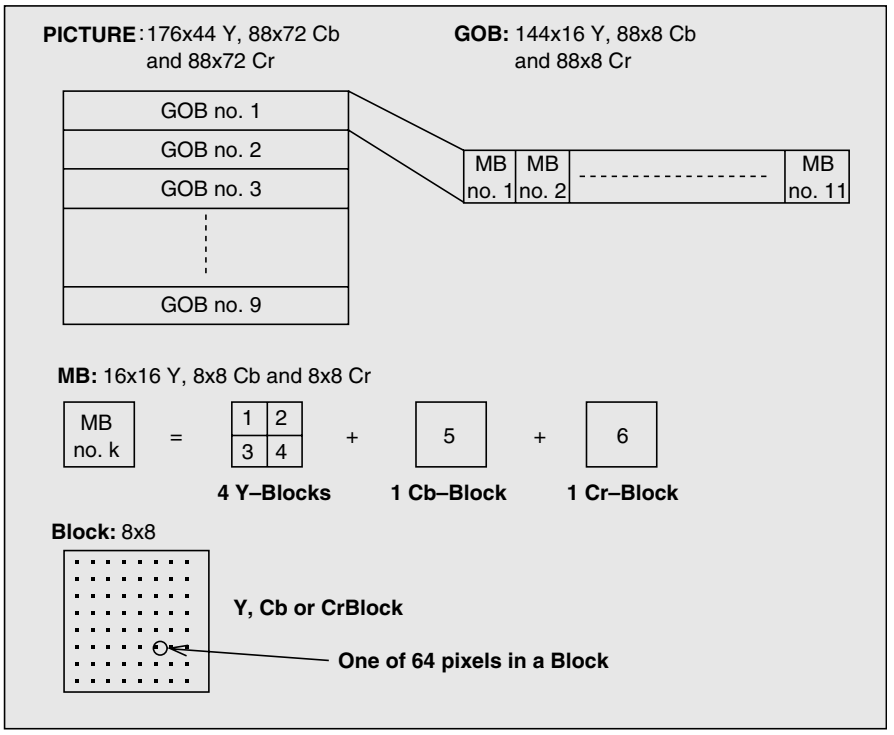



**FIGURE 26.24** Hierarchical layering structure for a QCIF frame in block-based video coders.

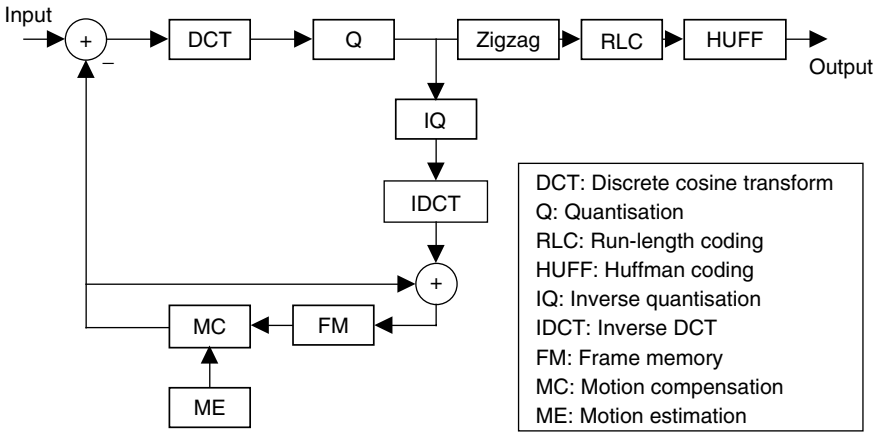



**FIGURE 26.25** Block diagram of a block-based video coder.

with $u, v, x, y = 0, 1, 2, \ldots, 7$, where $x$ and $y$ are the spatial coordinates in the pixel domain, $u$ and $v$ are the coordinates in the transform domain

$$C(u) = \frac{1}{\sqrt{2}} \quad \text{for } u = 0;\ 1 \text{ otherwise}$$

$$C(v) = \frac{1}{\sqrt{2}} \quad \text{for } v = 0;\ 1 \text{ otherwise}$$

#### 26.5.4.2 Quantization

Quantization is a process that maps the symbols representing the DCT transformed coefficients from one set of levels to a narrower one in order to minimise the number of bits required to transmit the symbols. Quantization in block-based coders is a lossy process and thus it has a negative impact on the perceptual quality of the reconstructed video sequence. The quantization parameter (Qp) is a user-defined parameter that determines the level of distortion that affects the video quality due to quantization. The higher the quantization level, Qp, the coarser the quantization process. Quantization uses different techniques based on the coding mode employed (INTRA or INTER), the position of the coefficient in a video block (DC or AC coefficients), and the coding algorithm under consideration.

#### 26.5.4.3 Raster Scan Coding

It is also known as zigzag pattern coding. The aim of zigzag coding the $8 \times 8$ matrix of quantised DCT coefficients is to convert the two-dimensional array into a stream of indices with a high occurrence of successive 0 coefficients. The long runs of zeros will then be efficiently coded as will be shown in the next subsection. The order of a zigzag pattern encoder is depicted in Fig. 26.26.

#### 26.5.4.4 Run-Length Coding

The run-length encoder takes the one-dimensional array of quantised coefficients as input and generates coded runs as output. Instead of coding each coefficient separately, the run-length coder searches for runs of similar consecutive coefficients (normally zeros after the DCT and quantisation stages) and codes the length of the run and the preceding nonzero level. A 1-bit flag (LAST) is sent after each run to indicate whether or not the corresponding run is the last one in the current block. Run-lengths and levels are then fed to the Huffman coder to be assigned variable length codewords before transmission on the video channel.

#### 26.5.4.5 Huffman Coding

Huffman coding, traditionally referred to as entropy coding, is a variable length coding algorithm that assigns codes to source-generated bit patterns based on their frequency of occurrence within the generated bit stream. The higher the likelihood of a symbol, the smaller the length of the codeword assigned to it and vice versa. Therefore, Entropy coding results in the optimum average codeword size for a given set of runs and levels.

#### 26.5.4.6 Motion Estimation and Prediction

For each MB in a currently processed video frame, a sum of absolute differences (SAD) is calculated between its pixels and those of each $16 \times 16$ matrix of pixels that lie inside a window (in the previous frame) of a user-defined size called the search window. The $16 \times 16$ matrix, which results in the least SAD, is considered to most resemble the current MB and referred to as the "best match." The displacement vector between the currently coded MB and the matrix that spatially corresponds to its best match in the previous frame is called the motion vector (MV) and the relative SAD is called the MB residual matrix. If the smallest SAD is less than a certain threshold then the MB is INTER coded by sending the MV and the DCT coefficients of the residual matrix, otherwise the MB is INTRA coded. The coordinates of the MV are transmitted differentially using the coordinates of one or

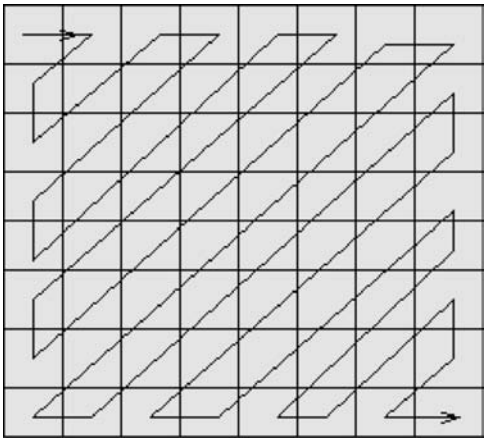


**FIGURE 26.26** Sequence of zigzag-coding coefficients of a quantised $8 \times 8$ block.

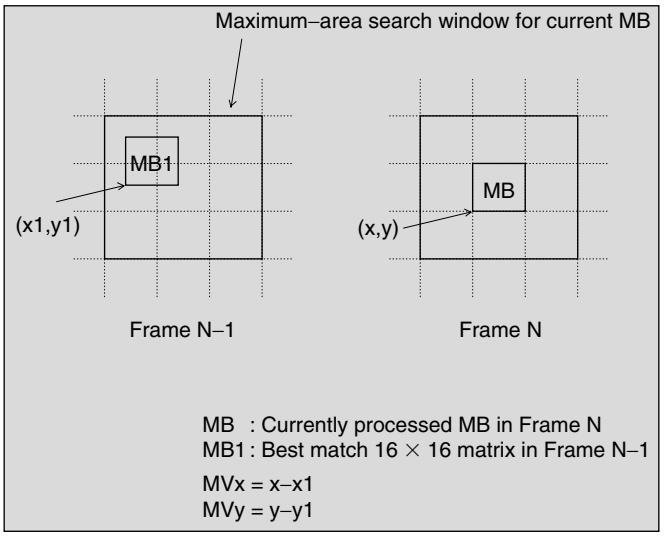



**FIGURE 26.27** Motion estimation process in a block-based video coder.

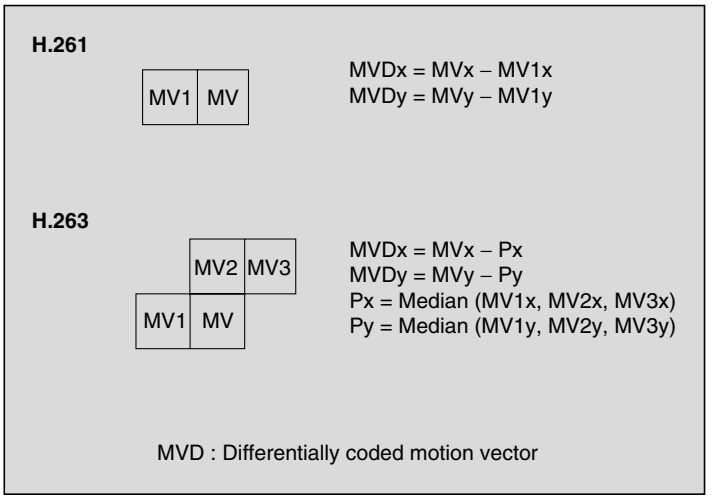



**FIGURE 26.28** Motion prediction in 2 ITU-T video coding standards.

more MVs corresponding to neighboring MBs (left MB in ITU-T H.261 or left, top, and top right MBs in ITU-T H.263 and ISO MPEG-4) within the same video frame. Figures 26.27 and 26.28 illustrate the motion estimation and prediction processes of contemporary video coding algorithms.

### 26.5.5 Subjective and Objective Evaluation of Perceptual Quality

The performance of a video coding algorithm can be simply subjectively evaluated by visually comparing the reconstructed video sequence to the original one. Two major types of subjective methods are used to assess the quality of perceptual video quality. In the first, an overall quality rating is assigned to the image (usually last decoded frame of a sequence). In the second, quality impairment is induced on a standard type image until it is completely similar to the reference image or vice versa.

Objectively, the video quality is measured by using some mathematical criteria, the most common of which is the peak-to-peak signal-to-noise ratio (PSNR) defined in Eq. 26.2.


**FIGURE 26.29** 150th Frame of original: (a) "Suzie" sequence and its compressed version at 64 kbit/s using, (b) H.261, (c) baseline H.263, and (d) Full-option H.263.

$$\text{PSNR} = 10\ \log_{10} \frac{255^2}{\frac{1}{M\times N}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}(x(i,j)-\hat{x}(i,j))^2} \tag{26.2}$$

For a fair comparison of perceptual quality between two video coding algorithms, the objective and subjective results must be evaluated at the same target bit rates. Because the bit rate in kbit/s is directly proportional to the number of frames coded per unit of time, the frame rate (f/s) has also to be mentioned in the evaluation process.

Figures 26.29 and 26.30 show the subjective and objective results, respectively, for coding 150 frames of the sequence "Suzie" at a bit rate of 64 kbit/s and a frame rate of 25 f/s.

## 26.5.6 Error Resilience for Mobile Video

Mobile channels are characterised by a high level of hostility resulting from high bit error ratios (BER) and information loss. Because of the bit rate variability and the spatial and temporal predictions, coded video streams are highly sensitive to transmission errors. This error sensitivity can be the reason for an ungraceful degradation of video quality, and hence the total failure of the video communication service. A single bit error could lead to a disastrous damage to perceptual quality. The most damaging effect of errors is that which leads to a loss of synchronisation at the decoder. In this case, the decoder is unable to determine the size of the affected variable-length video parameter and, therefore, drops the stream bits following the position of error until it resynchronises at the next synch word. Consequently, it is vital to employ an error resilience mechanism for the success of the underlying video communication service.

A popular technique used to mitigate the effects of errors is called error concealment [9]. It is a decoder-based zero-redundancy error control scheme whereby the decoder makes use of previously received error-free video data for the reconstruction of the incorrectly decoded video segment. A commonly used approach conceals the effect of errors on a damaged MB by relying on the content of the spatially corresponding MB in the previous frame. In the case where motion data is corrupted, the damaged

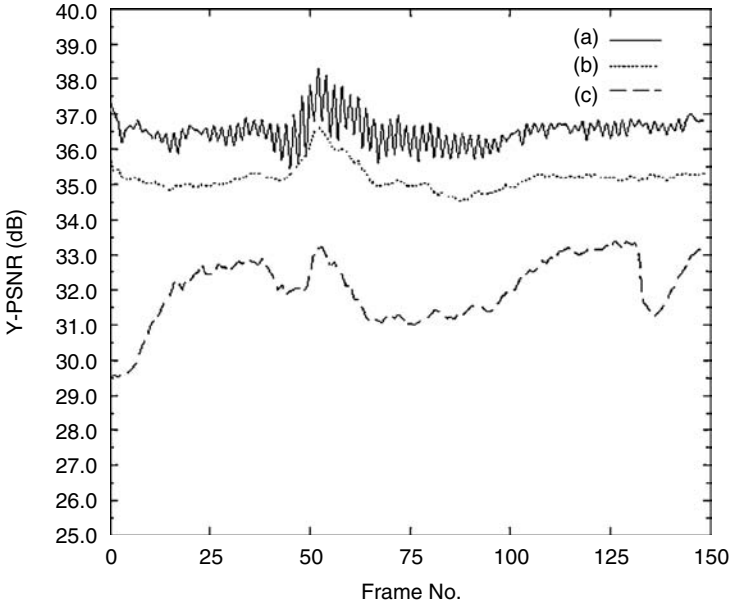



**FIGURE 26.30** PSNR values for Suzie sequence compressed at 64 kbit/s (a) baseline H.263, (b) full-option H.263, and (c) H.261.

motion vector can be predicted from the motion vectors of spatially neighboring MBs in the same picture. On the other hand, transform coefficients could also be interpolated from pixels in neighboring blocks.

However, error concealment schemes cannot provide satisfactory results for networks with high BERs and long error bursts. In this case, error concealment must be used in conjunction with error resilience schemes that make the coded streams more robust to transmission errors and video packet loss. In the literature, there are a large number of error resilience techniques specified in the standard ISO MPEG-4 [10] and the annexes to ITU-T H.263 defined in recommendations H.263+ [11] and H.263++ [12]. One of the most effective ways of preventing the propagation of errors in encoded video sequences is the regular insertion of INTRA-coded frames, which do not make use of any information from previously transmitted frames; however, this method has the disadvantage of making the traffic characteristics of a video sequence extremely bursty since a much larger number of bits are required to obtain the same quality levels as for INTER (predictively coded) frames. A more efficient improvement to INTRA-frame refresh consists of regular coding of INTRA MBs per frame, referred to as Adaptive INTRA Refresh (AIR), where the INTRA coded MBs are identified as part of the most active region in the video scene. The insertion of a fixed number of INTRA coded MBs per frame can smooth out the bit rate fluctuations caused by coding the whole frame in INTRA mode. In the following subsections, we present two major standard-compliant error resilience algorithms specified in the MPEG-4 video coding standard, namely data partitioning and two-day decoding with reversible codewords.

#### 26.5.6.1 Video Data Partitioning

The non error-resilient syntax of video coding standards suggests that video data is transmitted on a MB basis. In other words, the order of transmission is established such as all the parameters pertaining to a particular MB are sent before any parameter of the following MB is transmitted. This implies that a bit error detected in the texture data of an early MB in the video frame leads to the loss of all forthcoming MBs in the frame. Data partitioning changes the order of transmission of video data from a MB basis to a frame basis or a Visual Object Plane (VOP) basis in MPEG-4 terminology. Each video packet that corresponds to a VOP consists of two different partitions separated by specific bit patterns called markers (DC marker for INTRA coded VOPs and motion marker for INTER coded VOPs). The first

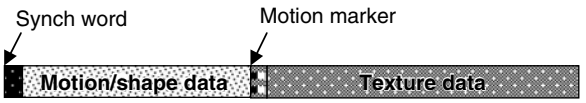



**FIGURE 26.31** Data partitioning for error resilient video communication.

partition contains the shape information, motion data, and some administrative parameters such as COD for INTRA frames and MCBPC of all the MBs in the VOP, while the second partition contains the texture data (i.e., the transform coefficients TCOEFF) of all the MBs inside the VOP and other control parameters such as CBPY. Using this partitioning structure as illustrated in Fig. 26.31, the errors that hit the data bits of the second partition do not lead to the loss of the whole frame since the error-sensitive motion data would have been correctly decoded upfront.

### 26.5.6.2 Two-Way Decoding and Reversible Variable-Length Codewords

Two-way decoding is used with reversible VLC words in order to reduce the size of the damaged area in a video bit stream. This error resilience technique enables the video decoder to reconstruct a part of the stream that would have been skipped in the ordinary one-way decoding due to loss of synchronisation. This is achieved by allowing the decoding of the variable-length codewords of the video bit stream in the reverse direction. The reversible codewords are symbols that could be decoded in both the forward and reverse directions. An example of reversible VLCs is a set of codewords where each one of them consists of the same number of the starting symbol, either 1 or 0. For instance, the set of variable-length codewords that is defined by 0100, 11001, 10101, 01010, 10011, 0010, consists of codewords that contain three 1s or 0s each, where the 1 or 0 is the starting symbol, respectively.

In conventional one-way decoding, the decoder loses synchronisation upon detection of a bit error. This is mainly due to the variable rate nature of compressed video streams and the variable-length Huffman codes assigned to various symbols that represent the video parameters. In order to restore its synchronisation, the decoder skips all the data bits following the position of errors until it falls on the first error-free synch word in the stream. The skipped bits are then discarded, regardless of their correctness, resulting in an effective error ratio that is larger than the channel BER by orders of magnitude. The response of the one-way video decoder to a bit error is depicted in Fig. 26.32.

With two-way decoding, a part of the skipped segment of bits can be recovered by enabling decoding in the reverse direction as shown in Fig. 26.33. Upon detection of a bit error, the decoder stops its operation searching for the next synch word in the bit stream. Upon gaining synchronization at the synch word, the decoder resumes its operation in the backward direction thereby rescuing the part of the bit stream, which has been discarded in the forward direction. If no error is detected in the reverse direction then the damaged area is confined to the MB where the bit error has been detected in the forward direction. If an error has also been flagged up in the backward direction, then the segment of bits between the positions of error in both the forward and backward directions is discarded as the error damaged area as shown in Fig. 26.33.

In many cases, a combination of error resilience techniques is used to further enhance the error robustness of compressed video streams to transmission errors of mobile environments. For instance,

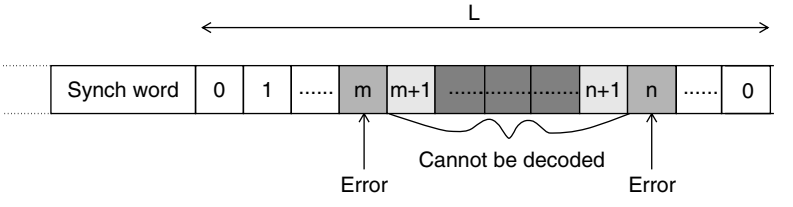



**FIGURE 26.32** One-way decoding of variable-length codes.

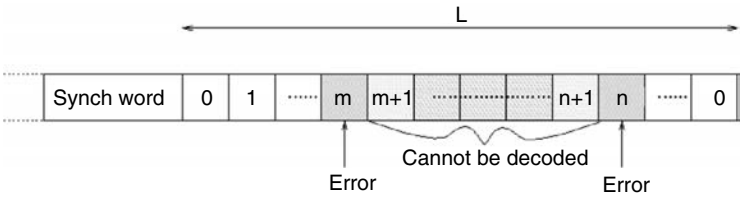



**FIGURE 26.33** Two-way decoding of variable-length codes.

both data partitioning and two-way decoding can be jointly employed to protect the error-sensitive motion data of the first video partition. The motion vectors and the administrative parameters contained in the first partition are all coded with reversible VLC words. The detection of a bit error in the forward direction triggers the decoder to stop its operation, regain synchronisation at the motion marker separating the two partitions in the corresponding VOP, and then decode backwards to salvage some of the correctly received bits that were initially skipped in the forward direction.

### 26.5.7 New Generation Mobile Networks

Packet-switched mobile access networks such as GPRS [13] and EGPRS [14] are intended to give subscribers access to a variety of mobile multimedia services that run on different networking platforms, let it be the core mobile network, i.e., UMTS, ATM, or even Internet. The packet-switched mobile access networks have a basic common feature in that they are all IP-based and allow time multi-slotting on a given radio interface. The multi-slotting capabilities enable the underlying networking platform to accommodate higher bit rates by providing the end-user with a larger physical layer capacity.

The real-time interactive and conversational services are very much delay-critical, so the provision of these services over mobile networks can only be achieved by using a service class capable of guaranteeing the delay constraints with one-delays in the order of 200 msec being required. In order to achieve such delay requirements, it is necessary to avoid using any retransmissions or repeat-requests scenarios by operating the RLC layer of the GPRS protocol stack in the unacknowledged mode of operations. Similarly, the transport layer protocol that must be employed is the user datagram protocol (UDP), which operates over IP and does not make use of any repeat-request system.

IP networks do not guarantee the delivery of packets and neither do they provide any mechanism to guarantee the orderly arrival of packets. This implies that not only does the inter-packet arrival time vary but it is also likely that packets may arrive out of order. Therefore, in order to transmit real-time video information, some transport-layer functionality must be overlaid on the network layer to provide timing information from which streaming video may be reconstructed. To offer this end-to-end network transport functionality, the IETF real-time transport protocol (RTP) [15] is used. RTP fulfills functions such as payload type identification, sequence numbering, timestamping, and delivery monitoring, and operates on top of IP and UDP for the provision of real-time services and video applications over the IP-based mobile networks.

On the other hand, the mobile access networks employ channel protection schemes that provide error control capabilities against multipath fading and channel interferers. For instance, GPRS employs four channel protection schemes (CS-1 to CS-4), offering flexibility in the degree of protection and data traffic capacity available to the user. Varying the channel coding scheme allows for an optimization of the throughput across the radio interface as the channel quality varies. The data rates provided by GPRS with the channel coding schemes enabled are 8 kbit/s for CS-1, 12.35 kbit/s for CS-2, 14.55 kbit/s for CS-3, and 20.35 kbit/s for CS-4; however, almost 15% of the bits in the payload of a radio block are used up by header information belonging to the overlying protocols. Therefore, the rates presented to the video source for each one of the channel coding schemes per time slot are 6.8 kbit/s for CS-1, 10.5 kbit/s for CS-2, 12.2 kbit/s for CS-3, and 17.2 kbit/s for CS-4. It is, however, envisaged that the CS-1 and CS-2

schemes will be used for video applications. Obviously, the available throughput to a single terminal will be multiples of the given rates per slot, depending upon the multi-slotting capabilities of the terminal. Conversely, EGPRS provides 9 channel coding schemes of different protection rates and capabilities and the choice of a suitable scheme is again a trade-off between the throughput and the error protection potential.

### 26.5.8 Provision of Video Services over Mobile Networks

Taking into perspective the traffic characteristics of a coded video source employing a fixed quantiser, we observe that the output bit rate is highly variable with high peaks taking place each time an INTRA-coded frame is transmitted. INTRA frames require roughly three times on average the bandwidth required for transmitting a predictively coded frame. Therefore, if the frequency of INTRA frames is increased for error control purposes as discussed in Section 26.5.6, the encoder will have to discard a number of frames following each INTRA coded frame until some bandwidth becomes available. Despite the fact that a fixed quantiser leads to a constant spatial quality, yet the frequent insertion of INTRA frames in the video sequence has a degrading effect on the temporal quality of the entire video sequence. In order to preventively cure this situation, it is advisable that a rate control mechanism be employed at the video encoder before the coded video bit stream is sent over the mobile channel. One method is to vary the used quantiser value in order to truncate the high-frequency DCT coefficients in accordance with the target bit rate of the video coder and the number of bits available to code a particular frame, VOP or MB. Coding an INTRA frame with a coarse quantiser results in a poor spatial quality but helps improve the temporal quality of the video sequence by maintaining the original frame rate and reducing the jittering effect caused by the disparity in size between INTRA and INTER coded frames.

The video delivery over mobile channels can take the form of real-time delay-sensitive conversational services, delay-critical (on-demand or live) streaming services, or delay-insensitive multimedia messaging applications. The latter requires guarantee on the error-free delivery of intended messages without placing any stipulation on the duration of transmission and therefore allows retransmissions of erroneous messages to take place. The former two categories of video services, however, are rather more delay-critical and necessitate the use of both application and transport layer end-to-end error control schemes for the robust transmission of compressed video in mobile environments.

The analysis of the GPRS protocol efficiency shows that a reduction of 15% in the data rate per time slot, as seen by the video encoder, is enough to compensate for all the protocol overheads. The video quality that can be achieved in video communications over the new generation mobile networks, is a function of the time slot/coding-scheme combination and the channel conditions during the time of video packet transmission. It is observed that in error-free conditions, CS-1 yields a sub-optimal quality due to the large overhead it places on the available bandwidth of each time slot; however, in error-prone conditions and for C/I ratios lower than 15 dB, CS-1 presents the best error protection capabilities and offers the best video quality as compared to other channel coding schemes. When eight time slots are used with CS-1, GPRS can offer a video payload data rate of 54.4 kbit/s. At this rate, it has been demonstrated that QCIF-resolution conversational MPEG-4 video services can be offered over GPRS for a frame rate of 10 f/s with fairly good perceptual quality, especially when frequency hopping is used; however, for highly detailed scenes involving a high amount of motion, the error-free video quality at high C/I ratios suffers both spatially and temporally because of the coarse quantiser used and the jitter resulting from the large number of discarded frames respectively. The error protection schemes of the GPRS protocol are used in conjunction with the application-layer error resilience techniques specified by the MPEG-4 video compression standard. Figure 26.34 shows the subjective video quality achieved by transmitting an MPEG-4 coded video sequence (at 18 kbit/s) over a GPRS channel with and without error resilience (AIR) when CS-1 and four time slots are used.

On the other hand, video services on EGPRS are less likely to encounter the same problems posed by the lack of bandwidth in the GPRS networks. When EGPRS employs the channel coding scheme MCS-9,

(a) (b)

**FIGURE 26.34** One frame of Suzie sequence encoded with MPEG-4 at 18 kbit/s and transmitted over a GPRS channel with C/I = 15 dB, with CS-1 and 4 time-slots used: (a) no error resilience and (b) AIR.

the terminal can be offered a data rate of 402.4 kbit/s when 8 time slots are employed. Obviously, at this data rate, there exists a much higher flexibility in selecting the operating picture resolution and the video content intended for transmission over the mobile network.

### 26.5.9 Conclusions

The provision of video services over the new generation mobile networks is made possible through the enabling technologies supported by the error protection schemes and the multi-slotting capabilities of the radio interface. Conversational video applications are delay-sensitive and thus do not support retransmissions of corrupted video data. To provide a user-acceptable video quality, the video application must employ an error resilience mechanism in conjunction with the physical layer channel coding schemes. A wide range of error resilience techniques have been developed in recent video compression algorithms and their annexed versions. The use of error resilience techniques for supporting the provision of video services over mobile networks helps enhance the perceptual quality, especially at times where the mobile channel is suffering from low C/I ratios resulting from high BERs and radio block loss ratios.

## References

1. ISO/IEC JTC1 10918 & ITU-T Rec. T.81: *Information Technology—Digital Compression and coding of continuous-tone still images: Requirements and guidelines*, 1994.
2. CCITT Recommendation H.261: *Video Codec for audiovisual services at $p \times 64$ kbit/s*, COM XV-R 37-E, 1990.
3. ISO/IEC CD 11172: *Coding of moving pictures and associated audio for digital storage media at 1.5 Mbit/s*, December 1991.
4. ISO/IEC CD 13818-2: *Generic coding of moving pictures and associated audio*, November 1993.
5. Draft ITU-T Recommendation H.263: *Video coding for low bit rate communication*, May 1996.
6. ISO/IEC JTC1/SC29/WG11N2802: *Information technology—Generic coding of audiovisual objects—Part 2: Visual*, ISO/IEC 14496-2, MPEG Vancouver meeting, July 1999.
7. Draft ITU-T Recommendation H.263 Version 2 (H.263+): *Video coding for low bit rate communications*, January 1998.
8. Rapporteur for Q.15/16—*Draft for H.263++, Annexes U, V and W to Recommendation H.263*, ITU Telecommunication Standardisation Sector, November 2000.
9. Y. Wang, and Q.F. Zhu, "Error control and concealment for video communication: a review," *Proc. of the IEEE*, Vol. 86, No. 5, pp. 974–997, May 1998.

10. R. Talluri, "Error resilient video coding in the MPEG-4 standard," *IEEE Communications Magazine*, pp. 112–119, June 1998.
11. S. Wenger, G. Knorr, J. Ott, and F. Kossentini, "Error Resilience Support in H.263+," *IEEE Transaction on Circuit and Systems for Video Technology*, Vol. 8, No. 7, Nov. 1998.
12. G. Sullivan, "*Rapporteur for Q.15/16—Draft for H.263++, Annexes U, V and W to Recommendation H.263*," ITU Telecommunication Standardisation Sector, November 2000.
13. Digital Cellular Telecommunications System (Phase 2+), "*General Packet Radio Service (GPRS); Overall description of the GPRS Radio Interface; Stage 2*," ETSI/SMG, GSM 03.64, V. 5.2.0, January 1998.
14. Tdoc SMG2 086/00, "*Outcome of Drafting Group on MS EGPRS Rx Performance*," EDGE Drafting Group, January 2000.
15. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "*RTP: A Transport Protocol for Real-Time Applications*," RFC1889, January 1996.

## 26.6 Pen-Based User Interfaces—An Applications Overview

*Giovanni Seni, Jayashree Subrahmonia, and Larry Yaeger*

### 26.6.1 Introduction

A critical feature of any computer system is its interface with the user. This has led to the development of user interface technologies such as mouse, touch screen, and pen-based input devices. They all offer significant flexibility and options for computer input; however, touch screens and mice cannot take full advantage of human fine motor control, and their use is mostly restricted to data selection, i.e., as pointing devices. On the other hand, pen-based interfaces allow, in addition to the pointing capabilities, other forms of input such as handwriting, gestures, and drawings. Because handwriting is one of the most familiar forms of communication, pen-based interfaces offer a very easy and natural input method.

A pen-based interface consists of a fine-tipped stylus and a transducer device that allows the movement of the stylus to be captured. Such information is usually given as a time ordered sequence of $x$–$y$ coordinates (digital ink) and an indication of inking, i.e., whether the pen is up or down. Digital ink can be passed on to recognition software that will convert the pen input into appropriate text or computer actions. Alternatively, the handwritten input can be organized into ink documents, notes, or messages that can be stored for later retrieval or exchanged through telecommunication means. Such ink documents are appealing because they capture information as the user composed it, including text in any mix of languages and drawings such as equations and graphs.

Pen-based interfaces are desirable in mobile computing (e.g., personal digital assistants [PDAs]) and mobile phones because they are scalable. Only small reductions in size can be made to keyboards before they become awkward to use; however, if they are not shrunk in size, they can never be very portable. This is even more problematic as mobile devices develop into multimedia terminals with numerous functions ranging from agenda and address book to wireless web browser, because of the increasing amounts of text that must be entered. Voice-based interfaces may appear to be a solution, but they entail all the problems that cell phones already have introduced in terms of disturbing bystanders and loss of privacy. Furthermore, using voice commands to control applications such as a web browser can be difficult and tedious; by contrast, clicking on a link with a pen, or entering a short piece of text by writing, is very natural and takes place in silence.

Recent hardware advances in alternative ink capture devices based on ultrasonic and optical tracking technologies have also contributed to the renewed interest in pen-based systems. These technologies avoid the need for pad electronics, thus reducing the cost, weight, and thickness of a pen-enabled system. Furthermore, they can sometimes be retrofitted to existing writing surfaces such as whiteboards [1,2] or used with paper, either specially marked [3,4] or plain [5].

This section reviews a number of applications, old and new, in which the pen can be used as a convenient and natural form of input. Our emphasis will be on the user experience, highlighting limitations of existing solutions and suggesting ways of improving them. We begin with a short review of currently available pen data acquisition technologies in Section 26.6.2. Then, in Section 26.6.3, we discuss handwriting recognition user interfaces for mobile devices and the need for making applications aware of the handwriting recognition process. In Section 26.6.4, we present Internet-related applications such as ink messaging. In Section 26.6.5 we discuss some methods for combining computer and paper inking, and then analyze some of the benefits and issues associated with working with digital ink on the computer. Finally, in Section 26.6.6, we present examples of synergistic interfaces being developed which combine the pen with other input modalities.

## 26.6.2 Pen Input Hardware

The function of the pen input hardware is to convert pen tip position over time into $X,Y$ coordinates at a sufficient temporal and spatial resolution for handwriting recognition and visual presentation [6].

A pen input system consists of a combination of pen, tablet, and in some cases, paper. Examples of these include PDAs and some electronic or graphics tablets. Some of these have a glass surface sitting directly atop the display. These integrated tablet-plus-displays allow you to point and write where you are looking, and are fairly intuitive to use. Others, like many graphics tablets, have an input device that is separate from the display. These opaque tablets are less intuitive, requiring the user to write in one place while they look in another. Users new to opaque tablets often find them difficult to use for text, image editing, and the like, but with practice, users can become quite proficient with the devices.

Paper-based systems provide another alternative for inputting digital ink, when used with special pens and, sometimes, special paper. They provide a natural-feeling writing surface, high resolution, and do not suffer from screen glare or parallax issues. However, they are necessarily always at a remove from the digital ink they are producing or from the user interface they might wish to control.

Both tablet-based and paper-based pen input systems must detect and report when the pen tip is in contact with the writing surface. Paper systems face an additional challenge because of the need to keep a consistent, familiar pressure between pen tip and paper, while still sensing this contact.

Pen hardware platforms available today use one of the following four kinds of technologies:

1. Magnetic tracking: Sequentially energized coils embedded in the pad couple a magnetic field into a pen tank circuit (coil and capacitor). Neighboring coils pick up the magnetic field from the pen, and their relative strength determines pen location [7]. The magnetic field can also be generated in the pen, requiring a battery that increases pen weight and thickness [8] but can help enable wireless tablets [9].
2. Electric tracking: The conductive properties of a hand and normal pen can be used for tracking [10]. A transmitter electrode in the pad couples a small displacement current through the paper to the hand, down through the pen, and back through the paper to an array of receiver electrodes. Pen location is calculated as the center of mass of the received signal strengths.
3. Ultrasonic tracking: Ultrasonic tracking is based on the relatively slow speed of sound in air (330 m/s). A pen generates a burst of acoustic energy and electronics in the pad measure the time of arrival to two stationary ultrasonic receivers [1,2]. The ultrasonic transmission is either synchronized to the pad, typically with an infrared signal, or a third ultrasonic receiver is used [11].
4. Optical tracking technology: Optical sensors are mounted in the tip of the pen [12,13] that can either provide relative tracking (like a mouse) or absolute position tracking (like a touch screen). Optics may also be mounted at the top of a pen [5] and used to determine the pen's position relative to some constant frame of reference, such as the edges of a piece of paper or a tablet PC, though the accuracy of these devices is not yet demonstrated. Yet another approach captures a sequence of small images of handwriting and assembles them to reconstruct the entire page [14].

#### 26.6.2.1 Discussion of Input Hardware

Magnetic tracking is the widest deployed system owing to high spatial resolution (>1000 dpi), acceptable temporal resolution (>100 Hz), reliability, and relatively low cost [7].

Magnetic and electric tracking require pad electronics and shielding, adding modest thickness and weight to portable devices. Electric tracking uses a normal pen but has no direct way to measure pen tip contact, and must rely on less reliable pen trajectory analysis [15].

Ultrasonic tracking does not require the same writing-surface electronics, thus potentially eliminating the added thickness issue. Relative tracking can reach 256 dpi, but absolute spatial resolution is limited to about 50 dpi because of the air currents that cause Doppler shifts.

Optical tracking with tip-mounted sensors offers high spatial (>2000 dpi) and temporal (>200 Hz) resolution, and can utilize a self-contained pen that remembers everything written. Tiny dots, acting like bar codes, can provide absolute positioning, and can also encode page number, eliminating overwrites when a person forgets to tell the digitizer they have changed pages (a challenge that pen hardware systems with paper interfaces have to address).

Optical tracking with top-mounted sensors is still in its infancy as of this writing, but prototype units are projected to produce >600 dpi relative spatial resolution, on the order of 0.25 mm absolute spatial resolution, and >100 Hz temporal resolution.

Optical methods based on CMOS technology should lend themselves to low-power, low-cost designs.

### 26.6.3 Handwriting Recognition

Handwriting is a very well-developed skill that humans have used for over 5000 years as a means of communicating and recording information. With the widespread acceptance of computers and computer keyboards, the future role of handwriting in our culture might seem questionable. However, as we discussed in the introduction, a number of applications exist where the pen can be more convenient than a keyboard. This is particularly so in the mobile computing space where keyboards are not ergonomically feasible.

Handwriting recognition is fundamentally a pattern classification task. The objective is to take an input graphical mark—the handwritten signal collected via a digitizing device—and classify it as one of a prespecified set of symbols. These symbols correspond to the characters or words in a given language encoded in a computerized representation such as ASCII (see Fig. 26.35). In this field, the term online has been used to refer to systems devised for the recognition of patterns captured with digitizing devices that preserve the pen trajectory; the term offline refers to OCR (optical character recognition) techniques, which instead take as input a static two-dimensional image representation, usually acquired by means of a scanner.

Handwriting recognition systems can be further grouped according to the constraints they impose on the user with respect to writing style (see Fig. 26.36a). The more restricted the allowed handwritten input, the easier the recognition task and the lower the required computational resources [16]. At the most restrictive end of the spectrum, in the boxed-discrete style, users write one character at a time

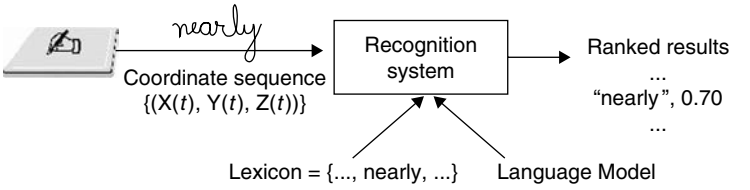



**FIGURE 26.35** The handwriting recognition problem. The image of a handwritten character, word, or phrase is classified as one of the symbols, or symbol strings, from a known list. Some systems use knowledge about the language in the form of dictionaries (or Lexicons) and frequency information (i.e., language models) to aid the recognition process. Typically, a score is associated with each recognition result.

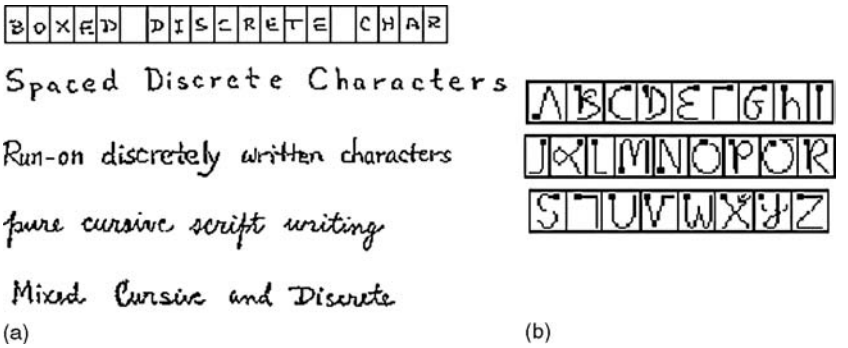



**FIGURE 26.36** Different handwriting styles. In (a), Latin characters are used from top to bottom, according to the presumed difficulty in recognition. (Adapted from Tappert, C.C., Adaptive on-line handwriting recognition, In 7th International Conference on Pattern Recognition, Montreal, Canada, 1984.) In (b), the Graffiti unistroke (i.e., written with a single pen trace) alphabet that restricts characters to a unique prespecified way that simplifies automatic recognition, the square dot indicates starting position of the pen.

within predefined areas. This removes one difficult step—character segmentation (the partitioning of strokes into letters)—from the recognition process. Further recognition accuracy can be achieved by requiring users to adhere to rules that restrict character shapes so as to minimize letter similarity (see Fig. 26.36b). Of course, such techniques require users to learn a new alphabet. At the least restrictive end of the spectrum, in the mixed style, users are allowed to write words or phrases the same way they do on paper—in their own personal style—whether they print, write in cursive, or use a mixture of the two.

Recognition of mixed-style handwriting is a difficult task owing to ambiguity in segmentation and large variations in letter style. Segmentation is complex because it is often possible to wrongly break up letters into parts that are in turn meaningful (e.g., the cursive letter d can be subdivided into letters c and l). Variability in letter shape is partly due to of coarticulation (the influence of one letter on another), and the presence of ligatures (connected characters), which frequently give rise to unintended, spurious letters being detected in the script. Writing styles also vary substantially from individual to individual.

In addition to the writing style constraints, the complexity of the recognition task is also determined by dictionary-size and writer-adaptation requirements. The size of a dictionary or language model can vary from extremely small (for tasks such as state name recognition) to huge or even open-ended (for tasks like proper name recognition). In open vocabulary recognition, any sequence of letters is a plausible recognition result, which is the most difficult scenario for a recognizer. Yet, open-ended, out-of-dictionary writing is essential for many situations, and the best recognizers balance probabilities at the letter and the word levels to prefer in-dictionary solutions while still permitting out-of-dictionary solutions.

In the writer-adaptation dimension, systems capable of out-of-the box recognition are called writer independent; i.e., they can recognize the writing of many writers. This gives a good average performance across different writing styles. However, there is a considerable improvement in recognition accuracy that can be obtained by customizing the letter models of the system to a writer's specific writing style. Recognition in this case is called writer dependent.

Despite these challenges, significant progress has been made in the building of writer-independent systems capable of handling unconstrained text and using dictionary sizes of 20,000 words [17–19] and more. The writer-independent recognizer that shipped in second generation Newton PDAs (the Print Recognizer, circa 1996), and was widely regarded as the world's first genuinely usable handwriting recognizer, used a language model with over 75,000 words in any combination with close to 200 prefixes and over 500 suffixes, plus regular expression grammars for punctuation, dates, times, phone numbers, postal codes, and money, in addition to being able to write completely out-of-dictionary sequences of characters [20]. When updated as Mac OS X's Inkwell [21–23], in 2002, explicit support for writing

URLs was added. Handwriting recognition is also an integral part of Microsoft's Windows XP Tablet PC Edition [24], introduced in 2002 for Tablet PCs, improved and extended in 2005's Service Pack 2, and also driving the so-called Ultra-Mobile PCs [25] introduced in 2006. For a comprehensive survey of the basic concepts behind written language recognition algorithms (see Refs. [26–28]).

### 26.6.3.1 Character-Based Interfaces

In Fig. 26.37 examples of character-based user interfaces for handwritten text input are presented that are representative of those found on many mobile devices. Because of the limited CPU and memory resources available on these platforms, handwritten input is often restricted to the boxed-discrete style, in which one character is entered at a time.

The following are the additional highlights of the user interface on these text input methods:

1. Special input area: Users are not allowed the freedom of writing anywhere on the screen. Instead, there is an area of the screen specially designated for the handwriting user interface, whether for text input or control. This design choice offers the following advantages:
   a. No toggling between edit/control and ink mode: Pen input inside the input method area is treated as ink to be recognized by the recognizer; pen input outside this area is treated as mouse events (for pressing on-screen buttons, selecting text, scrolling, etc.). Without this separation, special provisions, sometimes intrusive and nonintuitive, must be taken to distinguish between the two pen modes.
   b. Better user control: Within the specially designated writing window, it is possible to have additional GUI (graphical user interface) elements that help the user with the input task. For instance, there might be buttons for common edit keys such as backspace, newline, and delete. Similarly, a list of recognition alternates can be easily displayed and selected from. This is


(a) (b)

**FIGURE 26.37** Character-based text input method on today's mobile devices. In (a), user interface for English character input on a cellular phone. In (b), user interface for Chinese character input on a two-way pager.

particularly valuable because top-*N* recognition accuracy—a measure of how often the correct answer is among the highest ranked *N* results—is generally much higher than top-1 accuracy.

c. Consistent UI metaphor: Despite its ergonomic limitations, an on-screen keyboard is generally available as one of the text input methods on the device. Using a special input area for handwriting makes the user interface of alternative text entry methods similar.

2. Modal input. The possibilities of the user's input are selectively limited in order to increase recognition accuracy. Common modes include digits, symbols, uppercase letters, and lowercase letters in English, or traditional versus simplified in Chinese. By limiting the number of characters against which given input ink is matched, the opportunities for confusion and misrecognition are decreased, thus improving recognition accuracy. Writing modes represent another trade-off between making life simpler for the system or simpler for the user (and can cause difficulties if, for example, the system is expecting digits for a phone number field, but the user tries to enter 1–800–GO–FEDEX).
3. Natural character set. It is possible to use any character writing style commonly used in the given language, no need to learn a special alphabet. Characters can be multi-stroke, i.e., written with more than one pen trace.
4. Multi-boxed input. Having the user write every character in its own box provides valuable information for case disambiguation, helping the recognizer distinguish between an uppercase S and a lowercase s, between C and c, and so on, based simply on letter height relative to box height. It also almost entirely eliminates the character segmentation problem, since the strokes in a single box comprise a single character. In addition, when multi-stroke input is allowed, end of writing is generally detected by use of a timer that is set after each stroke is completed; the input is deemed concluded if a set amount of time elapses before any more input is received in the writing area. This time-out scheme is sometimes confusing to users, and gives the perception that recognition takes longer than it actually does. Multiple boxes give better performance in this regard because a character in one box can be concluded if input is received in another box, removing the need to wait for the timer to finish. (However, using a word-level language model, rather than recognizing just individual characters, almost always improves recognition accuracy, which implies that the best hypothesis about a given character may change multiple times as recognition proceeds, which can also be confusing to users. And if one's tablet technology provides entering- and exiting-proximity data, indicating when the pen is near the tablet surface, the end-of-writing timer can be conveniently short-circuited by terminating words or phrases when the pen leaves proximity of the tablet.)

Of all the restrictions imposed on users by these character-based input methods, modality is the one where user feedback has been strongest: people want modeless input. One challenge facing modeless recognition is the unavoidable increase in perplexity—the range and variability, and thus the confusability, in the set of possible answers—that results from having to always allow recognition of all modes—letters, numbers, symbols, words, dates, times, etc.—simultaneously. Another challenge results from the fact that distinguishing between letters that have very similar forms across modes can be virtually impossible without additional information. In English orthography, for instance, there are letters for which the lowercase version of the character is merely a smaller version of the uppercase version; examples include Cc, Oo, Ss, Uu, Ww, etc. Simple attempts at building modeless character recognizers can result in a disconcerting user experience because uppercase letters, or digits, might appear inserted into the middle of lowercase text. Such m1Xed ModE w0rdS (mixed mode words) look like gibberish to users.

In usability studies, the authors have further found that as the text data entry needs on wireless PDA devices shifts from short address book or calendar items to longer notes or e-mail messages, users deem writing one letter at a time to be inconvenient and unnatural.

### 26.6.3.2 More Natural User Interfaces

One known way of dealing with the character confusion difficulties described in the previous section is to use contextual information in the recognition process. At the simplest level this means recognizing characters in the context of their surrounding characters and taking advantage of visual clues derived from word shape. One simple technique is to maintain a running estimate of capital letter height (post-recognition, so even lowercase letters can inform the estimate), and use this to help disambiguate case and even numbers versus letters [20]. More sophisticated applications of this geometric context might look at relative character baselines, heights, sizes, and adjacency between pairs of adjacent characters [29].

Beyond shape, the interpretation of the textual context in which letters are recognized can assist in the disambiguation process. Even a simple bias toward case and mode consistency—so lowercase is more likely to follow lowercase, everywhere except the beginnings of words, and numbers are more likely to follow numbers—can improve recognition accuracy [20].

At a higher level, contextual knowledge can be in the form of lexical constraints, e.g., a dictionary of known words in the language may be used to restrict or guide interpretations of the input ink. These ideas naturally lead to the notion of a word-based text input method. By word, we mean a string of characters that, if printed in text using normal conventions, would be surrounded by white-space characters (see Fig. 26.38a).

Consider a mixed-case word recognition scenario where the size and position of a character, relative to other characters in the word, is taken into account during letter identification (see Fig. 26.38b). Such additional information would allow us to disambiguate between the lowercase and upper case version of letters that otherwise are very much alike. Figure 26.38b also illustrates how relative position within


**FIGURE 26.38** Word-based text input method for mobile devices. In (a), a user interface prototype. In (b), an image of the mixed-case word Wow, where relative size information can be used for distinguishing among the letters in the Ww pair. In (c), an image of the digit string 90187 where ambiguity in the identity of the first three letters can be resolved after identifying the last two characters as digits.

the word could enable us to correctly identify trailing punctuation marks such as periods and commas. A different kind of contextual information can be used to enforce some notion of consistency among the characters within a word. For instance, we could have a digits-string recognition context that favors word hypotheses where all the characters can be viewed as digits; in the image example of Fig. 26.38c, the recognizer would thus rank string "90187" higher than string "gol87."

The use of lexical dictionaries (really just word lists, though ubiquitously referred to as dictionaries) can further guide word-level recognition. For example, despite an observed tremendous variability in how people write their letters u and v, the presence of the word jump in a recognizer's language model can predispose it to produce jump, instead of jvmp. (Not to mention preferring mixed over m1Xed.) Of course, if a user actually wants to write jvmp, she may then have to write very carefully indeed, in order to overcome this bias, if and only if the recognizer allows writing outside of its dictionaries.

A strictly or rigidly applied language model might refuse to recognize words not in its dictionaries. This can produce disturbing and sometimes comical whole word substitutions when a word is misrecognized; e.g., a person might write "catching on?" and get back recognition results such as "egg freckles." The Newton PDA's first generation recognizer suffered from this problem. Together with an untenably small dictionary of just about 10,000 words, this resulted in the Doonesbury effect—those whole word substitutions—so named for the lampooning of the Newton in Gary Trudeau's Doonesbury cartoon strip (using the "egg freckles" example above, among others). A loosely applied language model in the second-generation Newton Print Recognizer allowed users to write outside the dictionaries [20] (besides having much larger dictionaries and generally higher recognition accuracy), and produced much better results for users. This looseness was achieved by incorporating a regular expression grammar that allowed any character anywhere, but at a much lower probability than the word-based parts of the overall language model.

In addition to the modeless input enabled by a word-based input method, there is a writing throughput advantage over character-based ones. In Fig. 26.39 we show the results of a timing experiment where eight users were asked to transcribe a 42-word paragraph using our implementation of both kinds of input methods on a keyboardless PDA device. The paragraph was derived from a newspaper story and contained mixed-case words and a few digits, symbols, and punctuation marks. The length of the text was assumed to be representative of a long message that users might want to compose on such devices. For comparison purposes, users were also timed with a standard on-screen

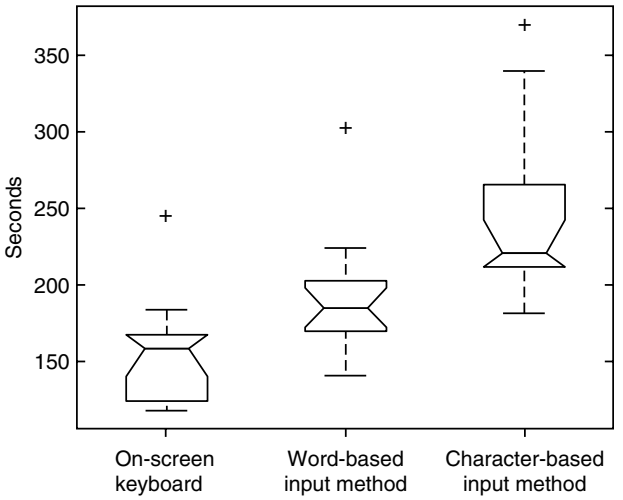



**FIGURE 26.39** Boxplots of time to enter a 42-word message using three different text input methods on a PDA device: an on-screen QWERTY keyboard, a word-based handwriting recognizer, and a character-based handwriting recognizer; median writing throughput was 15.9, 13.6, and 11.4 words/min, respectively.

(software) keyboard. Each timing experiment was repeated three times. The median times were 158, 185, and 220 s for the keyboard, word-based, and character-based input methods, respectively. Thus, entering text with the word-based input method was, on average, faster than using the character-based method.

In this study, the word-based input method did not have, on average, a time advantage over the soft keyboard; however, the user who was fastest with the word-based input method (presumably someone for whom the recognition accuracy was very high and thus had few corrections to do) was able to complete the task in 141 s, which is below the median soft keyboard time. Furthermore, the authors believe that the time gap between these two input methods will be (and may have been already) reversed with improved recognition accuracy and an intuitive, modeless means of error correction, such as were present in the later Newtons and in modern Tablet PCs. We also expect handwriting to offer improvements relative to soft-keyboard text entry in the case of European languages, which have accent marks requiring additional key presses on a soft keyboard.

As one can expect, the advantages of nonmodality and speed associated with a word-based input method over a character-based one comes at the expense of additional computational resources. Some word-based recognition engines have been shown to require a 10× increment in MIPS and memory resources compared to character-based engines. One should also say that, as evidenced by the variability in the timing data shown in the above plots, there isn't a single input method that works best for every user. It is thus important to offer users a variety of input methods from which to choose.

### 26.6.3.3 Write-Anywhere Interfaces

In the same way that writing words, as opposed to writing one letter at a time, constitutes an improvement in terms of naturalness of the user experience, we must explore recognition systems capable of handling continuous handwriting such as phrases. For any kind of computer—and especially the kind of mobile devices we have been considering here, with very limited screen real estate—this idea leads to the notion of a write-anywhere interface in which the user is allowed to write anywhere on the screen; i.e., on top of any application and system element on the screen (see Fig. 26.40).


**FIGURE 26.40** Write-anywhere text input method for mobile devices. Example of an Address Book application with the Company field appearing with focus. Handwritten input is not restricted to a delimited area of the screen but rather can occur anywhere. The company name Data Warehouse has been written.

A write-anywhere text input method is also appealing because there is no special inking area covering up part of the application in the foreground. However, a special mechanism is needed to distinguish pen movement events intended to manipulate user interface elements such as buttons, scrollbars, and menus (i.e., edit/control/mouse mode) and pen events corresponding to handwriting (i.e., ink/pen mode). The solution typically involves a tap and hold scheme wherein the pen has to be maintained down without dragging it for a certain amount of time in order to get the stylus to act temporarily as a mouse, otherwise its input is treated as ink. Alternative methods for distinguishing mousing from inking include treating the pen as a mouse, except when users hold down a particular barrel button on the side of the pen or a particular key on the keyboard, whereupon they get ink. The barrel button mousing-vs-inking option showed up in Mac OS X's Inkwell [21] as of the Tiger (OS X 10.4)

release, as did an additional, unusual option that permitted inking in the air above the tablet surface—while the pen is in close proximity to, but not actually pressing against the tablet—whenever the barrel button is pressed. These options, plus the use of a key on the keyboard to trigger inking were previously introduced in the Motion [30] professional video creation and editing application on the Mac.

When using the more common method of distinguishing mousing from inking—requiring the user to tap and hold to perform mousing—there is another potential user interface problem, which is that users expect certain user interface elements to perform immediate actions. Unlike a tap and drag operation, such as one might use to make a text selection, and for which a delayed action is tolerable, when users tap in a scrollbar or tap an open button, they expect the corresponding action to take place immediately. The same is likely true if a user attempts to drag a window by tapping and dragging the window's titlebar. To accommodate these expectations, a good pen interface should add two more important mouse-vs-ink disambiguations: (1) identify tapping (by the short time between pen down and pen up, and the short distance traveled) and treat the input as a mousing action, and (2) identify certain controls or regions of the screen as instant mousers. If the pen lands in an instant mouser, it immediately behaves as a mouse, instead of an ink pen. The first accommodation takes care of simple issues like tapping on a scrollbar or button, but the second accommodation is required to support immediate dragging of windows, scrollbar thumbs, movable icons, and the like. One must be careful, of course, not to identify too many user interface elements as instant mousers, lest the user feel that the decision about where to write on the screen is overly complicated.

An additional user interface issue with a write-anywhere text input paradigm is that there are usually no input method control elements visible anywhere on the screen. This frees up precious screen real estate, but hides functionality. For instance, access to recognition alternates might require a special pen gesture. As such, a write-anywhere interface will generally have more appeal to advanced users. Furthermore, recognition in the write-anywhere case is more difficult because there is no implicit information on word separation, orientation, or expected size of the text.

#### 26.6.3.4 Scrolling Input Window

An interesting and innovative approach to pen input on devices with severely constrained screen sizes is to allow the pen input window to scroll while the user is writing [31]. The originators of this idea refer to this approach as a treadmill technique, for the way the input window moves continuously underneath the user's writing. For a language written left-to-right, such as English, then, the inking area would continuously scroll to the left, so users could continue to write fairly normally, but in place on the screen. This combines the advantages of a dedicated writing area with the advantages of a seemingly large—in fact, unlimited—writing area. Though users must adapt their writing style somewhat and this technology has yet to make it out of the research lab, the adjustments users must make seem minor and the technique seems promising.

#### 26.6.3.5 Recognition-Aware Applications

Earlier in this section, we discussed how factors such as segmentation ambiguity, letter co-articulation, and ligatures make exact recognition of continuous handwritten input a very difficult task. To illustrate this point, consider the image shown in Fig. 26.41 and the set of plausible interpretations given for it. Can we choose with certainty one of these recognition results as the correct one? Clearly, additional information, not contained within the image, is required to make such a selection. One such source of information already mentioned is the dictionary, or lexicon, for constraining the letter strings generated by the recognizer. At a higher-level, information from the surrounding words can be used to decide, for example, between a verb and a noun word possibility. It is safe to say that the more constraints explicitly available during the recognition process, the more ambiguity in the input that can be automatically resolved. Less ambiguity results in higher recognition accuracy and thus an improved user experience.

(a)

| | |
|---|---|
| clog | dog |
| log | dug |
| lug | lag |
| clay | *clug* |
| *clcrj* | *doij* |

(b)

**FIGURE 26.41** Inherent ambiguity in continuous handwriting recognition. In (a), a sample image of a handwritten word. In (b), possible recognition results, strings not in the English lexicon are in italics.

For many common applications in PDA devices, e.g., contacts, agenda, and web browser, it is possible to specify the words and patterns of words that can be entered in certain data fields. Examples of structured data fields are telephone numbers, zip codes, city names, dates, times, URLs, etc. In order for recognition-based input methods to take advantage of this kind of contextual information, the text input framework on PDA devices needs to allow applications to specify the expected context for a given input field. When text is entered using a keyboard, lexical context is usually not required to obtain accurate text, although certain very precisely defined input fields, such as social security numbers, credit card numbers, and state two-letter codes, might benefit from such constraints. However, operating systems and applications employed largely on pen-based devices can improve the user experience significantly by supporting and providing specific contexts for text input fields and communicating them to recognizers.

One typically uses a grammar to define the permitted strings in a language, e.g., the language of valid telephone numbers. A grammar consists of a set of rules or productions specifying the sequences of characters or lexical items forming allowable strings in the defined language. Two common classes of grammars are BNF grammar or context-free grammar and regular grammar (see [32] for a formal treatment). Grammars are also used in the field of speech recognition and recently the W3C (World Wide Web Consortium) Voice Browser Working Group has suggested an XML-based syntax for representing BNF-like grammars [33]. In Fig. 26.42 we show a fragment of a possible grammar for defining telephone number strings. In an ink-aware text input framework, this grammar, together with the handwritten ink, could be passed along to the recognition engine when an application knows that the user is expected to enter a telephone number.

```
<rule id="digits0-9" scope="private">
  <one-of>
    <item>0</item>
    ...
    <item>9</item>
  </one-of>
</rule>
<rule-id="area-code" scope="private">
  <token>(</token>
  <count number="3">
    <ruleref uri="#digit0-9"/>
  </count>
  <token>)</token>
</rule>
<rule id="prefix" scope="private">
  <count number="3">
    <ruleref uri="#digit0-9"/>
  </count>
</rule>
```

```
<rule id="suffix" scope="private">
  <count number="4">
    <ruleref uri="#digit0-9"/>
  </count>
</rule>


<!-- Main rule -->
<rule id="phone-num" scope="public">
  <count number="0-1">
    <ruleref uri="#area-code"/>
  </count>
  <ruleref uri="#prefix"/>
  <count number="0-1">
    <token>-</token>
  </count>
  <ruleref uri="#suffix"/>
</rule>
```

**FIGURE 26.42** Example of an XML grammar-defining telephone numbers and written as per the W3C Voice Working Group Specification. There are four private rule definitions that are combined to make the main rule called phone-num.

An alternative approach, one used in the Newton for example, is for the text input framework to provide a set of common classes of input, such as phones, dates, times, general text, etc., and let the application specify one or more of these categories, as appropriate.

Given this kind of contextual information, recognizers should, in general, use the information to boost the probability of the relevant text strings, but should not, unless expressly requested, make the constraint a rigidly applied one. And it may be better to factor the soft constraints appropriate to a recognizer from the hard constraints the text input framework or the application itself might need to apply, in order to verify a correctly formatted credit card number or two-letter state code for example. Most text input fields are not so rigid, and users may be frustrated when they try to enter 1–800-MY-APPLe in a phone number field if the recognizer refuses to recognize anything but digits (and hyphens and parentheses).

Information about how the ink was collected, such as resolution and sampling rate of the capture device, whether writing guidelines or other writing size hints were used, spatial relationships to nearby objects in the application interface, etc., should also be made available to the recognition engine for improved recognition accuracy.

To be fully ink- or recognition-aware, the text input framework and applications need to provide an easy, in-place, modeless correction mechanism. This is best supported by *overwriting* and a *top-N* list of recognition alternatives. Overwriting is the ability to write directly over the text to be replaced, making a separate selection process unnecessary, and is particularly convenient if the interface permits overwriting individual characters. A list of recognition alternatives leverages the fact that recognition accuracy for, say, the top-5 alternatives is usually substantially higher than for just the top-1 alternative. Either the recognizer or the text input framework may also explicitly substitute particularly common or convenient alternatives in these lists, such as repeating the top choice but with altered case for the first letter. One user interface technique for accessing such an alternatives list with a write-anywhere recognition model is to expose the alternatives in a pop-up menu when the user taps twice and holds down the pen on the second tap. Another technique is to take advantage of a given system's existing contextual menu access method, such as right-clicking (Windows) or control-clicking (Macintosh) on a recognized word, to bring up the recognition alternates menu.

## 26.6.4 Ink and the Internet

Digital ink does not always need to be recognized in order to be useful. Two daily life applications where users take full advantage of the range of graphical representations that are possible with a pen are messaging, as when we leave someone a post-it note with a handwritten message, and annotation, as when we circle some text in a printed paragraph or make a mark in an image inside of a document. This subsection discusses Internet-related applications that will enable similar functionality. Both applications draw attention to the need for a standard representation of digital ink that is appropriate in terms of efficiency, robustness, and quality.

### 26.6.4.1 Ink Messaging

Two-way transmission of digital ink, possibly wireless, offers PDA users a compelling new way to communicate. Users can draw or write with a stylus on the PDA screen to compose a note in their own handwriting. Such an ink note can then be addressed and delivered to other PDA users, e-mail users, or fax machines. The recipient views the message as the sender composed it, including drawings and text in any mix of languages (see Fig. 26.43).

In the context of mobile-data communications it is important for the size of such ink messages to be small. There are two distinct modes for coding digital ink: raster scanning and curve tracing [34,35]. Facsimile (fax machine) coding algorithms belong to the first mode, and exploit the correlations within consecutive scan lines. Chain coding (CC), belonging to the second mode, represents the pen trajectory as a sequence of transitions between successive points in a regular lattice. It is known that curve tracing algorithms result in a higher coding efficiency if the total trace length is not too long. Furthermore, use of a raster-based technique implies the loss of all time-dependent information.


**FIGURE 26.43** Example of ink messaging application for mobile devices. Users can draw or write with a stylus on the device screen to compose an e-mail in their own handwriting; no automatic recognition is necessarily involved.

Message sizes of about 500 bytes have been reported for messages composed in a typical PDA screen size, using a CC-based algorithm known as multi-ring differential chain coding (MRDCC) [36]. MRDCC is attractive for transmission of ink messages in terms of data syntax, decoding simplicity, and transmission error control; however, MRDCC is lossy, i.e., the original pen trajectory cannot be fully recovered. If exact reconstructability is important, a lossless compression technique is required. This is likely to be the case if the message recipient might wish to run verification or recognition algorithms on the received ink, e.g., if the ink in the message corresponds to a signature that is to be used for computer authentication. One example of a lossless curve-tracing algorithm proposed by the ITU (international telecommunication union) is zone coding [37]. Our unpublished evaluation of zone coding, however, reveals there is ample room for improvement.

Additional requirements for an ink messaging application include support for embedded ASCII text, support for embedded basic shapes (such as rectangles, circles, and lines), and support for different pen-trace attributes (such as color and thickness).

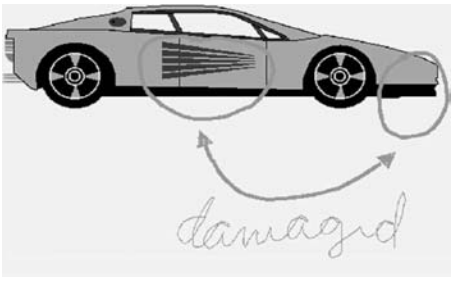
### 26.6.4.2 InkML and SMIL

SMIL, pronounced smile, stands for synchronized multimedia integration language. It is a W3C recommendation [38] defining an XML compliant language that allows a spatially and temporally synchronized description of multimedia presentations. In other words, it enables authors to choreograph multimedia presentations where audio, video, text, and graphics are combined in real-time. A SMIL document can also interact with a standard HTML page. SMIL documents are becoming common on the web as part of streaming technologies [39,40].

The following are the basic elements in a SMIL presentation: a **root-layout**, which defines things like the size and color of the background of the document; a **region**, which defines where and how a media element such as an image can be rendered, e.g., location, size, overlay order, scaling method; one or more media elements such as **text, img, audio**, and **video**; means for specifying a timeline of events, e.g., **seq,** and **par** indicate a block of media elements that will all be shown sequentially or in parallel, respectively, **dur** gives an explicit duration, **begin** delays the start of an element relative to when the document began or the end of other elements; means for skipping some part of an audio or a video (**clip-begin** and **clip-end**); means for adapting the behavior of the presentation to the end-user system capabilities (**switch**); means for freezing a media element after its end (**fill**); and a means for hyperlinking (**a**). For a complete introduction and tutorial see Refs. [41,42].

Digital ink is not currently supported as a SMIL native media type. One option would be to convert the ink into a static image, say in GIF format, and render it as an **img** element; however, this would preclude the possibility of displaying the ink as a continuous media (like an animation). Another option is to use the SMIL generic media reference **ref** (see Fig. 26.44); this option requires the existence of an appropriate MIME content-type/subtype.

Another W3C working group has produced a working draft of an Ink Markup Language, InkML [43], which can serve as the data format for representing ink entered with an electronic pen or stylus. InkML supports the input and processing of handwriting, gestures, sketches, music, and other notational languages in web-based and non-web-based applications, and provides a common format for the exchange of ink data between hardware devices and between software components such as handwriting



(a)

```
<smil>
  <head>
    <meta name = "title" content = "Ink and SMIL" />
    <root-layout width = "300" height = "200"
          background-color = "white" />
  </head>
  <body>
    <par>
      <img src = "car.gif" region = "main" />
      <ref src = "car.uni" region = "onmain"
          type = "ink/unipen" fill = "freeze"/>
      <audio src = "car.wav" />
    </par>
  </body>
</smil>
```

(b)

**FIGURE 26.44** Example of the role of digital ink in SMIL documents. In (a), a diagram or photo taken with a digital camera can be annotated with a pen; the digital ink can be coordinated with a spoken commentary. In (b), a corresponding short SMIL document fragment assuming the existence of an appropriate MIME content-type called ink and a subtype called unipen for representing the ink.

and gesture recognizers, signature verifiers, and other ink-aware modules. InkML is currently envisioned as an element of the broader W3C Multimodal Interaction Framework [44].

There also exists a nonstandard but open data format for digital ink called Unipen [45] that flexibly captures many aspects of data representation for ink acquisition, transmission, and recognition. A substantial body of handwriting data is available in this format, for the cost of the media only, from the International Unipen Foundation (iUF). The iUF also sponsors handwriting recognition accuracy competitions. The iUF is reportedly working on a fusion of the InkML and Unipen formats, called UPX, that will contain some format extensions they feel are necessary, but which should trivially allow export to InkML.

## 26.6.5 Extension of the Pen-and-Paper Metaphor

Use of the pen-and-paper paradigm dates back to almost 3000 BC. Paper, as we know it today, dates back to around 200 AD. Hence, the notion of writing with a pen on paper is an extremely natural way of entering handwritten information.

Archival and retrieval are two primary actions performed on handwritten information captured using traditional pen and paper. The problem, however, with regular pen and paper is that the process of retrieving information can be extremely inefficient. Retrieving information typically involves visually scanning the documents, which can be inefficient when the amount of handwritten information becomes large. One way to make the process efficient is to tag the information in a useful way. For example, a yellow sticker on pages that relate to a certain topic, or entering information about different topics into different notebooks, can make the process of looking for information on a topic more efficient, when using normal paper. The goal here is to extend the same functionality to electronic pen-and-paper systems [46].

Extending the pen-and-paper metaphor, one of the main applications for digital ink capture systems is to provide users with efficient ink archival/retrieval capabilities, by providing users the tools to tag information captured on the devices in a useful way. Different systems support different methods of tagging digitally acquired ink. Some systems combine ordinary paper with computers; some allow vocal annotation of notes; some perform real-time recognition, providing searchable text; and some retain ink but perform deferred or background recognition, providing searchable ink.

### 26.6.5.1 Combining Digital Ink and Physical Paper

Though most have so far failed in the marketplace, there have been several interesting attempts at leveraging the best of both real, physical paper and digital ink. Paper offers a completely familiar, intuitive writing environment and avoids the awkward and limited tactile feedback of writing with a metal or plastic tip on a slick glass surface. (An unpublished study at Apple Computer, during the continuing development of the Newton, showed that a modest roughening/texturing of the glass writing surface substantially increased recognition accuracy, all other things being equal, and all later

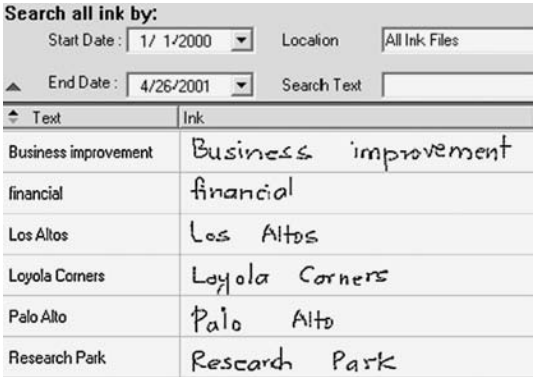



**FIGURE 26.45** Ink retrieval using keywords. Example of an application that uses the ASCII tags associated with handwritten ink to retrieve information from handwritten documents.

generation models of the Newton shipped with such a surface.) Digital ink offers the opportunity of recognition and textual or graphical search for easier retrieval of data from handwritten notes and drawings.

The now discontinued IBM ThinkPad TransNote [47] combined a laptop computer and a pad of paper overlying a digitizer into a single portfolio. In concert with IBM, Cross produced the now discontinued CrossPad [48], consisting of a pad of paper over a digitizer that could retain information about what was drawn on the pad. In both systems, anything written or drawn on the plain, unmarked paper was also digitized and could be uploaded to the computer. ACECAD still makes a DigiMemo pad [49] that functions much like the CrossPad.

Ink management software on the computer allowed users to archive handwritten notes and retrieve them, using either the time of creation of the notes or tags associated with keywords. The tags were text strings created using a handwriting recognition system. Figure 26.45 shows an example of a piece of the ink management software that displayed blocks of ink marked as keywords in the middle column and their tags in the left column. Users could retrieve handwritten documents by clicking on the keywords or typing a word in the search text box in the upper right-hand-top corner of the application.

In the application shown in Fig. 26.45, all the tags are text strings; however, one could easily extend the retrieval paradigm to use graphical queries and retrieve documents containing graphics, using features extracted from the graphical query. An example of this is shown in Fig. 26.46.

Anoto [3] and Logitech [4] use a tip-mounted optical sensor and specially marked paper to allow the recording of digital ink during drawing or writing. Though the pens are still somewhat expensive and lack a wireless means of connectivity or any kind of real-time use with a computer or mobile device, the cost of the special paper has dropped to the point that it is not the product killer many thought it would be when the technology was first introduced. Support for handwriting recognition is limited, but third party applications, such as Vision Objects' MyScript [50], provide more extensive functionality.

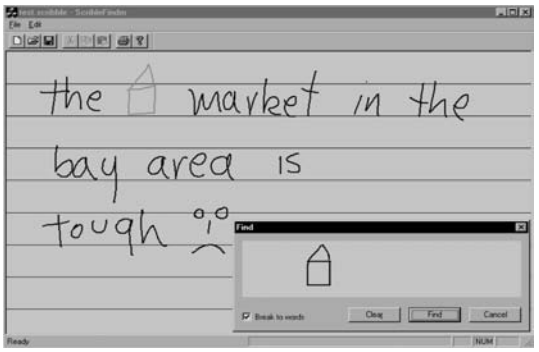



**FIGURE 26.46** Ink searching example. Users can search for an ink pattern inside a longer ink document, or collection of documents.

### 26.6.5.2 Digital Ink Documents and Handwriting Recognition

The previously discussed handwriting recognition methods may be used in at least three different modes: immediate, deferred, and background. With immediate recognition, handwritten input by the user is immediately converted to text. This has the advantage that, assuming recognition is accurate, the resulting text is more legible than handwriting and the user is left with a standard text document that can be archived, retrieved, and searched like any other computer document. Particularly when combined with a write-anywhere pen input model, immediate recognition can make a very effective text input method for most tasks

a user performs on a computer, including providing text input from a pen for applications that are not themselves ink-aware. It has the disadvantage that, if recognition accuracy is poor, unnoticed errors may creep into documents; and distinguishing between graphical drawings and text becomes either very difficult or requires a modal interface, in which the user is required to specify whether she is currently entering text or graphics.

Some systems allow deferred recognition, in which the pen input remains visible as ink, and may be selected for recognition at a later time. This has the advantage that the user's writing and/or drawing is unchanged, so no errors are introduced by recognition and graphics and text are treated identically. The disadvantage is that conversion to text requires an extra step and legibility may suffer, particularly for documents being shared amongst multiple users. Also, ink is not an acceptable data format for many applications.

Background recognition provides at least some of the advantages (and disadvantages) of both immediate and deferred recognition. With background recognition, the user's input remains visible as ink, but recognition is performed, predictably enough, in the background. This produces what is sometimes referred to as searchable ink. That is, even though the original handwritten text and drawings are untouched, because recognition results are obtained in the background, the user may ask the computer to search through the apparently ink-based document just as if it were stored in plain text. In fact, if the background recognition results retain a top-N list of recognition alternatives, or if the search allows a small amount of variance between the search terms and the recognition results, even modest recognition errors may be accommodated and still allow near perfect searching. This approach does imply, however, a complex ink-plus-recognition-results data format, which will only be usable by ink-aware applications. However, this problem is at least partially mitigated by the standard technique of copy operations providing multiple data formats and the paste operation selecting the data format best supported by the current application. For example, data copied from a note-taking application that retains ink and the top-5 recognition results could easily be pasted as plain text into an application that does not support ink.

Modern Tablet PCs, running Microsoft Windows XP for Tablet PC with Service Pack 2, provide all three recognition modes. Apple's Inkwell pen input framework in Mac OS X (10.3 and later) also supports all three recognition modes, as may be seen in Mage Software's inkBook [51]. The Newton PDA offered both immediate and deferred recognition at introduction, in 1993, and fans of the device and its (later generation) handwriting recognition capabilities continue to use the device and support each other through Web sites [52], e-mail lists [53], worldwide organizations [54], and worldwide conferences [55], some nine years after the product's demise (as of 2006), giving some indication of the value a well implemented pen-based system can bring to its users.

Tablet PCs offer the capability of directly creating and editing documents containing digital ink, and thus makes excellent note-taking devices for meetings and classrooms. Apple has so far failed to enter the tablet PC market, despite system level support for handwriting recognition in Mac OS X since the Jaguar release (10.2), but a third party, Axiatron, has stepped in to fill the void with the ModBook [56].

#### 26.6.5.3 Gesture Recognition

A gesture is handwritten ink that implies a particular action. In many cases, a gesture can be used to represent an action much more efficiently than it may be specified through a set of keyboard events. An example is the task of moving a portion of text from one position to another. Using a keyboard would involve selecting the portion of ink to be moved, copying it into a clipboard, deleting the selection, moving the cursor to the place in the document where the user would like to place the ink, and finally pasting the ink. Using a pen would allow users to indicate the same action by drawing a selection area around the ink to be moved and an arrow indicating the position to which the selection is to be moved. An example of this is shown in Fig. 26.47.

Even when simple keyboard alternatives exist for a particular action, transitioning back and forth between the pen and a keyboard can interrupt workflow. Modern pen input frameworks therefore often

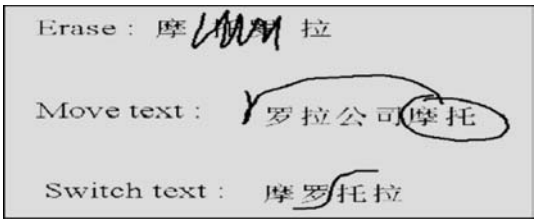



**FIGURE 26.47** Example of pen-and-paper-like editing. Users can perform erasing by scribbling directly on the unwanted text, moving text by circling and dragging, and transposing text by common gesturing.

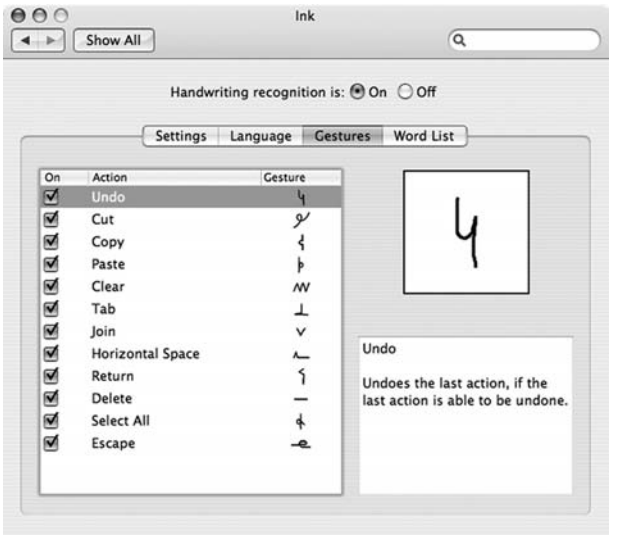



**FIGURE 26.48** Inkwell's suite of common action gestures in Mac OS X.

provide a suite of easily learnable gestures for carrying out common user interface actions, such as cut, copy, and paste. Figure 26.48 shows the suite of system gestures provided by Inkwell in Mac OS X.

Pen input and gesture recognition are also finding a home controlling important segments of the user interface in modern professional applications for creating and editing video, such as Apple's Motion [30], now incorporated into the Final Cut Studio suite [57]. A suite of 38 gestures (including two of Inkwell's system gestures and overriding a few others) is defined that controls many aspects of the application's user interface, such as > for play, – (from left to right) for advance one frame, – (from right to left) for back up one frame, and so on. These gestures are shown in Fig. 26.49. This kind of gestural user interface used to only be available on very expensive, dedicated hardware systems, but the presence of a modern pen input framework in Mac OS X, Inkwell, has allowed the capabilities to reach personal computers.

### 26.6.5.4 Smart Ink

One can extend the handwriting and gesture recognition system to allow users to associate more structure with groups of pen strokes as shown in Fig. 26.50, by combining handwriting recognition with the page segmentation features of OCR. The handwritten document on the left side is a typical handwritten page with text, tables and drawings, and the one on the right side is a version of the same document after being automatically interpreted by a smart ink recognition scheme. This association allows users to work with handwritten documents in more efficient ways, which may turn an electronic pen into a more effective way of entering information than a keyboard and mouse.

Research tools such as SILK and DENIM [58,59] further extend these ideas to include the assignment of dynamic behaviors to the hand-drawn objects, thus allowing graphic designers to quickly sketch a user-interface with an electronic pen. The tool addresses the needs of designers who prefer to sketch early interface ideas on paper or whiteboard and concentrate on behavior. This approach improves over existing user interface construction tools that focus largely on appearance, thus making it easy to specify widgets and their colors, alignment, fonts, etc., and showing what the interface will look like, but making it difficult to show what it will actually do.

Innovative new user interfaces continue to make strong use of pen, such as the BumpTop prototype [60], that uses the pen to control a 3D desktop metaphor incorporating physics, piles, and a number of novel document and window browsing techniques, and the iPen [61] concept system, that (in theory) combines an electronic pen, mobile display device, mobile phone, voice recorder, and mp3 player!

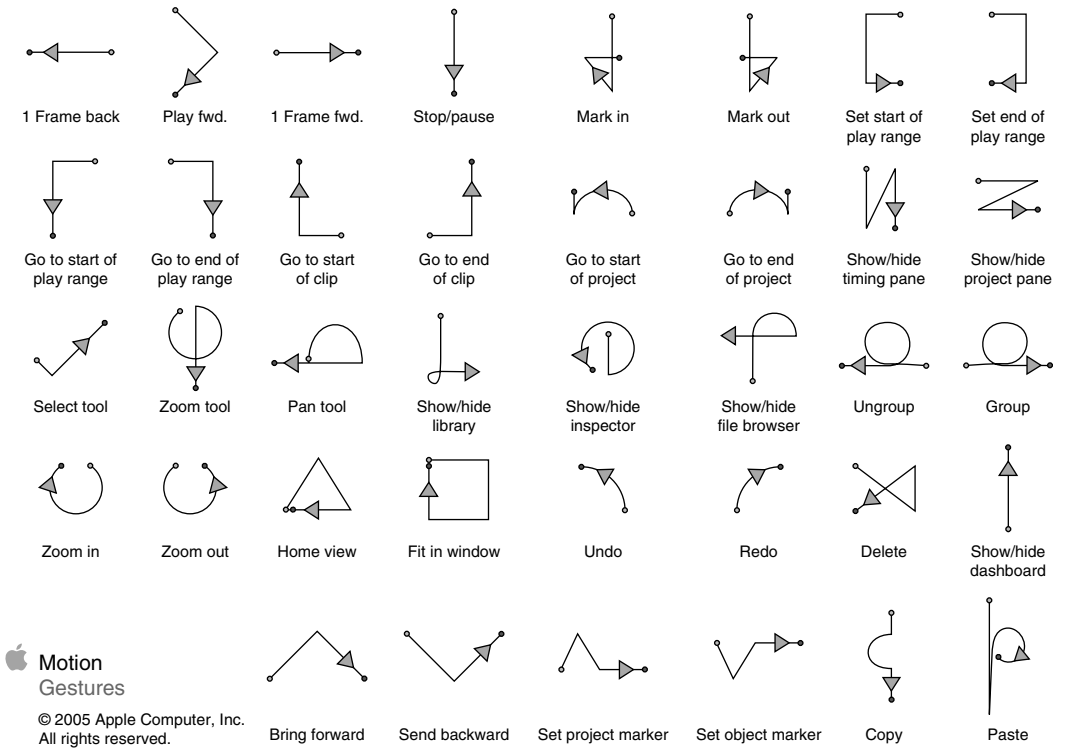



**FIGURE 26.49** The Motion application's suite of user interface control gestures.

## 26.6.6 Pen Input and Multimodal Systems

A multimodal interface is one that integrates multiple kinds of input simultaneously to achieve a desired result. With the increased availability of significant computing power on mobile devices and of efficient wireless connectivity enabling distributed systems, development of pen-based multimodal interfaces is becoming more and more feasible. The motivation is simple: create more robust, flexible, and user-friendly interfaces by integrating the pen with other input modalities such as speech. Greater robustness is achievable because cross-modal redundancy can be used to compensate for imperfect recognition on each individual mode. Greater flexibility is possible because users can choose from among various modes of achieving a task, or of issuing commands, employing the mode that is most appropriate at the time. Better user-friendliness will result from having computer interfaces that more closely resemble the multi-modality naturally present in human communication. In this section we review some successful multimodal systems that take advantage of the pen to produce synergistic interfaces, highlighting their common features.

Cohen et al. [62] combined speech and pen gestures to interact with a 2D representation, like a map, of the entities in a 3D scene such as the one generated with a battlefield simulator. An interactive map is displayed on a handheld device where the user can draw or speak to control the system. For example, while holding the pen at some location in the map, the user may say "XXXX platoon;" this command will result in the creation of a platoon simulation element labeled "XXXX" at the desired location. The user can then assign a task to the new platoon by uttering a command like "XXXX platoon follow this route" while drawing a line on the map.

Heyer and Julia [63] combined speech, pen gestures, and handwriting recognition in a travel planning application. Users interact with the map of a city, possibly displayed on a PDA device, to find out information about hotels, restaurants, and tourist sites. This information is accessed from a public database through the Internet. Pen and voice may be used by speaking a query such as "what is the

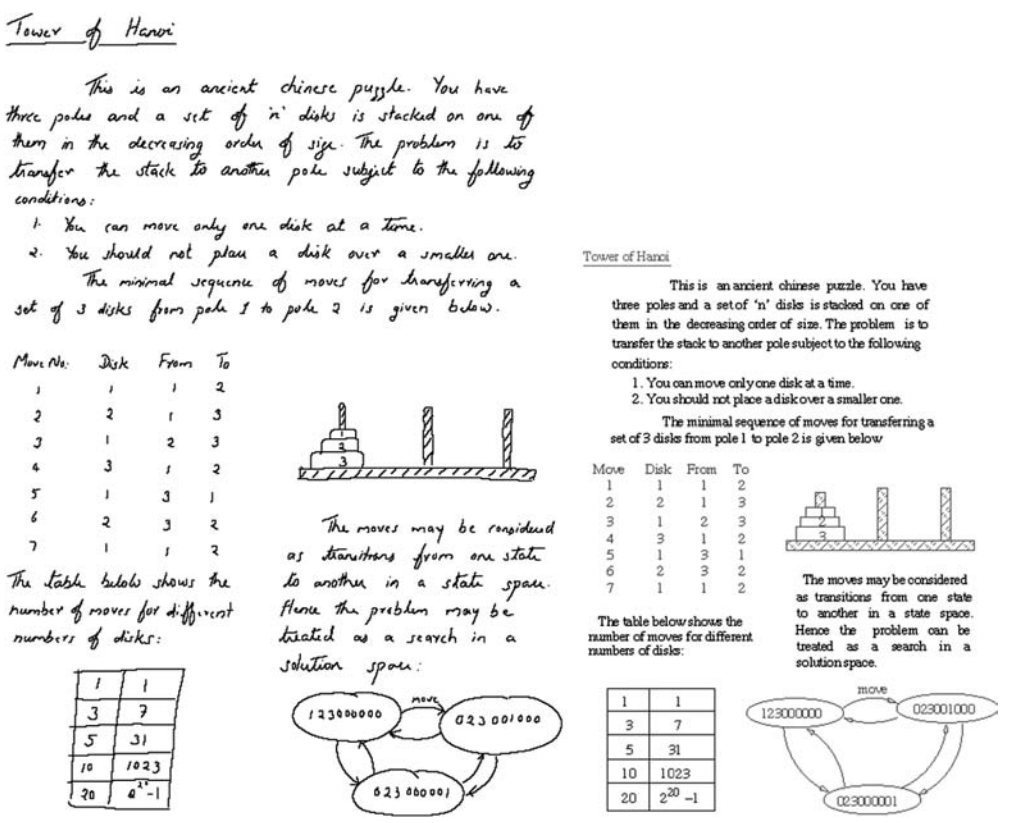



**FIGURE 26.50** Segmentation and recognition of online documents. Example of a typical handwritten page with text, tables, and drawings; and the desired segmentation interpretation.

distance from here to Fisherman's Wharf" while making a mark on the map. Pen-only gestures can also be used for control actions, such as moving the viewing area. Similarly, voice-only commands are allowed as in "show me all hotels with a pool."

Tue Vo and Wood [64] prototyped a multimodal calendar application called Jeanie. This is a very common application on PDA devices and one having several tasks that can be simplified by the multimodal method of pointing to or circling objects on the screen in addition to speaking commands. For example, a command combining spoken and handwritten input is reschedule this on Tuesday, uttered while holding the pen on a meeting entry in the appointment list. An example of a pen-only command is drawing an X on a meeting entry to cancel it.

Suhm et al. [65] have explored the benefits of multimodal interaction in the context of error correction. Specifically, they have integrated handwriting recognition in an automatic dictation system. Users can switch from continuous speech to pen-based input to correct errors. This work capitalizes on the fact that words that might be confusable in one modality (e.g., sound similar) are not necessarily so in another one (i.e., their handwritten shapes are likely to be different). Their study concluded that multimodal error correction is more accurate and faster than unimodal correction by re-speaking.

Multimodal applications such as these are generally built using a distributed agent framework. The speech recognizer, the handwriting recognizer, the gesture recognizer, the natural language understanding module, the database access module, etc., might each be a different agent—a computing process that provides a specific service and which runs either locally on the PDA device or remotely. These agents cooperate and communicate with each other in order to accomplish tasks for the user. One publicly

available software environment offering facilitated agent communication is the open agent architecture (OAA) from SRI [66].

A special agent is needed for integrating information from all input sources to arrive at a correct understanding of complete multimodal commands. Such a unification agent is sometimes implemented using semantic frames, a knowledge representation scheme from the early A.I. days [67], consisting of slots specifying pieces of information about the command. Recognition results from each modality agent are parsed into partially filled frames, which are then merged together to produce a combined interpretation. In the merging process information from different input modes is weighted, meaningless command hypotheses are filtered out, and additional feedback from the user might be requested.

### 26.6.7 Summary

As more electronic devices with pen interfaces have and continue to become available for entering and manipulating information, applications need to be more effective at leveraging this method of input. Pen is a mode of input that is very familiar for most users since everyone learns to write at an early age. Hence, users will tend to use this as a mode of input and control when available. Providing enhanced user-interfaces that will make it easier for users to take advantage of the pen interface in effective ways will make it easier for them to work with such devices.

Section 26.6 has given an overview of the pen input devices available today along with some of the applications that use the electronic pen either in isolation or in conjunction with other modes of input such as speech and the keyboard. The community has made great strides in addressing a number of the user-interface issues for capturing and manipulating information from electronic pens. Challenges, including recognizer accuracy, the distinguishing of geometric shapes from text, and designing more effective, possibly multimodal interfaces, remain to be addressed as purveyors of technology seek to provide higher and higher levels of satisfaction to users.

## Acknowledgment

The authors thank Thomas G. Zimmerman, Research Staff Member with the Human/Machine Interface Gadgets at IBM Research, for his input on Section 26.6.2, and Carlos McEvilly, Research Staff Member with the Motorola Human Interface Labs, for proofreading the first edition manuscript. For the second edition, the new, third author thanks Levi Thomas for her proofreading and copyediting, among many other things.

## References

1. Mimio electronic, interactive whiteboards. www.mimio.com.
2. Smart Technologies electronic, interactive whiteboards. www.smarttech.com.
3. Anoto digital pen and paper. www.anoto.com.
4. Logitech io digital pen. www.logitech.com/index.cfm?page = products/features/digitalwriting.
5. NaviScribe/ESPi (Electronic Scripting Products, Inc.) digital pen. www.naviscribe.com.
6. J. Subrahmonia and T. Zimmerman, Pen computing: Challenges and applications. In *IEEE Conference on Pattern Recognition*, Barcelona, Spain, 2000.
7. Wacom graphics tablets. www.wacom.com/productinfo.
8. ACECAD Flair graphics tablets. www.acecad.com.tw/products.html.
9. Hitachi BlueTooth graphics tablet. www.hitachi-soft.com/icg/products/BT1.html.domino.research.ibm.com/comm/wwwr_thinkresearch.nsf/pages/paper397.html.
10. T. Zimmerman and F. Hoffmann, IBM Research, patent pending, 1995.

11. N. Yamaguchi, H. Ishikawa, Y. Iwamoto, and A. Iida, Ultrasonic coordinate input apparatus. U.S. Patent 5,637,839, June 10, 1997.
12. O. Kinrot and U. Kinrot, Interferometry: Encoder measures motion through interferometry. *Laser Focus Worlds,* 36(3) March 2000. www.laserfocusworld.com/display_article/66078/12/ARCHI/none/Feat/INTERFEROMETRY:-Encoder-measures-motion-through-interferometr.
13. OTM Technologies VPen. www.otmtech.com/vpen.asp.
14. S. Nabeshima, S. Yamamoto, K. Agusa, and T. Taguchi, MEMO-PEN: A new input device. In *Conference Companion on Human Factors in Computing Systems* (Denver, Colorado, United States, May 07–11, 1995). I. Katz, R. Mack, and L. Marks, Eds. CHI '95, ACM Press, New York, NY, 256–257.
15. M.E. Munich and P. Perona, Visual input for pen-based computers. In *13th International Conference on Pattern Recognition*. Vienna, Austria, Aug. 25–29, 1996. www.vision.caltech.edu/mariomu/research/papers/icpr96.pdf.
16. C.C. Tappert, Adaptive on-line handwriting recognition. In *the 7th International Conference on Pattern Recognition*, Montreal, Canada, 1984.
17. G. Seni and T. Anastasakos, Non-cumulative character scoring in a forward search for online handwriting recognition. In *IEEE Conference on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, 2000.
18. K.S. Nathan, H.S.M. Beigi, J. Subrahmonia, G.J. Clary, and M. Maruyama, Real-time on-line unconstrained handwriting recognition using statistical methods. In *IEEE Conference on Acoustics, Speech, and Signal Processing*, Michigan, MI, 1995.
19. S. Jaeger, S. Manke, and A. Waibel, NPEN++: An online handwriting recognition system. In *Proceedings of the 7th International Workshop on Frontiers in Handwriting Recognition*, Amsterdam, The Netherlands, 2000.
20. L.S. Yaeger, B.J. Webb, and R.F. Lyon, Combining neural networks and context-driven search for on-line, printed handwriting recognition in the Newton. *AI Magazine, AAAI*, 19(1): 73–89, Spring 1998. beanblossom.in.us/larryy/Yaegeretal.AIMag.pdf.
21. Inkwell pen input framework in Mac OS X. www.apple.com/macosx/features/inkwell/.
22. Inkwell: *Using Ink Services* developer documentation. developer.apple.com/documentation/Carbon/Conceptual/using_ink/ink_intro/chapter_1_section_1.html.
23. Inkwell: *Ink Services Reference* developer documentation. developer.apple.com/documentation/Carbon/Reference/ink_services_ref/Reference/reference.html.
24. Programming the Tablet PC. windowssdk.msdn.microsoft.com/en-us/library/ms698573(VS.80).aspx.
25. Ultra-Mobile PC. www.microsoft.com/windowsxp/umpc/.
26. R. Plamondon and S.N. Srihari, On-line and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 22(1): 63–84, 2000.
27. C.C. Tappert, C.Y. Suen, and T. Wakahara, The state of the art in on-line handwriting recognition, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 12(10): 787–808, 1990.
28. R. Plamondon, D. Lopresti, L.R.B. Schomaker, and R. Srihari, "On-Line Handwriting Recognition," *Encyclopedia of Electrical and Electronics Eng.*, J.G. Webster, ed., vol. 15, pp. 123–146, Wiley Inter-Science, 1999.
29. R.F. Lyon and L.S. Yaeger, On-line hand-printing recognition with neural networks. (invited) *Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems* (1996), 201–212. beanblossom.in.us/larryy/LyonYaeger.MN96.pdf.
30. Motion video creation and editing software. www.apple.com/finalcutstudio/motion/.
31. G. Seni, TreadMill ink—enabling continuous pen input on small devices. In *Proceedings of the 8th International Workshop on Frontiers in Handwriting Recognition* (IWFHR'02), IEEE, 215–220, 2002.
32. H.R. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
33. Speech Recognition Grammar Specification for the W3C Speech Interface Framework. W3C Working Draft. www.w3.org/TR/speech-grammar. 2001.

34. H. Freeman, Computer processing of line-drawing data. *Computer Surveys*, 6(1): 57–97, March 1974.
35. T.S. Huang, Coding of two-tone images, *IEEE Transactions on Communications*, COM-25, 11, 1406–1424, November 1977.
36. J. Andrieux and G. Seni, On the coding efficiency of multi-ring and single-ring differential chain coding for telewriting application. *In IEEE Proc.—Vision, Image, and Signal Processing*, 148(4), 2001.
37. ITU-T Recommendation T.150. Terminal Equipment and Protocols for Telematic Services. 1993.
38. Synchronized Multimedia Integration Language (SMIL). W3C Recommendation. www.w3.org/TR/SMIL/. See also www.w3.org/AudioVideo/.
39. Real SMIL player. www.real.com.
40. SMIL support in QuickTime. www.apple.com/quicktime/technologies/interactivity/smil.html.
41. L. Hardman, A Smil Tutorial. homepages.cwi.nl/∼media/SMIL/Tutorial/. 1998.
42. W3C SMIL tutorial. www.w3schools.com/smil/default.asp.
43. Ink Markup Language (InkML). W3C Working Draft. www.w3.org/TR/InkML/.
44. Multimodal Interaction Framework. W3C Working Draft. www.w3.org/TR/mmi-framework/.
45. Unipen digital ink format at the Int. Unipen Foundation (iUF). www.unipen.org.
46. M. Lazzouni, A. Kazeroonian, D. Gholizadeh, and O. Ali, Pen and paper information recording system. US Patent 5,652,412. July, 1997.
47. IBM ThinkPad TransNote computers. www-132.ibm.com/search/transnote.html. www.research.ibm.com/thinkresearch/pages/2001/20010515_transnote.shtml.
48. CrossPad digital ink and paper tablet. www.cross.com/ProductSupport/CrossPad.aspx.
49. ACECAD DigiMemo digital pen with paper pad. www.acecad.com.tw/products.html.
50. VisionObjects MyScript note-taking app and handwriting recognition. www.visionobjects.com.
51. Mage Software inkBook note-taking app. www.magesw.com/inkbook/.
52. United Network of Newton Archives. www.unna.org.
53. NewtonTalk e-mail list. newtontalk.net.
54. Worldwide Newton Association. worldwide-newton.asso.eu.org.
55. Worldwide Newton Conference. wwnc.newtontalk.net.
56. Axiotron ModBook Tablet Mac. www.axiotron.com/index.php?id=modbook.
57. Final Cut Studio audio/video mastering software. www.apple.com/finalcutstudio/.
58. DENIM and SILK. guir.berkeley.edu/projects/denim/research/.
59. M.W. Newman and J. Landay, Sitemaps, storyboards, and specifications: A sketch of Web site design practice. In *Proceedings of the Designing Interactive Systems (DIS'00)*, New York, August 2000.
60. BumpTop prototype user interface. honeybrown.ca/Pubs/BumpTop.html.
61. iPen concept system. gp.co.at/works/ipen/.
62. P.R. Cohen, D. McGee, S.L. Oviatt, L. Wu, J. Clow, R. King, S. Julier, and L. Rosenblum, Multimodal interactions for 2D and 3D environments. *IEEE Computer Graphics and Applications*. July/August 1999.
63. A. Heyer and L. Julia, Multimodal maps: An agent-based approach. In *Multimodal Human-Computer Communication, Lecture Notes in Artificial Intelligence*. 1374. Bunt/Beun/Borghuis Eds. Springer 1998.
64. M. Tue Vo and C. Wood, Building an application framework for speech and pen input integration in multimodal learning interfaces. In *IEEE Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1996.
65. B. Suhm, B. Myers, and A. Waibel, Model-based and empirical evaluation of multimodal interactive error correction. In *Proceedings of the CHI 99 Conference*, Pittsburgh, PA, May 1999.
66. Open Agent Architecture. www.ai.sri.com/oaa.
67. E. Charniak and D. McDermott, *Introduction to Artificial Intelligence*. Addison-Wesley, Reading, MA, 1987.

## To Probe Further

Pen computing: http://hwr.nici.kun.nl/pen-computing/ A Web site hosted at the Nijmegen University with links related to practical issues in pen and mobile computing.

Handhelds: http://handhelds.org/ A Compaq-hosted Web site created to encourage and facilitate the creation of open source software for use on handheld and wearable computers.

Pen computing consumer products: http://www.pencomputing.com/ A commercial (paper) magazine and Web site devoted to news and reviews on pen and mobile computing products.

## 26.7 What Makes a Programmable DSP Processor Special?

*Ingrid Verbauwhede*

### 26.7.1 Introduction

A programmable DSP processor is a processor "tuned" towards its application domain. Its architecture is very different from a general-purpose von Neumann architecture to accommodate the demands of real-time signal processing. When first developed in the beginning of the 1980s, the main application was filtering. Since then, the architectures have evolved together with the applications. Currently, the most stringent demands for low-power embedded DSP processors come from wireless communication applications: second, 2.5, and third generation (2G, 2.5G, and 3G) cellular standards. The demand for higher throughput and higher quality of source and channel coding keeps growing while power consumption has to be kept as low as possible to increase the lifetime of the batteries.

In this section, first the application domain and its historical evolution will be described in Section 26.7.2. Then, in Section 26.7.3, the overall architecture will be described. In Section 26.7.4, the specifics of the DSP data paths will be given. In Section 26.7.5, the memory architecture and its associated address generation units are described. In Section 26.7.6, the specifics of the DSP pipeline will be explained. Finally, in Section 26.7.7, the conclusions will be given followed by some future trends.

### 26.7.2 DSP Application Domain

DSP processors were originally developed to implement traditional signal processing functions, mainly filters, such as FIRs and IIRs [5]. These applications decided the main properties of the programmable DSP architecture: the inclusion of a multiply-accumulate unit (MAC) as separate data path unit and a Harvard or modified Harvard memory architecture instead of a von Neumann architecture.

#### 26.7.2.1 Original Motivation: FIR Filtering

The fundamental properties of these applications were (and still are):

- *Throughput driven calculations and real-time operation.* Signal processing applications, such as speech and video, can be represented as an "infinite stream" of data samples that need to be processed at a rate determined by the application [20]. The *sample rate* is a fundamental property of the application. It determines at what rate the consecutive samples arrive for processing. For speech processing, this is the rate of speech samples (kHz range), for video processing this might be the frame rate or the pixel rate (MHz range) [3]. The DSP has to process these samples at this given rate. Therefore, a DSP operates under worst-case conditions. This is fundamentally different from general-purpose processing on a micro processor, which operates on an average case base, but which has an unpredictable worst-case behavior.
- *Large amounts of computations, few amounts of control flow operations.* DSP processors were developed to process large amounts of data in a very repetitive mode. For instance, speech filtering, speech coding, pixel processing, etc., require similar operations on consecutive samples, pixels, frames, etc. The DSP processor has adapted to this, by providing means of implementing these algorithms in a very efficient way. It includes zero-overhead looping, very compact instruction code, efficient parallel memory banks, and associated address generation units.

- *Large amount of data*, usually organized in a regular or almost regular pattern. Because of the real-time processing and the associated "infinite" amount of data that is processed, DSP processors usually have several parallel memory banks; each bank has its own address generation unit and parallel reads and writes are supported by the DSP.
- *Embedded applications.* DSP processors are developed for embedded applications, ranging from cellular phones, disk drives, cable modems, etc. The result is that all the program codes have to reside on the processor (no external memory, no cache hierarchy). Thus, the code size has to be minimized, as a result of which, till today there is a lot of assembly code written. Secondly, the power consumption has to be minimized since many of these applications run from batteries or have tight cooling requirements such as the usage of cheap plastic packages or enclosed boxes.

### 26.7.2.2 Modern Applications: Digital Wireless Communications

New applications drive the design of new DSP processors. State-of-the-art DSP processors will have more than one MAC, acceleration for Viterbi decoding, specialized instructions for Turbo decoding, and so on. Indeed, DSP processors have become the main workhorse for wireless communications for both the handsets and the base station infrastructure [22].

Second generation (2G) cellular standards required the introduction of optimized instructions for speech processing and for communication algorithms used in the channel coding and modulation/demodulation. The fundamental components of a wireless system are shown on Fig. 26.51.

#### *26.7.2.2.1 Speech Coding*

The source coder/decoder in 2G cellular standards (GSM, IS-136, IS-95 CDMA, Japanese PDC) is mainly a speech coder/decoder. The main function of a speech coder is to remove the redundancy and compress the speech signal and hence, reduce the bandwidth requirements for storage or transmission over the air. The required reduction in bit rate is illustrated in Fig. 26.52 for the Japanese PDC standard.

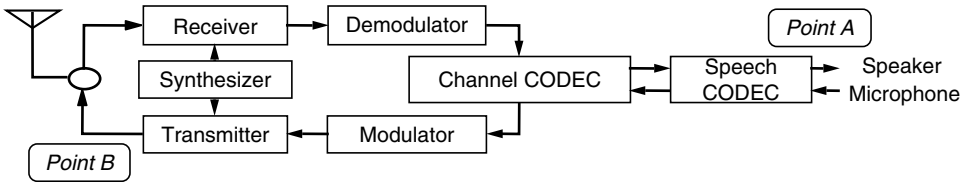



**FIGURE 26.51** Fundamental building blocks in a communication system.

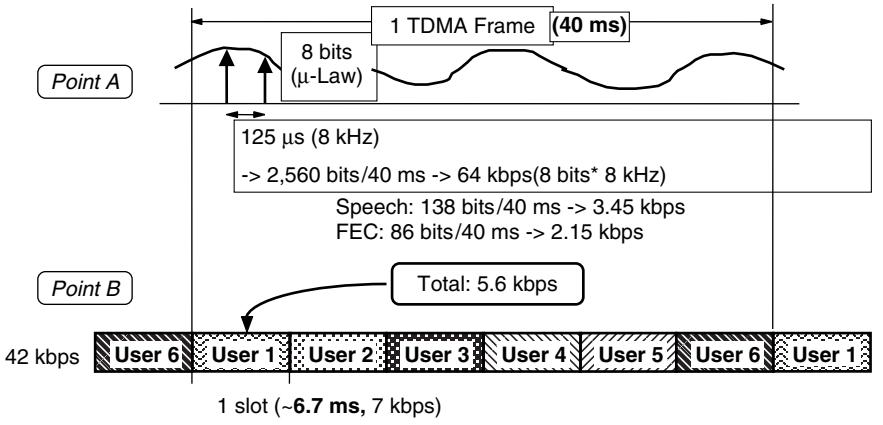



**FIGURE 26.52** Relationship between speech signal and the transmitted signal.

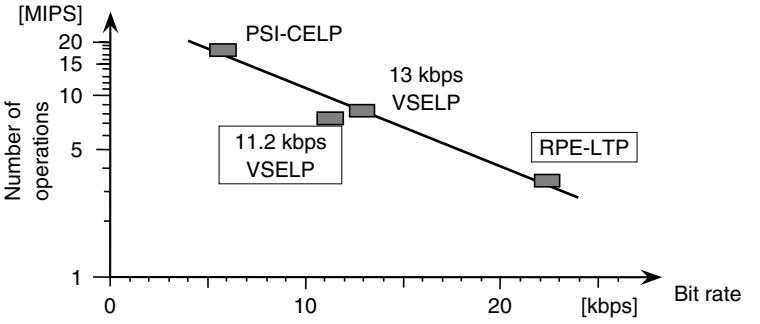



**FIGURE 26.53** MIPS requirement of several speech coders.

At point A, a "toll quality" digital speech signal requires the sampling of the analog speech waveform at 8 kHz. Each sample requires 8 bits of storage ($\mu$-law compressed) thus resulting in a bit rate of 64 kbits/s or 2560 bits for one 40 ms TDMA frame. This speech signal needs to be compressed to increase the capacity of the channel. One TDMA frame, which has a basic time period of 40 ms, is shared by six users. The bit rate at point B is 42 kbits/s. Thus, one user slot gets only 7 kbits/s. The 2560 bits have to be reduced to 138 bits, to which 86 bits are added for forward error correction (FEC), resulting in a total of 5.6 kbits/s.

The higher the compression ratio and the higher the quality of the speech coder, the more calculations, usually expressed in MIPS, are required. This is illustrated in Fig. 26.53. The first generation GSM digital cellular standard employs the Regular Pulse Excitation-Long Term Prediction (RPE-LTP) algorithm and requires a few thousand MIPS to implement it on a current generation DSP processor. For instance, it requires 2000 MIPS on the lode processor [21]. The Japanese half-rate coder Pitch Synchronous Innovation-Code Excited Linear Prediction (PSI-CELP) requires at least ten times more MIPS.

#### *26.7.2.2.2 Viterbi Decoding*

The function of the channel codec is to add controlled redundancy to the bit stream on the encoder side and to decode, detect, and correct transmission errors on the receiver side. Thus, channel encoding and decoding is a form of error control coding. The most common decoding method for convolutional codes is the Viterbi algorithm [4]. It is a dynamic programming technique as it tries to emulate the encoder's behavior in creating the transmitted bit sequence. By comparing to the received bit sequence, the algorithm determines the difference between each possible path through the encoder and the received bit sequence. The decoder outputs the bit sequence that has the smallest deviation, called the minimum distance, compared to the received bit sequence.

Most practical convolutional encoders are rate $1/n$, meaning that one input bit generates $n$ coded output bits. A convolutional encoder of constraint length $K$ can be represented by a finite state machine (FSM) with $K-1$ memory bits. The FSM has $2^{K-1}$ possible states, also called trellis states. If the input is binary, two next states are possible starting from the current state and the input bit. The task of the Viterbi algorithm is to reconstruct the most likely sequence of state transitions based on the received bit sequence. This approach is called the "most likelihood sequence estimation." These state transistions are represented by a trellis diagram. The kernel of the trellis diagram is the Viterbi butterfly as shown in Fig. 26.54b.

### 26.7.2.3 Next Generation Applications

Current generation DSP processors are shaped by 2G cellular standards, the main purpose of which is voice communication. 3G cellular standards will introduce new features: increased focus on data communication, e-mail, web browsing, banking, navigation, and so on.

2G standards can support short messages, such as the popular SMS messages in the GSM standard, but are limited to about 10 to 15 kbits/s. In the 2.5G cellular standards, provisions are made to support higher data rates. By combining GSM time slots, generalized packet radio services (GPRS) can support up to 115 kbits/s. But the 3G standards are being developed specifically for data services.

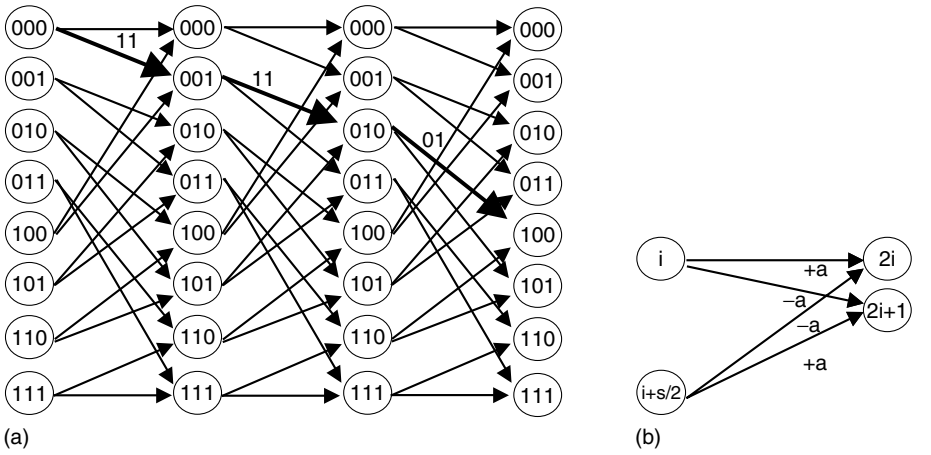



**FIGURE 26.54** Viterbi trellis diagram and one butterfly.

Wideband CDMA (WCDMA) will support up to 2 Mbits/s in office environments, lowered to 144 kbits/s for high mobile situations [6,13].

The increased focus on data services has large consequences for the channel codec design. The traditional Viterbi decoder does not provide a low enough bit error rate to support data services. Therefore, turbo codes are considered [2]. Turbo decoding (shown in Fig. 26.55) is a collaborative structure of soft-input/soft-output (SISO) decoders with the inclusion of interleaver memories between decoders to scatter burst errors [2]. Either soft-output viterbi algorithm (SOVA) [7] or maximum a posteriori (MAP) [1] can be used as SISO decoders. Within a turbo decoder, the two decoders can operate on the same or different codes. Turbo codes have been shown to provide coding performance to within 0.7 dB of the Shannon limit (after a number of iterations).

The log MAP algorithm can be implemented in a manner very similar to the standard Viterbi algorithm. The most important difference between the algorithms is the use of a correction factor on the "new path metric" value (the alpha, beta, and log-likelihood ratio values in Log MAP). This correction factor depends on the difference between the values being compared in the add-compare-select butterfly (as shown in Fig. 26.54). This means that the Viterbi acceleration units, that implement this add-compare-select operation, need to be modified. Turbo coding is one member of a large class of iterative decoding algorithms. Recently low density parity check codes (LDPC) that have gained renewed attention as another important class, which are potentially more easily translated to efficient implementations.

Other trends seem to place an even larger burden on the DSP processor. The Japanese i-Mode system includes e-mail, web browsing, banking, location finding in combination with the car navigation system,

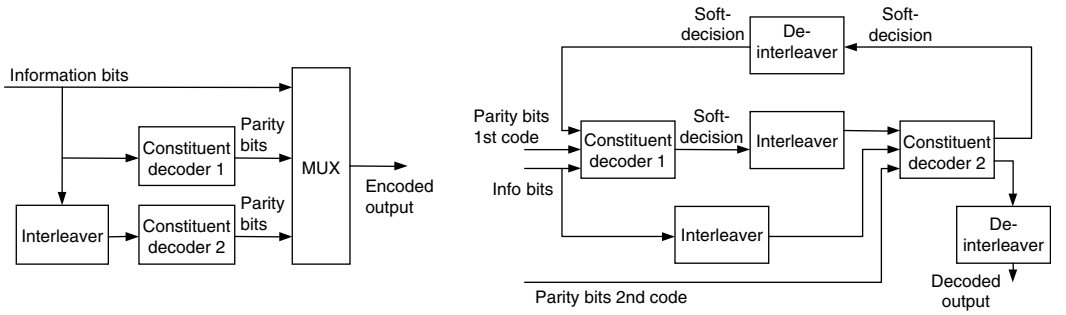



**FIGURE 26.55** Turbo encoder and decoder.

etc. Next generation phones will need to support video and image processing, and so on. Applications and upgrades will be downloadable from the Internet.

But at the same time, consumers are used to longer talk times (a couple of hours) and very long standby times (days or weeks). Thus, they will not accept a reduction of talk time nor standby time in exchange for more features. This means that these increased services have to be delivered with the same power budget because the battery size is not expected to grow nor is the battery technology expected to improve substantially.

### 26.7.3 DSP Architecture

The fundamental property of a DSP processor is that it uses a Harvard or modified Harvard architecture instead of a von Neumann architecture. This difference is illustrated in Fig. 26.56.

A von Neumann architecture has one unified address space, i.e., data and program, are assigned to the same memory space. In a Harvard architecture, the data memory map is split from the program memory map. This means that the address busses and data busses are doubled. Together with specialized address calculation units, this will increase the memory bandwidth available for signal processing applications. This concept will be illustrated by the implementation of a simple FIR filter. The basic equation for an $N$ tap FIR equation is the following:

$$y(n) = \sum_{i=0}^{i=N-1} c(i) \cdot x(n-i)$$

Expansion of this equation results in the following pseudo code statements:

```
y(0) = c(0)x(0) + c(1)x(–1) + c(2)x(–2) + ... + c(N – 1)x(1 – N);
y(1) = c(0)x(1) + c(1)x(0) + c(2)x( – 1) + ... + c(N – 1)x(2 – N);
y(2) = c(0)x(2) + c(1)x(1) + c(2)x(0) + ... + c(N – 1)x(3 – N);
  ...
y(n) = c(0)x(n) + c(1)x(n – 1) + c(2)x(n – 2) + ... + c(N – 1)x(n – (N – 1));
```

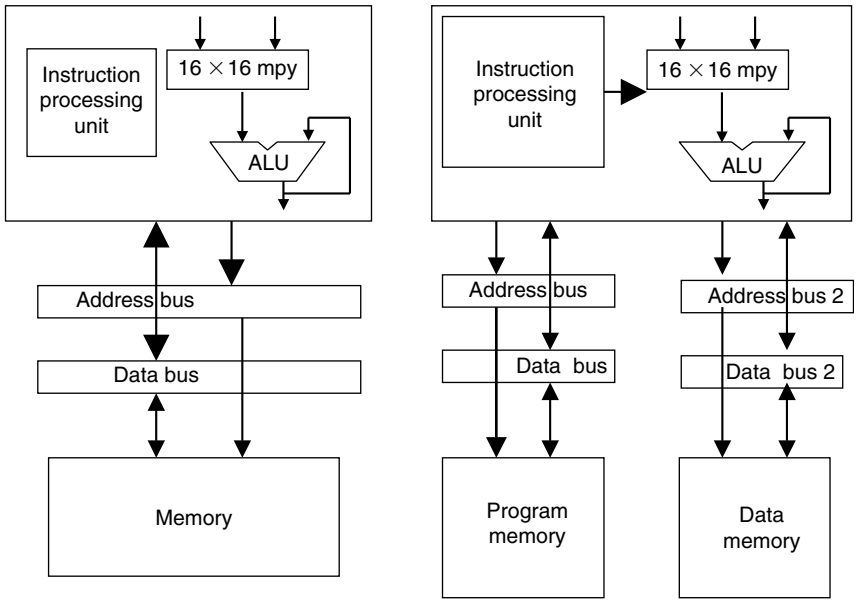



**FIGURE 26.56** von Neumann architecture and Harvard/modified Harvard architecture.

When this equation is executed in software or assembly code, output samples $y(n)$ are computed in sequence. To implement this on a von Neumann architecture, the following operations are needed. Assume that the von Neumann has a multiply and accumulate instruction (not necessarily the case). Assume also that pipelining allows to execute the multiply and accumulate in parallel with the read or write operations. Then one tap needs four cycles:

1. Read multiply-accumulate instruction.
2. Read data value from memory.
3. Read coefficient from memory.
4. Write data value to the next location in the delay line (because to start the computation of the next output sample, all values are shifted by one location).

Thus even if the von Neumann architecture includes a single cycle multiply-accumulate unit, it will take four cycles to compute one tap.

Implementing the same FIR filter on a Harvard architecture will reduce the number of cycles to three because it allows the fetch of the instruction in parallel with the fetch of one of the data items. This was a fundamental property that distinguished the early DSP processors. On the TMS 320Clx, released in the early '80s, it took $2N$ cycles for a $N$ tap filter (without the shift of the delay line) [5].

The modified Harvard architecture improves this idea even further. It is combined with a "repeat" instruction and a specialized addressing mode, the circular addressing mode. In this case, one multiply-accumulate instruction is fetched from program memory and kept in the one instruction deep instruction "cache." Then the data access cycles are performed in parallel: the coefficient is fetched from the program memory in parallel with the data sample being fetched from data memory. This architecture is found in all early DSP processors and is the foundation for all following DSP architectures. The number of memory accesses for one tap are reduced to two and these occur in the same cycle. Thus, one tap can execute in one cycle and the multiply-accumulate unit is kept occupied every cycle.

Newer generation of DSP processors have even more memory banks, accompanying address generation units and control hardware, such as the repeat instruction, to support multiple parallel accesses. The execution of a 32-tap FIR filter on the dual Mac architecture of the Lucent DSP 1621, shown in Fig. 26.57, will take only 19 cycles. The corresponding pseudo code is the following:

```
do 14 { //one instruction !
  a0 = a0+p0+p1
  p0 = xh*yh p1 = x1*y1
  y = *r0++ x = *pt0++
}
```

This code can be executed in 19 clock cycles with only 38 bytes of instruction code. The inner loop takes one cycle to execute and as can be seen from the assembly code, seven operations are executed in

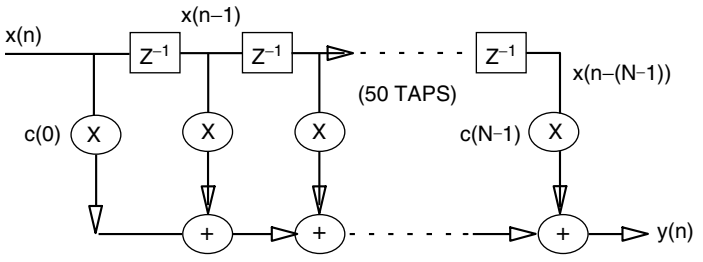



**FIGURE 26.57** Finite impulse response filter.

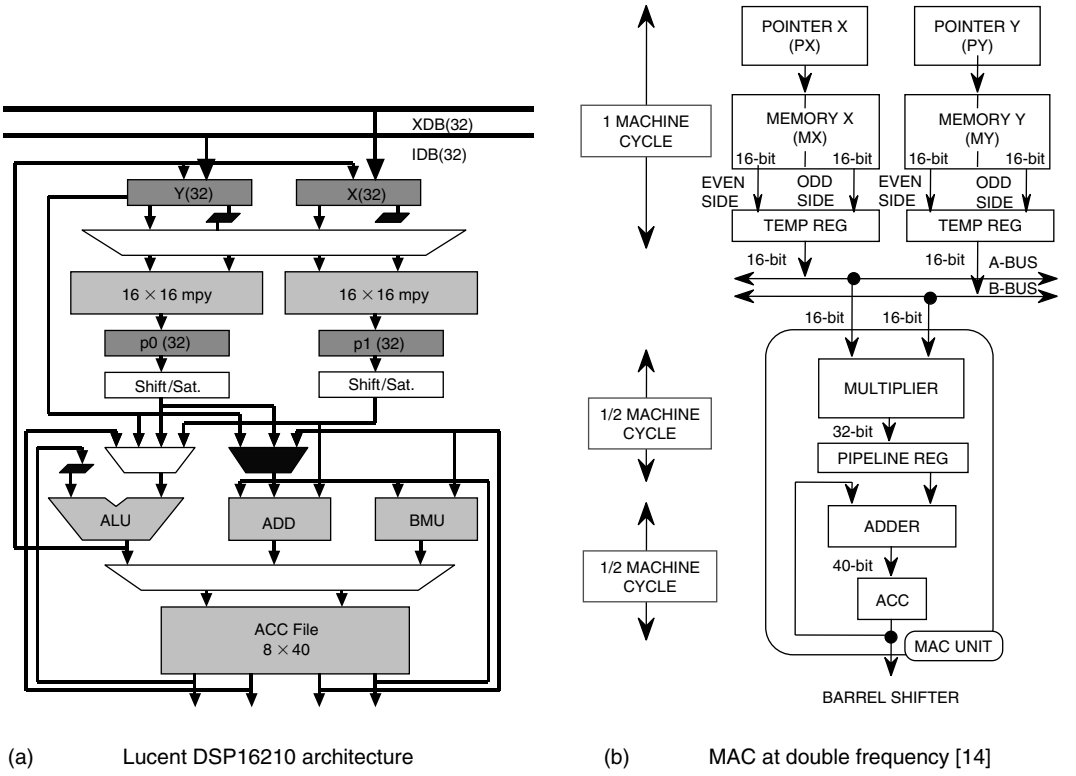



**FIGURE 26.58** DSP architectures for 0.5 cycle per FIR tap.

parallel: one addition, two multiplications, two memory reads, and two address pointer updates. Note that the second pointer update, *pt0++, updates a circular address pointer.

Two architectures which speed up the FIR calculation to 0.5 cycle per tap are shown in Fig. 26.58. The first one is the above mentioned Lucent DSP16210. The second one is an architecture presented in [9]. It has a multiply accumulate unit that operates at double the frequency from the memory accesses.

The difficult part in the implementation of this tight loop is the arrangement of the data samples in memory. To supply the parallel Mac data paths, two 32-bit data items are read from memory and stored in the *X* and *Y* register, as shown in Fig. 26.58. A similar split in lower and higher halfs occurs in the Intel/ADI Frio core [10]. Then the data items are split in an upper half and a low half and supplied to the two $16 \times 16$ multipliers in parallel or the left half and the right half of the TEMP registers in Fig. 26.58(b). It requires a correct alignment of the data samples in memory, which is usually a tedious work done by the programmer, since compilers are not able to handle this efficiently. Note that a similar problem exists when executing SIMD instructions on general purpose micro-processors.

Memory accesses are a major energy drain. By rearranging the operations to compute the filter outputs, the amount of memory accesses can be reduced. Instead of working on one output sample at a time, two or more output samples are computed in parallel. This is illustrated in the pseudo code below:

$$
\begin{aligned}
&\mathtt{y(0)} = \mathtt{c(0)x(0)} + \mathtt{c(1)x(-1)} + \mathtt{c(2)x(-2)} + \ldots + \mathtt{c(N-1)x(1-N)};\\
&\mathtt{y(1)} = \mathtt{c(0)x(1)} + \mathtt{c(1)x(0)} + \mathtt{c(2)x(-1)} + \ldots + \mathtt{c(N-1)x(2-N)};\\
&\mathtt{y(2)} = \mathtt{c(0)x(2)} + \mathtt{c(1)x(1)} + \mathtt{c(2)x(0)} + \ldots + \mathtt{c(N-1)x(3-N)};\\
&\ldots\\
&\mathtt{y(n)} = \mathtt{c(0)x(n)} + \mathtt{c(1)x(n-1)} + \mathtt{c(2)x(n-2)} + \ldots + \mathtt{c(N-1)x(n-(N-1))};
\end{aligned}
$$

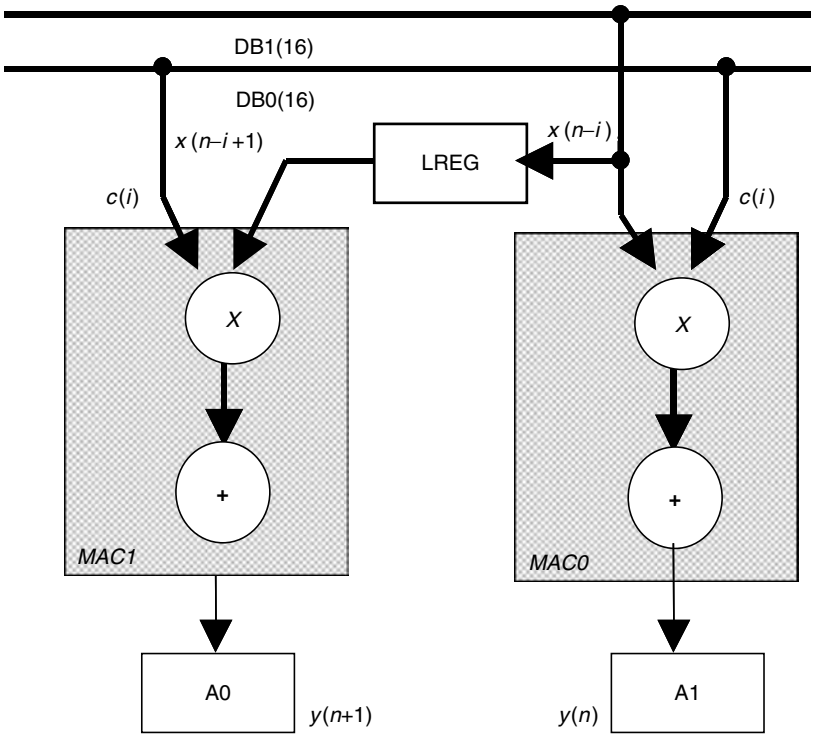



**FIGURE 26.59** Dual Mac architecture with delay register of the Lode DSP core.

In the lode architecture [21] a delay register is introduced between the two Mac units as shown in Fig. 26.59. This halves the amount of memory accesses. Two output samples are calculated in parallel as indicated in the pseudo code of Table 26.3. One data bus will read the coefficients, $c(i)$, the other data bus will read the data samples, $x(N-i)$, from memory. The first Mac will compute a multiply-accumulate for output sample $y(n)$. The second multiply-accumulate will compute in parallel on $y(n+1)$. It will use a delayed value of the input sample. In this way, two output samples are computed at the same time.

This concept of inserting a delay register can be generalized. When the datapath has $P$ Mac units, $P-1$ delay registers can be inserted and only $2N/(P+1)$ memory accesses are needed for one output sample. These delay registers are pipeline registers and hence if more delay registers are used, more initialization and termination cycles need to be introduced.

The idea of working on two output samples at one time is also present in the dual Mac processor of TI, the TIC55x. This processor has a dual Mac architecture with three 16-bit data busses. To supply both Macs with coefficient and data samples, the same principle of computing two output samples at the same time is used. One data bus will carry the coefficient and supply this to both Macs, the other two data busses will carry two different data samples and supply this to the two different Macs.

A summary of the different approaches is given in Table 26.2. Note that most energy savings are obtained from reducing the amount of memory accesses and secondly, from reducing the number of instruction cycles. Indeed the energy associated with the Mac operations can be considered as "fundamental" energy without it, no $N$ tap FIR filter can be implemented.

Modern processors have multiple address busses, multiple data busses and multiple memory banks, including both single and dual port memory. They also include mechanisms to assign parts of the physical memory to either memory space, program, or data. For instance for the C542 processor the on-chip dual access RAM can be assigned to the data space or to the data/program space, by setting a specific control bit (the OVLY bit) in a specific control register (the PMST register) [19].

**TABLE 26.2** Data Accesses, Mac Operations, Instruction Cycles, and Instructions for an $N$ Tap FIR Filter

| DSP | Data Memory Accesses | MAC Operations | Instruction Cycles | Instructions |
|---|---|---|---|---|
| von Neumann | $3N$ | $N$ | $4N$ | $2N$ |
| Harvard | $3N$ | $N$ | $3N$ | $3N$ |
| Modified Harvard with modulo arithmetic | $2N$ | $N$ | $N$ | 2 (repeat instruction) |
| Dual Mac or double frequency Mac | $2N$ | $N$ | $N/2$ | 2 (same) |
| Dual Mac with 3 data busses | $1.5N$ | $N$ | $N/2$ | 2 |
| Dual Mac with 1 delay registers | $N$ | $N$ | $N/2$ | 2 |
| Dual Mac with $P$ delay registers | $2N/(P+1)$ | $N$ | $N/(P+1)$ | 2 |

## 26.7.4 DSP Data Paths

The focus of the previous section was on the overall architecture of a DSP processor and its fundamental properties to increase the memory bandwidth. This will keep the data paths of the DSP operating every clock cycle. In this section, some essential properties of the DSP data paths will be described.

### 26.7.4.1 Multiply-Accumulate Unit

The unit that is most associated with the DSP is the Mac. It is shown in Fig. 26.60. The most important properties of the Mac unit are summarized below:

- The multiplier takes two 16-bit inputs and produces a 32-bit multiplier output. Internally the multiplication might be implemented as a $17 \times 17$ bit multiplier. This way the multiplier can implement both two's complement and unsigned numbers.
- The product register is a pipelined register to speed up the calculation of the multiply-accumulate operation. As a result the Mac operation executes in most processors in one cycle effectively although the latency can be two cycles.
- The accumulator registers are usually 40 bits long. Eight bits are designated as "guard" bits [11]. This allows the accumulation of $2^8$ products before there is a need of scaling, truncation, or saturation. These larger word lengths are very effective in implementing DSP functions such as filters. The disadvantage is that special registers such as these accumulators are very hard to handle by a compiler.

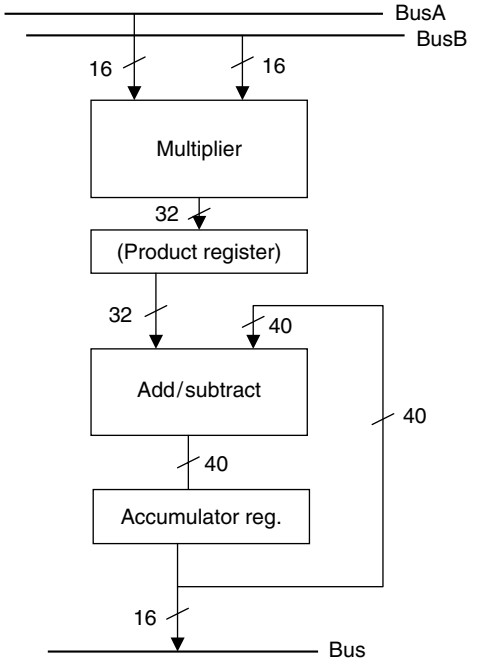



**FIGURE 26.60** Multiply accumulate unit.

### 26.7.4.2 Viterbi Acceleration Unit

Convolutional decoding and more specifically the Viterbi algorithm, has been recognized as one of the main, if not the most, MIPS consuming application in current and next generation standards. The key issue is to reduce the number of memory

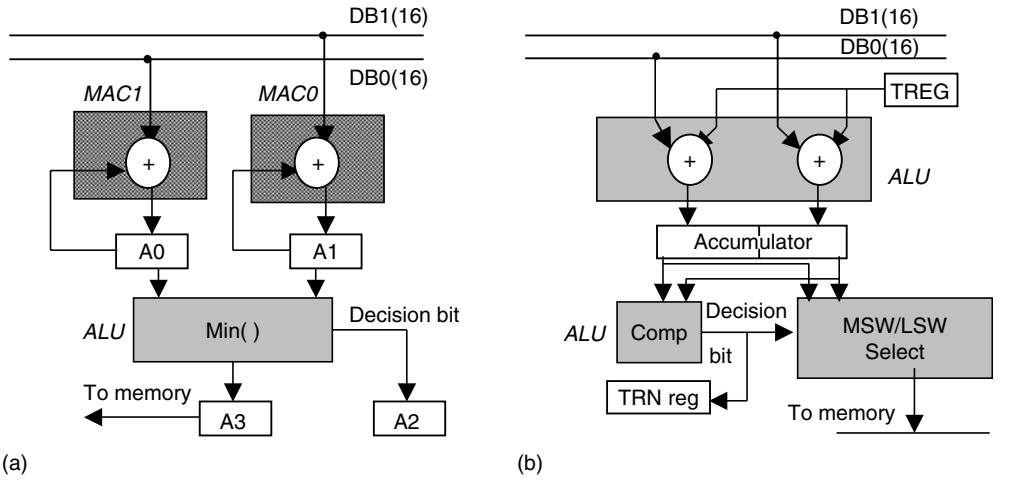



**FIGURE 26.61** Two data path variations to implement the add-compare-select operation.

accesses and secondly the number of operations to implement the algorithm. The kernel of the algorithm is the Viterbi butterfly as shown on Fig. 26.54. The basic equations executed in this butterfly are:

$$d(2i) = \min\{d(i) + a, d(i + s/2) - a\}$$
$$d(2i+1) = \min\{d(i) - a, d(i + s/2) + a\}$$

These equations are implemented by the "add-compare-select (ACS)" instruction and its associated data path unit. Indeed, one needs to add or subtract the branch metric from states $i$ and $i + s/2$, compare them, and select the minimum. In parallel, state $2i + 1$ is updated. The butterfly arrangement is chosen because it reduces the amount of memory accesses by half, because the two states that use the same data to update the same two next states are combined.

DSP processors have special hardware and instructions to implement the ACS operation in the most efficient way. The lode architecture [21] uses the two Mac units and the ALU to implement the ACS operation as shown in Fig. 26.61a. The dual Mac operates as a dual add/subtract unit. The ALU finds the minimum. The shortest distance is saved to memory and the path indicator, i.e., the decision bit is saved in a special shift register A2. This results in four cycles per butterfly.

The Texas Instruments TMS320C54x and the Matsushita processor described in [14,22] use a different approach that also results in four cycles per butterfly. This is illustrated in Fig. 26.61b. The ALU and the accumulator are split into two halves (much like SIMD instructions), and the two halves operate independently. A special compare, select, and store unit (CSSU) will compare the two halves, will select the chosen one, and write the decision bit into a special register TRN. The processor described in [14] describes two ACS units in parallel. One should note that without these specialized instructions and hardware, one butterfly requires 15 to 25 or more instructions.

## 26.7.5 DSP Memory and Address Calculation Units

Besides the data paths optimized for signal processing and communication applications, the DSP processors also have specialized address calculation units. As explained in Section 26.7.3, the parallel memory maps in the Harvard or modified Harvard architecture are essential for the data processing in DSP processors; however, to avoid overload on the regular data path units, specialized address generation units are included. In general, the number of address generation units will be same as the

**TABLE 26.3** Number of Parallel Address Generation Units for a Few DSP Processors

| | Generation Units | |
|---|---|---|
| Processor | Data Address | Program Address |
| C5× [18] | 1 (ARAU) | 1 |
| C54× [19] | 2 (DAGEN has two units: ARAU0, ARAU1) | 1 |
| Lode [21] | 2 (ACU0, ACU1) | 1 |
| Frio [10] | 2 | 1 |

maximum number of parallel memory accesses that can occur in one cycle. A few examples are shown in Table 26.3. Older processors, such as the C5× with a modified Harvard architecture, have one address generation unit serving the data address bus, and one program address generation unit serving the program address bus. When the number of address busses go up, so will the arithmetic units inside the address calculation unit. For instance the Frio [10] has two address busses served by two ALUs inside the data address generation unit.

The address generation units themselves are optimized to perform address arithmetic in an efficient way. This includes data paths with the correct word lengths. It also includes all the typical address modifications that are common in DSP applications. For instance indirect addressing with a simple increment can easily be done and expressed in the instruction syntax. More advanced addressing modes include circular buffering, which especially suits filter operations, and bit-reversed addressing, especially useful for fast Fourier transforms, and so on. There exist many good instruction manuals that describe the detailed operation of these specialized addressing modes, [11,18,19].

## 26.7.6 DSP Pipeline

The pipeline of a DSP processor is different from the pipeline of a general purpose control-oriented micro-processors. The basic slots of the DSP pipeline and the RISC pipeline are shown in Fig. 26.62. In a DSP processor, the memory access stage in parallel with the address generation (usually "post-modification") occurs before the execute stage. An example is described in [10]. In a RISC processor the memory access stage follows the execute stage [8], because the execute stage is used to calculate

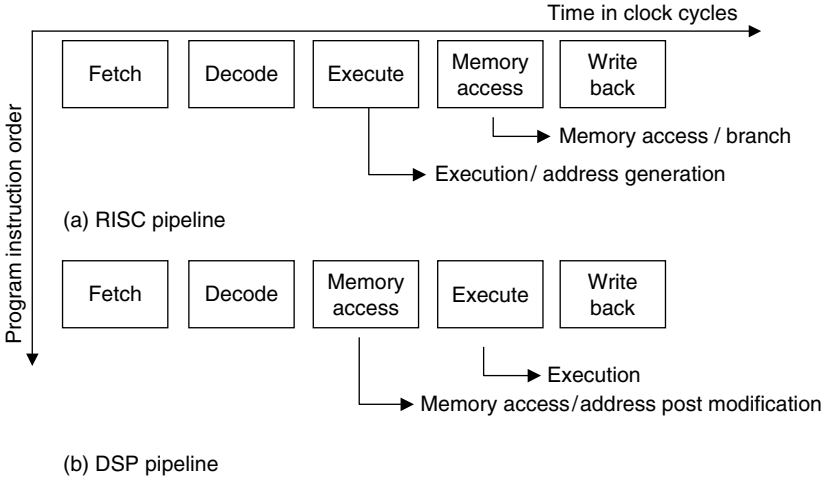



**FIGURE 26.62** Basic pipeline architecture for a RISC and a DSP processor.

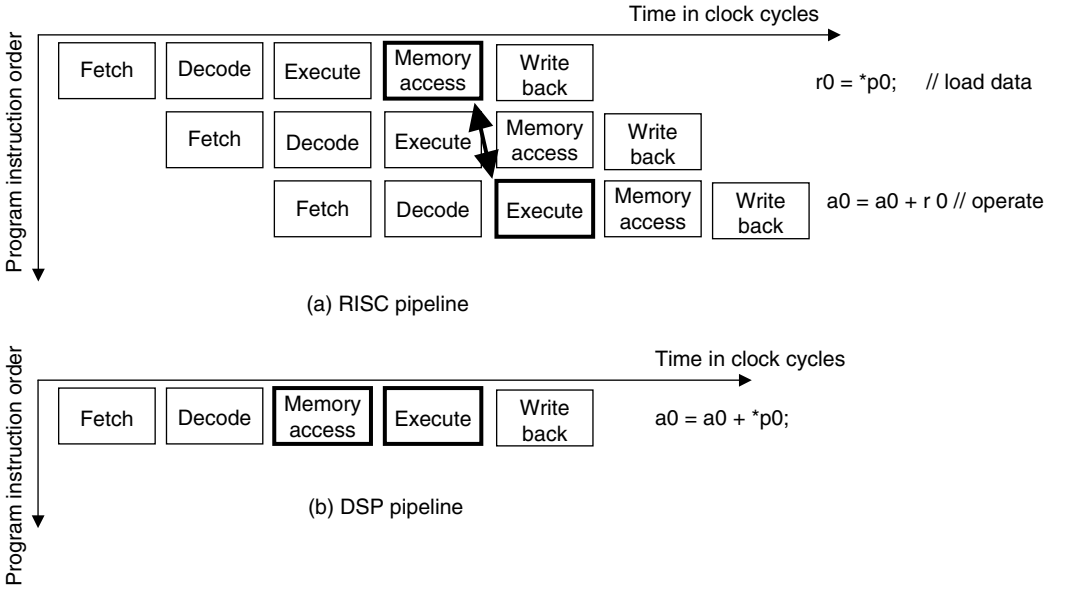



**FIGURE 26.63** Memory-intensive number crunching on a RISC and a DSP.

the address on the main ALU. The fundamental reason for this difference in pipeline structure is that DSP processors are optimized for memory intensive number-crunching type of applications (e.g., FIRs), while RISC type processors, including micro-controllers and micro-processors, are optimized for complex decision making. This is explained in Figs. 26.63 and 26.64. Typical for real-time compute intensive applications, is the continuous memory accesses followed by operations in the data path units. A typical example is the execution of the FIR filter as shown in the FIR pseudo code above. On a DSP processor, the memory access and the multiply-accumulate operation are specified in one instruction and follow each other in the pipeline stage. The same operation on a RISC machine will need three instruction slots. The first instruction slot will read the value from memory and only in the third instruction slot the actual computation takes place. If these delays are not obeyed, a data hazard will occur [8]. Fixing data hazards will lead to a large instruction and cycle overhead.

Similarly, it can be argued that branches have a larger penalty on DSP processors than on RISC machines. The reason is explained on Fig. 26.64. If a data dependent branch needs to be executed, e.g., "branch if accumulator is zero," then it takes that this instruction cannot follow immediately after the accumulator is set. In the simple examples of Fig. 26.64, there needs to be two, respectively three instruction cycles between the setting of the accumulator flag and the usage of it in the decode stage, by the RISC and DSP processor, respectively. Therefore, the RISC has an advantage for control dominated applications. In practice these pipeline hazards are either hidden to the programmer by hardware solutions (e.g., forwarding or stalls) or they are visible to the programmer, who can optimize his code around it. A typical example are the branch and the "delayed branch" instruction in DSP processor. Because an instruction is fetched in the cycle before it is decoded, a regular branch instruction will incur an unnecessary fetch of the next instruction in memory following the branch. To optimize the code in DSP processors, the delayed branch instruction is introduced. In this case, the instruction that follows the branch instruction in memory will be executed before the actual branch takes place. Hence, a delayed branch instruction takes effectively one cycle to execute while a regular branch will take two cycles to execute.

The delayed branch is a typical example on how DSP programmers are very sensitive to code size and code execution. Indeed, for embedded applications, minimum code size is a requirement.

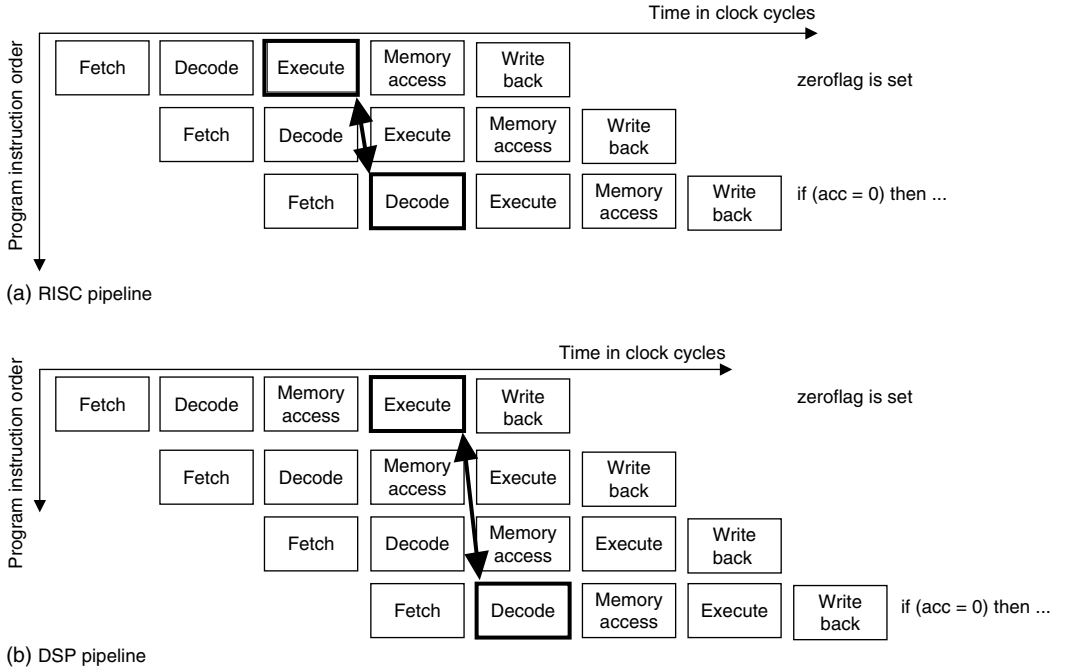



**FIGURE 26.64** Decision making (branch) on a RISC and a DSP.

## 26.7.7 Conclusions and Future Trends

DSP processors are a special type of processors, very different from the general purpose micro controller or micro processor architectures. As argued in this section, this is visible in all components of the processor: the overall architecture, the data paths, the address generation units, and even the pipeline structure.

The applications of the future will keep on driving the next generation of DSP processors. Several trends are visible. Clearly, there is a need to support higher level languages and compilers. Traditional compiler techniques do not produce efficient, optimized code for DSP processors. Compiler technology has only recently started to address the needs of low power embedded applications. But also the architectures will need changes to accommodate the compiler techniques. One drastic approach is the appearance of VLIW architectures, for which efficient compiler techniques are known. This results, however, in code size explosion associated with a large increase in power consumption. A hybrid approach might be a better solution. For example, the processor described in [10] has unified register files. Yet, it also makes an exception for the accumulators.

Another challenge is the increased demand for performance while reducing the power consumption. Next generation wireless portable applications will not only provide voice communication, but also video, text and data, games, and so on. On top of this the applications will change and will need reconfigurations while in use. This will require a power efficient way of runtime reconfiguration [16]. The systems on a chip that implement these wireless communication devices will include a large set of heterogeneous programmable and reconfigurable modules, each optimized to the application running on them. Several of these will be DSP processors and they will be crucial for the overall performance.

## Acknowledgments

The author thanks Mr. Katsuhiko Ueda of Matsushita Electric Co., Japan, for the interesting discussions and for providing several figures in this section.

## References


1. Bahl L., Cocke J., Jelinek F., Raviv J., "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Information Theory,* vol. IT-20, pp. 284–287, March 1974.
2. Berrou C., Glavieux A., Thitimajshima P., "Near shannon limit error-correcting coding and decoding: turbo-codes (1)," *Proc. ICC '93,* May 1993.
3. Catthoor F., De Man H., "Application-specific architectural methodologies for high-throughput digital signal and image processing," *IEEE Transactions on ASSP,* Feb. 1990.
4. Forney G., "The viterbi algorithm," *Proceedings of the IEEE,* vol. 61, no. 3, pp. 268–278, March 1973.
5. Gass W., Bartley D., "Programmable DSPs," Chapter 9 in *Digital Signal Processing for Multimedia Systems,* Parhi K., Nishitani T. (Eds.), Marcel Dekker Inc., New York, 1999.
6. Gatherer A., Stetzler T., McMahan M., Auslander E., "DSP-based architectures for mobile communications: past, present, future," *IEEE Communications Magazine,* pp. 84–90, Jan. 2000.
7. Hagenauer J., Hoeher P., "A viterbi algorithm with soft-decision outputs and its applications," *Proc. Globecom '89,* pp. 47.1.1–47.1.7, Nov. 1989.
8. Hennessy J., Patterson D., *Computer Architecture: A Quantitative Approach,* 2nd Edition, Morgan Kaufmann Publ., San Francisco, CA, 1996.
9. Kabuo H., Okamoto M., et al. "An 80 MOPS peak high speed and low power consumption 16-bit digital signal processor," *IEEE Journal of Solid-State Circuits,* vol. 31, no. 4, pp. 494–503, 1996.
10. Kolagotla R., et al., "A 333 MHz dual-MAC DSP architecture for next-generation wireless applications," *Proceedings ICASSP,* Salt Lake City, UT, May 2001.
11. Lapsley P., Bier J., Shoham A., Lee E.A., *DSP Processor Fundamentals: Architectures and Features,* IEEE Press, 1996.
12. Lee E.A., "Programmable DSP architectures: Part I and Part II," *IEEE ASSP Magazine,* pp. 4–19, Oct. 1988, pp. 4–14, Jan. 1989.
13. McMahan M.L., "Evolving cellular handset architectures but a continuing, insatiable desire for DSP MIPS," *Texas Instruments Technical Journal,* Jan.–Mar. 2000, vol. 17, no. 1, reprinted as Application Report SPRA650-March 2000.
14. Okamoto M., Stone K., et al., "A high performance DSP architecture for next generation mobile phone systems," *1998 IEEE DSP Workshop.*
15. Oliphant M., "The mobile phone meets the internet," *IEEE Spectrum,* pp. 20–28, Aug. 1999.
16. Schaumont P., Verbauwhede I., Keutzer K., Sarrafzadeh M., "A quick safari through the reconfiguration jungle," *Proceedings 38th Design Automation Conference,* Las Vegas, NV, June 2001.
17. Strauss W., "Digital signal processing, the new semiconductor industry technology driver," *IEEE Signal Processing Magazine,* pp. 52–56, March 2000.
18. Texas Instruments, *TMS320C5x* User's Guide, document SPRU056B, Jan. 1993.
19. Texas Instruments, *TMS320C54x DSP CPU Reference Guide,* document SPRU131G, March 2001.
20. Verbauwhede I., Scheers C., Rabaey J., "Analysis of multidimensional DSP specifications," *IEEE Transactions on signal processing,* vol. 44, no. 12, pp. 3169–3174, Dec. 1996.
21. Verbauwhede I., Touriguian M., "Wireless digital signal processing," Chapter 11 in *Digital Signal Processing for Multimedia Systems,* Parhi K., Nishitani T. (Eds.), Marcel Dekker Inc., New York, 1999.
22. Verbauwhede I., Nicol C., "Low power DSP's for wireless communications," *Proceeding ISLPED,* pp. 303–310, Aug. 2000.

# 27

# Data Security




Matthew Franklin
*University of California at Davis*


## 27.1 Introduction

Cryptography is the science of data security. This chapter gives a brief survey of cryptographic practice and research. The chapter is organized along the lines of the principal categories of cryptographic primitives: unkeyed, symmetric key, and asymmetric key. For each of these categories, this chapter defines the important primitives, gives security models and attacks scenarios, discusses constructions that are popular in practice, and describes current research activity in the area. Security is defined in terms of the goals and resources of the attacker.

## 27.2 Unkeyed Cryptographic Primitives

The main unkeyed cryptographic primitive is the cryptographic hash function. This is an efficient function from bit strings of any length to bit strings of some fixed length (say 128 or 160 bits). The description of the function is assumed to be publicly available. If $H$ is a hash function, and if $y = H(x)$, then $y$ is called the "hash" or "hash value" of $x$.

One desirable property of a cryptographic hash function is that it should be difficult to invert. This means that given a specific hash value $y$, it is computationally infeasible to produce any $x$ such that $H(x) = y$. Another desirable property is that it should be difficult to find collisions. This means that it is computationally infeasible to produce two inputs $x$ and $x'$ such that $H(x) = H(x')$. The attacker is assumed to know a complete specification of the hash function.

A cryptographic hash function can be used for establishing data integrity. Suppose that the hash of a large file is stored in a secure location, while the file itself is stored in an insecure location. It is infeasible for an attacker to modify the file without detection because a rehash of the modified file will not match the stored hash value (unless the attacker was able to invert the hash function). We will see other applications of cryptographic hash functions when we look at asymmetric cryptographic primitives in Section 27.4.

Popular choices for cryptographic hash functions include MD-5 [1], RIPEMD-160 [2], and SHA-1 [3]. It is also common to construct cryptographic hash functions from symmetric key block ciphers [4].

Recent results of Xiaoyun Wang and colleagues (e.g., published at Eurocrypt 2005 and Crypto 2005) show how to find random collisions faster than by exhaustive search for a number of popular hash functions (including MD5, SHA-1, and RIPEMD). These results are worrisome despite the apparent safety in the fact that practical security exploits usually require the ability to find meaningful collisions, whereas these attacks so far can only find collisions with a very specific structure. NIST recommends a transition from SHA-1 to other approved hash functions in the same family with higher security (SHA-224, SHA-256, SHA-384, SHA-512).

### 27.2.1 Random Oracle Model

One direction of recent research is on the random oracle model. This is a design methodology for protocols and primitives that make use of cryptographic hash functions. Pick a specific cryptographic hash function such as MD-5. Its designers may believe that it is difficult to invert MD-5 or to find collisions for it. However, this does not mean that MD-5 is a completely unpredictable function, with no structure or regularity whatsoever. After all, the complete specification of MD-5 is publicly available for inspection and analysis, unlike a truly random function that would be impossible to specify in a compact manner. Nevertheless, the random oracle model asserts that a specific hash function like MD-5 behaves like a purely random function. This is part of a methodology for proving security properties of cryptographic schemes that make use of hash functions.

This assumption was introduced by Fiat and Shamir [5] and later formalized by Bellare and Rogaway [6]. It has been applied to the design and analysis of many schemes (see, e.g., the discussion of optimal asymmetric encryption padding in Section 27.4.3.1).

Recently, a cautionary note was sounded by Canetti et al. [7]. They demonstrate by construction that it is possible for a scheme to be secure in the random oracle model and yet have no secure instantiation whatsoever when any hash function is substituted. This is a remarkable theoretical result; however, the cryptographic community continues to base their designs on the random oracle model, and with good reason. Although it cannot provide complete assurance about the security of a design, a proof in the random oracle model provides confidence about the impossibility of a wide range of attacks. Specifically, it rules out common attacks where the adversary ignores the inner workings of the hash function and treats it as a "black box." The vast majority of protocol failures are due to this kind of black box attack, and thus the random oracle model remains an invaluable addition to the cryptographer's tool kit.

## 27.3 Symmetric Key Cryptographic Primitives

The main symmetric key cryptographic primitives are discussed, including block ciphers, stream ciphers, and message authentication codes.

### 27.3.1 Symmetric Key Block Ciphers

A symmetric key block cipher is a parameterized family of functions $E_K$, where each $E_K$ is a permutation on the space of bit strings of some fixed length. The input to $E_K$ is called the plaintext block, the output is called the ciphertext block, and $K$ is called the key. The function $E_K$ is called an encryption function. The inverse of $E_K$ is called a decryption function and is denoted $D_K$.

To encrypt a message that is longer than the fixed-length block, it is typical to employ a block cipher in a well-defined mode of operation. Popular modes of operation include output feedback mode, cipher feedback mode, and cipher block chaining mode; see Ref. [8] for a good overview. In this way, the plaintext and ciphertext can be bit strings of arbitrary (and equal) length. New modes of operations are being solicited in connection with the development of the Advanced Encryption Standard (AES) (see Section 27.3.1.3).

The purpose of symmetric key encryption is to provide data confidentiality. Security can be stated at a number of levels. It is always assumed that the attacker has access to a complete specification of the parameterized family of encryption functions and to a ciphertext of adequate length. Beyond this, the specific level of security depends on the goals and resources of the attacker. An attacker might attempt a total break of the cipher, which would correspond to learning the key $K$. An attacker might attempt a partial break of the cipher, which would correspond to learning some or all of the plaintext for a given ciphertext. An attacker might have no resources beyond a description of the block cipher and a sample ciphertext, in which case the attacker is mounting a ciphertext-only attack. An attacker might mount a known-plaintext attack, if the attacker is given a number of plaintext–ciphertext pairs to work with (input–output pairs for the encryption function). If the attacker is allowed to choose plaintexts and then see the corresponding ciphertexts, then the attacker is engaged in a chosen-plaintext attack.

Symmetric key block ciphers are valuable for data secrecy in a storage scenario (encryption by the data owner for an insecure data repository, and subsequent decryption by the data owner at a later time), or in a transmission scenario (across an insecure channel between a sender and receiver who have agreed on the secret key beforehand).

Perhaps, the most popular symmetric key block cipher for the past 25 years has been the Data Encryption Standard (DES) [9], although it may be near the end of its useful life. NIST recently announced the AES block cipher, which we discuss in Section 27.3.1.3.

Most modern block ciphers have an iterated design, where a round function is repeated some fixed number of times (e.g., DES has 16 rounds). Many modern block ciphers have a Feistel structure [10], which is an iterated design of a particular type. Let $(L_{j-1}, R_{j-1})$ denote the output of the $(j-1)$th round, divided into two halves for notational convenience. Then the output of the $j$th round is $(L_j, R_j)$, where $L_j = R_{j-1}$ and $R_j = L_{j-1}$ XOR $f(R_{j-1}, K_j)$ for some function $f$. Here $K_j$ is the $j$th round key, derived from the secret key according to some fixed schedule. Note that a block cipher with a Feistel structure is guaranteed to be a permutation even if the function $f$ is not invertible.

#### 27.3.1.1 Differential Cryptanalysis

Differential cryptanalysis is a powerful statistical attack that can be applied to many symmetric key block ciphers and unkeyed cryptographic hash functions. The first publication on differential cryptanalysis is due to Biham and Shamir [11], but Coppersmith [12] has described how the attack was understood during the design of the DES in the early 1970s.

The central idea of differential cryptanalysis for block ciphers is to sample a large number of pairs of ciphertexts for which the corresponding plaintexts have a known fixed difference $D$ (under the operation of bitwise XOR). The difference $D$ leads to a good characteristic if the XOR of the ciphertexts (or of an intermediate result during the computation of the ciphertext) can be predicted with a relatively large probability. By calculating the frequency with which every difference of plaintexts and every difference of ciphertexts coincides, it is possible to deduce some of the key bits through a statistical analysis of a sufficiently large sample of these frequencies.

For a differential cryptanalysis of DES, the best attack that Biham and Shamir discovered requires $2^{47}$ chosen-plaintext pairs with a given difference. They note that making even slight changes to the S-boxes (nonlinear substitution transformation at the heart of DES) can lead to a substantial weakening with respect to a differential attack.

#### 27.3.1.2 Linear Cryptanalysis

Linear cryptanalysis is another powerful attack that can be applied to many symmetric key block ciphers and unkeyed cryptographic hash functions. Consider the block cipher as being a composition of linear and nonlinear functions. The goal of linear cryptanalysis is to discover linear approximations for the nonlinear components. These approximations can be folded into the specification of the block cipher, and then expanded to find an approximate linear expression for the ciphertext output bits in terms of plaintext input bits and secret key bits. If the approximations were in fact perfect, then enough plaintext–ciphertext pairs would yield a system of linear equations that could be solved for the secret

key bits; however, even when the approximations are far from perfect, they enable a successful statistical search for the key, given enough plaintext–ciphertext pairs. This is a known-plaintext attack, unlike differential cryptanalysis, which is chosen plaintext.

Linear cryptanalysis was introduced by Matsui and Yamagishi [13]. Matsui applied linear cryptanalysis to DES [14]. In his best attack, $2^{43}$ known plaintexts are required to break DES with an 85% probability. See Langford and Hellman [15] for close connections between differential and linear cryptanalysis.

#### 27.3.1.3 Advanced Encryption Standard (AES)

In 1997, NIST began an effort to develop a new symmetric key encryption algorithm as a Federal Information Processing Standard (FIPS). The goal was to replace the DES, which was widely perceived to be at the end of its usefulness. A new algorithm was sought, with longer key and block sizes, and with increased resistance to newly revealed attacks such as linear cryptanalysis and differential cryptanalysis. The AES was to support 128-bit block sizes, and key sizes of 128 or 192 or 256 bits. By contrast, DES supported 64-bit block sizes and a key size of 56 bits.

Fifteen algorithms were proposed by designers around the world. This was reduced to five finalists, announced by NIST in 1999: MARS, RC6, Rijndael, Serpent, and TwoFish. In 2000, Rijndael was selected as the AES. Rijndael has a relatively simple structure; however, unlike many earlier block ciphers (such as DES), it does not have a Feistel structure.

The operation of Rijndael proceeds in rounds. Imagine that the block to be encrypted is written as a rectangular array of byte-sized words (four rows and four columns). First, each byte in the array is replaced by a different byte, according to a single fixed lookup table (S-box). Next, each row of the array undergoes a circular shift by a fixed amount. Next, a fixed linear transformation is applied to each column in the array. Last, the entire array is exclusive-or with a round key. All of the round keys are calculated by expanding the original secret key bits according to a simple key schedule. Note that the only nonlinear component is the S-box substitution step. Details of Rijndael's operation can be found in Ref. [16].

### 27.3.2 Symmetric Key Stream Ciphers

Stream ciphers compute ciphertext one character at a time, where the characters are often individual bits. By contrast, block ciphers compute ciphertext one block at a time, where the block is much larger (64 bits long for DES, 128 bits long for AES). Stream ciphers are often much faster than block ciphers. The typical operation of a stream cipher is to exclusive-or message bits with a key stream. If the key stream were truly random, this would describe the operation of a one-time pad. The key stream is not truly random, but it is instead derived from the short secret key.

A number of stream ciphers have been optimized for hardware implementation. The use of linear feedback shift registers is especially attractive for hardware implementation, but unfortunately these are not sufficiently secure when used alone. The Berlekamp–Massey algorithm [17] allows a hidden linear feedback shift register to be determined from a very short sequence of output bits. In practice, stream ciphers for hardware often combine linear feedback shift registers with nonlinear components to increase security. One approach is to apply a nonlinear function to the output of several linear feedback shift registers that operate in parallel (nonlinear combination generator). Another approach is to apply a nonlinear function to all of the states of a single linear feedback shift register (nonlinear filter generator). Still another approach is to have the output of one linear feedback shift register determine when a step should be taken in other linear feedback shift registers (clock-controlled generator).

Some stream ciphers have been developed to be especially fast when implemented in software, e.g., RC5 [18]. Certain modes of operation for block ciphers can be viewed as symmetric key stream ciphers (output feedback mode and cipher feedback mode).

### 27.3.3 Message Authentication Codes

A message authentication code (MAC) is a keyed cryptographic hash function. It computes a fixed-length output (tag) from an input of any length (message). When both the sender and the receiver know

the secret key, a MAC can be used to transmit information with integrity. Without knowing the secret key, it is very difficult for an attacker to modify the message or the tag so that the hash relation is maintained. The MAC in the symmetric key setting is the analog of the digital signature in the asymmetric key setting. The notion of message authentication in the symmetric key setting goes back to Gilbert et al. [19].

Security for MACs can be described with respect to different attack scenarios. The attacker is assumed to know a complete specification of the hash function, but not the secret key. The attacker might attempt to insert a new message that will fool the receiver, or the attacker might attempt to learn the secret key. The attacker might get to see some number of message–tag pairs, either for random messages or for messages chosen by the attacker.

One popular MAC is the CBC-MAC, which is derived from a block cipher (such as DES) run in cipher block chaining mode. Another approach is to apply an unkeyed cryptographic hash function after the message has been combined with the key according to some prepackaging transform. Care must be taken with the choice of transform; one popular choice is HMAC [20]. The UMAC construction [21] has been optimized for extremely fast implementation in software, while maintaining provable security. Jutla [22] recently showed especially efficient methods for combining message authentication with encryption, by using simple variations on some popular modes of operation for symmetric key block ciphers.

## 27.4 Asymmetric Key Cryptographic Primitives

Two asymmetric key cryptographic primitives are discussed in this section: public key encryption schemes and digital signature schemes.

### 27.4.1 Public Key Encryption Schemes

A public key encryption scheme is a method for deriving an encryption function $E_K$ and a corresponding decryption function $D_K$ such that it is computationally infeasible to determine $D_K$ from $E_K$. The encryption function $E_K$ is made public, so that anyone can send encrypted messages to the owner of the key. The decryption function $D_K$ is kept secret, so that only the owner of the key can read encrypted messages. The functions are inverses of each other, so that $D_K(E_K(M)) = M$ for every message $M$. Unlike the symmetric key setting, there is no need for the sender and receiver to preestablish a secret key before they can communicate securely.

Security for a public key encryption scheme relates to the resources and goals of the attacker. The attacker is assumed to have a complete description of the scheme, as well as the public encryption key $E_K$. Thus, the attacker is certainly able to encrypt arbitrary messages (chosen-plaintext attack). The attacker might be able to decrypt arbitrary messages (chosen-ciphertext attack, discussed in detail in Section 27.4.3.1). The goal of the attacker might be to deduce the decryption function $D_K$ (total break), or simply to learn all or some information about the plaintext corresponding to a particular ciphertext (partial break), or merely to guess which of two plaintexts is encrypted by a given ciphertext (indistinguishability).

The idea of public key encryption is due to Diffie and Hellman [23]. Most popular public key encryption schemes base their security on the hardness of some problem from number theory. The first public key encryption proposed remains one of the most popular today—the RSA scheme due to Rivest et al. [24]. Other popular public key encryption schemes are based on the discrete logarithm problem, including ElGamal [25] and elliptic curve variants [26].

For efficiency purposes, public key encryption is often used in a hybrid manner (called key transport). Suppose that a large message $M$ is to be encrypted using a public encryption key $E_K$. The sender chooses a random key $k$ for a symmetric key block cipher such as AES. The sender then transmits $E_K(k)$, $\mathrm{AES}_k(M)$. The first component enables the receiver to recover the symmetric key $k$, which can be used to decrypt the second component to recover $M$. The popular e-mail security protocol PGP uses this method (augmented with an integrity check).

It is also possible to use a key agreement protocol to establish a secret key over an insecure public channel, and then to use the secret key in a symmetric key block cipher. The idea is due to Diffie and Hellman [23], and the original Diffie–Hellman key agreement protocol is still widely used in practice.

## 27.4.2 Digital Signature Schemes

A digital signature scheme is a method for deriving a signing function $S_K$ and a corresponding verification function $V_K$, such that it is computationally infeasible to derive $S_K$ from $V_K$. The verification function $V_K$ is made public, so that anyone can verify a signature made by the owner of the signing key. The signing function $S_K$ is kept secret, so that only the owner of the signing key can sign messages. The signing function and verification function are related as follows: if the signature of a message $M$ is $S_K(M)$, then it should be the case that $V_K(S_K(M)) = \text{valid}$ for all messages $M$.

Security for a digital signature scheme depends on the goals and resources of the attacker [27]. The attacker is assumed to know a complete specification of the digital signature scheme and the verification function $V_K$. The attacker might also get to see message–signature pairs for random messages (known message attack), or for arbitrary messages chosen by the attacker (chosen message attack). The goal of the attacker might be to derive the signature function (total break) or to forge a signature on a particular message (selected message forgery) or to forge any message–signature pair (existential message forgery).

In practice, a signing function is applied not to the message itself but rather to the hash of the message (i.e., to the output of an unkeyed cryptographic hash function applied to the message). The security of the signature scheme is then related to the security of the hash function. For example, if a collision can be found for the hash function, then an attacker can produce an existential message forgery under a chosen message attack (by finding a collision on the hash function, and then asking for the signature of one of the colliding inputs).

One of the most popular digital signature schemes is RSA (based on the same primitive as RSA public key encryption, where $S_K = D_K$ and $V_K = E_K$). Other popular digital signature schemes include the digital signature algorithm (DSA) [28] and ElGamal [25].

## 27.4.3 Advanced Topics for Public Key Cryptography

### 27.4.3.1 Chosen Ciphertext Security for Public Key Encryption

As discussed earlier, a number of definitions for the security of a public key encryption scheme have been proposed. Chosen ciphertext security is perhaps the strongest natural definition, and it has emerged as the consensus choice among cryptographers as the proper notion of security to try to achieve. This is not to say that chosen ciphertext security is necessary for all applications, but instead of having a single encryption scheme, that is, chosen ciphertext secure will allow it to be used in the widest possible range of applications.

The strongest version of definition of chosen ciphertext security is due to Rackoff and Simon [29], building from a slightly weaker definition of Naor and Yung [30]. It can be described as a game between an adversary and a challenger. The challenger chooses a random public key and corresponding private key $[E_K, D_K]$, and sends the public key $E_K$ to the adversary. The adversary is then allowed to make a series of decryption queries to the challenger, sending arbitrary ciphertexts to the challenger and receiving their decryptions in reply. After this stage, the adversary chooses two messages $M_0$ and $M_1$ whose encryptions the adversary thinks will be particularly easy to distinguish between. The adversary sends $M_0$ and $M_1$ to the challenger. The challenger chooses one of these messages at random; call it $M_b$, where b is a random bit. The challenger encrypts $M_b$ and sends the ciphertext C to the adversary.

Now the adversary attempts to guess whether C is an encryption of $M_0$ or $M_1$. To help him with his guess, he is allowed to engage in another series of decryption queries with the challenger. The only restriction is that the adversary may never ask the challenger to directly decrypt C. At some point, the adversary makes his guess for $M_b$. If the adversary can win this game with any nonnegligible advantage (i.e., with probability $1/2$ plus $1/k^c$, where $k$ is the length of the private key and $c$ is any positive

constant), then we say that the adversary has mounted a successful chosen ciphertext attack. If no adversary (restricted to the class of probabilistic polynomial time turning machines) can mount a successful chosen ciphertext attack, then we say that the cryptosystem is chosen ciphertext secure.

This might seem like overkill for a definition of security. Unlimited access to a decryption oracle might seem like an unrealistically strong capability for the attacker. Merely distinguishing between two plaintexts might seem like an unrealistically weak goal for the attacker. Nevertheless, this definition has proven to be a good one for several reasons. First, it has been shown to be equivalent to other natural and strong definitions of security [31]. Second, Bleichenbacher [32] showed that a popular standard (RSA PKCS #1) was vulnerable to a chosen ciphertext attack in a practical scenario.

In the random oracle model, chosen ciphertext security can be achieved by combining a basic public key encryption scheme such as RSA with a simple prepackaging transform. Such a transform uses random padding and unkeyed cryptographic hash functions to scramble the message before encryption. The prepackaging transform is invertible, so that the message can be unscrambled after the ciphertext is decrypted.

The optimal asymmetric encryption padding (OAEP) transform takes an $m$-bit message $M$, a random bit string $R$ of length $s$, and outputs $\mathrm{OAEP}(M, R) = ((M \parallel 0^s) \text{ xor } H(R)) \parallel (R \text{ xor } G((M \parallel 0^s) \text{ xor } H(R)))$. Here $G$ and $H$ are unkeyed cryptographic hash functions that are assumed to have no exploitable weaknesses (random oracles). This can be viewed as a two-round Feistel structure (e.g., DES is a 16-round Feistel structure). Unpackaging the transform is straightforward. The OAEP transform is used extensively in practice, and has been incorporated in several standards. OAEP combined with RSA yields an encryption scheme that is secure against a chosen ciphertext attack [33,34].

Shoup [35] shows that OAEP+, a variation on OAEP, yields chosen ciphertext security when combined with essentially any public key encryption scheme: $\mathrm{OAEP+}\ (M, R) = ((M \parallel W(M, R)) \text{ xor } H(R)) \parallel (R \text{ xor } G(M \parallel W(M, R)) \text{ xor } H(R))$, where $G$, $H$, and $W$ are unkeyed cryptographic hash functions that behave like random oracles. Boneh [36] shows that even simpler prepackaging transforms (essentially one-round Feistel structure versions of OAEP and OAEP+) yield chosen ciphertext secure encryption schemes when combined with RSA or Rabin public key encryption.

Without the random oracle model, chosen ciphertext security can be achieved using the elegant Cramer–Shoup cryptosystem [37]. This is based on the hardness of the decision Diffie–Hellman problem (see Section 27.4.3.3). Generally speaking, constructions in the random oracle model are more efficient than those without it.

### 27.4.3.2 Threshold Public Key Cryptography

In a public key setting, the secret key (for decryption or signing) often needs to be protected from theft for long periods against a concerted attack. Physical security is one option for guarding highly sensitive keys, e.g., storing the key in a tamper-resistant device. Threshold public key cryptography is an attractive alternative for safeguarding critical keys.

In a threshold public key cryptosystem, the secret key is never in one location. Instead, the secret key is distributed across many locations. Each location has a different "share" of the key, and each share of the key enables the computation of a share of the decryption or signature. Shares of a signature or decryption can then be easily combined to arrive at the complete signature or decryption, assuming that a sufficient number of shareholders contribute to the computation. This sufficient number is the threshold that is built into the system as a design parameter. Note that threshold cryptography can be combined with physical security by having each shareholder use physical means to protect his individual share of the secret key.

Threshold cryptography was independently conceived by Desmedt [38], Croft and Harris [39], and Boyd [40], building on the fundamental notion of secret sharing [41,42]. Satisfactory threshold schemes have been developed for a number of public key encryption and digital signature schemes. These threshold schemes can be designed so as to defeat an extremely strong attacker who is able to travel from shareholder to shareholder, attempting to learn or corrupt all shares of the secret key (proactive security). Efficient means are also available for generating shared keys from scratch by the shareholders

themselves, so that no trusted dealer is needed to initialize the threshold scheme [43,44]. Shoup [45] recently proposed an especially simple and efficient scheme for threshold RSA.

#### 27.4.3.3 New Hardness Assumptions for Asymmetric Key Cryptography

A trend has occurred in recent years toward the exploration of the cryptographic implications of new hardness assumption. Classic assumption includes the hardness of factoring a product of two large primes, the hardness of extracting roots modulo a product of two large primes, and the hardness of computing discrete logarithms modulo a large prime (i.e., solving $g^x = y \bmod p$ for $x$).

One classic assumption is the Diffie–Hellman assumption. Informally stated, this assumption is that it is difficult to compute ($g^{ab} \bmod p$) given ($g^a \bmod p$) and ($g^b \bmod p$), where $p$ is a large prime. This assumption underlies the Diffie–Hellman key agreement protocol. The decisional Diffie–Hellman assumption has proven to be useful in recent years. Informally stated, this assumption is that it is difficult to distinguish triples of the form ($g^a \bmod p$, $g^b \bmod p$, $g^{ab} \bmod p$) and triples of the form ($g^a \bmod p$, $g^b \bmod p$, $g^c \bmod p$) for random $a$, $b$, $c$. Perhaps most notably, the Cramer–Shoup chosen ciphertext secure encryption scheme is based on this new assumption.

The security of RSA is based on a root extraction problem related to the hardness of factoring: given message $M$ and modulus $N = pq$ of unknown factorization and suitable exponent e, compute $M^{1/e} \bmod N$. Recently, a number of protocols and primitives have been based on a variant of this assumption called the strong RSA assumption: given $M$ and $N$, find e and $M^{1/e} \bmod N$ for any suitable e. For example, a provably secure signature scheme can be based on this new assumption without the need for the random oracle assumption [46].

The RSA public key scheme is based on arithmetic modulo $N$, where $N = pq$ is a product of two primes (factors known to the private key holder but not to the public). Recently, Paillier [47] has proposed a novel public key encryption scheme based on arithmetic modulo $p^2q$. His scheme has nice homomorphic properties, which enable some computations to be performed directly on ciphertexts. For example, it is easy to compute the encryption of the sum of any number of encrypted values, without knowing how to decrypt these ciphertexts. This has many nice applications, such as for secure secret ballot election protocols.

The Phi-hiding assumption was introduced by Cachin et al. [48]. This is a technical assumption related to prime factors of $p-1$ and $q-1$ in an RSA modulus $N = pq$. This assumption enables the construction of an efficient protocol for querying a database without revealing the queries that are being made to the database (private information retrieval).

Lastly, a number of hardness assumptions (including the bilinear Diffie–Hellman assumption and the decisional bilinear Diffie–Hellman assumption) have been proposed for proving the security of constructions in pairing-based cryptography (see Section 27.4.3.4).

#### 27.4.3.4 Pairing-Based Cryptography

In the early 1990s, Menezes et al. [49] demonstrated a new attack on the discrete logarithm problem over certain elliptic curves. The main technical tools were pairing functions that mapped pairs of points on the elliptic curve to elements of a different group (where the discrete logarithm problem might be easier). In 2000, Joux [50] showed that these pairing functions had positive uses in cryptography, by using them to construct an efficient three-party variant of Diffie–Hellman key exchange. Independently, and around the same time, similar ideas were explored by Sakai, Ohgishi, and Kasahara [63].

The flexibility and versatility of these tools has made pairing-based cryptography a very active field in recent years. In this section, we briefly mention a few new constructions that have particularly interesting features for secure systems design.

The problem of identity-based encryption (or IBE)—originally posed by Adi Shamir in 1984 [51]—is to construct a public key encryption scheme in which the public key can be any desired bit string (e.g., the e-mail address of the recipient), while the corresponding private key can be derived by a trusted third party that knows the master key for the system. In 2001, an efficient pairing-based construction for IBE was proposed [52].

Gentry and Silverberg [53] showed a pairing-based construction for hierarchical identity-based encryption (HIBE), in which a hierarchy of master keys can be created on the fly. This can limit the damage if any one particular master key is exposed. HIBE is a useful building block for other pairing-based cryptographic constructions with novel functionality. For example, Canetti et al. [54] used HIBE to construct a forward secure encryption scheme, in which the public encryption key stays fixed while the private decryption key evolves overtime. Exposure of the private key at any moment allows the attacker to decrypt all current and future ciphertexts, but past ciphertexts remain secure.

Pairing-based cryptography has also influenced the design of digital signature schemes. For example, a very short signature scheme (e.g., just 160 bits at current security levels) based on pairings was proposed and analyzed by Boneh et al. [55]. Another example is the aggregate signature scheme of Boneh et al. [56] that allows many different signatures of different messages by different parties to be combined mathematically into a small string such that all of the signatures can still be verified.

#### 27.4.3.5 Privacy Preserving Protocols

Using the cryptographic primitives described in earlier sections, it is possible to design protocols for two or more parties to perform useful computational tasks while maintaining some degree of data confidentiality. Theoretical advances were well established with the completeness theorems in Ref. [57] and others; however, practical solutions have often required special-purpose protocols tailored to the particular problem.

One important example—both historically and practically—is the problem of conducting a secret ballot election [58,59]. This can be viewed as a cryptographic protocol design problem among three types of parties: voters, talliers, and independent observers. All types of parties have different security requirements. Voters want to guarantee that their individual ballots are included in the final tally, and that the contents of the ballots remain secret. Talliers want to produce an accurate final count that includes all valid ballots counted exactly once, and no invalid ballots. Independent observers want to verify that the tally is conducted honestly. One of the best secret ballot election protocols currently known for large-scale elections is probably [60] the protocol based on threshold public key encryption.

## 27.5 Other Resources

An excellent resource for further information is the *CRC Handbook of Applied Cryptography* [61], particularly the first chapter of this handbook, which has an overview of cryptography that is highly recommended. Ross Anderson's book on security engineering [62] is a recommended resource, especially for its treatment of pragmatic issues that arise when implementing cryptographic primitives in practice. See also the frequently asked questions list maintained by RSA Labs (www.rsa.com/rsalabs). Paulo Barreto's *Pairing-Based Crypto Lounge* has a lot of good information about pairing-based cryptography (http://paginas.terra.com.br/informatica/paulobarreto/pblounge.html).

## References

1. Rivest, R., The MD5 message-digest algorithm, Internet Request for Comments, RFC 1321, April 1992.
2. Dobbertin, H., Bosselers, A., and Preneel, B., RIPEMD-160: A strengthened version of RIPEMD, *Proceedings of Fast Software Encryption Workshop*, Gollman, D. (Ed.), Springer-Verlag, LNCS, Heidelberg, 1039, 71, 1996.
3. FIPS 180-1, Secure hash standard, Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, May 11, 1993.
4. Preneel, B., Cryptographic hash functions, *Eur. Trans. Telecomm.*, 5, 431, 1994.

5. Fiat, A. and Shamir, A., How to prove yourself: Practical solutions to identification and signature problems, *Advances in Cryptology—Crypto'93*, Springer-Verlag, LNCS, Heidelberg, 773, 480, 1994.
6. Bellare, M. and Rogaway, P., Random oracles are practical: A paradigm for designing efficient protocols, *Proceedings of the 1st ACM Conference on Computer and Communications Security*, Fairfax, VA, 62, 1993.
7. Canetti, R., Goldreich, O., and Halevi, S., The random oracle model revisited, *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing*, Dallas, TX, 209, 1998.
8. Davies, D. and Price, W., *Security for Computer Networks,* 2nd ed., John Wiley & Sons, New York, 1989.
9. FIPS 46, Data encryption standard, Federal Information Processing Standards Publication 46, U.S. Dept. of Commerce/N.B.S., National Technical Information Service, Springfield, VA, 1977 (revised as FIPS 46-1: 1988; FIPS 46-2:1993).
10. Feistel, H., Notz, W., and Smith, J., Some cryptographic techniques for machine-to-machine data communications, *Proc. IEEE* 63, 1545, 1975.
11. Biham, E. and Shamir, A., Differential cryptanalysis of DES-like cryptosystems, *J. Cryptol.,* 4, 3, 1991.
12. Coppersmith, D., The data encryption standard (DES) and its strength against attacks, *IBM J. Res. Dev.*, 38, 243, 1994.
13. Matsui, M. and Yamagishi, A., A new method for known plaintext attack of FEAL cipher, Advances in *Cryptology—Eurocrypt'92*, Springer-Verlag, LNCS, Heidelberg, 658, 81, 1993.
14. Matsui, M., Linear cryptanalysis method for DES cipher, *Advances in Cryptology—Eurocrypt'93*, Springer-Verlag, LNCS, Heidelberg, 765, 386, 1994.
15. Langford, S. and Hellman, M., Differential-linear cryptanalysis, *Advances in Cryptology—Crypto'94*, Springer-Verlag, LNCS, Heidelberg, 839, 17, 1994.
16. National Institute of Standards and Technology, Advanced Encryption Standard (AES), http://csrc.nist.gov/encryption/aes/.
17. Massey, J., Shift-register synthesis and BCH decoding, *IEEE Trans. Inf. Th.*, 15, 122, 1969.
18. Rivest, R., The RC5 encryption algorithm, *Fast Software Encryption, Second International Workshop*, Springer-Verlag, LNCS, Heidelberg, 1008, 86, 1995.
19. Gilbert, E., MacWilliams, F., and Sloane, N., Codes which detect deception, *Bell Sys. Tech. J.,* 53, 405, 1974.
20. Bellare, M., Canetti, R., and Krawczyk, H., Keying hash functions for message authentication, *Advances in Cryptology—Crypto'96*, Springer-Verlag, LNCS, Heidelberg, 1109, 1, 1996.
21. Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P., UMAC: Fast and secure message authentication, *Advances in Cryptology—CRYPTO'99*, Springer-Verlag, LNCS, Heidelberg, 1666, 216, 1999.
22. Jutla, C., Encryption modes with almost free message integrity, *Advances in Cryptology—Eurocrypt 2001*, Springer-Verlag, LNCS, Heidelberg, 2045, 529, 2001.
23. Diffie, W. and Hellman, M., New directions in cryptography, *IEEE Trans. Inf. Th.,* 22, 644, 1976.
24. Rivest, R., Shamir, A., and Adleman, L., A method for obtaining digital signatures and public-key cryptosystems, *Commn. ACM,* 21, 120, 1978.
25. ElGamal, T., A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Trans. Inf. Th.,* 31, 469, 1985.
26. Koblitz, N., Elliptic curve cryptosystems, *Math. Comput.,* 48, 203, 1987.
27. Goldwasser, S., Micali, S., and Rivest, R., A digital signature scheme secure against adaptive chosen-message attacks, *SIAM J. Comput.,* 17, 281, 1988.
28. Kravitz, D., Digital signature algorithm, U.S. Patent #5, 231, 668, July 27, 1993.
29. Rackoff, C. and Simon, D., Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack, *Advances in Cryptology—Crypto'91*, Springer-Verlag, LNCS, Heidelberg, 576, 433, 1992.
30. Naor, M. and Yung, M., Public-key cryptosystems provably secure against chosen ciphertext attacks, *Proceedings of the ACM Symposium on Theory of Computing*, Seattle, WA, 33, 1989.

31. Dolev, D., Dwork, C., and Naor, M., Non-malleable cryptography, *SIAM J. Comput.,* 30, 391, 2000.
32. Bleichenbacher, D., Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1, *Advances in Cryptology—CRYPTO'98*, Springer-Verlag, LNCS, Heidelberg, 1462, 1, 1998.
33. Bellare, M. and Rogaway, P., Optimal asymmetric encryption, *Advances in Cryptology—Eurocrypt'94*, Springer-Verlag, LNCS, Heidelberg, 950, 92, 1995.
34. Fujisaki, E., Okamoto, T., Pointcheval, D., and Stern, J., RSA-OAEP is secure under the RSA assumption, *Advances in Cryptology—Crypto 2001*, Springer-Verlag, LNCS, Heidelberg, 2139, 260, 2001.
35. Shoup, V., OAEP reconsidered, *Advances in Cryptology—Crypto 2001*, Springer-Verlag, LNCS, Heidelberg, 2139, 239, 2001.
36. Boneh, D., Simplified OAEP for the Rabin and RSA functions, *Advances in Cryptology—Crypto 2001*, Springer-Verlag, LNCS, Heidelberg, 2139, 275, 2001.
37. Cramer, R. and Shoup, V., A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack, *Advances in Cryptology—Crypto'98*, Springer-Verlag, LNCS, Heidelberg, 1462, 13, 1998.
38. Desmedt, Y., Society and group oriented cryptography: A new concept, *Advances in Cryptology—Crypto'87*, Springer-Verlag, LNCS, Heidelberg, 293, 120, 1988.
39. Croft, R. and Harris, S., Public-key cryptography and re-usable shared secrets, in *Cryptography and Coding,* Beker, H. and Piper, F. (Eds.), Clarendon Press, Oxford, 189, 1989.
40. Boyd, C., Digital multisignatures, in *Cryptography and Coding,* Beker, H. and Piper, F. (Eds.), Clarendon Press, Oxford, 241, 1989.
41. Shamir, A., How to share a secret, *Comm. ACM,* 22, 612, 1979.
42. Blakley, R., Safeguarding cryptographic keys, *Proceedings of AFIPS National Computer Conference*, Arlington, VA, 313, 1979.
43. Pedersen, T., A threshold cryptosystem without a trusted party, *Advances in Cryptology—Eurocrypt'91*, Springer-Verlag, LNCS, Heidelberg, 547, 522, 1992.
44. Boneh, D. and Franklin, M., Efficient generation of shared RSA keys, *J. ACM,* 48, 702, 2001.
45. Shoup, V., Practical threshold signatures, *Advances in Cryptology—Eurocrypt 2000,* Springer-Verlag, LNCS, Heidelberg, 1807, 207, 2000.
46. Cramer, R. and Shoup, V., Signature schemes based on the strong RSA assumption, *ACM Trans. Inf. Sys. Sec.,* 3, 161, 2000.
47. Paillier, P., Public key cryptosystems based on composite degree residuosity classes, *Advances in Cryptology—Eurocrypt'99*, Springer-Verlag, LNCS, Heidelberg, 1592, 223, 1999.
48. Cachin, C., Micali, S., and Stadler, M., Computationally private information retrieval with polylogarithmic communication, *Advances in Cryptology—EUROCRYPT'99*, Springer-Verlag, LNCS, Heidelberg, 1592, 402, 1999.
49. Menezes, A., Okamoto, T., and Vanstone, S.A., Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Inf. Th.,* 39, 1639, 1993.
50. Joux, A., A one round protocol for tripartite Diffie–Hellman, *Proceedings of the ANTS IV*, Springer-Verlag, LNCS, Heidelberg, Leiden, The Netherlands, 1838, 385, 2000.
51. Shamir, A., Identity-based cryptosystems and signature schemes, *Proceedings of Crypto*, Springer-Verlag, LNCS, Heidelberg, Santa Barbara, CA, 196, 47, 1984.
52. Boneh, D. and Franklin, M., Identity-based encryption from the Weil pairing. *SIAM J. Comput.*, 32, 586, 2003.
53. Gentry, C. and Silverberg, A., Hierarchical ID-based cryptography, *Proceedings of Asiacrypt*, Springer-Verlag, LNCS, Heidelberg, Queenstown, NZ, 2501, 548, 2002.
54. Canetti, R., Halevi, S., and Katz, J., A forward-secure public-key encryption scheme, *Proceedings of Eurocrypt*, Springer-Verlag, LNCS, Heidelberg, Warsaw, Poland, 2656, 255, 2003.
55. Boneh, D., Lynn, B., and Shacham, H., Short signatures from the Weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.
56. Boneh, D., Gentry, C., Lynn, B., and Shacham, H., Aggregate and verifiably encrypted signatures from bilinear maps, *Proceedings of Eurocrypt*, Springer-Verlag, LNCS, Heidelberg, Warsaw, Poland, 2656, 416, 2003.

57. Goldreich, O., Micali, S., and Wigderson, A., How to play any mental game or a completeness theorem for protocols with honest majority, *Proceedings of ACM Symposium on the Theory of Computing*, 218, New York, NY, 1987.
58. Cohen, J. and Fisher, M., A robust and verifiable cryptographically secure election scheme, *Proceedings of the IEEE Symposium on Foundations of Computer Science*, Portland, OR, 372, 1985.
59. Benaloh, J. and Yung, M., Distributing the power of a government to enhance the privacy of voters, *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Calgary, Canada, 52, 1986.
60. Cramer, R., Schoenmakers, B., and Genarro, R., A secure and optimally efficient multi-authority election scheme, *Eur. Trans. Telecomm.*, 8, 481, 1997.
61. Menezes, A., van Oorschot, P., and Vanstone, S., *Handbook of Applied Cryptography,* CRC Press, Boca Raton, FL, 1997.
62. Anderson, R., *Security Engineering: A Guide to Building Dependable Systems*, John Wiley & Sons, New York, 2001.
63. Sakai, R., Ohgishi, K., and Kasahara, M., Cryptosystems based on pairing, *2000 Symposium on Cryptography and Information Security (SCIS2000)*, Okinawa, Japan, Jan. 26–28, 2000.

# Index

## A

## D

## E

## F



## G



## H



## I

## N



## O

## P

## Q



## R

## S

## T



## U

## V



## W

## X



## Y



## Z

Computer Science/ Computer Engineering/ Computing

# Digital Systems and Applications

New design architectures in computer systems have surpassed industry expectations. Limits, which were once thought of as fundamental, have now been broken. **Digital Systems and Applications** details these innovations in systems design as well as cutting-edge applications that are emerging to take advantage of the fields increasingly sophisticated capabilities. This book features new chapters on parallelizing iterative heuristics, stream and wireless processors, and lightweight embedded systems.

**This fundamental text—**

- Provides a clear focus on computer systems, architecture, and applications
- Takes a top-level view of system organization before moving on to architectural and organizational concepts such as superscalar and vector processor, VLIW architecture, as well as new trends in multithreading and multiprocessing. includes an entire section dedicated to embedded systems and their applications
- Discusses topics such as digital signal processing applications, circuit implementation aspects, parallel I/O algorithms, and operating systems
- Concludes with a look at new and future directions in computing
- Features articles that describe diverse aspects of computer usage and potentials for use
- Details implementation and performance-enhancing techniques such as branch prediction, register renaming, and virtual memory
- Includes a section on new directions in computing and their penetration into many new fields and aspects of our daily lives

6195
ISBN 0-8493-8619-5
90000
9 780849 386190
www.crcpress.com

CRC Press
Taylor & Francis Group
an informa business
www.taylorandfrancisgroup.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
270 Madison Avenue
New York, NY 10016
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK