METRIC

MIL-HDBK-62
13 September 1996

DEPARTMENT OF DEFENSE HANDBOOK

DOCUMENTATION OF DIGITAL ELECTRONIC SYSTEMS WITH VHDL

This handbook is for guidance only.
Do not cite this document as a requirement.

AMSC/NA **FSC 5962**

[illegible]

FOREWORD

1. This handbook is approved for use by all Departments and Agencies of the Department of Defense (DoD).

2. This handbook is for guidance only. This handbook cannot be cited as a requirement. If it is, the contractor does not have to comply.

3. This handbook was developed to provide guidance to Department of Defense personnel who are writing requests for proposals for military digital electronic systems, DoD contractors who are developing very high-speed integrated circuit (VHSIC) hardware description language (VHDL) models for the Government, and DoD engineers, scientists, and management or independent validation and verification contractors who are evaluating or reviewing models delivered to the Government. It documents the state of the art and existing technologies for VHDL model development. Addressed in the handbook are which VHDL models are required to be delivered with a contract, which VHDL models should be developed during the different stages of the lifetime of a system, and how VHDL models can be structured to be consistent with modeling standards.

4. This handbook was developed under the auspices of the US Army Materiel Command's Engineering Design Handbook Program, which is under the direction of the US Army Industrial Engineering Activity. Research Triangle Institute (RTI) was the prime contractor for this handbook under Contract No. DAAA09-86-D-0009. The handbook was authored by Dr. Geoffrey A. Frank and edited by Ray C. Anderson of RTI. Development of this handbook was guided by a technical working group that included Mr. Gerald T. Michael, US Army Research Laboratory, chairman; Dr. John W. Hines, US Air Force Wright Laboratory; Mr. J. P. Letellier, US Naval Research Laboratory; and Mr. Michael A. Frye, US Department of Defense, Defense Logistics Agency.

5. Beneficial comments (recommendations, additions, deletions) and any pertinent data that may be of use in improving this document should be addressed to Defense Supply Center Columbus, ATTN: Director-VA, 3990 East Broad Street, Columbus, OH 43216-5000, by using the Standardization Document Improvement Proposal (DD Form 1426) appearing at the end of this document or by letter.

CONTENTS

CHAPTER 3
VHDL CONCEPTS

CHAPTER 4
DoD REQUIREMENTS FOR THE USE OF VHDL

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

A

AC	=	alternating current
ALU	=	arithmetic and logic unit
ANSI	=	American National Standards Institute
ASCII	=	American standard code for information interchange
ASIC	=	application-specific integrated circuit
ATE	=	automatic test equipment
ATPG	=	automatic test pattern generator

B

BIM	=	bus interface module
BIT	=	built-in test
BIU	=	bus interface unit
BSDL	=	boundary scan definition language

C

CAD	=	computer-aided design
CAE	=	computer-aided engineering
CALS	=	computer-aided acquisition and logistics support
CDR	=	Critical Design Review
CDRL	=	contract data requirements list
COTS	=	commercial off-the-shelf
CMOS	=	complementary metal-oxide semiconductor
CPU	=	central processing unit
CSP	=	communicating sequential process

D

DASC	=	Design Automation Standards Committee
DESC	=	Defense Electronics Supply Center
DID	=	data item description
DoD	=	Department of Defense

E

ECAD	=	electronic computer-aided design
EDIF	=	electronic design interchange format
EDS	=	electronic data sheet
EIA	=	Electronic Industries Association
ESD	=	electrostatic discharge
EW	=	electronic warfare

F

FCR	=	fault containment region
FDIR	=	fault detection, isolation, and recovery
FFT	=	fast Fourier transform
FIFO	=	first in, first out
FSM	=	finite state machine

H

HSDB	=	high-speed data bus
HW	=	hardware
HWCI	=	hardware configuration item

I

IC	=	integrated circuit
IEEE	=	Institute of Electrical and Electronic Engineers
IGES	=	International Graphics Exchange Standard
IIR	=	infinite impulse response
I/O	=	input/output
IPC	=	Institute for Interconnecting and Packaging Electronic Circuits
ISA	=	instruction set architecture

J

JTAG	=	Joint Test Action Group

L

LRM	=	language reference manual
LRM	=	line-replaceable module
LRU	=	line-replaceable unit
LSSD	=	level-sensitive scan design

M

MCM	=	multichip module
MUT	=	module under test

N

NMOS	=	negative metal-oxide semiconductor

P

PDR	=	Preliminary Design Review
PI	=	processor interface
PLA	=	programmable logic array
PMS	=	processor memory switch

Q

QPL	=	qualified products list

R

R	=	reset
RAM	=	random-access memory
RFP	=	request for proposal
ROM	=	read-only memory
RTL	=	register-transfer level

S

S	= set
SA/0	= stuck at zero
SA/1	= stuck at one
SDF	= standard delay format
SPSP	= special-purpose signal processor
SW	= software

T

TAP	= test access port
TIREP	= Technology Independent Representation of Electronic Products
TMS	= test mode select
TRR	= Test Readiness Review

U

UUT	= unit under test

V

VHDL	= very high-speed integrated circuit (VHSIC) hardware description language
VHSIC	= very high-speed integrated circuit
VITAL	= VHDL initiative toward ASIC libraries
VLSI	= very large-scale integrated
VML	= VHDL model library
V&V	= validation and verification

W

WAVES	= Waveform and Vector Exchange Specification
WGP	= waveform generator procedure

CHAPTER 1 INTRODUCTION

The goals, scope, and intended audience of the handbook are described in this chapter. Included are references to industry standardization efforts related to the goals of this handbook. Also provided is an overview of each chapter of the handbook.

1-1 PURPOSE

This handbook describes the use of the very high-speed integrated circuit (VHSIC) hardware description language (VHDL) to document the design of military digital electronic systems. This handbook is designed to help Government personnel involved in the acquisition of military digital electronic systems understand the following issues related to the use of VHDL models to document military digital electronic systems:

1. What VHDL models are required to be delivered with a contract? In particular, this handbook discusses the guidelines described in MIL-HDBK-454 (Ref. 1) and the requirements of the VHDL data item description (DID) (Ref. 2). (The VHDL DID provides comprehensive requirements for VHDL models that include the need for extensive auxiliary and testing support files. This handbook contains approaches to structuring VHDL models so that DID requirements and intent can be met without an excessive number of auxiliary and testing support files. Government personnel can use information in this handbook to tailor definitions of items in the DID to fit their project needs. Contractors can use the information to propose the organization and content of VHDL models they will deliver to the Government.)

2. Which VHDL models should be developed during the different stages of the lifetime of a system? The Department of Defense (DoD) requirements now mandate delivery of VHDL models after a system or chip has been fabricated and is ready for deployment, but VHDL models have great potential to support the evaluation of system and chip designs before fabrication is started. The types of VHDL models appropriate for delivery early in the system design process are discussed in this handbook. This information may be useful to DoD personnel during the preparation of requests for proposals (RFPs). This handbook may also be useful to DoD personnel in preparing phased development programs for which multiple awards are made in the early phases of the program to prepare competing designs (which should include VHDL models of the designs).

3. How can VHDL models be structured to be consistent with modeling standards? It is of critical importance to the DoD that VHDL models of compatible pieces of hardware are themselves compatible. Because VHDL is such an expressive language, different descriptions may not be easily interfaced if standards for defining interfaces are not observed. Guidelines and reference modeling standards to ensure compatibility between VHDL models are described in this handbook. In particular, standards for descriptions of test vectors such as the Waveform and Vector Exchange Specification (WAVES) standard (Ref. 3), standard bus interfaces such as the Institute of Electrical and Electronics Engineers (IEEE) Standards 1149.1 (Ref. 4) and 1149.5 (Ref. 5) or test and maintenance, and standard data-type descriptions such as IEEE Standard 1164 (Ref. 6) are discussed.

1-2 SCOPE

Use of VHDL to model military digital electronic systems is described in this handbook. In particular, this handbook addresses the development of models compliant with the VHDL DID (Ref. 2) and MIL-HDBK-454 (Ref. 1). Digital electronics are only part of most military systems. Most modern weapons platforms use sensors and actuators that are tightly coupled with the digital electronic systems; however, the modeling of these sensors and actuators is outside the scope of this handbook. Many military electronic systems have both digital and analog components. The modeling of only the digital components is discussed in this handbook. Researchers are currently exploring the use of VHDL for analog components and considering changes to the language to allow VHDL to better support modeling of analog and hybrid components and subsystems. Although VHDL models are frequently used to provide test beds for testing software before the hardware is fabricated, this handbook does not discuss the issues of developing tests for software.

The handbook is not intended to provide a working knowledge of VHDL. On the other hand, the handbook introduces VHDL terms and concepts so it can serve as a stand-alone reference document for readers familiar with VHDL.

1-3 INTENDED AUDIENCE

This handbook is intended for use by DoD personnel who are writing requests for proposals for digital electronic systems, DoD contractors who are developing VHDL models to be delivered to the Government, and DoD personnel or independent validation and verification contractors who are evaluating or reviewing models that have been delivered to the Government. DoD personnel include people who are writing RFPs for the development of digital electronic systems, are serving on proposal review teams, are negotiating the deliverables and tailoring the DIDs associated with a contract, are part of Government validation and verification (V&V) teams, or are in government laboratories tracking the evolution of technology for the design of digital electronic systems. VHDL tool vendors or VHDL library vendors may also find this handbook useful in terms of understanding the needs of DoD contractors.

Users of this handbook should have some formal training or some experience with electrical engineering and/or computer science and should have experience reading and writing VHDL models.

Although the user does not need a complete understanding of VHDL to read this handbook, he or she will need to understand VHDL to implement the suggestions made in this handbook and to understand the example VHDL programs.

1-4 HISTORY, PURPOSE, AND SCOPE OF VHDL

1-4.1 HISTORY OF VHDL

The VHSIC program was created to ensure that the digital microelectronic systems in the weapon systems fielded by the DoD would be at least comparable to state-of-the-practice commercial technology. Over its 10-yr lifetime this program developed tools and technology for the design, manufacture, and use of state-of-the-art integrated circuits (ICs).

At the start of the VHSIC program in 1980, the DoD was already experiencing a problem with the obsolescence of ICs. VHSIC studies (Ref. 7) indicated that by 1990, 80% of nonmemory ICs in military electronic systems would be application-specific integrated circuits (ASICs). At the same time the VHSIC studies also indicated that the average lifetime of a fabrication process would be two years. Since the acquisition process for DoD systems was seven to ten years, a majority of the ICs in a DoD system would be obsolete before the system could be fielded.

VHDL began as a research effort under the DoD VHSIC program to document fully the DoD digital systems (Ref. 8). As experience with the language was gained, the language was improved by incorporating additional features. The language was subsequently standardized by the IEEE and adopted by the American National Standards Institute (ANSI) as ANSI/IEEE Std 1076-1987 (Ref. 9). This standard was updated in 1994 by IEEE Std 1076-1993 (Ref. 10).

1-4.2 THE PURPOSE OF VHDL

VHDL was developed to provide a standardized language to describe formally the behavior and structure of DoD digital electronic systems (Ref. 8). These descriptions serve as a procurement device by specifying exactly what functions a new device would have to perform in order to replace an old device. Through simulation of these descriptions the ability of the design of a new device to perform the same required functions as the old device can be more accurately estimated before being physically verified. Furthermore, the VHDL descriptions may contain timing information. As a result, the performance of competing designs can be compared before the devices are built. This performance simulation provides an ability to perform an impartial assessment of proposals for integrated circuits and for complex electronic systems containing many ICs.

Because VHDL has been standardized, it is now being used as the primary hardware description language for commercial computer-aided design (CAD) vendors, and it is likely that this trend will continue. VHDL is also coming into use as an exchange standard between tool sets provided by different vendors.

As previously stated, VHDL was developed to serve the need of the DoD to document the functionality of digital electronic systems delivered to it by the defense industry (Ref. 8). This documentation is required to procure new systems and to assist in the maintenance of fielded systems. VHDL provides a powerful, technology-independent way to describe a wide range of electronic hardware systems from individual integrated circuits to large multiprocessor systems. It supports top-down and bottom-up design methodologies or mixtures of the two.

For new systems a VHDL model can be provided by the DoD that specifies the exact functional behavior desired of the system. This description can then be offered to potential bidders for competitive procurement. Bidders can be required to submit VHDL models of their proposed designs, and these can be simulated and compared with the original DoD model. The VHDL models could be evaluated as part of the overall proposal evaluation process. This step ensures that bidders understand the functions the system is to perform and that the designs will meet functional requirements.

VHDL also provides important benefits after a system is fielded. As fielded systems fail and are repaired, additional spare parts must be acquired as stocks of original spare parts are exhausted. For electronic systems this need requires that the DoD provide, among other things, a complete functional specification of the desired parts to potential bidders. This functional specification must be technology independent because it is often impossible or excessively expensive to acquire parts in the original technology; thus it becomes desirable to reimplement the function in a different technology. Technology independence permits the separation of the behavior function (plus timing) from its implementation, which makes incorporating new technologies easier.

Until the advent of VHDL there was no standard way to provide this functional specification. Documentation delivered with the original systems was usually in a technology-dependent, proprietary format that was not supportable long term. This obsolescence raised the cost and technical risk of reprocuring new parts because using technically obsolete engineering data is expensive and time-consuming. VHDL offers the technical means to provide functional, timing, and other specifications for digital electronic systems in a form that will be useful long after the original system is delivered.

1-4.3 THE SCOPE OF VHDL

VHDL supports describing hardware at many levels of abstraction from an entire system composed of individual racks of equipment down to gate-level descriptions of integrated circuits. VHDL includes primitive functions for gate-level operations. VHDL supports processes, a rich data abstraction facility, and synchronization capabilities for algorithmic descriptions. VHDL allows different levels of abstraction to be mixed in the same description, and this flexibility can reduce both the amount of time for simulation and the introduction of unnecessary detail. VHDL also provides for the specification of detailed hardware timing requirements. Timing specification is particularly important when the VHDL description represents a hardware component that must be integrated with other components, as is almost always the case. VHDL also supports annotating designs and allows the user to specify physical types and their units, which can be used as attributes for a design.

The DoD is actively incorporating VHDL requirements into procedures used to develop military electronic computers. VHDL is required documentation under Guideline 64 of MIL-HDBK-454, which defines the requirements for VHDL descriptions to accompany any digital electronics that are being added to the DoD qualified products list (QPL). A data item description, DI-EGDS-80811 (Ref. 2), defines the detailed characteristics of a VHDL model to be delivered to the Government. VHDL models of systems will become part of the Computer-Aided Acquisition and Logistic Support (CALS) Program (Ref. 11) usage guidelines.

1-5 RELATED INDUSTRY STANDARDS

Realizing the benefits for customers of standardized models and modeling languages, the electronics industry is developing commercial standards for electronic systems. This is a continuing process. For example, the IEEE requires updates of its standards every five years.

The DoD recognizes and strongly supports VHDL standardization efforts, including the following: (1) the IEEE VHDL (1076) standardization, (2) the IEEE Design Automation Standards Committee (DASC) standards, (3) the Joint Test Action Group (JTAG) definition of test interface standards, including the IEEE 1149.1 boundary scan test bus and the IEEE 1149.5 test and maintenance bus, as well as the Boundary Scan Definition Language (BSDL) (Ref. 12), a VHDL style that describes implementations of IEEE Std 1149.1 boundary scan test circuitry, (4) the IEEE 1164 standard logic package, and (5) the IEEE 1029.1 WAVES test vector standards.

The IEEE has adopted and standardized VHDL as IEEE Std 1076 (Ref. 10). The standard is the *VHDL Language Reference Manual* (LRM). The VHDL DID requires the use of IEEE Std 1076. This handbook uses the VHDL LRM as its definition of VHDL. IEEE standards are revised approximately every five years; therefore, the IEEE VHDL standard released in 1988 was revised in 1993. The revised VHDL standard is the DoD-required standard until it is again revised. The LRM is described in more detail in Chapter 3.

The IEEE DASC is developing standards to support the interoperability of VHDL models. One aspect of this effort is IEEE Std 1164, which defines a standard set of values for signals that includes values for unknowns and high-impedance values. IEEE Std 1164 is discussed in Chapter 7. A second aspect is the VHDL initiative toward ASIC libraries (VITAL) (Refs. 13 and 14), which is developing a standard for use in the sign-off process for chip designs by fabrication vendors.

The JTAG is developing a standard VHDL practice to describe implementations of the IEEE 1149.1 boundary scan test circuitry (Ref. 4). This practice provides a method used to describe modifications to a low-level structural model of an integrated circuit in order to incorporate the circuitry required for a boundary scan built-in test capability. The IEEE 1149 series of standards is discussed in Chapter 8.

The WAVES IEEE Std 1029.1 (Ref. 3) is intended to create a standard representation of test vectors or waveforms for electronic devices. It uses features of VHDL to describe procedures used to generate test vectors and waveforms and to describe methods used to ensure the output of the module under test matches the required output. WAVES provides a common format used to describe test vectors for many different automatic test equipment (ATE) machines and a common output format for automatic test pattern generation software. This standard reduces the amount of work required to interface ATE machines with many VHDL parts models. MIL-HDBK-454 (Ref. 1) states that the VHDL models delivered to the Government should be compatible with WAVES and requires the use of WAVES for any test vectors or waveforms delivered with the model. The WAVES standard is discussed in Chapter 7.

1-6 OVERVIEW

In Chapter 2 the use of hierarchies in modeling computer hardware is discussed, and the concepts of behavioral and structural models of electronic systems are described. These concepts are essential to VHDL models compliant with the VHDL DID. Models with mixed levels of abstraction are discussed. Also discussed is the use of simulation to support functional correctness checking and performance evaluation. Examples of these concepts are presented.

In Chapter 3 the use of VHDL to capture the structure and behavior of electronic computers is discussed. Aspects of VHDL that support the reuse of VHDL models are presented. The development and use of libraries of VHDL descriptions for reuse of both VHDL programs within a model and between models, as well as the annotation of VHDL models with descriptive information, are described.

Chapter 4 discusses two Government documents concerning the use of VHDL: MIL-HDBK-454 (Ref. 1) and the VHDL DID, DI-EGDS-80811 (Ref. 2). The need for VHDL descriptions of all application-specific integrated circuits and all digital electronic components on the DoD qualified products list is discussed. The required structure and contents of VHDL descriptions provided to the Government, as defined by the VHDL DID, are presented. In particular, the requirement for both structural and behavioral models of each component of an electronic subsystem is described. This chapter provides guidelines to be used to tailor the VHDL DID and discusses an example of a tailored VHDL DID. This chapter also contains required annotations for VHDL models.

Chapter 5 contains a description of the construction and use of behavioral VHDL models. Common techniques used to create behavioral VHDL models, specify the timing for behavioral models, and annotate behavioral models are described. Also discussed are the usefulness of behavioral models in top-down design and the simulation of models with mixed levels of abstraction.

Chapter 6 discusses the construction and use of structural VHDL models. Common techniques used to create structural VHDL models, including automatic synthesis and schematic capture, are described. Applications of structural models for hybrid model simulation, physical design, testability analysis, and annotation with layout and testability information are also described in this chapter.

The preparation of VHDL models for simulation is detailed in Chapter 7. The process of configuring a model from libraries of component descriptions is described. Techniques that support the interoperability of models are emphasized. In component libraries these models can be combined freely to provide hybrid structural and behavioral models of systems. The development of test benches and test vectors to check the correctness and completeness of the model rather than the development of test vectors to check the correctness of the component design is discussed. Also discussed are the use of parameterized timing models and the selection of timing options for simulation.

Chapter 8 discusses issues surrounding VHDL modeling of the test and diagnostic functions of digital electronic systems. This chapter describes measures of and techniques for testability and describes different levels of testability based on the IEEE 1149 hierarchy of testing interfaces. The use of behavioral modeling to verify that the test bus and test controller systems respond properly to error conditions detected by on-chip BIT without requiring gate-level implementation details is emphasized. The use of detailed structural models as the starting point for built-in test structure generation, such as boundary scan, is discussed. This chapter also emphasizes that detailed structural models are necessary for evaluation of many testability measures.

Chapter 9 describes the preparation of a VHDL model for delivery to the Government. The contents and organization of the files delivered to the Government, as specified in the VHDL DID, are described. The files that must be delivered include not only the VHDL source models but also test vectors, annotations, certain other external files, and documentation. Chapter 9 also includes recommendations for VHDL model style and recommendations for naming files and organizing libraries.

REFERENCES

1. MIL-HDBK-454M, *General Guidelines for Electronic Equipment*, 28 April 1995.
2. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, 11 May 1989.
3. IEEE Std 1029.1-1992, *Waveform and Vector Exchange Specification (WAVES)*, IEEE Design Automation Standards Subcommittee, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1992.
4. IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary Scan Architecture*, IEEE Standards Board, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1990.
5. P. McHugh, “IEEE P1149.5 Module Test and Maintenance Bus”, *IEEE Design and Test of Computers* (December 1992).
6. IEEE Std 1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1993.
7. *Very High-Speed Integrated Circuits Final Program Report*, VHSIC Program Office, Office of the Under Secretary of Defense for Acquisition, Washington, DC, September 1990.
8. J. Hines, “Where VHDL Fits Within the CAD Environment”, *24th ACM*/IEEE Design Automation Conference Proceedings*, Miami Beach, FL, June 1987, The Institute of Electrical and Electronics Engineers, Inc., New York, NY.
9. ANSI/IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, March 1988.
10. ANSI/IEEE Std 1076-1993, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, April 1994.
11. MIL-STD-1840B, *Automated Interchange of Technical Information*, 1992.
12. K. Parker and S. Oresjo, “A Language for Describing Boundary Scan Devices”, *Proceedings of the IEEE International Test Conference*, Los Alamitos, CA, 1990, The Institute of Electrical and Electronics Engineers, Inc., New York, NY.
13. V. Berman, “An Analysis of the VITAL Initiative”, *VHDL Boot Camp*, VHDL International Users’ Forum, San Jose, CA, October 1993, VHDL Interna-

*Association for Computing Machinery

tional Users' Forum, c/o Conference Management Services, Menlo Park, CA.

14. O. Levia and F. Abramson, "ASCI Sign-Off in VHDL", *VHDL Boot Camp*, VHDL International Users' Forum, San Jose, CA, October 1993, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

BIBLIOGRAPHY

J. R. Armstrong, *Chip-Level Modeling in VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.

D. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, Norwell, MA, 1989.

R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.

D. Perry, *VHDL*, McGraw-Hill Book Co., Inc., New York, NY, 1991.

VHSIC Annual Report for 1986, AD-A191-027, VHSIC Program Office, Office of the Under Secretary of Defense for Acquisition, Washington, DC, December 1986.

VHSIC Annual Report for 1987, AD-A199-880, VHSIC Program Office, Office of the Under Secretary of Defense for Acquisition, Washington, DC, December 1987.

VHSIC Annual Report for 1988, AD-A223-725, VHSIC Program Office, Office of the Under Secretary of Defense for Acquisition, Washington, DC, December 1988.

CHAPTER 2
HARDWARE DESCRIPTION CONCEPTS

As digital electronic systems approach complexity levels of hundreds of millions of devices, the hardware architect needs techniques to reduce the design complexity to an understandable level without eliminating any design detail. Two mechanisms used to control complexity are hierarchy and abstraction. Techniques that create models of hardware by using hierarchy and different levels of abstraction are described. The concepts of structural and behavioral models of digital electronic systems are essential to very high-speed integrated circuit (VHSIC) hardware description language (VHDL) models that comply with the VHDL data item description (DID); these concepts are described in this chapter. Models with mixed levels of abstraction, in which a hierarchical model of a system contains behavioral elements at different levels of abstraction, are discussed. Also discussed are the uses of simulation to support functional correctness checking and performance evaluation. Examples of these concepts are presented.

2-1 INTRODUCTION

A hardware design is usually developed by constructing a series of models that become less abstract (and thus more implementation specific) as the design process progresses. This iterative design process is known as top-down design. The goal of this process is to allow the hardware architect the flexibility to construct and evaluate models of very different design alternatives rapidly during the early stages of the design process. In the later stages of the process, the models become more detailed, more accurate, and more difficult and expensive to change and evaluate. Thus in these later stages the architect cannot explore as many options.

Before a hardware design begins the project manager must specify milestones, i.e., when models of the design are to be completed, verified, and evaluated. Evaluation occurs as part of a tradeoff between different designs or as part of the verification of the correctness of the design. Models may be verified by simulating the models and comparing the simulation results with expected results or against each other. When the project manager specifies the milestones, he or she must clearly indicate for each milestone the level of abstraction of the model to be delivered, the approach to verification to be used, and the types of evaluations to be performed on the model. For a military contract these milestones are specified in the contract data requirements list (CDRL) and its associated DIDs. This chapter discusses some possible levels of abstraction that can be provided for hardware models. This chapter also describes the two types of models identified in subpar. 10.2.1 of the VHDL DID (Ref. 1): behavioral and structural models. Discussion of design methodologies is beyond the scope of this handbook.

Hierarchy is a method of controlling the complexity of hardware models. A hierarchical description decomposes a hardware module into modules of lesser complexity and specifies how these modules are connected together. A module represents a logical or a physical part of a larger hardware system. Interconnections represent the electrical connections between modules that are used to carry information. Hierarchies can be organized functionally or physically. Hierarchy also provides a means for incrementally developing and validating the design in a top-down fashion. In a top-down design process the hardware is partitioned into a collection of interconnected modules, behavioral models are created for each of the modules, and the complete model is verified. A second iteration of design is performed by partitioning each of the top-level modules into their components and then verifying the refined model.

Both behavioral and structural models can be developed for the same digital system. These models serve different purposes. A behavioral model describes the functions and timing of the system independently of any specific implementation. Subpar. 10.2.1 of the VHDL DID (Ref. 1) requires delivery of a behavioral VHDL model of the entire system and delivery of a behavioral model of each module of the system. A behavioral model is often classified in terms of its level of abstraction, which is determined by the functions it performs, the data types used in the model, and the level of granularity of the events that determine its timing.

A structural model describes the physical structure of a specific implementation by specifying components and their interconnections. Components are described either structurally or behaviorally. Structural models of components create another level of hierarchy. A component of a structural model described behaviorally is called a leaf module. The level of abstraction of a structural model is the same as the level of abstraction of its leaf modules if the leaf modules all have a common level of abstraction. If a structural model has leaf modules with different levels of abstraction, the structural model is a mixed level of abstraction model. Subpar. 10.2.1 of the VHDL DID (Ref. 1) requires delivery to the Government of a structural VHDL model of a hardware system. The leaf-level models of the structural model must meet specific requirements described in the VHDL DID. In the top-down design process the behavioral models at a given level become the reference models for the various choices of structural models at that level. These intermediate behavioral models should be delivered along with the subsequently created structural models.

2-2 LEVELS OF ABSTRACTION IN MODELS OF DIGITAL ELECTRONIC SYSTEMS

2-2.1 OVERVIEW

Several levels of abstraction are commonly used during the design of digital electronic systems. There are no hard and fast boundaries between levels, but standardization efforts and common usage are beginning to develop widely accepted definitions. For example, Institute of Electrical and Electronics Engineers (IEEE) Std 1164 (Ref. 2) defines data types and functions for the gate level of abstraction. The VHDL initiative toward ASIC libraries (VITAL) (Ref. 3) defines the level of granularity of timing for this level of abstraction.

Fig. 2-1 illustrates the relationship between structural, functional, and timing representations. Fig. 2-1 also shows three orthogonal axes of hardware description: functional, structural, and timing.

In Fig. 2-1 the origin represents little or no fidelity in the model; the fidelity of the structure, function, and timing aspects of the model increase along their respective axes. If any one of the three axes is deleted, one plane remains. Thus the three planes that can be created are also important. Behavioral models include function and timing but provide no fidelity in their representation of structure. This lack of structural fidelity does not mean that the behavioral models do not have structure but that the structure of a behavioral model does not faithfully represent the physical structure of the hardware being modeled. Similarly, performance models faithfully represent the structure and timing of a hardware system but do not represent the functionality of the hardware being modeled with any fidelity. The final plane is that of functional models with structure but without any fidelity in the timing of the system. Delta delay models, i.e., the delay of an operation is represented with the smallest possible delay describable in VHDL, are used for this purpose. In a top-down design the designer develops a series of models of the system with increasing fidelity. Fig. 2-1 is similar to Gajski's Y-chart (Ref. 4) but (following the VHDL DID) does not distinguish between structural and functional domains. Instead it distinguishes timing as a separate axis.

Table 2-1 lists some of the levels of abstraction in common use. (Table 2-1 is similar to other tables in the literature (Refs. 5, 6, and 7).) In a top-down design process the hardware architect starts at the level of abstraction that makes sense for the design problem to be solved. Models at lower levels of abstraction are used for the incremental refinement of the model. The gate level is the lowest level of abstraction typically used in a VHDL design. At the lowest level the digital electronic system is not treated as a digital system at all. Instead the circuits are modeled as analog devices, and the waveforms produced by the system are currents and voltages, not logic values. Although there has been experimental work in modeling analog systems using VHDL, it is not common practice. Other tools, such as SPICE (a public domain integrated circuit simulation program), are used at this level of modeling.

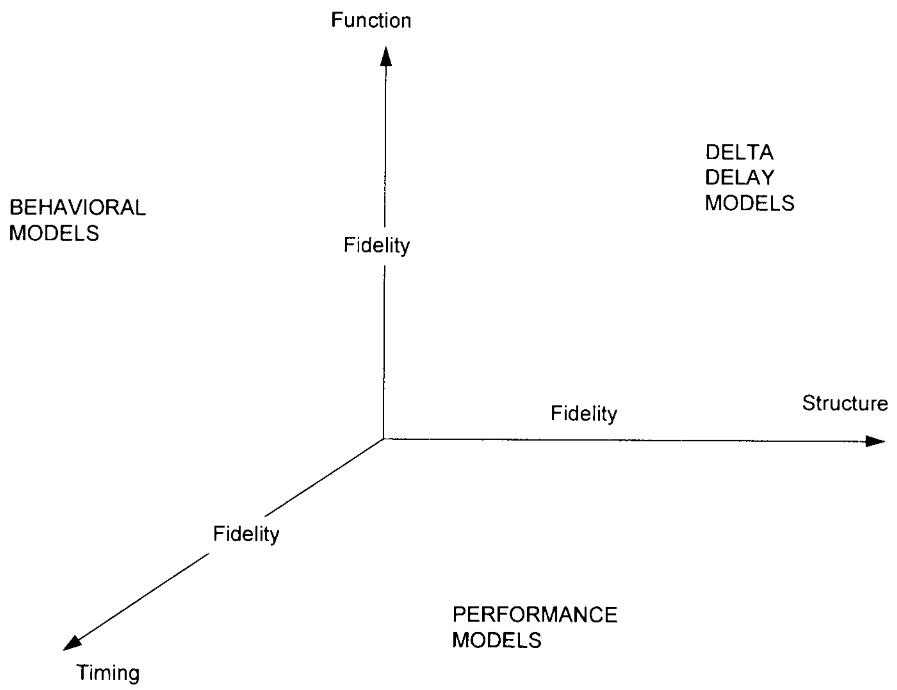


Figure 2-1. Functional Models, Structural Models, and Levels of Abstraction

TABLE 2-1. FEATURES OF BEHAVIOR, STRUCTURE, AND TIMING AND DIFFERENT LEVELS OF ABSTRACTION

LEVEL OF ABSTRACTION	TYPICAL BEHAVIORAL MODEL FUNCTIONS	TYPICAL STRUCTURAL MODEL COMPONENTS	TYPICAL TIMING MEASURES
Network	Message send Message receive	Processors memories Network elements	Message response time
Algorithmic	Signal processing Primitive operations (e.g. filter, fast Fourier transform)	Processors Memories Busses	Throughput
Instruction Set Architecture	Instruction level functions (e.g. Add, Mpy)	Program-accessible registers	Instruction times
Register Transfer	Register-arithmetic and logic unit (ALU) Register operations (e.g. Load Accumulator)	Registers Internal busses ALUs	Clock times
Gate	Boolean operations (e.g. AND, OR, NOT)	Gates Flip-flops	Gate delays
Analog	Differential equations	Transistors, resistors, etc	Actual time

2-2.2 NETWORK MODELS

The highest level of abstraction represented in Table 2-1 is the network model, also known as a processor memory switch (PMS) model (Ref. 8). The primitive components of a structural model at this level of abstraction are processors, memories, and switches; switches include interface modules as well as switching components in a switched network or routing components in a packet switching network. The time units used are application specific but are related to the response time of the hardware to application stimuli and to throughput rates for application-specific units of work. This level of model is usually developed in order to make tradeoffs between alternative system architectures and to assess the risk of a design by finding potential bottlenecks or weak points in the design. It may also be used as a proof of concept to demonstrate that an architectural concept is feasible. This level of model may also be used to specify interface protocols for components and to demonstrate that the components will be able to work together. A model at this level may become the arbiter for deciding whether variations in designs will be tolerated.

This is the level of abstraction at which two special forms of VHDL models are often created and used: performance models and interface models.

2-2.2.1 Performance Models

Performance models at this level are used to understand and balance the processing load and the input/output (I/O) requirements of multiprocessor systems and their interconnects.

Performance models may provide only timing information and thus may not simulate the functions of the system. The designer can use these models to estimate response time and component utilization and to find potential performance bottlenecks in a design.

A performance model is useful for demonstrating the feasibility of a system architecture, but it is not a sufficient behavioral model for delivery under the terms of the VHDL DID. However, a contract monitor could require a performance model during the concept exploration stage of the development of a weapon system.

2-2.2.2 Interface models

Interface models* combine high-level and incomplete models of the processor and memory components with detailed and complete bus or network interface modules. The model of a processor used in an interface model is designed to provide appropriate workloads for the busses or interconnects in terms of the size and frequency of messages sent and received. On the other hand, the model of the interface is very detailed, and the function and timing are accurate specifications of the interface protocol.

Even though an interface model is useful for demonstrating the compatibility of components, it is not a sufficient behavioral model for delivery under the terms of the VHDL DID. However, a contract monitor could request an interface model during the concept exploration stage of the development of a weapon system.

*These models are also known in industry as bus functional models.

2-2.3 ALGORITHMIC MODELS

An algorithmic model describes the functions of a system in a "program-like" or algorithmic manner. Because the inputs and outputs of an algorithmic model are not usually described at the bit level, an algorithmic model will not necessarily provide a completely accurate model of the external interface to the system. However, it will provide the same overall functionability as a register-transfer-level (RTL) or gate-level model. For example, an algorithmic description of a floating-point processor performs all the functions of the processor but uses a simulator-dependent representation of the floating point numbers. If the floating point format of the simulator is different from the floating point format specified for the system being designed, the algorithmic model may not produce the same answers, even at the abstract level, as the hardware being designed. However, the values produced by the simulator would be accurate enough to evaluate the quality of the design. Thus an algorithmic description can use the primitive data types and operations that the simulator provides as a way to simplify the description and increase the speed of the simulation at the cost of precision, accuracy, and the use of formats that are potentially different from the actual hardware to be developed.

An algorithmic model can be used to verify that the functions of a digital system are correct, but depending on the number representation used, it may not provide the bit-accurate results needed to verify outputs from the simulations of more detailed models.

2-2.4 INSTRUCTION SET ARCHITECTURE MODELS

An instruction set architecture (ISA) model includes the complete set of instructions recognized by a given processor (Ref. 8). An ISA model provides the externally visible state and functions that the processor can perform. The timing of an ISA model is typically defined in terms of the times required to perform each of the instructions of the processor instruction set. This timing may be expressed in terms of processor clock cycles or in absolute time, e.g., microseconds. ISA models can support simulated execution of software if the compilers and operating system load modules are available.

An ISA model accurately describes all the functions and data types provided by the hardware that are accessible to the user. In particular, a correct ISA model of a programmable device correctly executes any valid program for the device. Thus an ISA model of a programmable device can be used to debug software written for that device, and inputs and outputs of an ISA model can be translated into forms that are completely compatible with more detailed models. An ISA model of a programmable subsystem may therefore be used in combination with more detailed models of other subsystems. ISA models are appropriate forms of behavioral models for delivered systems because they are accurate to the bit level and thus are compatible with both the behavioral and structural models of all adjacent components.

2-2.5 REGISTER-TRANSFER MODELS

A register-transfer-level model describes the functions and data types accessible to the user of the system and includes descriptions of the internal memory (or registers) and the internal data paths of the hardware. Some registers in a typical central processing unit (CPU) are accessible to the programmer and therefore are part of an ISA description, but some registers may not be directly accessible to the programmer, such as a memory address register, cache memory, or microcode instruction register. This internal memory structure is part of what distinguishes different implementations of the same architecture and thus is not appropriate in an ISA model except as an aid to understanding the model.

Register-transfer-level models use arithmetic and logical operations such as add, subtract, and compare. These operations access data in registers and return results to registers. Since the registers are clocked memory elements, the clock time is the key timing measure.

The register-transfer-level model is a particularly important class of models because commercially available hardware synthesis technology can be used to generate detailed integrated circuit designs from appropriate register-transfer-level models. Synthesis of gate-level structural models from register-transfer-level models is discussed in Chapter 6.

2-2.6 GATE-LEVEL MODELS

Gate-level models are the lowest level of abstraction generally modeled using VHDL. Gate-level models are structural models constructed with primitive elements (also known as the leaf-level modules) that represent Boolean logic functions, e.g., AND, OR, NOT, and basic logic functions such as flip-flops and multiplexors. IEEE Std 1164 (Ref. 2) provides a standard set of primitive functions and data-type definitions for gate-level models. The VITAL initiative (Ref. 3) is working on a standard set of timing definitions for this level of model. The typical timing measures for this level of abstraction are gate delays, which are dependent upon the technology used to implement the design and may also be parameterized to reflect the ambient temperature of the device, the power applied to the device, and the layout of the circuit in terms of both feature size and the lengths of the wires or vias connecting the circuits. Gate-level models are considered low-level structural models because the behavior of the leaf modules in these models is simple and well-understood. Structural models are discussed in par. 2-4. Gate-level models are typically technology dependent, particularly with respect to timing. They are the basis for application-specific integrated circuits (ASIC) foundry sign-off, where they are used to verify the behavior of the integrated circuits that will be manufactured.

2-2.7 USES OF ABSTRACTION AND HIERARCHICAL DECOMPOSITION IN THE DESIGN PROCESS

During the process of designing a system, the system may be represented at several levels of abstraction. “Top-down design” and “bottom-up design” refer to the sequence in which models at different levels of abstraction and different levels of hierarchical decomposition are developed. When a new model of a design is to be created, the designer can choose to define a new level of hierarchy or to change the level of abstraction, or some combination of these approaches can be chosen. Top-down design is the process of (1) partitioning a module into submodules, (2) defining the interfaces between the submodules, (3) allocating resources and requirements to those submodules, (4) verifying that the partitioned form of the design is consistent with the unpartitioned design in both function and performance and that the resource and requirements constraints have been met, and finally (5) recursively applying the same process to the components. During this process the design evolves from the highest level of abstraction to the lowest level of abstraction.

This process can be captured in VHDL. To do so, a behavioral model of a module is created and annotated with attributes that reflect quantitative resource and requirements budgets. The partitioning of the module is represented by converting the behavioral model into a structural model, in which the components of the structural model define the submodules and the ports and port mappings specify the interfaces between the submodules. Verification of the design is done in VHDL through, for example, simulation. VHDL provides a strong type-checking capability, which aids verification. VHDL tools can check the consistency of the interfaces between submodules at analysis time.

Bottom-up design is the process of creating higher level models by connecting together known lower level models. A classic example of bottom-up design is the process of creating combinational logic functions by connecting gate-level functions. VHDL supports bottom-up design with structural models, in which the known lower level models are specified by component declarations and the interconnections of the components are specified by the port maps in the component instantiation statements.

The process of transforming a model at one level of abstraction into a model at a lower level of abstraction is called synthesis (Ref. 9). Refining the hierarchy of a structural model is an effective way to transform a high-level model into a low-level model. For example, an ISA model can be converted into a register-transfer-level model by creating a register-transfer-level model for each leaf module in the ISA description. The program-accessible registers in the ISA model are defined as physical components, and the internal busses connecting these registers and the ALU are specified. The implementation of the instruction fetch and decode mechanisms and the translation of a logical address to a physical address is defined in terms of physical components. Also the RTL model of the ALU is created. Thus a more specific model is created by replacing the top-level behavioral model with a structural model or with a model at a lower level of abstraction. This process is done most easily if the functional hierarchy of the behavioral model is similar to the physical hierarchy of the implementation. For the Sobel processor described in subpar. 2-3.3, the functional decomposition is consistent with a physical decomposition at the top level. In particular, the four filter functions also occur as physical components in the parallel implementation. Synthesis is a difficult process because it is a many-to-many mapping. For example, a behavioral model may have two separate functions that compute memory addresses and summing pixels, but the corresponding RTL model may use the same ALU for both. On the other hand, calls to the same pixel add routine may be allocated to different ALUs to achieve parallelism.

The most common way to check the functional correctness of a hardware model is through simulation. The VHDL approach to checking functional correctness uses a test bench. A test bench is a part of a VHDL model that reads or generates a set of test vectors and sends them to the module being tested. The test bench collects the responses made by the module to the test vectors and checks the results produced by the module against a specification of correct results. Simulation can be used in this way to verify that the model is functionally correct at least to the extent that it provides correct responses to the input test vectors.

Simulation can also be used to estimate the performance of the finished hardware. Because a behavioral model often includes timing information, simulation can be used to verify that the model performs within its performance limits over a variety of external test conditions, e.g., changes in temperature or changes in voltage. The simulation results in trace files listing the names of signals, the times that the signals change values, and their new values. These trace files can be postprocessed to estimate the throughput of the hardware, the delay times from input to output, and the amount of time that different components are kept busy during the simulation. Simulation results can be used to identify performance problems in the hardware design, such as insufficient throughput, excessive response time to stimuli, and the possible race conditions that make the behavior of the hardware vary erratically.

2-3 BEHAVIORAL DESCRIPTIONS OF HARDWARE DESIGNS

2-3.1 THE PURPOSE OF BEHAVIORAL DESCRIPTIONS

Behavioral models provide a description of the function of a hardware system independent of any particular implementation. A behavioral model is a “black box" in the sense that any internal hierarchy or structure is provided as an aid to description or understanding and is not necessarily meant to serve as a definition of the organization of any implementation.

Behavioral models play a key role in top-down system design and provide an important form of documentation of a hardware system. Designers can use behavioral models of subsystems to evaluate the performance and functional correctness of the system architecture. In these models, timing budgets are used in the subsystem behavioral model. Simulations of the behavioral models of the subsystems can demonstrate that the subsystems meet their timing budgets and therefore demonstrate that the system architecture is feasible.

Designers can use behavioral models to construct prototypes of systems before an implementation has been specified. Prototypes help validate a proposed design by allowing the designer to understand the functions, timing, and interactions of the proposed hardware subsystems in a system context. Behavioral models can help the customer understand the potential risks associated with particular implementation decisions. For example, a behavioral model may indicate which parts of a design are likely to be the slowest, the largest, or the most complex. These risk indicators can help the customer evaluate proposed implementations.

Behavioral models also play an important role in the verification of an implementation by defining correct response to stimuli. A designer creates a set of functional test stimuli, or test vectors, and simulates the behavioral model using the test vectors to generate the correct responses to the test vectors. The designer then creates an implementation model and simulates the implementation model using the same test vectors used to simulate the behavioral model. Finally, the designer verifies that the implementation model is consistent with the behavioral model by comparing the results generated by the implementation model with the results generated by the behavioral model. If the results are equivalent, the implementation model represents a correct implementation of the functions and timing of the behavioral model.

Commercial computer-aided design (CAD) tool vendors currently provide or sell synthesis tools that accept register-transfer-level behavioral models and generate gate-level structural models and chip designs including logic designs and layouts. Research is continuing on raising the level of abstraction of the input to synthesis tools. Behavioral models that are compatible with synthesis tools are particularly valuable to the Department of Defense (DoD) in system maintenance, upgrade, and replacement of obsolete parts. For example, if the DoD needs to replace an electronic circuit that is no longer available and has a complete VHDL behavioral description of the circuit compatible with a synthesis tool, it may be possible to generate at relatively low cost a replacement circuit that is optimized and validated with respect to some currently available fabrication process.

By capturing the system in an implementation-independent, simulatable form, behavioral models provide an important starting point for system upgrades and improvements to add functions, reduce size, weight, or power, and keep systems up with the state of technology advances. Behavioral models also provide a model for hardware that conceals the proprietary implementation details. This capability allows the implementor to protect the implementation design while completely describing the system function. The behavioral model of a proprietary hardware system may include implementation-specific information such as timing, power consumption, weight, or heat dissipation while protecting the implementation details.

Behavioral models at a high level of abstraction are also usually more efficiently simulated than detailed structural models. High-level behavioral models can often achieve simulation times two or three orders of magnitude shorter than those for detailed structural models. Generally, simulation times are closely related to the number of events scheduled by the simulator. Reducing the number of events by a factor of *N* is likely to decrease the simulation time by a factor greater than *N*. This decrease is possible because (1) VHDL simulators typically store events in queues, (2) simulation time is the product of the number of events simulated and the average time to insert events in the queue, and (3) the average insertion time is a function of queue size. Detailed structural models may require hundreds, thousands, or even millions of events to be scheduled to complete a function; a high-level behavioral block may be able to compute the same function in a single event. To have a useful behavioral model of a subsystem that also improves simulation speed, the model must be compatible with both structural and behavioral models of all adjacent subsystem components. Achieving this requirement allows the modeler to mix and match structural and behavioral models in order to configure a simulation model emphasizing a particular portion of the system. The modeler uses a detailed structural model of the part of the system that is of interest and high-level behavioral models of other parts of the system to minimize simulation time. These mixed abstraction models are described in greater detail in par. 2-5.

2-3.2 THE USE OF HIERARCHY IN BEHAVIORAL DESCRIPTIONS

Because the behavior of a digital electronic system may be very complex, some form of hierarchy and structure is often necessary to make a given behavioral model comprehensible to humans. The hierarchy of a behavioral description should be fashioned to improve understanding rather than to describe an implementation. For this reason, a modeler should prefer decomposition of a behavioral model into functions and subfunctions over physical decompositions into boards, integrated circuits, registers, and gates. One part of an object-oriented hierarchy style is a definition of functions that provide all access to a data structure. VHDL packages are well suited to this style of decomposition. This approach supports information hiding since the details of the data structure are not known to the user, only to the developer of the access routines and the data structure. For example, memory is a data structure that could be modeled in VHDL using either a very large array or access types. A package of functions for reading and writing to the memory could be used to provide the same interface to either implementation and could be expanded to include functions for computing the physical address of a word in memory by using the different addressing modes of the processor. Applying object-oriented techniques to VHDL is currently being researched (Ref. 10).

In a VHDL context the hierarchy of behavioral models is specified in terms of the hierarchy of function calls, which may be used to support object-oriented programming features, particularly data abstraction and information hiding. The hierarchy of function calls also may be used to define a decomposition of the functional requirements for the system being modeled.

Behavioral models of a system may be structured hierarchically for the following reasons:

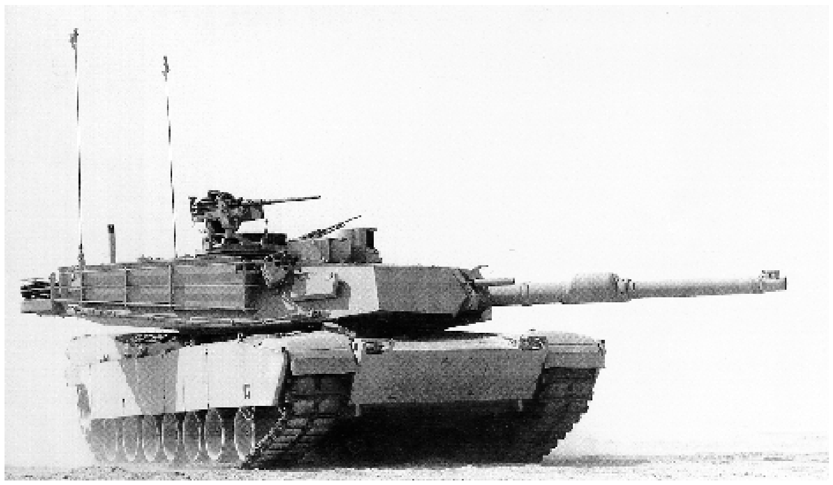
1. Hierarchical models help to simplify and organize a behavioral model into comprehensible sections. A hierarchically structured behavioral model reflects good software engineering practice by partitioning the description into simple functions that may be reused. A good behavioral model emphasizes comprehension, even at the cost of some efficiency. VHDL provides several mechanisms to improve the comprehensibility of behavioral models including functions and the overloading of infix operators so that common mathematical functions can be defined by the user for different data types. These mechanisms are described in Chapter 3.

2. Hierarchical behavioral models can reuse functions and procedures. The sharing of functions and procedures within and between components is an important aspect of good modeling practice. VHDL provides functions, procedures, and packages containing data-type definitions, functions, and procedures as mechanisms that promote reuse both within and between processes. These mechanisms are described further in Chapter 3.

3. Hierarchical models can make use of graphical block diagrams as an aid to understanding the textual behavioral model. This approach is particularly valuable when a CAD tool is used to generate a VHDL behavioral model from a graphical block diagram.

2-3.3 EXAMPLE OF A BEHAVIORAL DESCRIPTION

In this subparagraph a hierarchical behavioral model of an edge detection processor, from Ref. 11, is described. Edge detection is a common filtering procedure used in many military and civilian image processing systems including automatic target recognition systems. Fig. 2-2 shows a


(A) Input Image

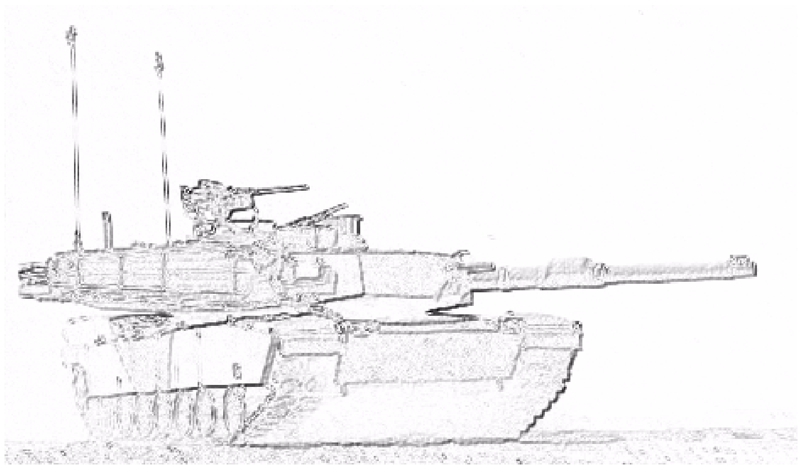

(B) Edge Magnitude Output Image

Figure 2-2. Example Input Image and Edge Magnitude Output of an Edge Detection Processor (Ref. 11)

test input image and the edge magnitude output of such a system.

Fig. 2-3 shows the hierarchy of function calls for a behavioral model of the edge detection system. At the top of the hierarchy is the edge detection processor, which is a behavioral model. This process calls six functions: the horizontal filter, the vertical filter, the left diagonal filter, the right diagonal filter, the magnitude function, and the direction function. The first four of these functions in turn make use of another function, the weight function.

The behavioral model of the edge detection system makes use of data abstraction to simplify the modeling of the system. The VHDL definitions of the data types for this behavioral model are shown in Fig. 2-4. This VHDL package declaration describes the pixel data type, the index types that are used to address pixels in the image, and the data type for the image, which is defined as a two-dimensional array of pixels. The directional output of the system is described as an enumerated type that lists the eight points of the compass. A scan line is defined as a subtype of the image data type. Pixels are defined in terms of the built-in data-type integer. During implementation the definitions of the pixel data type can be refined to specify the number of bits in the word. Using data abstraction the developer allows this implementation decision to be abstracted out of the behavioral model. Fig. 2-4 also specifies the data types for the parameters of the functions used to implement the system including the four filter functions, the magnitude and direction functions, and the weight function.

Fig. 2-5 specifies the interface to the edge detector in VHDL, i.e., as an entity interface. The input to the system is a sequence of pixels that are loaded in scan line order. The output from the system is a pair containing magnitude and direction values for each pixel in the output. This entity interface is common to both behavioral and structural architecture bodies and subsequently can be configured with either.

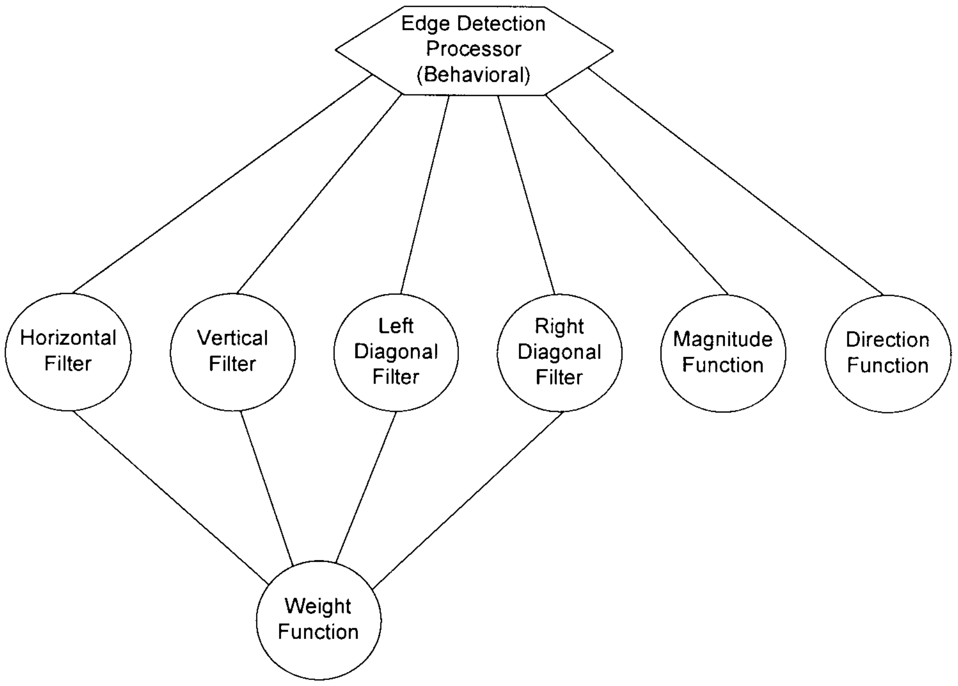


Figure 2-3. Hierarchy of Functions in a Behavioral Model

```
package image_processing is
constant num_lines: natural := 512;
constant line_len: natural := 512;
type x_index is range 1 to line_len;
subtype x_out_index is x_index range 2 to line_len - 1;
type y_index is range 1 to num_lines;
subtype y_out_index is y_index range 2 to num_lines - 1;
subtype pixel is integer;
subtype filter_out is integer;
type direction is (N, NE, E, SE, S, SW, W, NW);
type image is array(x_index, y_index) of pixel;
type scan_line is array(image'range(1)) of pixel;
type pix3 is array (1 to 3) of pixel;
function horizontal_filter
    ( A: image;
      I: x_index;
      J: y_index )
    return filter_out;
function vertical_filter
    ( A: image;
      I: x_index;
      J: y_index )
    return filter_out;
function left_diagonal_filter
    ( A: image;
      I: x_index;
      J: y_index )
    return filter_out;
function right_diagonal_filter
    ( A: image;
      I: x_index;
      J: y_index )
    return filter_out;
function magnitude
    ( H,V,LD,RD:  filter_out)
    return pixel;
function direct
    ( H,V,LD,RD:  filter_out)
    return direction;
function weight
    ( X1,X2,X3: pixel)
    return filter_out;
function shift
    ( A: pix3;
      B: pixel)
    return pix3;
end image_processing;
```

Figure 2-4. Image Data Abstractions and Functions

```
   -- The sobel algorithm library contains the packages,
   -- entity declarations, and architecture bodies for
   -- the algorithm level model of the sobel processor.
library sobel_algorithm;
use sobel_algorithm.image_processing.all;
use sobel_algorithm.timing.all;
   -- The IEEE library and the 1164 standard logic
   -- package are used in the algorithm model only
   -- for the clock.
library IEEE;
use ieee.std_logic_1164.all;
entity edge_detector is
      port (P:     in pixel;
            Clock: in std_ulogic;
            E:     out pixel;
            D:     out direction );
end edge_detector;
```

Figure 2-5. Interface Specifications for an Edge Detection Processor

This entity interface references two VHDL libraries: the IEEE library, which contains the standard logic package, and an application-specific library called `sobel_algorithm`. This entity interface uses one package from this second VHDL library, the one containing data-type definitions and function specifications for this application. The clock signal uses the `std_ulogic` data type from the IEEE package. Separating the application-specific details such as the scan line size and number of scan lines per image frame into a package makes it easier to reuse the design entity in different applications. Collecting these details in one place also makes it easier to modify the entire design, should that ever be necessary.

Fig. 2-6 describes the behavior of the edge detector in VHDL. The architecture body contains a single process. The body of the process consists of two sets of nested loops. The first set of nested loops creates an internal buffer for a frame of the image by reading the pixels in scan line order, one pixel per clock. The timing of the input is controlled using the `rising_edge` function that is specified in the IEEE Std 1164 standard logic package (Ref. 2). The second set of nested loops produces the outputs in scan line order by calling on functions to compute the output values. The functions called by the second loop refer to pixels stored in the internal frame buffer.

The output of pixels in the second loop is delayed by the `pixel_output_delay`, which is a constant in the timing package. This approach to implementation-independent timing has its limitations. In this example, this abstract behavior does not capture some of the benefits of pipelining, in which some resulting pixels may be sent out of the edge detector before some input pixels arrive.

Fig. 2-7 describes three of the functions from the image-processing package that are used by the edge detector: the horizontal filter, the vertical filter, and the weight function. The calling relationship between the horizontal and vertical filters and the weight function shown in Fig. 2-3 is the result of the weight function calls in the bodies of the horizontal and vertical filters. The other functions in the image-processing package (not shown but required) are implemented in a similar manner.

```
architecture behavior of edge_detector is
begin
   sobel: process
   variable A:  image;      -- Internal frame buffer for image
   variable H:  filter_out; -- Temporary storage for results of
                            -- horizontal filter
   variable V:  filter_out; -- Temporary storage for results of
                            -- vertical filter
   variable LD: filter_out; -- Temporary storage for results of
                            -- left diagonal filter
   variable RD: filter_out; -- Temporary storage for results of
                            -- right diagonal filter
   begin
      -- Construct a complete image frame by reading
      -- in the pixels in scan line order
      for i in x_index loop
         for j in y_index loop
            wait until rising_edge(Clock);
            A(i,j) := P;
         end loop;
      end loop;
assert (false) report "array read in";
      wait for pixel_output_delay;
      -- For each pixel in the output image
      -- compute the values of all the filters,
      -- then use these filter values to compute
      -- the magnitude and direction outputs
      for i in x_out_index loop
         for j in y_out_index loop
            wait until rising_edge(Clock);
            H  := horizontal_filter(A,i,j);
            V  := vertical_filter(A,i,j);
            LD := left_diagonal_filter(A,i,j);
            RD := right_diagonal_filter(A,i,j);
            E  <= magnitude(H,V,LD,RD);
            D  <= direct(H,V,LD,RD);
          end loop;
       end loop;
    end process sobel;
end behavior;
```

Figure 2-6. Behavioral Model for an Edge Detection Processor

```
package body image_processing is
function horizontal_filter
    ( A: image;
      I: x_index;
      J: y_index )
    return filter_out is
begin
    return weight( A(I-1,J-1), A(I,J-1), A(I+1,J-1))
           - weight(A(I-1,J+1), A(I,J+1), A(I+1,J+1));
end horizontal_filter;
function vertical_filter
    ( A: image;
      I: x_index;
      J: y_index )
    return filter_out is
begin
    return weight( A(I-1,J-1), A(I-1,J), A(I-1,J+1) )
           - weight(A(I+1,J-1), A(I+1,J), A(I+1,J+1) );
end vertical_filter;
function weight
   ( X1,X2,X3: pixel)
    return filter_out is
begin
    return X1+ 2 * X2 + X3;
end weight;
function shift
    ( A: pix3;
      B: pixel)
    return pix3 is
begin
    return A(2 to 3) & B;
end shift;
–– Other functions are omitted
end image_processing;
```

Figure 2-7. Example Functions for a Behavioral Model

2-4 STRUCTURAL DESCRIPTIONS OF HARDWARE DESIGNS

2-4.1 THE PURPOSE OF STRUCTURAL DESCRIPTIONS

The primary purpose of a structural model is to capture the physical organization of a particular implementation. To capture the physical organization, the hierarchy of a structural model should follow the hierarchy of the physical design. Structural models of hardware are traditionally represented by schematic diagrams of the connections between physical components. When VHDL is used to represent structural models, VHDL components are used to describe the physical components (such as integrated circuits and boards), and signals are used to describe the electrical connections between physical components. VHDL uses ports to describe the interfaces between signals and components. Ports allow the reuse of components in the same way that formal parameters allow the reuse of functions.

Low-level structural models can provide detailed documentation of a particular implementation, but because of this implementation dependence, they are not appropriate for specifications to be used in the competitive procurement of new designs.

Structural models may be required in order to allow analysis of the design that is specific to the implementation. For example, the VHDL DID requires structural models to have sufficient detail to support logic-level fault simulation. Fault sets for digital hardware are typically defined in terms of failures at the bit level in the gate-level descriptions of the hardware. To evaluate the effectiveness of a set of test vectors, single-bit faults are injected into a gate-level structural model during simulation. This faulty simulation output is then compared to the output of the fault-free model to check the ability of a test vector to distinguish between the faulty and flawless models. This process is described in more detail in Chapter 8.

Gate-level structural models are required to synthesize built-in test structures. The boundary scan approach requires that combinational logic be separated from sequential logic by fully observable and controllable test nodes. Computer-aided engineering (CAE) tools are emerging that can synthesize the boundary scan test nodes and their interconnections if the system separates combinational and sequential logic at the gate level. This synthesis and its corresponding test-vector generation require detailed structural models at the gate level.

2-4.2 THE USE OF HIERARCHY IN STRUCTURAL DESCRIPTIONS

Hierarchy is important in structural models as a means of conveying the logical or physical decomposition of the hardware. Subpar. 10.2.3 of the VHDL DID (Ref. 1) requires that the hierarchy of a structural model follow the hierarchical organization of the physical design. This organization is useful in several ways. A hierarchical structure that corresponds to the physical organization supports the design and acquisition of upgrades by identifying physical interfaces between components that can be developed separately, and it can document maintenance issues. For example, a physically oriented hierarchical model reflects the organization of the hardware into line-replaceable modules (LRMs). Also a hierarchical structure that corresponds to the physical organization documents boundaries between different technologies. A good structural hierarchy reflects the composition of boards into an interconnected set of integrated circuits with specific layout and routing. This partitioning facilitates the use of appropriate CAD tools for the design of integrated circuits and the design of boards.

The interconnection of components in a structural model should represent the physical interconnections. For example, each data-carrying wire on the board should have a corresponding signal in the VHDL model. The relationship between signals and wires may not be one-to-one, e.g., a 16-bit bus, which contains 16 individual wires, may be represented by a single signal in the VHDL model. This correspondence is one way of checking the consistency of the model with the physical hardware.

The physical hierarchy for a military digital electronic system has several levels that should be represented in a structural model. For example, a specification of a military system written to conform with MIL-STD-490 (Ref. 12) partitions the system into segments and the segments into configuration items including hardware configuration items (HWCIs). The HWCIs are further partitioned into prime items and critical items. A structural model of a digital electronic system should be consistent with this partitioning.

Hardware block diagrams and schematic diagrams are graphical representations of hardware data flow. VHDL provides mechanisms to represent this same hardware data flow formally. When a hardware block diagram is used to provide graphical documentation for a VHDL structural model, the following guidelines should be observed to make the relationship between the VHDL model and the block diagram clear and unambiguous:

1. There should be a one-to-one correspondence between the blocks in the diagram and component instantiations in the VHDL model.
2. Block names should be directly translatable into VHDL component instances. Either `InputBus` or `input_bus` is acceptable.The VITAL specification recommends names that use capital letters to separate words rather than underscores.
3. There should be a one-to-one relationship between interconnections in the block diagram and signals in the VHDL source program.
4. If the interconnections in the block diagram are labeled, the labels should be directly translatable into VHDL signal names.
5. All signals referenced in a VHDL process should have a corresponding interconnect in the block diagram.

Guideline 1 requires distinct instance labels but allows components to be reused. For example, the edge detection processor described in subpar 2-3.3 reuses the adder component within all of the filters. Guideline 2 encourages the user to translate automatically graphical block names into instance labels. (The block names may contain blanks that are translated into underscores in the VHDL source program.) Guideline 3 encourages the user to implement multibit busses and interconnects as bit vectors or higher level data types. For example, the behavioral model of the edge detection processor uses the integer data type for its signals. In a structural model these signals are translated into bit vectors. The use of single signals is essential for the mixed level of abstraction models described in par. 2-5.

A number of commercial CAD tools have the capabilities to create schematic representations of VHDL structural models and to create VHDL structural models directly from the schematic representation of the CAD tool.

2-4.2.1 Hierarchical Decomposition Based on Physical Elements

During design the digital electronic system is partitioned into subsystems. At the top level the system as a whole is described. The next level is a partitioning of the system into subsystems. The structural model should follow the partitioning described for the system into HWCI as described in the Level A specifications (Ref. 12). A structural model should preserve the partitioning into HWCIs of the physical system because it is a standard unit for acquisition.

The structural model should also be consistent with the physical hardware at the level of the line-replaceable unit (LRU). LRU partitioning is significant for logistics and support because it represents the basic unit used to maintain the system in the field. Any changes in boundaries between LRUs can have a significant effect on logistics and support; therefore, the structural model should accurately represent those boundaries. Furthermore, LRUs are important boundaries of the system for diagnostic and testing purposes. Field maintenance personnel must be able to isolate faults to an individual LRU. Thus a structural model should be able to simulate built-in test (BIT) diagnostic capabilities and interfaces to external test equipment at the level of its LRUs.

Another level of partitioning that should be represented in a structural model is the board. Partitioning the structural model to correspond to the physical partitioning of the hardware into boards assists in the automatic placement and routing of boards and in the thermal and power analysis of the boards. Furthermore, delays between boards are likely to be much greater than delays within a specific board; this difference must be represented. In some cases the boards may be LRUs, so the components in the LRU-level partitioning and the board-level partitioning may be the same.

Partitioning of the structural model should also correspond to the partitioning of a board design into multichip modules (MCMs) and integrated circuits (ICs) as appropriate. Different CAD tools and optimization criteria may apply at the MCM/IC level versus the board level, so partitioning of a structural model to represent MCMs can aid in the synthesis, analysis, and optimization of a design using MCMs.

Partitioning of the structural model should also correspond to the partitioning of an MCM design into packaged very large-scale integrated (VLSI) circuits. Because packaged VLSI circuits are the lowest possible practical level for repair through replacement, isolation of faults to specific integrated circuits is an important design consideration. Also, if the model accurately represents the boundaries of VLSI circuit packages, VLSI CAD tools can be used to synthesize, analyze, and optimize VLSI circuits.

Structural models for components of a circuit should also follow the partitioning used by the CAD tools to design the circuit. For example, the hierarchy of the structural model of a VLSI circuit should follow the boundaries of standard cells or macrocells used by the CAD tool. In general, if a CAD tool is used to design a circuit and to generate a VHDL model automatically for the circuit, the generated description follows the hierarchy of the design. A CAD tool that flattens a design hierarchy before producing a structural model of the design should not be used to generate models for delivery to the Government. Using CAD tools to generate detailed and hierarchical structural models is a recommended practice since it reduces costs and helps to keep the model consistent with the physical hardware.

2-4.2.2 Leaf Modules in a Hierarchical Structural Description

If a component is represented by a behavioral model and does not have a structural model, the component is called a leaf module. Subpar. 10.2.1.1 of the VHDL DID (Ref. 1) specifies three valid leaf module options:

1. Modules selected from a Government list of valid uses of leaf modules referenced or contained in the contract
2. Modules corresponding to a collection of hardware elements that together exhibit a stimulus-response behavior but whose interaction is best modeled at the electrical or physical level
3. Modules whose detailed design has not yet been completed but whose behavior is required as a delivery disclosure at specified times during the contract.

The first option for a leaf module allows the contractor to use models from a Government source of validated models. The Government requires VHDL models for the electronic components delivered to it. These requirements are discussed in Chapter 4. Once these models have been validated, they can be supplied to contractors for use in VHDL models of hardware systems that use the products. The Government and the contractor may also negotiate to include other VHDL models, such as models not in the qualified products list (QPL) that are developed by the contractor or by other Government contractors. These negotiations must be reflected in the tailored VHDL DID for the specific contract.

The second option identifies a common set of primitive elements used in designs whose elements are not easily described accurately with VHDL behavioral models. As described in subpar. 10.2.1.1 of the VHDL DID (Ref. 1), these elements include digital logic gates, analog circuit blocks, and power supplies. Functional models of digital logic gates are defined as part of the IEEE Std 1164 (Ref. 2) specification of standard signal formats. This specification includes truth tables and a resolution function for using a nine-value state/strength logic system for AND, OR, NOT, NOR, NAND, and XOR. This functional specification is being augmented with timing information and standard formats for back-annotation by the VITAL effort (Ref. 3).

The third option is designed to cover situations in which the Government wants VHDL models delivered during the design cycle, i.e., before design of all of the components has been completed. In this case high-level behavioral models may be used as leaf modules to specify the current state of the design. As the design progresses into more detail, these behavioral models are augmented with structural models.

2-4.3 EXAMPLES OF STRUCTURAL DESCRIPTIONS

In this subparagraph two examples of structural VHDL models are presented: one at algorithmic level and one at a register-transfer level. The algorithmic model uses the data-type definitions and some of the functions of the `sobel_algorithm` library presented in subpar. 2-3.3. The entity interface declarations and architecture bodies for this level of model are included in this library. The register-transfer-level model uses different data-type definitions, in which the number of bits in each word is specified. These definitions and the entity interfaces and architecture bodies that reference these packages are in the `sobel_structure` library.

2-4.3.1 Algorithmic-Level Structural Description

Fig. 2-8 shows a hierarchy for an algorithmic structural model of the edge detection system described in subpar. 2-3.3. This model is at the algorithmic level because the data types have not yet been refined to bit vectors; therefore, the inputs and outputs of the model are not bit-for-bit representations of the inputs and outputs of the real device. However, the structural model does reveal much of the physical organization of the system as it will be implemented. As shown in Fig. 2-8, this model continues to use some of the elements of the behavioral model, particularly the weight function, and it uses the data-type definitions previously used in the behavioral model. This structural model implements the

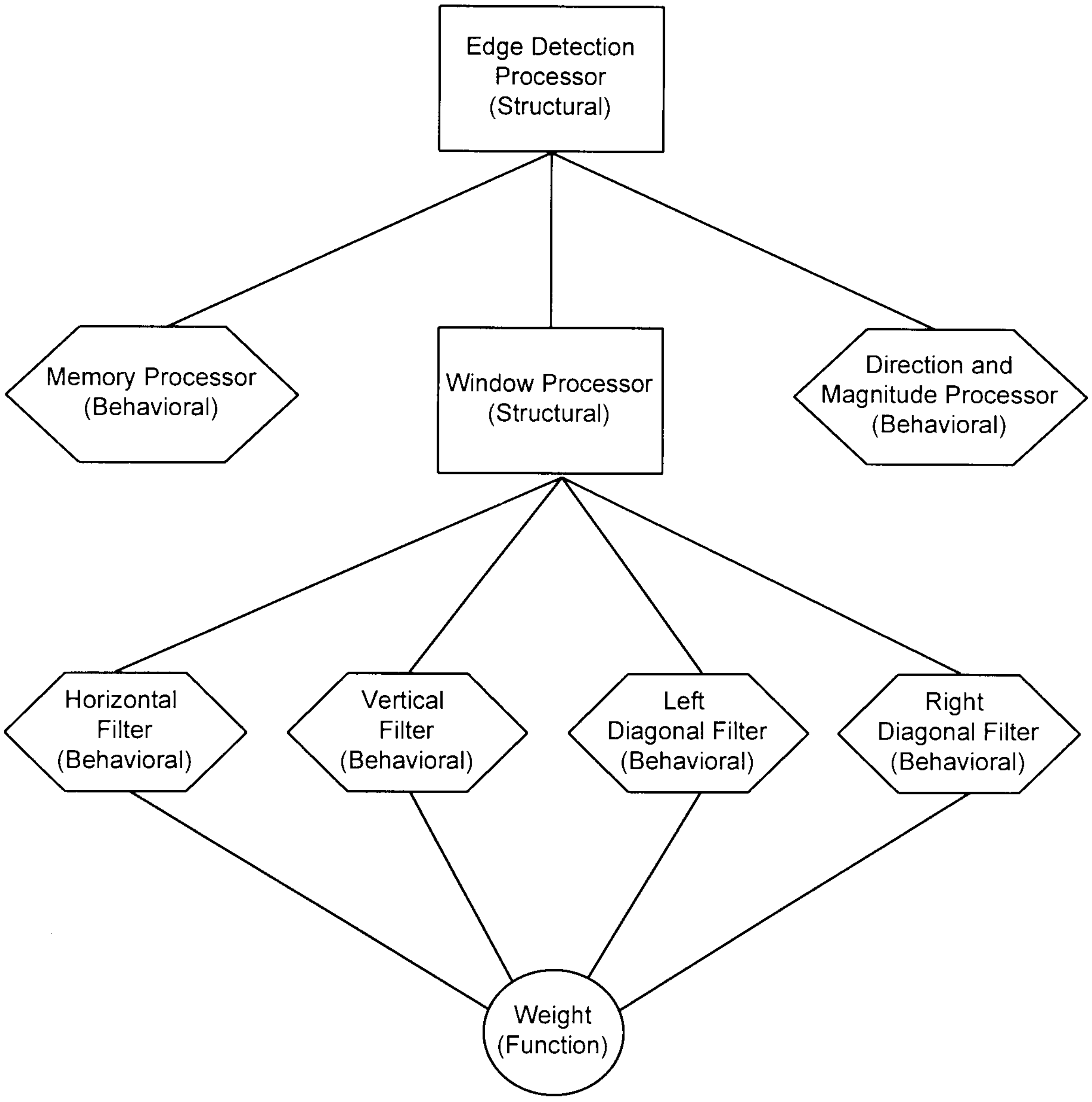

Edge Detection Processor (Structural)
Memory Processor (Behavioral)
Window Processor (Structural)
Direction and Magnitude Processor (Behavioral)
Horizontal Filter (Behavioral)
Vertical Filter (Behavioral)
Left Diagonal Filter (Behavioral)
Right Diagonal Filter (Behavioral)
Weight (Function)

Figure 2-8. Hierarchy of Components in an Algorithmic-Level Structural Model

same function but with different timings due to a pipelined approach. The top levels of the structural hierarchy reflect the physical partitioning used in the circuit design. At this point the filter functions have been converted into design entities, and an additional entity, the memory processor, has been added to the design.

Structural models are often represented by hardware block diagrams. A hardware block diagram for the edge detection processor is shown in Fig. 2-9. The components are represented by rectangles; the interconnects are shown as lines connecting the components. Attributes may be associated with the components, interconnects, and interfaces in a block diagram. Names are usually given to the components and may also be given to interconnects and interfaces.

Fig. 2-9 shows a top-level hardware block diagram of the first-level partitioning of the edge detection processor. It shows three interconnected components: a buffer memory, a window processor, and a magnitude and direction processor. The buffer memory loads the image in scan line order, one pixel at a time. The buffer memory passes three scan lines parallel to the window processor, as indicated by interconnections P1, P2, and P3.

The window processor computes the horizontal, vertical, and left and right diagonal filters. The outputs of these filters are signals labeled H (for horizontal edges), V (for vertical edges), LD (for left diagonal edges), and RD (for right diagonal edges). The direction and magnitude processor outputs E, the magnitude of the edge (a measure of the level of the contrast between the areas separated by the edge), and D, the direction of the edge

.

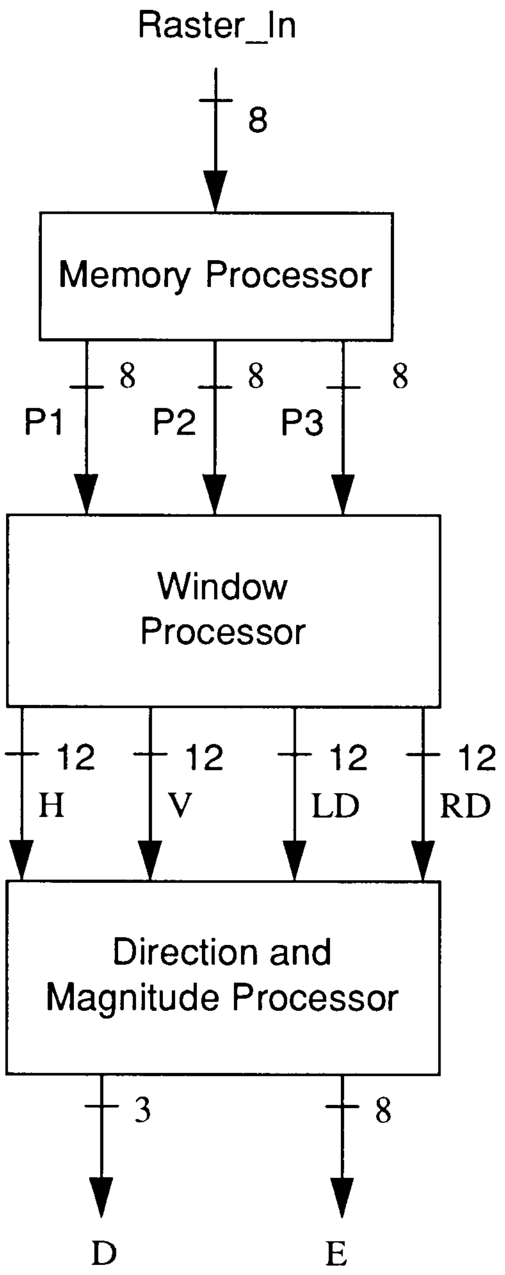


Figure 2-9. A Hardware Block Diagram for the Edge Detection Processor (Ref. 11)

Fig. 2-10 shows the VHDL structural architecture body for the edge detector. Similarly to the behavioral architecture for the edge detector shown in Fig. 2-6, this architecture uses the image-processing package for data-type definitions. The port maps for the component instantiations reflect the connections shown in the block diagram, Fig. 2-9.

Just as structural models can be hierarchical, block diagrams also demonstrate the hierarchy. Fig. 2-11 depicts a structural model of the window processor, which is one component of the edge detection system shown in Fig. 2-9. Fig. 2-11 shows the data flow for the window processor component of the edge detector. It has three input ports labeled P1, P2, and P3. It has four output ports labeled H, V, LD, and RD. The input and output interface names are identical to the corresponding interconnect names in the top-level block diagram to make the relationship between the VHDL model and the block diagram clear.

Figs. 2-12 and 2-13, respectively, show the entity interface declaration and the structural architecture body for the window processor. This VHDL design unit references the same two libraries as the higher level structural model of the edge detection processor. The port maps for this model reflect the connectivity shown in Fig. 2-11.

Fig. 2-14 shows the entity interface declaration for the horizontal filter. This same interface could be used with either a behavioral or a structural architecture body. Because the interface uses the `std_ulogic` data type for the clock, it references the `std_logic_1164` package in the `IEEE` library. Similarly, since it uses the algorithmic-level data-type specifications, it references the `image_processing` package in the `sobel_algorithm` library.

Fig. 2-15 shows a behavioral architecture body for the horizontal filter. Since this behavioral model is designed to be independent of any particular implementation, no attempt has been made to optimize the number of computations or the use of memory. However, the `weight` and `shift` functions are used to eliminate unnecessary redundancy in the program and improve readability. Two variables are internal to the process; they serve as buffers for the input pixels from two scan lines. The data in these two buffers are used as parameters to the weight function. The behavior of the horizontal filter is described in two parts. The first part updates the state of the filter, which is defined by the values of the pixel buffers `NEXT_LINE` and `LAST_LINE`.

The second part computes the output for the filter as the difference of the weighted sums of the two input lines. The function `weight` provides a common mechanism for the computation of the weighted sum. The horizontal filter process calls it twice, and the other filters use it as well.

```
architecture structure of edge_detector is
    component mem_processor
        port ( P:          in  pixel;
               Clock:      in  std_ulogic;
               P1, P2, P3: out pixel );
    end component;
    component window_processor
        port ( P1, P2, P3:    in  pixel;
               Clock:         in  std_ulogic;
               H, V, LD, RD:  out filter_out );
    end component;
    component mag_dir_processor
        port ( H, V, LD, RD:  in  filter_out;
               Clock:         in  std_ulogic;
               E:             out pixel;
               D:             out direction );
    end component;
    signal P1: pixel;       -- Tap onto 1st scan line buffer in Mem Proc
    signal P2: pixel;       -- Tap onto 2nd scan line buffer in Mem Proc
    signal P3: pixel;       -- Tap onto 3d scan line buffer in Mem Proc
    signal H:  filter_out; -- Temp storage for results of horizontal filter
    signal V:  filter_out; -- Temp storage for results of vertical filter
    signal LD: filter_out; -- Temp storage for results of left diag filter
    signal RD: filter_out; -- Temp storage for results of right diag filter
begin
    MP:  mem_processor port map (P, Clock, P1, P2, P3);
    WP:  window_processor port map (P1, P2, P3, Clock, H, V, LD, RD);
    MDP: mag_dir_processor port map (H, V, LD, RD, Clock, E, D);
end structure;
```

Figure 2-10. Structural Model for an Edge Detection Processor

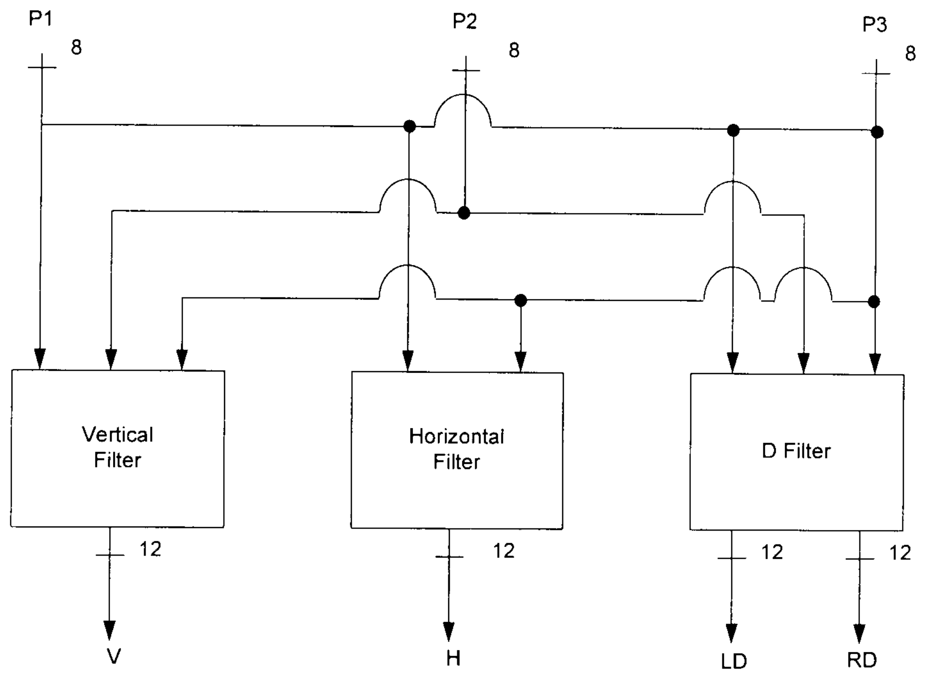


Figure 2-11. A Hardware Block Diagram for the Window Processor of the Edge Detection Processor

```
   -- The sobel algorithm library contains the packages,
   -- entity declarations, and architecture bodies for
   -- the algorithm level model of the sobel processor.
library sobel_algorithm;
use sobel_algorithm.image_processing.all;
use sobel_algorithm.timing.all;
   -- The IEEE library and the 1164 standard logic
   -- package are used in the algorithm model only
   -- for the clock.
library IEEE;
use ieee.std_logic_1164.all;

entity window_processor is
   port ( P1:    in  pixel;
          P2:    in  pixel;
          P3:    in  pixel;
          Clock: in  std_ulogic;
          H:     out filter_out;
          V:     out filter_out;
          LD:    out filter_out;
          RD:    out filter_out );
end window_processor;
```

Figure 2-12. VHDL Entity Interface for the Window Processor

```
architecture structure of window_processor is
    component horizontal_filter
        port ( P1:    in  pixel;
               P3:    in  pixel;
               Clock: in  std_ulogic;
               H:     out filter_out );
    end component;
    component vertical_filter
        port ( P1:    in  pixel;
               P2:    in  pixel;
               P3:    in  pixel;
               Clock: in  std_ulogic;
               V:     out filter_out );
    end component;
    component left_diagonal_filter
        port ( P1:    in  pixel;
               P2:    in  pixel;
               P3:    in  pixel;
               Clock: in  std_ulogic;
               LD:    out filter_out );
    end component;
    component right_diagonal_filter
        port ( P1:    in  pixel;
               P2:    in  pixel;
               P3:    in  pixel;
               Clock: in  std_ulogic;
               RD:    out filter_out );
    end component;
begin
    HF:  horizontal_filter port map (P1, P2, P3, Clock, H);
    VF:  vertical_filter port map (P1, P2, P3, Clock, V);
    LDF: left_diagonal_filter port map (P1, P2, P3, Clock, LD);
    RDF: right_diagonal_filter port map (P1, P2, P3, Clock, RD);
end structure;
```

Figure 2-13. VHDL Structural Architecture Body for the Window Processor

```
   -- The sobel algorithm library contains the packages,
   -- entity declarations, and architecture bodies for
   -- the algorithm level model of the sobel processor.
library sobel_algorithm;
use sobel_algorithm.image_processing.all;
use sobel_algorithm.timing.all;
   -- The IEEE library and the 1164 standard logic
   -- package are used in the algorithm model only
   -- for the clock.
library IEEE;
use ieee.std_logic_1164.all;
entity horizontal_filter is
   port ( P1:    in pixel;
          P3:    in pixel;
          Clock: in std_ulogic;
          H:     out filter_out );
end horizontal_filter;
```

Figure 2-14. Interface for the Horizontal Filter

```
architecture behavior of horizontal_filter is
   variable NEXT_LINE: pix3;  -- a 3 stage buffer of pixels
                              -- from the next scan line
   variable LAST_LINE: pix3;  -- a 3 stage buffer of pixels
                              -- from the last scan line
begin
   h_filter: process
   begin
      wait until rising_edge(CLOCK);
      NEXT_LINE := shift(NEXT_LINE,P3);
      LAST_LINE := shift(LAST_LINE,P1);
      H <= weight(LAST_LINE(1), LAST_LINE(2), LAST_LINE(3))
           - weight(NEXT_LINE(1), NEXT_LINE(2), NEXT_LINE(3))
           after pixel_output_delay;
   end process h_filter;
end behavior;
```

Figure 2-15. Behavioral Model for the Horizontal Filter

2-4.3.2 Register-Transfer-Level Structural Description

Fig. 2-16 shows the hierarchy of design entities and the types of their architecture bodies in a register-transfer-level structural description. Each node in the tree has a corresponding VHDL entity interface and at least one architecture body. Not all of the VHDL code for the models is shown here. This model has four levels of hierarchy. At the top of the hierarchy is the edge detection processor, which has a structural architecture body. This architecture body uses three components: the memory processor, the window processor, and the direction and magnitude processor. All three of these components use structural architecture bodies. The window processor makes use of four filter processors as components, and the magnitude and direction processor has two components. All six of these components use structural architecture bodies. The leaf-level modules in this model are first-in, first-out (FIFO) buffers, adders, subtractors, delays, multiplexors, comparators, encoders, and absolute value processors. These modules use behavioral architecture bodies described at the register-transfer level. Thus this structural model is a register-transfer-level model.

Fig. 2-17 is a block diagram of the structural model of the horizontal filter whose behavioral description is shown in Fig. 2-15. The model uses three adders. Delay units are used to postpone certain signals for one clock cycle. The subtractor `SUB` performs a subtraction on the incoming data. The first adder `ADD1` adds the difference between the current inputs and the difference between the inputs of the previous cycle (provided by `DELAY1`). The second adder `ADD2` adds the current and previous sums. A VHDL structural body corresponding to this block diagram is shown in Fig. 2-18.

The leaf nodes shown in Fig. 2-17 are macrocells from a standard library included with the synthesis tool used to implement the VLSI circuit for the edge detector. The goal of this design was to minimize the number of cells required to perform the function. Thus there is little resemblance between the structural model shown in Fig. 2-17 and the behavioral description shown in Fig. 2-15. Algebraic manipulation of the function described in the behavioral model verifies the equivalence of this structural model and the behavioral model.

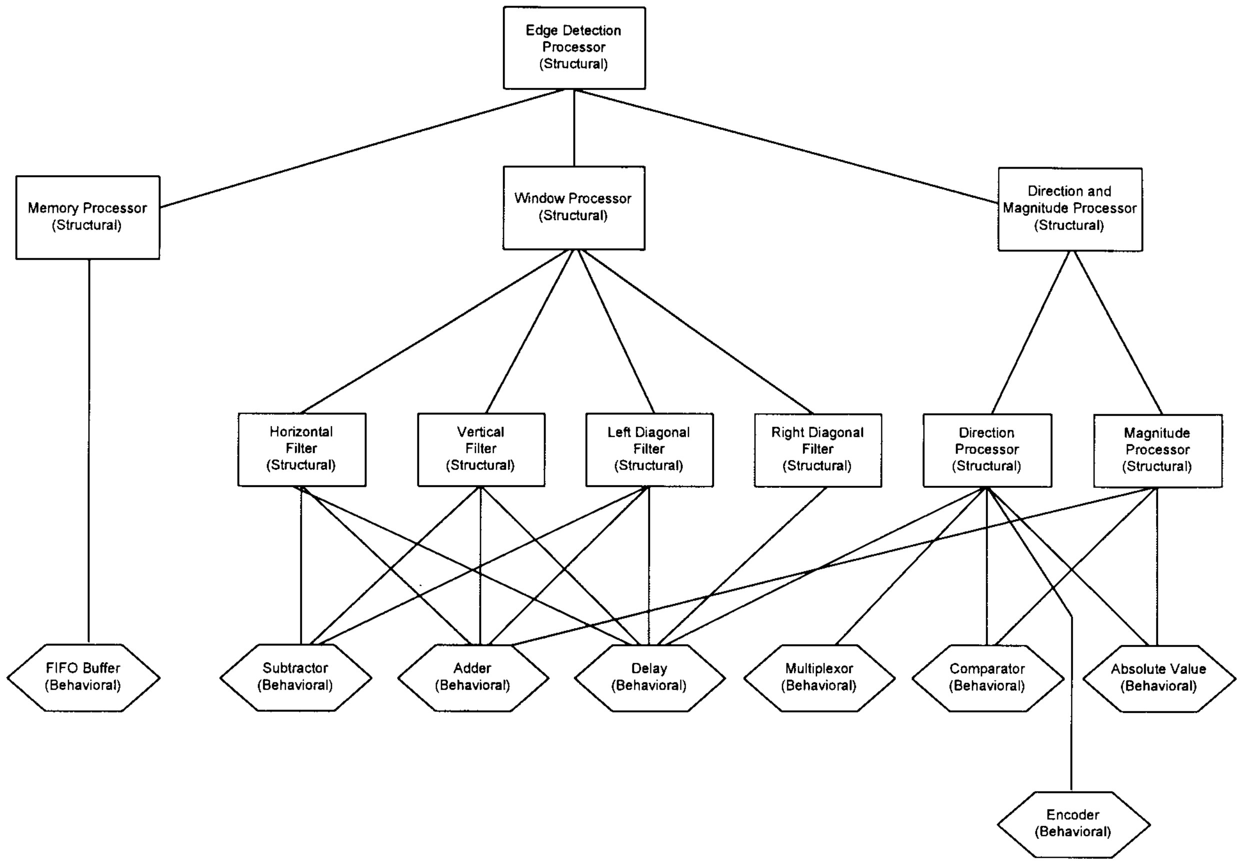

Edge Detection Processor (Structural)
Memory Processor (Structural)
Window Processor (Structural)
Direction and Magnitude Processor (Structural)
Horizontal Filter (Structural)
Vertical Filter (Structural)
Left Diagonal Filter (Structural)
Right Diagonal Filter (Structural)
Direction Processor (Structural)
Magnitude Processor (Structural)
FIFO Buffer (Behavioral)
Subtractor (Behavioral)
Adder (Behavioral)
Delay (Behavioral)
Multiplexor (Behavioral)
Comparator (Behavioral)
Absolute Value (Behavioral)
Encoder (Behavioral)

Figure 2-16. Hierarchy of Functions in a Structural Model

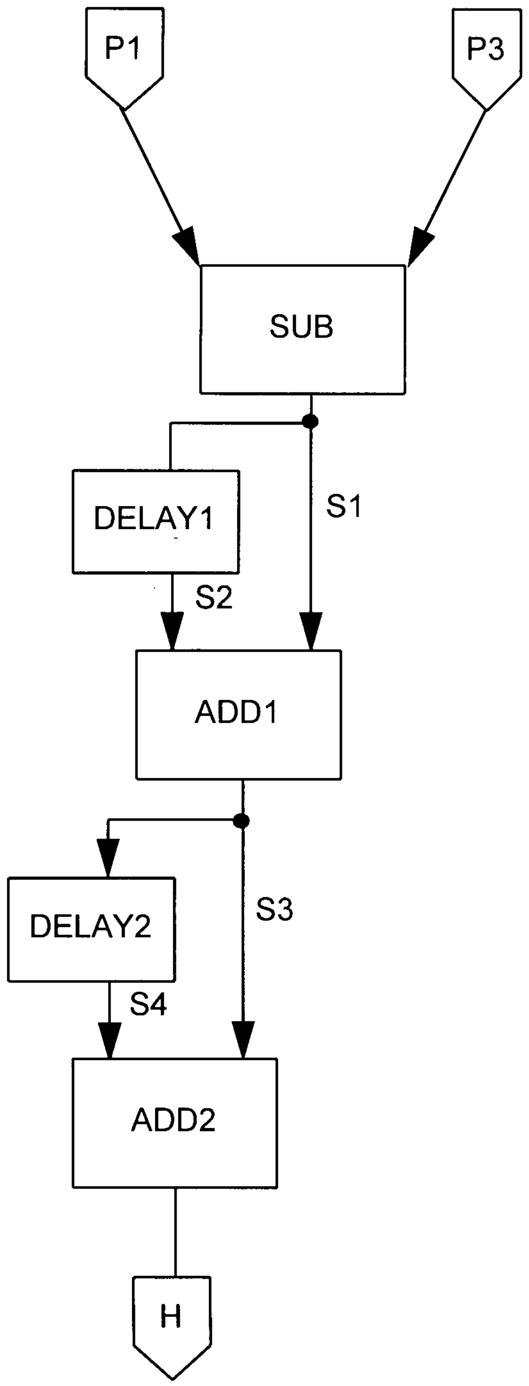


Figure 2-17. Block Diagram of the Horizontal Filter Processor (Ref. 11)

```
architecture structure of horizontal_filter is
   component subtractor
       port ( A1:    in  pixel;
              A2:    in  pixel;
              Clock: in  std_ulogic;
              DIFF:  out filter_out );
   end component;
   component adder
       port ( A1:    in  filter_out;
              A2:    in  filter_out;
              Clock: in  std_ulogic;
              SUM:   out filter_out );
   end component;
   component delay
       port ( A_IN:  in  filter_out;
              Clock: in  std_ulogic;
              A_OUT: out filter_out );
   end component;
   signal S1: filter_out; -- Connects difference to 1st
        -- delay and 1st adder
   signal S2: filter_out; -- Connects 1st delay to 1st adder
   signal S3: filter_out; -- Connects 1st adder to 2nd delay
        -- and 2nd adder
   signal S4: filter_out; -- Connects 2nd delay to 2nd adder
begin
   SUB:    subtractor port map (P1, P3, Clock, S1);
   DELAY1: delay port map (S1, Clock, S2);
   ADD1:   adder port map (S1, S2, Clock, S3);
   DELAY2: delay port map (S3, Clock, S4);
   ADD2:   adder port map (S3, S4, Clock, H)
end structure;
```

Figure 2-18. Structural Architecture of the Horizontal Filter

2-5 MIXED ABSTRACTION MODELS

2-5.1 THE PURPOSE OF MIXED LEVEL OF ABSTRACTION MODELS

Hierarchical models may not have the same level of detail down the path to each leaf. For example, in the same model of a computer the central processing unit (CPU) may be modeled in terms of its instruction-set behavior, whereas an application-specific integrated circuit (ASIC) may be modeled at the gate level. These mixed-abstraction-level models allow detailed simulation of part of a system and achieve high simulation speeds because the high-level behavioral parts of the model simulate more quickly than the detailed structural parts.

Given a complete VHDL model database with both behavioral and structural architecture bodies for all of the modules, the architect can configure a model using low-level structural architectures for some components and high-level behavioral architectures for the rest of the system. The resulting model achieves higher simulation speed through the use of the high-level behavioral architecture bodies and yet provides detailed simulation for the part of the system where low-level structural architecture bodies are used.

2-5.2 DESIGNING MODULES FOR MIXED ABSTRACTION MODELS

Subpar. 10.2.1 of the VHDL DID (Ref. 1) requires delivery of both structural and behavioral models of all modules other than the leaf modules. Models conforming with this requirement allow users of the models to build and simulate mixed abstraction versions of the models. The modules of a model need to be carefully designed if mixed abstraction versions of the model are to be configured quickly and efficiently. VHDL provides a mechanism to configure mixed abstraction models, the configuration specification. The configuration specification describes which representation of a module is to be used, e.g., for a particular instance of a module. This mechanism can be used to select behavioral or structural representations.

Behavioral models must be designed to interface with structural models of neighboring modules. In particular, the data types for the external interfaces must be chosen carefully so that structural models can be interfaced at a later stage. In general, the structural VHDL models use low-level data types such as the IEEE standard logic types (Ref. 2) as the data types of their input and output ports. The behavioral model should be prepared to interface with such data types. In some cases a single behavioral input or output may correspond to an array of standard logic values.

One VHDL mechanism that supports interfacing behavioral and structural models is the type conversion function. Type conversion functions can be associated with the ports of structural models in either component instances or configuration specifications. In the early stages of model development, the project manager should develop a standard set of data types for the module interfaces. All models should be constructed with these standard data types. VHDL provides a mechanism (the package) to share a single definition of a data type across all parts of a model.

2-5.3 AN EXAMPLE OF A MIXED LEVEL OF ABSTRACTION MODEL

The hierarchy of a mixed level of abstraction model is shown in Fig. 2-19. This model uses the register-transfer-level behavioral structural models of the components of the horizontal filter processor in conjunction with ISA-level behavioral models of the vertical and diagonal processors and with algorithmic-level behavioral models of the memory processor and the magnitude and direction processor. Because integer formats are used in the behavioral models for the memory processor and the magnitude and direction processor, type conversion functions are required to convert the integers to and from the 8-bit array inputs and the 12-bit array outputs of the structural model of the horizontal processor.

REFERENCES

1. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, Department of Defense, Washington, DC, 11 May 1989.
2. IEEE Std 1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (std_logic_1164)*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1993.
3. IEEE Std 1076.4, *IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, December 1995.
4. D. Gajski and R. Kuhn, "Guest Editors' Introduction: New VLSI Tools", IEEE Computer **16**, 11-4 (1983).
5. G. A. Frank, C. U. Smith, and J. L. Cuadralo, "An Architecture Design and Assessment System for Software/Hardware Codesign", *Proceedings of the 22nd Design Automation Conference,* Las Vegas, NV, June 1985, pp. 417-24, IEEE Computer Society Press, Los Alamitos, CA.
6. J. Schoen, *Performance and Fault Modeling With VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.
7. R. A. Walker and D. E. Thomas, "A Model of Design Representation and Synthesis", *Proceedings of the 22nd Design Automation Conference*, Las Vegas, NV,

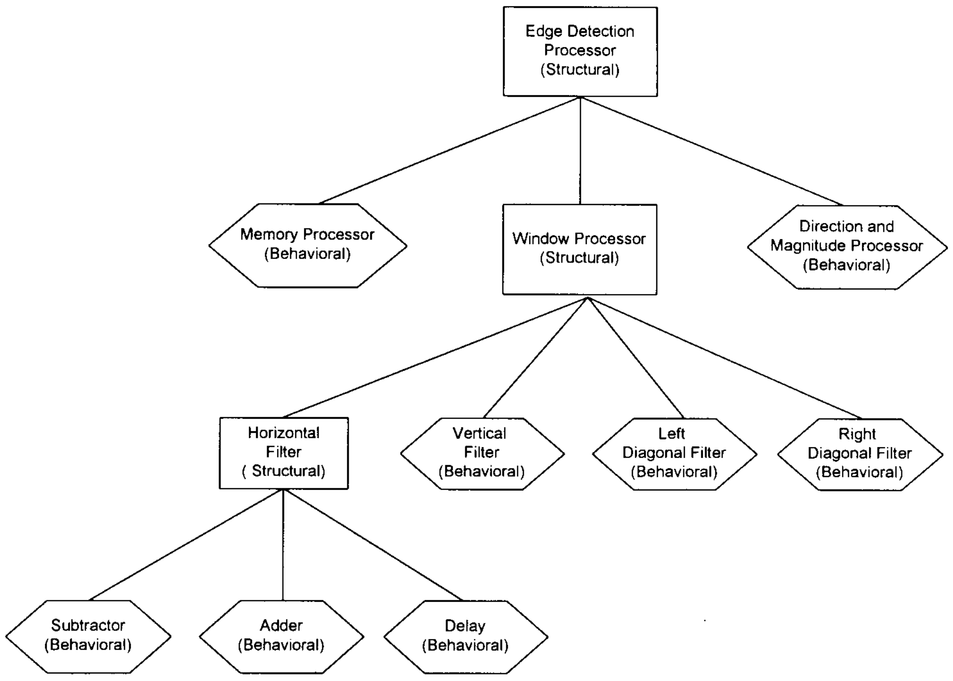

Edge Detection Processor (Structural)
Memory Processor (Behavioral)
Window Processor (Structural)
Direction and Magnitude Processor (Behavioral)
Horizontal Filter (Structural)
Vertical Filter (Behavioral)
Left Diagonal Filter (Behavioral)
Right Diagonal Filter (Behavioral)
Subtractor (Behavioral)
Adder (Behavioral)
Delay (Behavioral)

Figure 2-19. Hierarchical Organization of a Mixed Level of Abstraction Model

June 1985, pp. 453-9, IEEE Computer Society Press, Los Alamitos, CA.

8. D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill Book Co., Inc., New York, NY, 1982.
9. J. R. Armstrong and F. G. Gray, *Structured Logic Design Using VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.
10. J. E. DeGroat and G. S. Powley, Jr., "Object-Oriented Generation of VHDL Synthesizable Architectural Building Blocks", *VHDL: Champions of the Second Generation, Proceedings VHDL International Users' Forum Spring Conference*, 2-6 April 1995, San Diego, CA, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
11. C. O. Scheper and R. L. Baker, "VHSIC Silicon Compiling Tools Applied to Image Processing", *1987 Digest of Papers, Government Microcircuit Applications Conference*, GOMAC-87 B119187, Orlando, FL, 26-29 October 1987, US Army Laboratory Command, Fort Monmouth, NJ.
12. MIL-STD-490A, *Specification Practices*, 4 June 1985.

BIBLIOGRAPHY

A. G. Stanculescu, A. S. Tsay, N. D. Zamfiresccu, and D. L. Perry, "Switch-Level VHDL Descriptions", *IEEE International Conference on Computer-Aided Design, Digest of Technical Papers*, 1989, pp. 180-3, IEEE Computer Society Press, Los Alamitos, CA.

J. R. Armstrong, "Accurate Timing Modeling in Behavioral Models", SIGDA Newsletter **18**, 72-5 (December 1988).

J. R. Armstrong, *Chip-Level Modeling With VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

D. L. Barton, "Behavioral Descriptions in VHDL", VLSI Systems Design **9**, 28-33 (June 1988).

M. J. Chung and S. Kim, "An Object-Oriented VHDL Design Environment", *Proceedings of the 27th ACM/IEEE Design Automation Conference*, Piscataway, NJ, 1991, Institute of Electrical and Electronics Engineers, Inc., New York, NY.

A. Dewey, "The VHSIC Hardware Description Language (VHDL) Program", *ACM IEEE 21st Design Automation Conference Proceedings 84*, Albuquerque, NM, 23-27 June 1984, Institute of Electrical and Electronics Engineers, Inc., New York, NY.

R. Ernst and J. Bhasker, "Simulation-Based Verification for High-Level Synthesis", IEEE Design & Test of Computers **8**, 14-20 (March 1991).

F. T. Hady, J. H. Aylor, R. D. Williams, and R. Waxman, "Uninterpreted Modeling Using the VHSIC Hardware Description Language (VHDL)", *1989 IEEE International Conference on Computer-Aided Design, Digest of Technical Papers*, 1989, IEEE Computer Society Press, Los Alamitos, CA.

P. Hunter, R. Harr, and H. Carter, "General Requirements for VHDL Behavioral Models", SIGDA Newsletter **18**, 21-33 (December 1988).

M. Neighbors, "Extending the VHDL Simulation Environment into the Requirements Analysis Level—An Example (SINCGARS-ICOM Radio)", *Proceedings of the Tactical Communications Conference, Vol. 1, Tactical Communications, Challenges of the 1990s*, Fort Wayne, IN, 24-26 April 1990, pp. 435-53, Institute of Electrical and Electronics Engineers, Inc., New York, NY.

N. Park and A. Parker, "A Software Package for Synthesis of Pipelines From Behavioral Specifications", IEEE Transactions on Computer-Aided Design **7** (March 1988).

R. A. MacDonald and R. Waxman, "Operational Specification of the SINCGARS Radio in VHDL", *Proceedings of the Tactical Communications Conference, Vol. 1, Tactical Communications, Challenges of the 1990s*, Piscataway, NJ, 1990, pp. 415-33, Institute of Electrical and Electronics Engineers, Inc., New York, NY.

R. J. Hookway, "System Simulation Using VHDL", *Automated Design and Engineering for Electronics-West, Proceedings of the Technical Sessions*, 31 March-2 April 1987, pp. 21-33, Cahners Exposition Group, Anaheim, CA.

D. W. Runner and E. H. Warshawsky, "Synthesizing Ada's Ideal Machine Mate", VLSI Systems Design **9**, 30, 32-3, 36, 38-9 (October 1988).

B. R. Stanisic, "VHDL Modeling for Analog-Digital Hardware Designs (VHSIC Hardware Description Language)", *IEEE International Conference on Computer-Aided Design, Digest of Technical Papers*, 1989, pp. 184-7, IEEE Computer Society Press, Los Alamitos, CA.

S. Tandri, M. H. Abd-El-Barr and C. McCrosky, "High-Level Specification, Simulation and Layout Generation of Systolic Arrays—A Case Study", *Conference Proceedings, CCVLSI '90, Canadian Conference on Very Large-Scale Integration,* pp. 6.6/1-6, Carlton University, Ottawa, Ontario, Canada, 1990.

A. Sama and J. Armstrong, "Behavioral Modeling of RF Systems With VHDL", *Proceedings of the Spring 1991 VHDL Users' Group Meeting*, 1991, Cincinnati, OH, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

CHAPTER 3
VHDL CONCEPTS

This chapter presents an overview of VHDL. The use of VHDL to capture the behavior and structure of digital electronic systems is discussed. Aspects of VHDL that support the reuse of models and source code are presented. The development and use of VHDL libraries, for reuse of VHDL descriptions both within a model and between models, and the annotation of VHDL models with descriptive information are described. Also described is structuring VHDL models to improve their readability and reuse.

3-1 INTRODUCTION

This chapter introduces and defines very high-speed integrated circuit (VHSIC) hardware design language (VHDL) terminology in a conceptual framework and shows how VHDL features can be used to describe digital systems. This chapter discusses how to use VHDL features to describe the structure, function, and timing of a digital system; how to annotate models, handle errors, and reuse VHDL code; and how to manage the configuration of simulation models through the use of late binding. VHDL terminology introduced in this chapter is used throughout this handbook. The intent of this chapter is to provide information on what a contract monitor could or should see in a VHDL hardware model delivered to the Government, not to provide a detailed tutorial on VHDL. Detailed VHDL tutorials are listed in the chapter Bibliography.

The VHSIC hardware description language was developed to provide a standardized language to describe the behavior and structure of Department of Defense (DoD) digital electronic systems formally (Ref. 1). The language is formal because it has well-defined syntax and semantics. VHDL began as a research effort under the DoD VHSIC program (Ref. 1). As experience with the language was gained, it was improved over a period of several years by incorporating additional features. The language was subsequently standardized by the Institute of Electrical and Electronics Engineers (IEEE) as Standard (Std) 1076-1987 (Ref. 2). The IEEE requires standards to be recertified approximately every five years; therefore, an update to VHDL was completed in 1993, IEEE Std 1076-1993 (Ref. 3). Tool support for the standards generally lags behind the standardization process, so it is important for a contract monitor to understand what features of the latest version of VHDL are used in models and which tool sets support those features. This understanding is particularly important if different subcontractors are using different VHDL development environments or if the VHDL tool environment of the contracting or validation and verification (V&V) organization is different from that of the prime contractor.

3-2 BASIC VHDL CONCEPTS

3-2.1 VHDL DESIGN ENTITIES

The design entity is the primary VHDL concept that represents a component of an electronic system. This component can be either a physical component (such as an integrated circuit or a printed circuit board) or a logical component (such as a memory comprising several circuits on a board or an arithmetic and logic unit (ALU) occupying only a portion of an integrated circuit).

In a VHDL model a design entity consists of an entity interface and exactly one of its corresponding architecture bodies. (One entity interface can have several associated architecture bodies.) When a VHDL model is configured, a specific architecture body is selected for the design entity through the use of either configuration declarations or configuration specifications. Fig. 3-1 illustrates the relationship between design entities, entity interfaces, and architecture bodies.

The VHDL data item description (DID) (Ref. 4) requires each physical module of an electronic system to be documented with one or more design entities. The VHDL DID expects that all physical modules that are not considered primitive, or leaf, modules should have both a behavioral design entity and a structural design entity. Primitive, or leaf, modules are documented with a behavioral design entity.

All design entities for the same hardware component and at the same level of abstraction should have a common entity interface. This approach encourages reuse of models because changes in the design of a particular component can be encapsulated in the architecture body, without causing changes in the rest of the VHDL model. For example, consider a VHDL entity interface for a microprocessor such as a 1750A (Ref. 5). The entity interface for this microprocessor may have one architecture body that implements an instruction-set-architecture (ISA)-level model of the microprocessor, another architecture body that provides a register-transfer-level model, and another architecture body that provides a gate-level model of the same device. Suppose that this entity interface is bound to an instance in a larger model of a board that includes other components for the main memory system and input/output (I/O) subsystems. The ISA design entity can be used to verify software written for the microprocessor or to test the I/O subsystem model. To verify the test and maintenance functions, the gate-level design entity can be used. The register-transfer-level design entity can be used to synthesize a new version of the microprocessor using new integrated circuit (IC) technology. All of these design entities can be simulated in the context of the board model without changing the VHDL code of the board

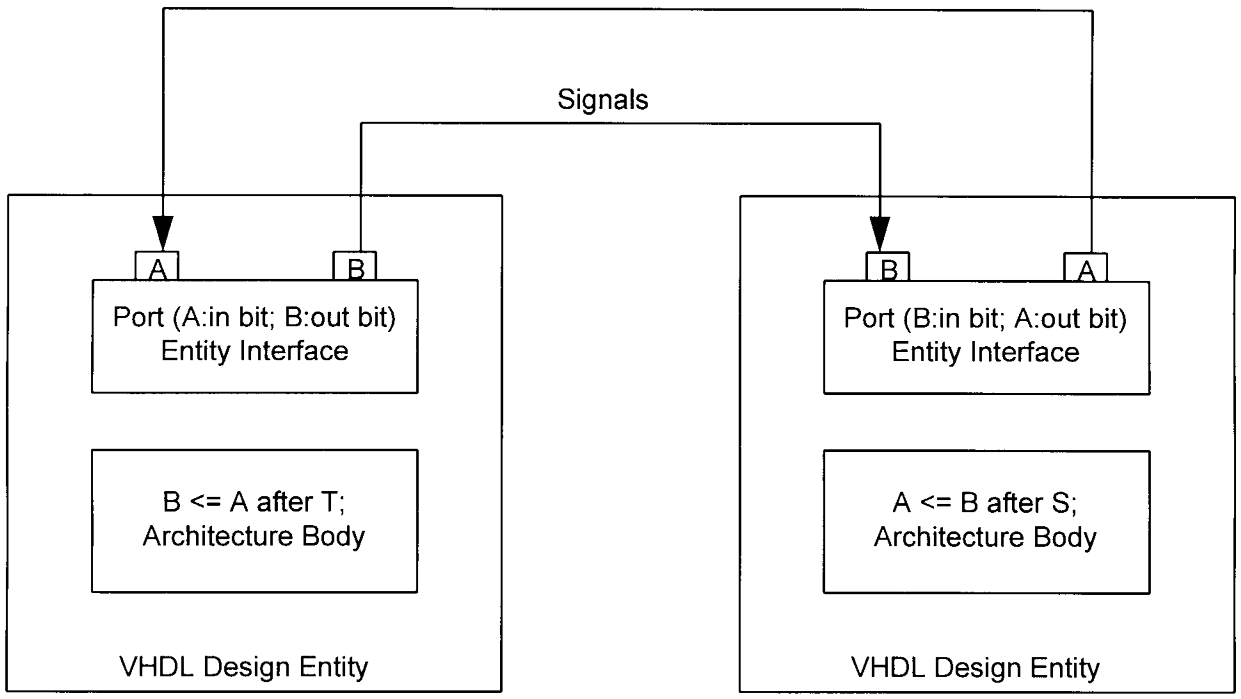


Figure 3-1. Design Entities, Entity Interfaces, and Architecture Bodies

or the design entities selected for the other subsystems provided they use the same entity interface and the architectures implement the same behavior.

Division of a design entity into an entity interface and an architecture body also allows the system designer to delay the choice of an architecture body until later in the design process. This approach allows the system designer to make tradeoffs between different implementations for a device simply by selecting a different architecture body. VHDL has mechanisms to select architecture bodies without changing the contents of any of the entity interfaces or architectures for the system that uses the component. These mechanisms are discussed in subpar. 3-8.3. This feature allows a major reduction of risk because anytime a model is modified, there is a risk that new errors will be introduced into the model.

3-2.1.1 Entity Interfaces

The entity interface declaration specifies the interface of the entity, i.e., the external view of a design entity. This external view includes ports, generic and local constants, attributes, and error checking of the inputs to the design entity. The entity declaration provides information about this external interface to other architectures using the design entity. This information includes external electrical connections, which are specified with port declarations, and generic constraints, such as the acceptable range of operating temperatures for the device. An entity interface declaration can also specify a mechanism to detect unacceptable behavior (such as timing violations) during simulation.

Appropriate entity interface declarations are essential for interoperability of VHDL models. A contract monitor receiving a model should assess the likelihood of its reuse and the changes that may occur in the model when it is reused to ensure that the model is developed to support that reuse scenario. In particular for an entity interface declaration, this assessment requires choosing the data types used to define the ports and on the generics to be used in the model.

For a design entity that represents a physical device, the ports specify the external electrical connections of the device. For example, if an integrated circuit is being modeled by a VHDL design entity, the ports of its entity interface describe the individual pins on the integrated circuit package. For more abstract models, particularly at the algorithmic level, these ports may represent the busses that a processing element accesses. Fig. 2-10 and its related discussion provide an example of a more abstract port.

A port is defined by a name, a mode, and a type. The port name is used to identify a particular port; all port names for an entity interface must be unique with respect to the other ports of the entity interface. If a physical device is being modeled, the VHDL DID (Ref. 4) requires a given port name to correspond to the physical electrical connection of the component. For example, the number of ports and the port names for an IC model must correspond to the number of pins and the pin names of the device being modeled.

The allowable port directions (or modes) are `in`, `out`, `inout`, `buffer` or `linkage`. The port modes define the allowable direction of data flow through a port. They also determine the sources of the signal connected to that port. For example, ports labeled `in` and `inout` are sinks for a signal; ports labeled `out`, `inout`, and `buffer` are sources. Ports labeled `buffer` or `linkage` provide other special functions not germane to this discussion; the reader is referred to the *VHDL Language Reference Manual* (Ref. 3) or VHDL texts such as those cited in the Bibliography for more information.

VHDL has mechanisms to define the type of a port and to check the consistency between the type of the port and the type of its associated signal. This latter mechanism assures that incompatible types are not connected. Ports can have default values, which are used when an instance of an input port is left unconnected or when an output port is undriven.

Fig. 3-2 shows the entity interface declaration for the edge detection processor discussed in subpar. 2-4.3. This entity interface has four ports named `P` (for input of image pixels), `Clock` (for synchronization), `E` (for output of the edge image), and `D` (for output of the edge direction information). Ports `P` and `Clock` are `in` ports and ports `E` and `D` are `out` ports. The type of input port `P` and output port `E` is `pixel`, a user-defined type specified in the package `image_processing` in the library `sobel_algorithm`. The type of input port `Clock` is `std_ulogic`, an IEEE Std 1164 type (Ref. 6), which is specified in the package `std_logic_1164` in the library `IEEE`. Edge detector is a simple entity interface declaration not containing any of the other possible declarations or any error-checking statements. These additional capabilities are discussed in pars. 3-6 and 3-7.

3-2.1.2 Architecture Bodies

An architecture body describes the relationships between the inputs and outputs of the corresponding entity declaration. Such relationships may include both function and timing. Multiple architecture bodies can be associated with a particular entity interface, although only one can be associated with a given instance of an entity interface. This instance-by-instance binding capability provides flexibility in the construction and use of hardware descriptions and eliminates the risk that would result from having to change entity interfaces every time a different architecture is used. Since different architecture bodies for a component can be selected without modifying the code for the architectures that use the component, the risk that would result from requiring modifications of the architectures is eliminated. Furthermore, different architecture bodies can be selected without requiring that the architectures be reanalyzed, i.e., recompiled, and this procedure can significantly reduce the time to prepare a model for simulation.

A good design is modularized to support design tradeoffs and to anticipate possible changes in the design so they are appropriately partitioned into design entities. Good partitioning allows changes to be implemented by substituting different architecture bodies without any modification of the associated entity interface. One situation in which changes are expected is during top-down development of a hardware module. The level of detail in a top-down design changes, so different architecture bodies can reflect the addition of different levels of detail to the design. For example, two architecture bodies may perform exactly the same logical function but differ in their timing and implementation. Pars. 2-3 and 2-4 describe different architecture bodies for the same entity declaration. One is a behavioral model; one is a structural model.

One frequently used testing methodology uses two models that process the same inputs; their outputs are compared for equality. These two models should have the same entity interface but different architecture bodies. One architecture body is considered the reference model; its outputs are the standard to which the outputs of the second architecture body are compared. In a top-down development process the reference architecture body usually represents a more abstract view of the system, and the one being tested represents a more detailed view. Research is being performed to develop functional verification tools that provide an alternative to simulation for this verification (Ref. 7).

```
      -- The sobel algorithm library contains the packages,
      -- entity declarations, and architecture bodies for
      -- the algorithm level model of the sobel processor.
library sobel_algorithm;
use sobel_algorithm.image_processing.all;
      -- The IEEE library and the 1164 standard logic
      -- package are used in the algorithm model only
      -- for the clock.
library IEEE;
use ieee.std_logic_1164.all;
entity edge_detector is
      port (P:     in pixel;
            Clock: in std_ulogic;
            E:     out pixel;
            D:     out direction );
end edge_detector;
```

Figure 3-2. A VHDL Entity Interface Declaration

An architecture body contains both declarations and statements. These statements may include processes, component instantiations, and concurrent signal assignment statements. These kinds of statements execute concurrently and use the signals declared either in the architecture body or as the ports of the entity interface. Such signals exchange information and synchronize the actions of the architecture.

Depending upon the statements it contains, an architecture body is considered to have one of three styles: behavioral, structural, or mixed. The behavioral style is normally constructed using processes and concurrent signal assignment statements and includes signals for communication between processes and variables for communication within each process. In a behavioral style, each process provides a sequential execution paradigm. Behavioral models constructed using the concurrent signal assignment as the primary construct are sometimes called data flow style models. The structural style uses only component instantiations to specify design entities at the next lower level of the hierarchy and connects these components using signals. The leaf, or primitive, elements of a structural model are the lowest level of the design hierarchy and are always written in a behavioral style. The mixed style combines processes and component instances in the same architecture.

These styles of VHDL models are designed to support models that serve different purposes. Chapter 2 discusses the purposes of behavioral and structural models at different levels of abstraction. In pars. 3-3 and 3-4 the key VHDL concepts for these styles and the roles of these styles in support of the purposes of structural and behavioral models, as described in Chapter 2, are discussed.

3-2.2 THE VHDL CONCEPT OF TIME

A VHDL simulation is the computation of a series of events sequenced by time. In VHDL an event is a change in the value of a signal. Thus each event is associated with a signal and has a value (the new value of the signal) and a time. The interval between events can be very large or very small, so simulation time can be advanced by arbitrary amounts. VHDL models are usually simulated by a discrete event simulator in order to cope with variably sized time steps.

A VHDL simulation is a two-step process (Ref. 8). First, all signal values are updated. After the signal values are updated, the processes that are sensitive to changes in the signal values are executed. After all processes have been executed, the process repeats and signal values are updated. This cycle is repeated and terminates only when the simulator runs out of events, the simulation time advances to the maximum possible value, or the simulation is stopped by the user or by an error.

The run time of most VHDL simulators is determined largely by the number of events in a simulation. Reducing the number of events required in a simulation is likely to reduce its run time. Thus a behavioral model with a few large processes and a small number of signals usually executes faster than a structural model with many component instantiations and many signals. Using more abstract signal data types also reduces the number of events. For example, if a signal has a data type of a 32-bit integer, a simple event represents a new 32-bit value. On the other hand, if the same 32-bit integer is represented as 32 one-bit signals, up to 32 events may be required to represent the same change in value.

3-2.3 SIGNALS

In VHDL, signals provide a means of communication between and processes, concurrent signal assignments, and components. During simulation, changes in signal values may activate processes or signal assignment statements, which in turn compute new values for signals.

The declaration of a signal specifies its type. The type of a signal must be consistent with the type of any port to which the signal is connected. Also the type of a signal must be consistent with the value on the right-hand side of a signal assignment statement.

To ensure interoperability of VHDL models, the signal type declarations should be made available to those entities connected by the signals. A necessary condition for interoperability of design entities is that the user can connect two design entities together with one or more signals. VHDL type checking can be used to ensure that models meet at least this interoperability condition. A valuable technique to ensure interoperability is the use of packages to encapsulate signal type definitions and their associated functions and make them globally available. This approach has been taken in IEEE Std 1164 (Ref. 6), which defines a standard set of types for signals in logic-level models. The specification of signal data types that are used by multiple design entities is an important early milestone for a VHDL system design.

3-2.3.1 Signal Assignment Statements

Signal assignment statements are the VHDL constructs that specify the future values of signals and the times at which those values are to be assigned. Computation of the future values of signals is the essence of the function of the model; computation of the times at which the signal will assume those values is the essence of the timing of the model.

VHDL has two different kinds of signal assignment statements: sequential and concurrent. These two types of signal assignment statements are valid in different contexts: sequential signal assignment statements are valid only inside a process or subprogram, whereas concurrent signal assignment statements are valid in concurrent contexts, such as within an architecture or block statement.

Simulation events in VHDL are generated by signal assignment statements. Execution of a signal assignment statement causes one or more transactions to be scheduled for the future. Each transaction has a time and a value that represents a possible value of the signal at a specified time in the future. These transactions are stored in queues called drivers. VHDL uses a concept called a driver to capture the possible future values of a signal. As simulation time advances, transactions are removed from the queue as their times pass from the future to the present and become the present driving value of the driver.

A signal assignment statement edits the transactions in the associated driver. Editing refers to transactions being added to, deleted from, or inserted into the driver queue. The interaction of signal assignment statements and drivers is called propagation. VHDL supports two models of signal value propagation: inertial delay (the default) and transport delay. These two models allow users to model accurately certain physical properties of hardware. In the transport delay model each signal value, no matter how short its duration, is propagated. This approach is important for modeling edge triggered devices, in which a short-duration pulse may cause the device to change state. Inertial delays are intended to model circuits for which an input must persist for some minimum time before the circuit responds. If the input has a shorter duration than the minimum inertial delay, the circuit does not respond.

Each concurrent signal assignment statement has a unique driver, but all sequential signal assignment statements writing to the same signal in the same process share the same driver. The user should be careful not to make invalid assumptions about the editing rules for sequential signal assignment statements sharing the same driver because these editing rules are different from those for concurrent signal assignment statements, which have different drivers. See, for example, Ref. 8, pp. 70-82, for a detailed discussion.

Fig. 3-3 shows a sequential signal assignment statement extracted from the horizontal filter shown in Fig. 2-15. In this case the value of the signal `H` is specified by a complex expression that averages elements of the three element buffers `LAST_LINE` and `NEXT_LINE`. The time that `H` assumes this value is the current simulation time plus the value of `pixel_output_delay`.

The timing of a component is likely to change based on a number of factors, such as the operating temperature of the component or the technology with which the component is implemented. To ensure reuse of the VHDL model of a component, the timing information should be parameterized so that changes in these external factors can be made without requiring changes to the VHDL model. As shown by examination of Fig. 2-5 and its reference to a timing package, the delay in Fig. 3-3 is parameterized by using a deferred constant. An alternative approach is to use generics and pass the delay information down the hierarchy of design entities. These approaches are discussed subpars. 3-6.1 and 3-6.2. Standardization of the timing of components is most advanced at the gate level; standards such as VHDL initiative toward ASIC libraries (VITAL) (Ref. 9) and EIA 567-A (Ref. 10) are included. Par. 6-5 and subpar. 6-3.3.3 describe mechanisms used to parameterize timing information in models at the gate level.

3-2.3.2 Resolution Functions

A signal `S` may have several drivers, one for each concurrently executing source of future values for `S`. All of the sequential signal assignment statements within a single process share the same driver for `S`, and each driver maintains a queue of possible future values for its associated signal. These future values are time stamped. The contents of these queues must be merged to determine the future value of the signal. VHDL includes a mechanism, referred to as a resolution function, that determines how conflicts in future values of a signal are resolved. Whenever a new value needs to be assigned to `S` (and `S` has multiple sources of values), a resolution function is called to compute the value of the signal based on the current values of the sources of the signal. These resolution functions are defined by the user. The resolution function returns a value that is then assigned as the driving value of the signal. When a signal is declared, a resolution function may be associated with that signal. If no resolution function is associated with the signal, the signal is considered unresolved. For example, the input port `Clock` for the edge detector design entity whose interface is shown in Fig. 3-2 is an unresolved data type `std_ulogic`. The "u" in the name indicates an unresolved data type. An unresolved data type is used for efficiency reasons because there is only one driver for the `Clock` signal.

Fig. 3-4 shows an example resolution function called a "wired-and" resolution function. It is associated with a four-value logic data type called `MVL`. This resolution function returns a `'0'` whenever any of its inputs are `'0'`, it returns an `'X'` if there is an `'X'` input but no `'0'` input, it returns a `'Z'` if all inputs are `'Z'`, and it returns a `'1'` otherwise. The input to a resolution function is always a vector, and a resolution function must be able to respond properly to a zero length vector, which may occur if all inputs are disconnected.

One or more resolution functions is a necessary part of any data type definition designed to specify signals. A resolution function may be defined for a data type `T` that is used for signals. When a data type declaration for signals is used to ensure interoperability of models, it should be equipped with a resolution function. This action guarantees that the declaration can be used in situations in which signals have multiple drivers. If a signal has a single driver, it may be de-

```
H <= weight(LAST_LINE(1), LAST_LINE(2), LAST_LINE(3))
          - weight(NEXT_LINE(1), NEXT_LINE(2), NEXT_LINE(3))
          after pixel_output_delay;
```

Figure 3-3. Example Signal Assignment Statement

```
     TYPE MVL IS ( 'X',  -- Forcing  Unknown
                   '0',  -- Forcing  0
                   '1',  -- Forcing  1
                   'Z',  -- High Impedance
                 );
     TYPE MVL_Vector IS ARRAY (NATURAL RANGE <> ) OF MVL;
     -----------------------------------------------------------------
-- resolution function
FUNCTION WiredAnd (Inputs: MVL_Vector) RETURN MVL IS
   TYPE LogicTable IS ARRAY(MVL,MVL) OF MVL;
   CONSTANT AndTable: LogicTable :=
          ( -- 'X'  '0'  '1'  'Z'
              ('X', '0', 'X', 'X'), -- 'X'
              ('0', '0', '0', '0'), -- '0'
              ('X', '0', '1', '1'), -- '1'
              ('X', '0', '1', 'Z'), -- 'Z'
          );
   VARIABLE Result: MVL := '1';
BEGIN
   FOR i IN Inputs'range LOOP
      Result := AndTable(Result, Inputs(i));
      IF Result = '0' THEN
         RETURN '0';
      END IF;
   END LOOP;
   RETURN Result;
END WiredAnd;
```

Reprinted with permission. Copyright © by Paul J. Menchini.

Figure 3-4. Example of a Resolution Function (Ref. 11)

clared as an unresolved signal. Unresolved signals typically require less simulation overhead than resolved signals and are therefore more efficient. A typical abstract data type for signals is provided in both a resolved and unresolved form. IEEE Std 1164 (Ref. 6) includes a resolution function in its VHDL package specifying the data types for logic-level signals.

3-3 VHDL SUPPORT FOR BEHAVIORAL DESIGN

One of the most powerful features of VHDL is its ability to support abstract, technology-independent descriptions of hardware in the form of behavioral models. Behavioral models model the function and timing of an electronic system. VHDL has features that allow creation of implementation-independent behavioral architecture bodies.

VHDL provides support for behavioral modeling with both concurrent and sequential execution modes. A behavioral architecture body may contain multiple processes, all of which execute concurrently. However, statements within a given process are executed sequentially.

3-3.1 PROCESSES

Processes are the VHDL construct that supports sequential modes of execution. A process contains a sequence of statements executed sequentially when the process is activated.

Control constructs, which may occur in processes, include loops, conditionals, and assignment statements. Assignment statements in processes include variable assignment statements and sequential signal assignment statements. Sequential signal assignment statements allow processes to update signal values over time.

Processes cannot be nested, but the function of a process can be organized hierarchically through the use of functions and subroutines. Communication between statements within a process and between a process and the functions and subroutines that it calls can be accomplished using variables. In most simulations, assignment to variables is much more efficient than assignment to signals, so the use of variables is preferred to the use of signals. The current value of a signal can also be assigned to a variable as a way to communicate from the external environment into a process. Communication from a process to its external environment is accomplished through signal assignments.

A process may have an explicit sensitivity list, which specifies a list of signals such that the change in value of any signal on the list will cause the process to be activated. Wait statements in a process specify when the process will be suspended and when it will resume. A process must have either a sensitivity list or a wait statement.

The state of a process, as defined by its variables, persists through a simulation. In contrast, variables local to a subroutine are not persistent and are reinitialized each time the subroutine is called.

3-3.2 WAIT STATEMENTS

Wait statements provide a mechanism used to suspend a process and may be used to synchronize processes. When a wait statement is executed, execution of the process containing the wait statement is suspended until the conditions of the wait statement are satisfied. When the conditions are met, execution of the process resumes.

The optional clauses of a wait statement (sensitivity clause, condition clause, and timeout clause) provide a variety of ways to control execution of a process. The sensitivity clause of a wait statement contains a list of signals referred to as a sensitivity list. Changes in the current values of signals on the list may (depending upon the condition clause of the wait statement) cause the process to resume execution. A wait statement with a timeout clause can be used to introduce timing delays into functional models. See subpar. 2-3.3 for discussion of some of the limitations of this approach in defining timing.

3-3.3 A BEHAVIORAL DESIGN EXAMPLE

Fig. 3-5 shows a behavioral architecture body for the edge detection processor described in subpar. 2-3.3 and shown in

```
architecture behavior of edge_detector is
begin
   sobel: process
   variable A:  image;      -- Internal frame buffer for image
   variable H:  filter_out; -- Temporary storage for results of
                            -- horizontal filter
   variable V:  filter_out; -- Temporary storage for results of
                            -- vertical filter
   variable LD: filter_out; -- Temporary storage for results of
                            -- left diagonal filter
   variable RD: filter_out; -- Temporary storage for results of
                            -- right diagonal filter
   begin
      -- Construct a complete image frame by reading
      -- in the pixels in scan line order
      for i in x_index loop
         for j in y_index loop
            wait until rising_edge(Clock);
            A(i,j) := P;
         end loop;
      end loop;
assert (false) report "array read in";
      wait for pixel_output_delay;
      -- For each pixel in the output image
      -- compute the values of all the filters,
      -- then use these filter values to compute
      -- the magnitude and direction outputs
      for i in x_out_index loop
         for j in y_out_index loop
            wait until rising_edge(Clock);
            H  := horizontal_filter(A,i,j);
            V  := vertical_filter(A,i,j);
            LD := left_diagonal_filter(A,i,j);
            RD := right_diagonal_filter(A,i,j);
            E  <= magnitude(H,V,LD,RD);
            D  <= direct(H,V,LD,RD);
          end loop;
       end loop;
    end process sobel;
end behavior;
```

Figure 3-5. Example of a Behavioral Model

Fig. 2-6. The entity interface declaration for this design entity is shown in Fig. 3-2.

This architecture body consists of a single process. The process contains two major nested loops. In both loops, the `for` loop control structure is used. The first loop uses a wait statement to synchronize loading pixel values from the input signal `P` into the variable `A`. The wait statement has in its condition clause the second input signal `Clock`. The wait statement uses the `rising_edge` routine from the IEEE `std_logic_1164` package to catch the rising edge of the clock signal.

The second loop computes and outputs the pixel values. The output is accomplished by signal assignment statements assigning values to the output signals `E` and `D`. These signal assignment statements also specify the timing in a parameterized way through the use of the constant `pixel_output_delay`. The wait statement in the loop synchronizes the output of pixels with the clock and prevents the values from overwriting each other.

Between the two loops is a separate wait statement that causes the delay associated with computing the output pixel values. An assertion statement (discussed in subpar. 3-7.1) is used to assist debugging by printing a message when input of the image is complete.

This architecture body uses the package of data type definitions and function specifications in the package `image_processing`, shown in Figs. 2.4 and Fig. 2.7. References to the `image_processing` package in the `sobel_algorithm` library are allowed because the entity interface shown in Fig. 3-2 includes `library` and `use` statements referring to this library and package, and these references are inherited by the architecture body. Use of this package allows the system to be parameterized in several ways. For example, the number of pixels in the image can be changed without changing the architecture body. Similarly, the number of bits of precision in a pixel can be changed, or the data type for a pixel could be changed from integer to an unsigned natural number or a bit vector.

This behavioral model uses functions to achieve a hierarchical organization. The calling hierarchy for this model is shown in Fig. 2-3.

This behavioral model specifies both the function and timing of the system. The timing information is introduced through the `wait` statements in the input and output loops and the `wait` statement between the two loops. The timing information is parameterized because the delays are specified with constants. The value of the `pixel_output_delay` constant is specified in the `timing` package.

3-4 VHDL SUPPORT FOR STRUCTURAL DESIGN

Structural models can be used to model the actual or proposed physical structure of a digital system. VHDL structural architecture bodies support hierarchy by allowing a design entity to bind other design entities to instances of its components. The generic maps of component instantiation statements allow attribute values to flow down through the structural hierarchy with appropriate modifications at each level.

3-4.1 STRUCTURAL ARCHITECTURE BODIES

A structural architecture uses only component instances and their interconnections to define its structure. The components are bound to design entities during elaboration. This binding provides support for hierarchical structural models.

A VHDL structural description can be visualized as an unpopulated board. The component declarations define a parts list for the board and specify the pins on those parts. In this analogy the ports specified in the declaration of component `C` define the pins of `C`. The component instances are sockets whose pins are wired to the traces on the board. The port maps of component instantiation statements define the wiring of the pins to the traces. The traces can be internal signals that are traces local to the board or interface signals that connect to the board edge connector through the ports specified in the entity declaration to onboard socket pins.

Lower level design entities represent the devices to be plugged into the sockets. “Binding” is the act of doing so. The design entity to be bound to a component may be selected on an instance-by-instance basis by means of a configuration.

The major language features supporting structural descriptions are component declarations and component instantiations. These features are described in subpar. 3-4.2.

3-4.2 COMPONENTS

In VHDL, components represent the outlines of individual hardware entities from which a larger design entity is composed. Before a component can be used in a model, it must be declared with a component declaration statement. A component is incorporated into a model by means of the component instantiation statement. Multiple component instantiation statements may refer to the same component declaration, just as a typical hardware board may use many copies of the same circuit.

The link between physical components and the corresponding components in the VHDL model can be reinforced through the naming of components and the annotation of component instances. VHDL allows different attribute values to be associated with different instances of the same component. The EIA 567 standard (Ref. 10) describes the concept of an electronic data sheet, in which a data sheet is associated with each component in the “parts list”. Attributes in the electronic data sheet are used to compute timing for elements of the model. This concept is described in more detail in par. 3-6 and Chapter 5.

3-4.2.1 Component Declarations

Component declarations can be thought of as defining an inventory of components, which can be reused as many times as necessary in the model. Component declarations specify and name the ports and generic constants of components. A port clause in a component declaration serves to declare the ports of the component. It must contain the names, directions, types, and default values, if any, of the ports in the component. VHDL analyzers check that component instances refer to components that have been declared and that the port map for the instance is consistent with the parts of the component declaration. This consistency checking helps to catch errors in VHDL models during the model development process.

Component declarations can be placed in packages, and this placement allows them to be reused. This approach is particularly valuable if a common or prescribed parts list is required across multiple hardware modules.

3-4.2.2 Component Instantiations and Interconnections

A component that has previously been declared can be used in an architecture body via a component instantiation statement. A component instantiation statement gives the component instance a unique name and associates the ports of the component with the signals that convey information to and from the component. Use of component instantiations results in a structural model that is a network of components connected by signals.

Component instantiation statements provide three types of associations: (1) associating a design entity (with a particular architecture body) with the component instance, (2) associating ports in the entity interface with locally declared signals, and (3) associating the values of generic constants with the generics defined in the entity interface. These associations can occur in two places: in the structural architecture body (This is a form of early binding or “hard wiring” of the information.) or in a separate configuration declaration (This is a form of late binding.). An early binding can be changed only by editing and analyzing the architecture body, whereas a late binding can be changed without editing the architecture body. The use of external configuration declarations is discussed in subpar. 3-8.3.

Signals are associated with the ports of a component instance in the port map of the component instantiation statement. When the same signal is used in multiple port maps, a signal net is defined. The direction information in the port declaration of the component determines the sources and sinks for the net.

Component declarations (and their corresponding instantiations) in VHDL are placeholders. The design entities used to model the components cannot be specified in either the component declaration or the component instantiation. This lack of dependency supports top-down model development because the lower level design entities need not be defined and analyzed before the higher level design entities are created. However, when both levels of design entities are defined and a configuration specification is used to associate the lower level design entity with the component instance at a higher level, the consistency-checking capabilities of VHDL ensure the consistency of the models. This consistency allows a behavioral model of a component to be replaced by a structural model by changing and reanalyzing the configuration information.

3-4.3 A STRUCTURAL DESIGN EXAMPLE

Fig. 3-6 contains a structural architecture body for the horizontal filter described in subpar. 2-4.3.2. The entity interface declaration corresponding to this architecture body is shown in Fig. 2-14. This architecture body has three input ports, `P1` and `P3` of type `pixel` and `Clock` of type `std_ulogic`. It has a single output port `H` of type `filter_out`.

This example illustrates several points. It shows the use of component and signal declarations, the use of component instantiations, and the association of signals with the ports of a component.

The architecture body in Fig. 3-6 declares three components: an adder, a subtractor, and a delay. In each of these three components the port list names the ports and defines the direction and the data type of the ports. The data types are specified in the `image_processing` package in the `sobel_structure` library. The image-processing package is shown in Fig 2-4, and the package body is illustrated in Fig. 2-7.

The particular design entity, i.e., an entity interface and a corresponding architecture body, to be bound to each instance is selected in a separate configuration specification. The use of configuration specifications adds flexibility by deferring selection of particular versions until the model is ready to be simulated. Configuration specifications and declarations are discussed in subpar. 3-8.3.

The architecture body in Fig. 3-6 declares four signals: `S1`, `S2`, `S3`, and `S4`. These signals are used to carry information among the component instances in the model. All four signals have the same user-defined type, `filter_out`. This type is used for all in ports and out ports of the components except for the data input ports of the `subtractor` component and for the `Clock` ports on the `adder` and `delay` components.

The `begin` in Fig. 3-6 designates the start of the executable statement part of the architecture body. This part contains the component instantiation statements that describe the structure of the architecture body. Each component instance has a label, which must be unique within a particular architecture body. After the label is the name of the component being instantiated. This model shows that a single component can be replicated as many times as needed, e.g., there are two instances of `delay` and two instances of `adder`. Each replication, however, must have a unique instance label. The instance labels for the adders are `ADD1` and `ADD2`.

Lastly, each component instance is connected to the signals by associating each signal with a particular port in the order in which the ports are listed in the component declaration. This association could also be done by name, which

```
architecture structure of horizontal_filter is
    component subtractor
        port ( A1:    in  pixel;
               A2:    in  pixel;
               Clock: in  std_ulogic;
               DIFF:  out filter_out );
    end component;
    component adder
        port ( A1:    in  filter_out;
               A2:    in  filter_out;
               Clock: in  std_ulogic;
               SUM:   out filter_out );
    end component;
    component delay
        port ( A_IN:  in  filter_out;
               Clock: in  std_ulogic;
               A_OUT: out filter_out );
    end component;
    signal S1: filter_out; -- Connects difference to 1st delay and 1st adder
    signal S2: filter_out; -- Connects 1st delay to 1st adder
    signal S3: filter_out; -- Connects 1st adder to 2nd delay and 2nd adder
    signal S4: filter_out; -- Connects 2nd delay to 2nd adder

begin
    SUB:  subtractor port map (P1, P3, Clock, S1);
    DEL1: delay port map (S1, Clock, S2);
    ADD1: adder port map (S1, S2, Clock, S3);
    DEL2: delay port map (S3, Clock, S4);
    ADD2: adder port map (S3, S4, Clock, H);
end structure;
```

Figure 3-6. A Structural Architecture Body

would allow the signal port pins to be listed in an arbitrary order. The sources and sinks for the signals are implied by the port list in the entity declaration and the port lists in the component declarations. For example, `Clock` is a signal with an external source and five sinks, one for each component instance. Similarly, `S1` has as its source the port `DIFF` in the instance with label `SUB` and has as its sinks the port `A1` of `ADD1` and the port of `A_IN` of `DEL1`. Also `H` has the port `SUM` of `ADD2` as its only source and has one or more external sinks.

3-5 VHDL SUPPORT FOR DATA ABSTRACTION

Data abstraction is the practice of extracting the essential characteristics of data by creating user-defined data types and disregarding certain implementation details. Data abstraction is a powerful tool used to control the complexity of models. It allows a complex data structure to be defined in a single place in the code and thereby assures consistency in the definition throughout a model. It also is a tool to ensure consistent definitions of the operations on a user-defined data type. These aspects are critical to the interoperability of models.

Data abstraction is also a powerful tool used to isolate changes and thereby reduces the risk associated with mass changes in software. A single data type may have many different implementations at different levels of abstraction. These implementation details should be hidden from those users who do not have a need to know the structure. Thus, when the implementation of the data type changes, the changes in the VHDL code can be isolated to that section of the code which provides the implementation details. For example, the data type `pixel` has different representations at different levels of abstraction in the edge detector model. In the algorithm-level model `pixel` is defined as an integer. In the gate-level model `pixel` is defined as a bit vector with a specific number of bits.

Data abstraction is implemented in VHDL with user-defined types. A user-defined data type consists of a type definition together with the definitions of the functions that act on the data type. Examples of abstract data types include the `direction` and `image` types defined in Fig. 2-4.

VHDL has capabilities that allow the user to create new data types, and it has capabilities to overload subprogram and enumeration literal names. VHDL also enhances interoperability by supporting the definition and use of type conversion functions to interface design entities that were written using different interface data types.

3-5.1 USER-DEFINED TYPES

VHDL has two mechanisms that allow the user to create new scalar types, and it has two methods used to create composite types.

The user can create a new scalar type in VHDL by defining an enumerated type or by defining a physical type. Physical types are described in subpar. 3-6.3. An enumerated type is defined by listing all of the possible values of an object of that type. An example of an enumerated type is the IEEE Std 1164 `std_ulogic` type, which consists of nine possible values, as shown in Fig. 3-7. Enumerated types have an explicit ordering specified by the order in which they are listed in the declaration. Variables and signals can have values that are enumerated types and can be assigned values that are enumerated types. Values of enumerated types can be compared using the relational operators =, /=, <, <=, >, and >=. For example, `'X' < 'Z'` because `'X'` is listed before `'Z'` in the declaration in Fig. 3-7.

VHDL includes four built-in enumerated types: `character`, `boolean`, `bit`, and `severity_level`. VHDL includes additional built-in logical operators for the `boolean` and `bit` enumerated types: `and`, `or`, `nand`, `nor`, `not`, `xor`. The `severity_level` built-in type is described in par. 3-7.

A composite type is created by aggregating simpler types. There are two kinds of composite types: arrays and records. An array type is created by aggregating a collection of elements of the same subtype. The elements of an array are selected by using an index. For example, a bit vector is created by aggregating a homogeneous array of bits. A record type is created by aggregating a heterogeneous collection of elements, each of which must be named at analysis time. A bus with multiple control, address, and data lines can be created by aggregating a type for the control lines (which may again be a composite type), a type for the address lines, and a type for the data lines.

VHDL also supports access types, which are similar to the pointer data types of C and PL/I. However, signals cannot be declared as access types. VHDL also supports file types for use in the input of test vector files and in the output of messages and trace data. Signals also may not be a file type.

Subtypes are another option available to the user. A VHDL subtype inherits the operations defined for the parent type but restricts the possible values of variables, constants, or signals declared as the subtype. Error messages are generated when an operation produces a value that is not within the subtype. For example, an array type may be defined with an unrestricted, i.e., integer, range. A subtype of the array may be defined as having a restricted range, e.g., “0 to 10”.

Subtypes are an important mechanism used to define labeled types without also defining the functions allowed for the data type. As such, they are a relatively simple method of using a VHDL analyzer to support consistency checking.

3-5.2 TYPE CONVERSION FUNCTIONS

Type conversion functions provide a mechanism used to make incompatible design entities work together. For example, type conversion functions may be required to make models at different levels of abstraction interoperate. If one design entity uses integer types for its I/O ports and another design entity uses bit vectors, type conversion functions can be used to make these two design entities interoperate.

Type conversions can be specified in component instantiation statements. A port map specification in a component instantiation statement can list a type conversion function applied to a signal rather than listing only a signal as being connected to the port. This procedure allows late binding of type conversion to a signal. The IEEE Std 1164 package (Ref. 6) includes type conversion functions for some commonly used logic types. These functions are included in the IEEE Std 1164 package to support the interoperability of 1164-compatible models with models that were not built with the full 1164 logic set. Similarly, the IEEE synthesis package (Ref. 12) provides type conversion functions used

```
-------------------------------------------------------------------
-- logic state system  (unresolved)
-------------------------------------------------------------------
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
-------------------------------------------------------------------
```

Figure 3-7. An Enumerated Type: The IEEE Std 1164 Unresolved Logic Data Type (Ref. 6)

to convert twos complement and sign-magnitude integers into bit vectors and vice versa.

3-5.3 OVERLOADED OPERATORS

An operator is a computation that is a recognized part of the VHDL language. Binary operators (operators with two operands) typically use the infix notation, e.g., `A + B`, rather than the more general function notation, e.g., `+(A,B)`. The VHDL language comes with a set of operators that are defined on the built-in data types of the language. An overloaded operator is an operator that performs functions depending upon the type of its operands. For example, the addition operator may be overloaded to perform integer addition if its operands are integers and real addition if its operands are real numbers. Operator overloading increases the readability of VHDL models and allows the same operations on different types to be identically named.

The IEEE Std 1164 package (Ref. 6) includes definitions of the overloaded operators "`and`", "`nand`", "`or`", "`nor`", "`xor`", and "`not`". The package provides overloading for situations in which both operands are either `std_ulogic` or both are `std_logic`.

3-6 VHDL SUPPORT FOR ANNOTATING MODELS

Annotation is the practice of incorporating information into the model that may not be directly related to the function of the model but that can provide a more accurate description of a particular implementation. An example is the temperature range over which the device is expected to operate.

Information that is not used during simulation can also be incorporated into a VHDL model as a form of documentation that can be processed by VHDL analyzers. This kind of information can be used by external tools that can extract it from the VHDL description.

The major VHDL features that support design annotation are constants, attributes, and physical types. VHDL allows the user to define attributes, to associate attributes with VHDL signals, design entities, and components, and to use attribute values to compute the function and timing of VHDL components. VHDL supports the definition of data types called physical types, which are designed to support the definition of attributes. VHDL allows the user to define physical types and units and relations between units of the same physical type. VHDL includes a single built-in physical type, the type time. VHDL allows constants to be defined and shared by multiple design units through the use of packages, it supports deferred definition for constant values, and it supports parameterized models through the use of generics.

Because user-defined attributes are constants, they can be assigned values at elaboration time by generics, just as other constants can. Attributes have the advantage of being attached to specific objects; constants are not. Constants can have their value definitions deferred and can be collected into packages; therefore, it is easier to access common constant values from multiple design entities.

The approach taken by VITAL (Ref. 9) and EIA 567 (Ref. 10) is to use constants defined in packages, and part of the constant record structure is the link back to the originating part, not attributes. Attributes are used for purposes other than back annotation, e.g., the `VITAL_Level` attribute associated with an architecture body. One of the things that VHDL 93 (Ref. 3) provides to make attributes easier to use is the built-in path attribute. This attribute simplifies finding a specific instance in which an attribute value needs to be set.

For constants or attributes to be used effectively to document a model, they must be used consistently throughout the model. The EIA 567 (Ref. 10) defines a set of constants for device models and functions that use these constants to define and check the timing of the models. The EIA constants describe an electronic data sheet, which has three views: physical, electrical, and timing. Each of these views is specified with a VHDL package that defines a collection of data types including, in particular, data types for the constants.

3-6.1 ATTRIBUTES

Attributes are the primary VHDL construct that supports the annotation of models with user-specified data. This information might include vendor part numbers, drawing numbers, power dissipation, or almost any other information a user might want to include. VHDL includes predefined attributes that provide information about named entities. Of particular value in this regard are attributes describing the state of signals, such as `stable` or `event`.

The value of an attribute is accessible to the VHDL design unit in which the attribute is declared; tools have been developed that interact with VHDL analyzers to access and manipulate these values. Attribute values can be used to compute the timing or modify the function of a design entity. Attributes have types, which are assigned by attribute declarations. Because VHDL provides an extensive facility with which to define and check types, the VHDL type mechanism provides great flexibility in including additional information and checking the consistency of the information added to a VHDL description. Attributes that are not predefined are constants, but they may be given values by generics after analysis. Generics are discussed in subpar. 3-6.2.

Attributes should be distinguished from comments as a form of documentation. VHDL allows comments, but VHDL analyzers ignore the text of comments. Thus a VHDL analyzer has no control over the consistency of information in comments. However, VHDL attributes are parsed and type checked by VHDL analyzers. VHDL analyzers will also compute the value of attribute expressions that are assigned values at analysis time.

3-6.2 GENERIC CONSTANTS

Generic constants are an important mechanism used to parameterize VHDL models. Parameterized models are easier to reuse because they are designed to support some level of change external to the model. Generic constants are elaboration-time parameters. Since their values are constant during simulation, they do not imply the performance penalty associated with run-time parameters. Both the EIA 567 (Ref. 10) and VITAL (Ref. 9) use generics to define the value of timing parameters.

VHDL expressions can include generic constants whose values are fixed when the model is elaborated. The value of a generic constant is specified when it is used in a component instantiation statement or when its design entity is referenced in a configuration specification. The use of configuration declarations to set generic constant values is shown in subpar. 3-8.3.2.

Configuration declarations provide a mechanism within VHDL to do back annotation (Ref. 13). The VHDL structural model is analyzed, and netlists are extracted from the analyzed model. External timing tools are used to analyze the netlist and compute timing values based on factors such as parasitic capacitance. The external tool then generates a configuration declaration containing the timing information it has computed. When the model is ready for simulation, the configuration declaration is elaborated so that the timing attributes of the model are assigned the values computed by the external tool.

Fig. 3-8 illustrates the use of generic constants and attributes in an entity interface. In this example an attribute is declared as part of an entity interface declaration. The value of the attribute is computed from the values of generic constants that are inherited either from a component instantiation or a configuration specification.

In Fig. 3-8 a function `derate` is assumed to take the generic constants `base_delay` and `temperature` as arguments and return the appropriate value. This function is called when the model is elaborated. The delay computed by this function has been parameterized in terms of the two parameters, `base_delay` and `temperature`.

An architecture body for the interface of Fig. 3-8 is shown in Fig. 3-9. This body uses the attribute with the name `fcn_delay` and is associated with the entity interface `t_or` for the time delay in the signal assignment statement.

The attribute value has been used in the signal assignment statements in place of a fixed time value. Thus, the same entity-architecture pair can be reused many times with possibly different values for the generic constants without rewriting the VHDL source code.

3-6.3 PHYSICAL TYPES

Physical types represent measurable physical quantities. VHDL provides facilities to define physical types and to check that those types are used consistently in the model. A physical type definition is characterized by an integer range

```
LIBRARY timed;
USE timed.util;
ENTITY t_or IS
    GENERIC(temperature: float;
            base_delay:  time );
    PORT( in1:  IN bit;
          in2:  IN bit;
          out1: OUT bit );
    ATTRIBUTE fcn_delay: time;
BEGIN
    ATTRIBUTE fcn_delay OF t_or:entity IS
        util.derate(base_delay, temperature);
END t_or;
```

Figure 3-8. Entity Interface Declaration With Generic Constants and an Attribute

```
ARCHITECTURE behavior OF t_or IS
BEGIN
    or_proc : PROCESS(A, B)
        BEGIN
           Y <=  A or B AFTER t_or' fcn_delay;
    END PROCESS;
END behavior;
```

Figure 3-9. Architecture Body Using an Attribute

and a base unit of measurement. Secondary units of measurement may also be declared for a physical type, along with an equation defining the relationship of the secondary unit of measurement to some primary unit of measurement. VHDL comes equipped with a single built-in physical type: time.

Fig. 3-10 illustrates the declaration of a physical type, which in this case is a type measuring distance. The base unit of measurement is the angstrom; other secondary units of measurement are also specified in both metric and English units.

```
type Distance is range 0 to 1E18
   units
   -- base unit:
      A;             -- angstrom
   -- metric units:
         nm   =   10 A;   -- nanometer
         um   = 1000 nm;  -- micron
         mm   = 1000 um;  -- millimeter
         cm   =   10 mm;  -- centimeter
         m    =  100 cm;  -- meter
         km   = 1000 m;   -- kilometer
   -- english units:
         mil  = 25400 A;  -- mil
         inch = 1000 mil; -- inch
         ft   = 12 inch;  -- foot
         yd   = 3 ft;     -- yard
         fm   = 6 ft;     -- fathom
         mi   = 5280 ft;  -- mile
         lg   = 3 mi;     -- league
```

Copyright © 1993. IEEE. All rights reserved.

Figure 3-10. Example of a Physical Type Declaration (Ref. 3)

Physical types provide a powerful mechanism to increase the understandability and consistency of attribute definitions. The EIA 567 (Ref. 10) physical and electrical views use physical types extensively to create an electronic data sheet. The EIA 567 physical and electrical views are discussed in more detail in subpar. 6-3.3.

3-7 ERROR HANDLING WITH VHDL

When a model is used as part of a larger system, it is possible that some of its operating conditions may be violated. For example, a timing violation may be observed that may cause incorrect operation of the circuit. Users should be informed of these violations so the incorrect operating condition can be corrected. The VHDL DID (Ref. 4) requires certain types of error checking; subpar. 7-4.3 describes these requirements in more detail.

VHDL provides a special mechanism to detect errors: assertion statements. Another way to flag errors is to extend the data types for signals to include error states. These approaches are described in the following subparagraphs.

3-7.1 ASSERTION STATEMENTS

Assertion statements are one mechanism to detect and report errors. Assertion statements provide a relatively simple way to check some of the behavioral or operating conditions of a model and can be used to check signal timing at the ports of an entity interface. For example, assertion statements may be used in entity interfaces, architecture bodies, and subprogram bodies. Assertion statements appear in any sequential or concurrent statement part. One use of passive processes is to encapsulate assertion statements. Passive processes can be defined in packages and can be made available for use in multiple design entities.

As shown in Fig. 3-11, an assertion statement consists of a condition, an optional report clause, and an optional severity clause. The condition must evaluate to a Boolean value. If the condition of an assertion statement evaluates to `FALSE`, the report string is displayed with the designated severity. The report clause string is displayed in an implementation-dependent fashion. There are four possible values of a severity code: `note`, `warning`, `error`, and `failure`.The action of the simulator for each level of severity is implementation dependent, and some simulators allow the user to specify the action to be taken and/or the severity level that will terminate the simulation. Synthesis tools may use the assertion conditions as invariants.

A concurrent assertion statement executes when a signal that is referenced in the condition section of the assertion

```
entity RSFF is
    port ( R, S:     in     Bit;
           Q, QBar: buffer Bit);
    begin
       CheckInputConstraint:
          assert R = '0' or S = '0'
             report "R and S inputs both asserted!"
             severity Failure;
end RSFF;
```

Reprinted with permission. Copyright © by Paul J. Menchini.

Figure 3-11. An Assertion Statement (Ref. 11)

statement changes value. Sequential assertion statements in processes or subprograms are executed in the order in which they appear. Assertion violations are reported with either a default or a user-specified message.

Fig. 3-11 shows an example of an assertion statement for an R(reset)S(set) flip-flop. This type of flip-flop cannot tolerate `'1'` values on both the `S` (set) and `R` (reset) inputs at the same time. The assertion statement `CheckInputConstraint` is designed to detect this anomaly.

More elaborate error detection can be done with passive processes. Assertion statements can invoke functions in their condition or in their report and severity expressions. However, assertions do not provide the same degree of computational sophistication that is available in a passive process since functions do not maintain state between calls. Passive processes do retain state and therefore can provide testing of assertions that require some history to be maintained.

Both VITAL (Ref. 9) and EIA 567 (Ref. 10) provide functions designed to check timing and to be used in the condition parts of assertion statements.

3-7.2 HANDLING SIGNAL ERROR STATES

VHDL allows designers to specify any logic level convention desired. To help detect and propagate errors, logic level conventions are often extended to include signal error states. This approach has been used in IEEE Std 1164 (Ref. 6), which includes the `'X'` and `'W'` states to mark errors and 'U' to mark uninitialized objects, as shown in Fig. 3-7. Because any logic level may appear as a signal value during the course of a simulation, it is important that processes using `std_logic` signals be able to handle all possible signal levels, including signal error states. Furthermore, error states should be propagated to the outputs so that when an error occurs, it can be detected at the external boundary of the system. One method used to provide these error handling capabilities is to overload the operators for the normal functions. This approach has been used in IEEE Std 1164 (Ref. 6); Fig. 3-12 illustrates this approach. The figure shows the logic table for the logical `and` function. The results for `and` applied to the values `'0'` and `'1'` match the traditional definition, but the definition has been extended to deal with all nine values defined in the IEEE Std 1164 data type `std_ulogic` (Fig. 3-7). This table for the `and` function should be compared with the table for the `WiredAnd` resolution function shown in Fig. 3-4, which also propagates its error states.

Effective use of packages to encapsulate error state data types and functions can prevent the need to change VHDL models that use the logic, as discussed in subpar. 3-8.2.

3-8 VHDL SUPPORT FOR SHARING AND REUSE

VHDL was developed with many features that support sharing and design reuse. These features help to minimize effort duplication and to ensure that consistent models are used throughout a design. Sharing and reuse are supported in VHDL by VHDL design libraries, packages, and configuration declarations.

VHDL libraries impose a structure on the models available to the user. VHDL libraries store design units that can be made available to the user. The user must indicate which libraries are used by a model. Depending upon the implementation, the library may also be useful for configuration management and access control.

Packages provide a mechanism to collect VHDL source statements for some common purpose. Such statements include data type declarations, attribute declarations, and subprogram declarations. These declaration statements can then be included in other VHDL design units. The package provides a common location for the source code so that revisions need to be made only once. Revisions of the package

```
-- truth table for "and" function
  CONSTANT and_table : stdlogic_table := (
  --      ----------------------------------------------------
  --      | U    X    0    1    Z    W    L    H    -        |   |
  --      ----------------------------------------------------
          ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
          ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
          ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
          ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
          ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
          ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
          ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
          ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
          ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
  );
```

Figure 3-12. An Example of Error Propagation: IEEE Std 1164 AND Operator Table (Ref. 6)

are then automatically used whenever a design unit that references the package is subsequently analyzed.

Configuration declarations provide a feature for late binding of architecture bodies to entity interfaces and late binding of values to generic constants. Because the configuration declarations can exist as separate files, they can reduce editing of other design units and thus reduce risk.

3-8.1 VHDL DESIGN LIBRARIES

VHDL libraries are used to store information that can be used (or reused) to construct new VHDL models and to provide a mechanism to partition a large design into manageable pieces. A library may contain a collection of frequently used parts, data and function definitions common to all VHDL design units in a model, or data and function definitions common to a particular model. Libraries have names that can be referenced via a VHDL library clause to make the contents of a library accessible. The contents of libraries may be made available for reference by use clauses or may be referenced directly using expanded names.

VHDL design libraries are the repository of VHDL design units. Existing design units can be referenced in a VHDL description by using expanded name of the library unit. The `use` clause makes the contents of a design unit visible, just as the `library` clause makes the library itself visible. The `use` clause provides a "shortcut" so a user does not have to repeat the expanded name of the library unit in every reference.

VHDL requires that an entity interface declaration `ent` must be analyzed, i.e., compiled, before any architecture body is associated with it. However, a VHDL design unit that references an entity interface declaration does not have to be modified or even reanalyzed when the architecture body is changed. Separating the analysis of entity interface declarations from the analysis of associated architecture bodies is a major risk reduction factor because anytime a program is modified, there is a significant possibility errors will be introduced. It is also a significant factor in reducing the time required to analyze a large model. Thus a well-designed VHDL model takes advantage of design entities as a mechanism to modularize the model as well as a mechanism to document the relationship between physical components and the VHDL model.

VHDL has two predefined libraries: `work` and `std`. Library `work` is the library specified by the user into which library units are analyzed. It usually contains the library units of a model under construction. The name `work` is intended as a temporary name for the current library. When the current library has been developed, it should be given a name, and appropriate references to this library should be inserted in the source code for the design units in the library. Library `std` contains the predefined VHDL packages `standard` and `textio`, which provide definitions and functions needed for all VHDL models. Packages associated with other IEEE standards are in other libraries, such as the `ieee` library being used by IEEE Std 1164 (Ref. 6).

The binding of library names to external storage is implementation dependent and therefore may vary from vendor to vendor and from design environment to design environment. This dependency results from variations in the file-naming conventions in different operating systems. One common implementation-dependent restriction on file names is the length of the name. To support portability of libraries, it is recommended that library names be no longer than eight characters.

Standards organizations are creating and populating their own libraries. For example, IEEE Std 1164 (Ref. 6) has defined a package called `std_logic_1164`, which is stored in the library `ieee`. Another example is IEEE Std 1029.1, the Waveform and Vector Exchange Specification (WAVES) standard (Ref. 14), which uses four libraries: (1) a WAVES standard library, (2) a library that contains code specific to particular automated test equipment (ATE), (3) the `work` library, which is where the module under test is stored, and (4) a local standard library. The partitioning of design units into the WAVES library is described in subpar. 7-3.1.

3-8.1.1 Declaring and Using Libraries

Libraries are referenced in a VHDL description in `library` and `use` clauses, and they may also be referenced in expanded names. The `library` clause specifies the particular libraries, and the `use` clause specifies what library units or declarations within a library are to be directly visible to the unit in which the clause occurs.

Each design unit implicitly contains the following context clause:

```
library std,work; use std.standard.all;
```

Because the current design unit is initially placed in the `work` library, it needs to have access to other design units in the same library. This implicit context clause provides this access and also makes the VHDL library `std` available. As mentioned in the previous paragraph, the `std` library contains predefined VHDL standard packages, such as `textio`.

A library clause and a use clause in a design unit context clause are shown in Fig. 3-13. In this example four libraries are named. Two libraries are implicitly named: `work` (the default working library) and `std` (the VHDL standard library), and two libraries are explicitly named: `ieee` (which contains the IEEE Std 1164 data type definitions) and `custom` (a library of predefined gate-level models). In library `std` there is a package named `standard`, the contents of which are made visible by the implicit use clause `use std.standard.all`. In addition, the IEEE Std 1164 definitions in library `ieee` are made visible by the use clause `use ieee.std_logic_1164.all`;.

In Fig. 3-13 the design unit is the architecture body `structure` of the design entity `imply`. The `custom` library contains models for the components that are used in this architecture body. The configuration specifications bind the component declarations of `c_or` and `c_inv` to specific design entities specifying both the entity interfaces `ttl_invert` and `ttl_or` in the library `custom` and the architecture bodies (both of which are named `behavior`). The component instantiations connect the external ports of `imply` (called A and Y) and the internal signal of `imply` (called `nota`) to the ports of the components as named in the entity declarations in the library `custom`. Fig. 3-14 illustrates how the library clauses, use clause, and configuration specifications in Fig. 3-13 link these design units together to create a VHDL model.

```
LIBRARY custom;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ARCHITECTURE structure OF imply IS
    -- Signal Declarations
    SIGNAL nota: std_ulogic;
    -- Component Declarations
    COMPONENT c_or
        PORT (in1:  IN  std_ulogic,
              in2:  IN  std_ulogic,
              out1: OUT std_ulogic);
    END COMPONENT;
    COMPONENT c_inv
        PORT (in1:  IN  std_ulogic,
              out1: OUT std_ulogic);
    END COMPONENT;
    -- Component Specifications
    FOR or1: c_or  USE
        ENTITY custom.ttl_or(behavior);
    END FOR;
    FOR inv1:c_inv USE
        ENTITY custom.ttl_invert(behavior);
    END FOR;
BEGIN
    -- Component Instantiations
    inv1:c_inv PORT MAP (in1 => A,  out1 => nota)
    or1: c_or  PORT MAP (in1 => nota, in2 => B, out1 => C)
END structure;
```

Figure 3-13. Using a Component Library to Configure a Structural Architecture Body

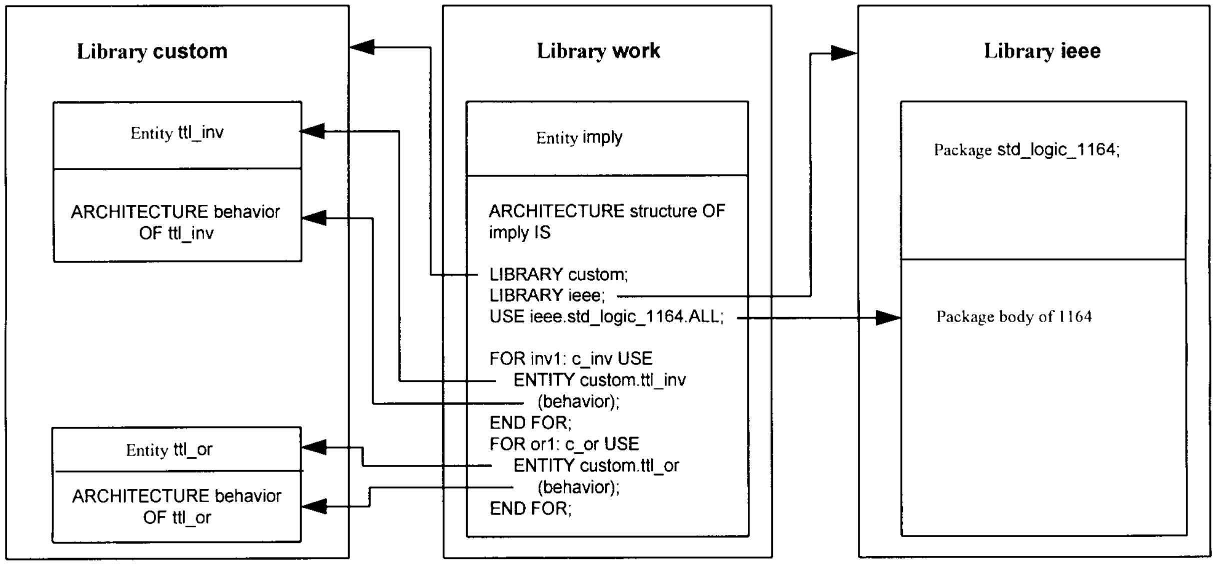


Figure 3-14. Use of Library and Use Clauses to Access External Libraries

Fig. 3-15 and Fig. 3-16 show the use of configuration specifications to select from libraries appropriate design entities and their architecture bodies for components in a structural model. Selection of appropriate architecture bodies is a key step in configuring a VHDL model. A single entity interface can have several associated architecture bodies. Different architecture bodies can represent different implementations of the same entity interface. Selecting an architecture body is a means of trading off or evaluating alternative implementations. Different architecture bodies may represent different levels of abstraction of a design entity. In this case selecting an architecture body determines the level of abstraction to be used for a particular component. Subpar. 10.2.1 of the VHDL DID requires both behavioral and structural models for all modules that are not leaf modules. Thus selecting an architecture for each component is an essential step in configuring a DID-compliant VHDL model.

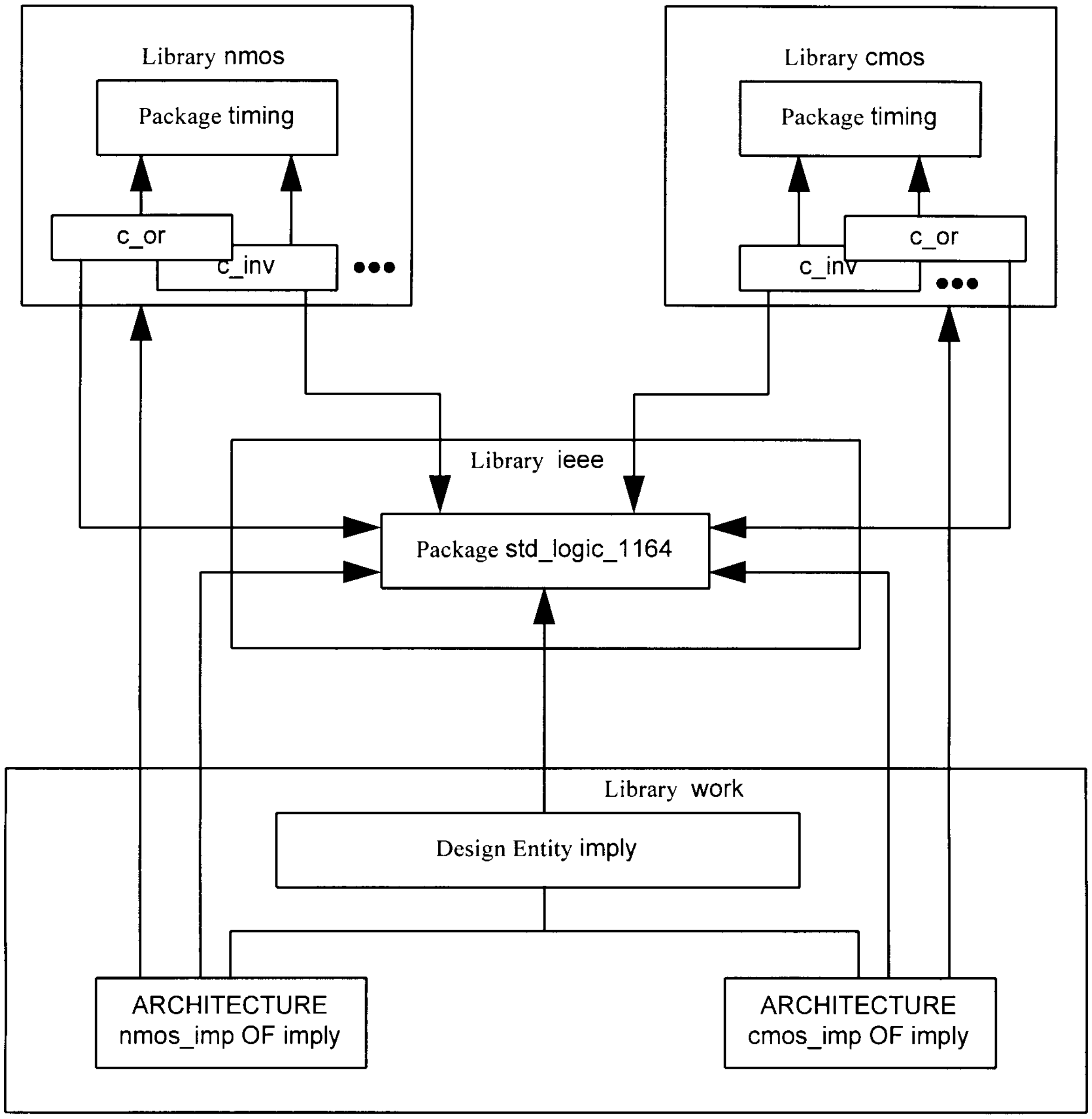


Figure 3-15. Using Different Architecture Bodies to Select Libraries

```
LIBRARY nmos;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ARCHITECTURE nmos_imp OF imply IS
    -- Signal declarations
    SIGNAL nota: std_ulogic;
    -- Component declarations
    COMPONENT c_or
        PORT (in1:  IN  std_ulogic,
              in2:  IN  std_ulogic,
              out1: OUT std_ulogic);
    END COMPONENT;
    COMPONENT c_inv
        PORT (in1:  IN  std_ulogic,
              out1: OUT std_ulogic);
    END COMPONENT;
    -- Component Specifications
    FOR c_or  USE
       nmos.c_or(behavior);
    END FOR;
    FOR c_inv USE
       nmos.c_inv(behavior);
    END FOR;
BEGIN
    -- Component Instantiations
    inv1:c_inv PORT MAP (in1 => A,  out1 => nota)
    or1: c_or  PORT MAP (in1 => nota, in2 => B, out1 => C)
END nmos_imp;
```

Figure 3-16. Technology-Dependent Architecture Body Using Configuration Specifications

3-8.1.2 Constructing Libraries

Setting up a VHDL library system involves an implementation-dependent procedure used to establish library names and their correspondence to external storage. Once a design library `lib` has been established, the VHDL analyzer adds design units to `lib` by binding `lib` to the library `work`. When all of the design units in `lib` have been analyzed, work on a new library can proceed by changing the binding of library `work`.

The VHDL source code for a design unit is usually stored in a text file. The VHDL analyzer parses the VHDL source code contained in the file and checks that it conforms to the language definition. The VHDL analyzer also builds an internal representation of the design unit and maintains a directory of the VHDL libraries and their contents.

Design units are divided into two classes: primary units and secondary units. Primary units specify interfaces. They include entity declarations, package declarations, and configuration declarations. Secondary units are the bodies associated with primary units, and they include architecture bodies and package bodies. All secondary units associated with a primary unit `prim` must be kept in the same library as `prim`.

For a VHDL analyzer to process a design unit `ent`, it must have previously analyzed all design units referenced by `ent`. In particular, secondary units must be analyzed after their corresponding primary units have been analyzed, and every design unit must be analyzed after all design units to which it refers. Thus a set of VHDL libraries (a model database) may have a complex set of dependencies that determines at least a partial order in which design units must be analyzed. This analysis order should be specified for a user in order to recreate a VHDL simulation model from the VHDL source code. The VHDL DID (Ref. 4) requires that this analysis order be provided with models delivered to the Government. The WAVES (Ref. 14) header file also requires this information.

The organization of design units into libraries is an important part of the configuration management of a VHDL design database. The partitioning of design units in a design database into libraries is usually done for one of two reasons. The first reason is to control read and write access to elements of a particular library. Write access is particularly important to establish who has the right to change the contents of a library. For example, if a large project has several teams, each team may have a separate library in which it is allowed to store modules. In fact, there may be separate libraries for different levels of confidence. Each user has a work library, and the team may have a team library. After unit testing, a design unit may be promoted from the user's work library to a common team library. After integration testing across all of the units for which the team is responsible, the library units in the team library may be promoted to a project library. The promotion of team library units becomes a formal milestone in the project schedule.

A second approach to partitioning design units into libraries is to collect design units that represent a particular design approach into a library. The goal of this approach is to isolate a set of changes (or differences) to a specific library. This approach is taken in WAVES (Ref. 14), in which all of the design units specific to a particular type of automatic test equipment are stored in a single library. If a design database contains multiple types of ATE, it will have multiple libraries, each containing different versions of the same design units.

3-8.2 VHDL PACKAGES

Packages in VHDL provide a way to share information, both within a single model and across models. A package may contain type declarations, attribute declarations, subprogram declarations, and other declarations. A package should be written and analyzed only once. Once analyzed, the information in a package is available for use by other VHDL library units within the same library and in external libraries, as shown in Fig. 3-14.

A VHDL package consists of two parts: the package declaration and the package body. The information in a package declaration can be used by the analyzer to check for certain types of errors, e.g., type mismatch errors. The package body contains the specifications of the values of any constants not defined in the package declaration and the bodies of any subprograms declared in the package declaration. The package body is analyzed separately from the package declaration.

The standards efforts related to VHDL make extensive use of VHDL packages as a way to use VHDL analyzers to enforce compliance with the standards. The IEEE Std 1164 (Ref. 6) uses a package to specify an abstract data type for extended logic. EIA 567 (Ref. 10) uses three packages to define its electronic data sheet. The `textio` package in the predefined VHDL standard library contains a collection of utility functions for textual input and output. The WAVES standard (Ref. 14) also uses packages to define standard data types.

3-8.2.1 Constructing VHDL Packages

Packages are particularly important as ways to define abstract data types such as the IEEE Std 1164 (Ref. 6) extended logic. The IEEE Std 1164 package declaration defines its logic data type as an enumerated type, as shown in Fig. 3-7. The package declares a resolution function, overloaded operators, and type conversion functions. Its package body provides the semantic definitions of the functions and operators. Fig. 3-12 shows a table of constants that is declared in the 1164 package body. This table is interpreted by the body of the `and` function. The IEEE Std 1164 allows alternative implementations of its package body in order to provide greater execution efficiency.

3-8.2.2 Declaring and Using Packages

The information in a package can be made visible selectively, or all of the information can be made visible with a `use` clause. The data type definitions for `image` and `filter_out` in Fig. 3-5 are made visible by the use clause `use sobel_algorithm.image_processing.all;` in Fig. 3-2. This same use clause makes such functions as `horizontal_filter` (which are stored in the package `image_processing`) accessible to the process. Similarly, the constant `pixel_output_delay` is defined in the package `timing`. The packages `image_processing` and `timing` are stored in the library `sobel_algorithm`.

Packages are an important mechanism of back annotation. EIA 567 (Ref. 10) specifies the package declarations; the user provides the package bodies as a form of back annotation. This structure makes extensive use of deferred constants to implement back annotation. A deferred constant is a constant with a package declaration whose value is specified in the package body. Any design unit that references a package has an analysis dependency on only the package declaration, not on the package body. As long as the package declaration has been analyzed, the package body can be constructed and analyzed at the user's leisure. (Of course, the package body must exist by the time the model is elaborated.) Thus a user can construct a VHDL structural model by referencing the constant in the package declaration, extract the netlist from the structural model, process the netlist with an external tool that generates the package body (including the value of the constant), analyze the generated package body, and then simulate the system using the back-annotated constant value. This EIA 567-compliant approach is described in more detail in subpar. 6-3.3.

VHDL places some restrictions on the use of deferred constants for back annotation. In particular, a package declaration can have only one associated package body, which must reside in the same library as the package declaration. In contrast, one entity declaration may have many architecture bodies, all of which must reside in the same library. For example, in a library there cannot exist a single timing view package declaration and separate package bodies for minimum time, maximum time, and nominal time. The EIA 567 standard (Ref. 10) includes all three times in a single package.

3-8.3 CONFIGURATION SPECIFICATIONS AND DECLARATIONS

Before a model can be simulated, the exact configuration of library units included in the simulation must be specified. That is, each component instance in the model must have a specific design entity (both entity declaration and architecture body) associated with it. Furthermore, all generic constants must be given a value. These associations can be made with VHDL configuration specifications or with declarations.

The ability to configure a model permits creation of many variations on a basic model without having to rewrite the VHDL source code. Different configurations are useful for exploring alternative implementations of functions, for incorporating different levels of abstraction into a simulation model, and for changing the values of parameters. Configuration specifications and declarations can be used to change the values of parameters by specifying values for generic constants.

3-8.3.1 Constructing Configuration Specifications and Declarations

A specific configuration (binding) can be provided either in the block in which the component instance appears or in a separate configuration declaration. Using a configuration specification "hard-wires" a body-particular design entity to a component instance. This method is useful when no alternative configurations are available or desirable. A change in the configuration in this case requires a modification to the source code containing the instance and its subsequent reanalysis. In contrast, use of a configuration declaration allows deferral of the final configuration decisions until after analysis of the instance. For example, when a VHDL model is used to document existing hardware, it may be desirable to use configuration specifications to define the timing information. If a VHDL model is used during the design of a component, i.e., when changes in layout and timing are frequent, the use of generics and configuration declarations may be preferred to reduce analysis time.

One way to specify timing information is through use of configuration specifications combined with use of deferred constants. VHDL constrains the way deferred constants can be used to define values for global parameters. Within a single library each package declaration has at most one body. Thus if different values are required for deferred constants, packages with the same interface but different bodies must be installed in different libraries.

There are two approaches to using these libraries in a structural model. The first approach, shown in Fig. 3-15, uses a single design entity for a system and separate structural architecture bodies. Each of the structural architecture bodies references the same package name but in a different technology library. The second approach, shown in Fig. 3-17, uses a single architecture body and two configuration declarations to associate design units from one or another of the libraries with the components of the body.

Fig. 3-15 illustrates the first approach with the `imply` function described in subpar. 3-8.1.1. Fig. 3-15 shows two technology libraries each containing a package of timing information and design entities that reference the IEEE standard logic package stored in a third library. Two design entities `c_inv` and `c_or` are shown in each library. Also shown is a work library containing a higher level entity interface (in this case, logical function `imply`) and two structural architecture bodies for the entity. The two architecture bodies use different contexts, i.e., different `library` and `use` clauses, to bind design entities to the component instances in the common architecture. The arrows represent the combination of `library` and `use` clauses. Fig. 3-16 shows the VHDL source code for one of these architecture bodies.

The two timing packages in the different libraries declare the same constants with the same types, but they assign different values to the constants. To make this division clear, the constants can be deferred so that the package declarations are identical and the differences occur only in the package bodies. The timing packages could also contain different derating functions for the different technologies. Because the two packages have the same names and the same constants and the design entities have the same names and the same port types and interfaces, entities in the `nmos` and `cmos` libraries can be used interchangeably, but they will have different timings.

These technology-dependent timing packages can be shared by many architectures and thus can provide a very compact representation of technology-dependent timing. Technology-dependent packages can also be used to define type conversion functions for appropriate subsets of the IEEE Std 1164 logic package or to define conversion functions to map the IEEE Std 1164 logic values to higher level data types.

Fig. 3-17 illustrates the second approach, in which there is only one architecture body, but two configuration declarations are used to select the appropriate libraries. In this approach the `library` clauses that reference the technology libraries are in the configuration declarations rather than being in the structural architecture body.

Because structural VHDL models can be constructed hierarchically, configuration declarations can also be constructed hierarchically. The nesting of block configurations reflects the hierarchy of the model being configured. The hierarchy can also be described piecemeal by having a configuration declaration reference another configuration declaration within the binding indication of a component configuration. In this case the hierarchy of dependencies of the configuration declarations reflects the hierarchy of the configured model.

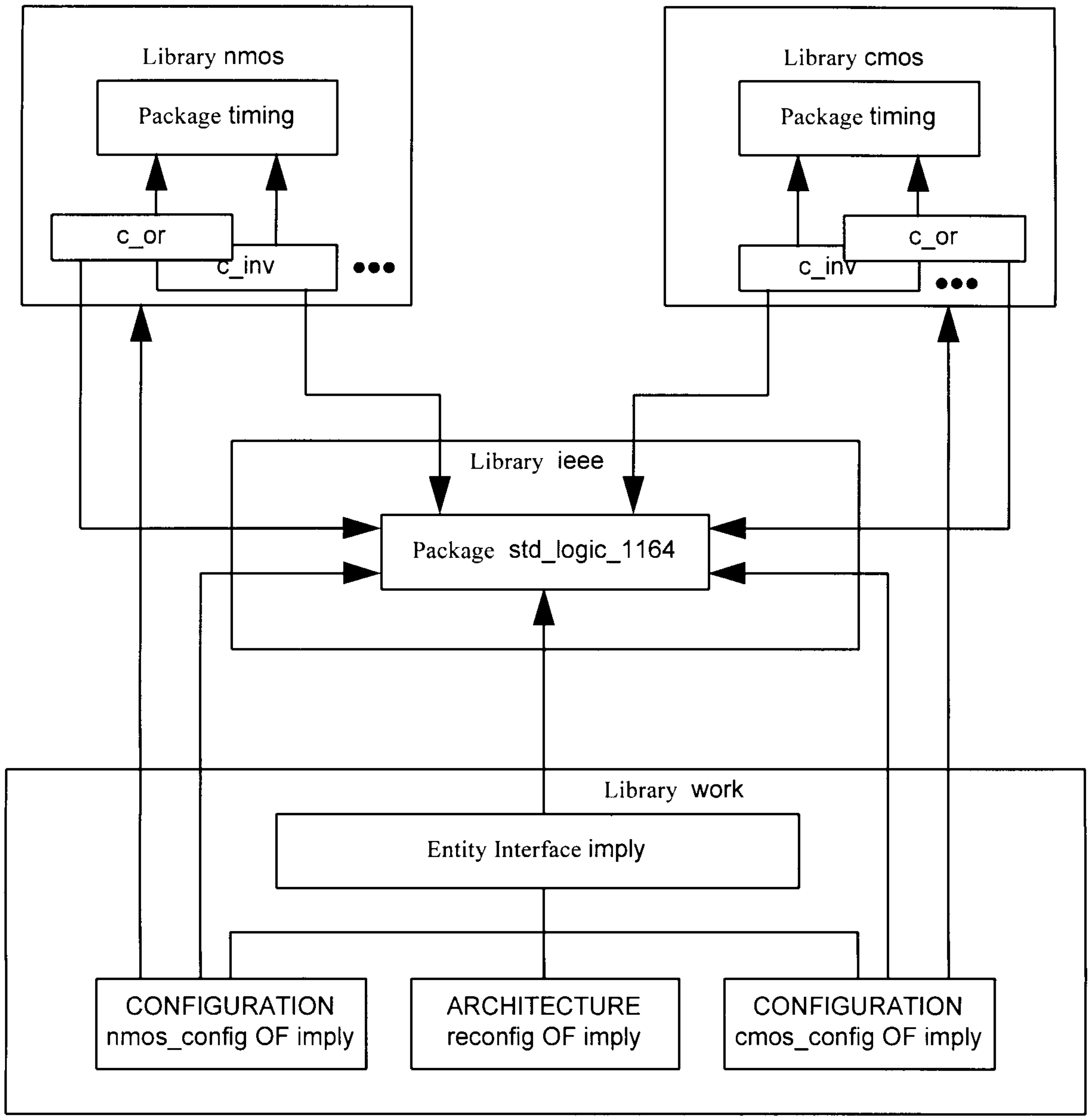


Figure 3-17. Use of Configuration Declarations to Select Alternative Design Libraries

3-8.3.2 Using Configuration Specifications and Declarations

Configuration specifications can be used within an architecture body (or block statement) when it is desired to specify a unique configuration for the architecture (or block). Once a component instance has been configured this way, it cannot be reconfigured without modifying the source code of the architecture body or block. A configuration declaration should be used when the overall model configuration may change during the course of model development and simulation. Fig. 3-13 illustrates the use of configuration specifications inside an architecture body. Fig. 3-16 shows the architecture body, which uses configuration specifications to include the negative metal-oxide semiconductor (NMOS) technology-specific timing information. In this architecture body the `nmos` library is specified. Furthermore, the configuration specifications require that the specific design entities in the `nmos` technology library be used for all instances of the `c_or` and `c_inv` components. Because the two architectures for `imply` have the same structure, these bindings could be delayed until elaboration, as discussed in subpar. 3-6.2. If the two architectures have different internal structures, this method of selection is necessary.

Fig. 3-18 shows a different version of the architecture body for `imply` that is designed for use with a separate configuration declaration. A corresponding configuration declaration is shown in Fig. 3-19.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ARCHITECTURE reconfig OF imply IS
    -- Component declarations
    COMPONENT c_or
        PORT (in1:  IN std_ulogic,
              in2:  IN std_ulogic,
              out1: OUT std_ulogic);
    END COMPONENT;
    COMPONENT c_inv
        PORT (in1:  IN std_ulogic,
              out1: OUT std_ulogic);
    END COMPONENT;
    -- Signal declarations
    SIGNAL nota: std_ulogic;
BEGIN
    -- Component instantiations
    inv1: c_inv PORT MAP (in1 => A, out1 => nota);
    or1:  c_or  PORT MAP (in1 => B, in2 => nota, out1 => Y);
END reconfig;
```

Figure 3-18. A Reconfigurable Architecture Body

```
LIBRARY nmos;
CONFIGURATION nmos_config OF imply IS
    FOR reconfig
        FOR or1: c_or
            USE ENTITY nmos.c_or(behavior);
        END FOR;
        FOR inv1: c_inv
            USE ENTITY nmos.c_inv(behavior);
        END FOR;
    END FOR;
END nmos_config;
```

Figure 3-19. Use of a Configuration Declaration to Select Design Entities From a Library

Both of the examples shown in Figs. 3-15 and 3-17 use deferred constants to provide the timing information. This practice allows determination of the timing values to be deferred until the package bodies are analyzed. An alternative approach that provides greater flexibility is the use of generic constants for timing information. For example, the architecture body shown in Fig. 3-18 can be used with the configuration declaration shown in Fig. 3-19.

The configuration declaration shown in Fig. 3-19 specifies the library, entity, and architecture body for each component instance of the architecture body `reconfig`. The outer `FOR` loop specifies the architecture body; the inner `FOR` loops specify the design entities to be used for each of the component instances. This configuration declaration contrasts with the configuration declaration in Fig. 3-20, in which the inner `FOR` loops specify values for generics as well as the design entities.

In Fig. 3-20 it is assumed that a design library named `timed` exists and that this library contains the parameterized design entities like the `t_or` entity shown in Fig. 3-9. These design entities provide parameterized timing by using generic constants. The configuration declaration in Fig. 3-20 selects the entity interface and architecture body, and it defines the values of the generics of the design entities. These values can be back annotated. In particular, this configuration declaration can be created after the structural architecture for `imply` and the entity interfaces and architecture bodies in the library `timed` have been analyzed. A tool can be used to generate the configuration declarations shown in Fig. 3-20.

```
LIBRARY timed;
CONFIGURATION max_time OF imply IS
    FOR reconfig
        FOR or1: c_or
            USE ENTITY timed.t_or(behavior)
               GENERIC MAP ( 50.0, 10 NS );
        END FOR;
        FOR inv1: c_inv
            USE ENTITY timed.t_inv(behavior)
               GENERIC MAP ( 50.0, 20 NS );
        END FOR;
    END FOR;
END max_time;
```

Figure 3-20. Using a Configuration Declaration to Specify Generic Constant Values

REFERENCES

1. J. Hines, "Where VHDL Fits Within the CAD Environment", *24th ACM/IEEE Design Automation Conference Proceedings*, Miami Beach, FL, June 1987, pp. 491-4, Association of Computing Machinery, Baltimore, MD.
2. ANSI/IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 31 March 1988.
3. ANSI/IEEE Std 1076-1993, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, © September 1993.
4. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, Department of Defense, Washington, DC, 11 May 1989.
5. MIL-STD-1750A, *Military Standard Sixteen-Bit Computer Instruction Set Architecture*, 15 December 1989.
6. IEEE Std 1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (`std_logic_1164`)*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, © May 1993.
7. P. Wilsey, D. Benz, and S. Pandey, "A Model of VHDL for the Analysis, Transformation, and Optimization of Digital System Designs", *Conference on Hardware Description Languages (CHDL '95)*, pp. 611-6, August 1995.
8. R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.
9. IEEE Std 1076.4-1995, *IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, December 1995.
10. EIA 567-A, *VHDL Hardware Component Modeling and Interface Standard*, Electronic Industries Association, Washington, DC, March 1994.
11. P. Menchini, *Top-Down Design With VHDL*, First Annual Rapid Prototyping of Application (ARPA)-Specific Signal Processors (RASSP) Conference, Arlington, VA, August 1994, ARPA Electronic Systems Technology Office, Arlington, VA.
12. IEEE Std 1076.3, *IEEE Standard for VHDL Language Synthesis Package*, (Draft Standard), The Institute of Electrical and Electronics Engineers, Inc., New York, NY, September 1995.
13. O. Levia and F. Abramson, "ASIC Sign-Off in VHDL", *VHDL Boot Camp, Proceedings of the Fall VIUF*, San Jose, CA, October 1993, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
14. IEEE Std 1029.1-1991, *Waveform and Vector Exchange Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1991.

BIBLIOGRAPHY

J. R. Armstrong and F. G. Gray, *Structured Logic Design Using VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

J. Ashenden, *The VHDL Cookbook*, University of Adelaide, Adelaide, South Australia, 1990.

J. M. Berge, A. Fonkua, S. Maginot, and J. Roulliard, *VHDL '92: New Features of the VHDL Hardware Description Language*, Kluwer Academic Publishers, Norwell, MA, 1994.

J. M. Berge, A. Fonkua, S. Maginot, and J. Roulliard, *VHDL Designer's Reference*, Kluwer Academic Publishers, Norwell, MA, 1992.

J. Bhasker, *A VHDL Primer*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1994.

S. Carlson, *Introduction to HDL-Based Design Using VHDL*, Synopsis, Inc., Mountain View, CA.

D. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, Norwell, MA, 1989.

B. Cohen, *VHDL Coding Styles and Methodologies: An In-Depth Tutorial*, Kluwer Academic Publishers, Norwell, MA, 1995.

A. Dewey, *Analysis and Design of Digital Systems With VHDL*, Addison-Wesley Publishing Company, Inc., Piscataway, NJ, 1992.

A. Dewey, "The VHSIC Hardware Description Language (VHDL) Program", *ACM IEEE 21st Design Automation Conference Proceedings 84*, Piscataway, NJ, 1984, Association of Computing Machinery, Baltimore, MD.

Enabling Design Creativity, Proceedings of the VHDL International Users' Forum Fall 1991 Meeting, Newport Beach, CA, October 1991, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

R. Harr and A. Stancluescu, Eds., *Applications of VHDL to Circuit Design*, Kluwer Academic Publishers, Norwell, MA, 1989.

S. Leung and M. A. Shanblatt, *ASIC System Design With VHDL: A Paradigm*, Kluwer Academic Publishers, Norwell, MA, 1989.

R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.

Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill Book Co., Inc., New York, NY, 1993.

D. Perry, *VHDL*, McGraw-Hill Book Co., Inc., New York, NY, 1991.

J. Schoen, *Performance and Fault Modeling With VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

Standard No. 1076-CONC-1990, *The Sense of VASG*, 1990. (This publication is the companion document to IEEE Std 1076-1987.)

Using VHDL for Electronic Product Design, Proceedings of the VHDL Users' Group Spring 1991 Meeting, Cincinnati, OH, April 1991, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

Using VHDL in System Design, Test, and Manufacturing, Proceedings of the Spring 1992 VHDL International Users' Forum, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

VHDL Boot Camp, Proceedings of the Fall 1993 VHDL International Users' Forum, San Jose, CA, October 1993, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

VHDL: Windows of Opportunity, Proceedings of the VHDL Users' Group Fall 1990 Meeting, Oakland, CA, October 1990, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

CHAPTER 4
DoD REQUIREMENTS FOR THE USE OF VHDL

In this chapter the two primary Government documents concerning the use of VHDL are discussed: (1) MIL-HDBK-454 and (2) the VHDL DID, DI-EGDS-80811. The need for VHDL descriptions of all application-specific integrated circuits (ASICs) and all qualified digital electronic integrated circuits in board-level designs is discussed. The DID-required structure and contents of VHDL descriptions provided to the Government are presented. In particular, the requirement for both structural and behavioral models of each component of a digital electronic subsystem is described. This chapter also describes the required annotations to VHDL models.

4-1 INTRODUCTION

The two primary documents that describe the requirements of very high-speed integrated circuit (VHSIC) hardware description language (VHDL) models to be delivered to the Government are MIL-HBK-454 (Ref. 1) and the VHDL Data Item Description (DID), DI-EGDS-80811(Ref. 2). MIL-HDBK-454 describes the criteria for selection and application of various types of electronic equipment. In particular, Guideline 64 of MIL-HDBK-454 describes such criteria for microelectronic devices and provides guidance to deliver VHDL models of application-specific integrated circuits (ASIC) and microelectronic circuits used in board designs. Further, these models should comply with requirements stated in the VHDL DID. The VHDL DID lays out comprehensive requirements for VHDL models and the necessary auxiliary and testing support files.

VHDL is also required by MIL-STD-1840 (Ref. 3) for the exchange of digital data relating to electrical or electronic applications. MIL-STD-1840 requires one or more of the following formats:

1. Electronic Design Interchange Format (EDIF) (Ref. 4)
2. VHDL (Ref. 5)
3. International Graphics Exchange Standard (IGES) (MIL-D-28000) (Ref. 6)
4. Institute for Interconnecting and Packaging Electronic Circuits (IPC) (Ref. 7).

Subpar. 4.4.11.2 of MIL-STD-1840 cites the VHDL DID (Ref. 2) and Electronic Industries Association standard EIA-567 (Ref. 8) as the application protocols used to organize and write the VHDL code. Though MIL-HDBK-454, the VHDL DID, and MIL-STD-1840 require the use of VHDL, they provide little or no practical guidance on the organization of VHDL models and support files.

This chapter contains approaches to structuring the VHDL models so that DID requirements and intent can be met with appropriate auxiliary and testing support files. These approaches are written to readily support the tailoring of items in the DID to fit project requirements and the structuring of VHDL models so that they can be delivered to the Government at an affordable cost.

4-2 MIL-HDBK-454 GUIDELINES FOR THE USE OF VHDL

MIL-HDBK-454 describes the common guidelines to be used in military specifications for electronic equipment. It contains 78 individual guidelines covering a variety of issues relating to electronic equipment. Guideline 64 of MIL-HDBK-454 (Ref. 1) covers microelectronic devices and recommends delivery of VHDL models for microelectronic circuits under specific situations. Microelectronic circuits include monolithic integrated circuits, hybrid integrated circuits, and multichip modules.

Subpar. 4.1.3 of Guideline 64 of MIL-HDBK-454 (Ref. 1) describes a sequence of choices to be used to acquire microelectronic circuits. Subpar. 4.5.1 of Guideline 64 recommends delivery of structural and behavioral models for ASICs and cites the VHDL DID (Ref. 2). Otherwise, a nonstandard part approval must be requested. MIL-HDBK-454 lays out the requirements for the documentation and testing of nonstandard and standard parts on the Qualified Products List and of other microcircuits.

Subpar. 4.5.3 of Guideline 64 recommends documentation of digital qualified devices used in board-level applications with behavioral VHDL descriptions. These behavioral descriptions must enable test generation and support fault detection/isolation to the circuit pins.

4-2.1 DOCUMENTATION OF ASICs DEVELOPED FOR THE GOVERNMENT WITH VHDL

One form of a nonstandard microelectronic circuit used increasingly in military electronic systems is an ASIC. An ASIC is any microcircuit customized to perform a specific function. By dedicating all resources on the device to a specific function, ASICs provide high throughput for a given level of power, weight, and size. The rapidly increasing capability of electronic computer-aided design (ECAD) tools has made it possible to design and fabricate ASICs at a reasonable cost. However, the small number of copies of ASICs makes them especially vulnerable to becoming unavailable due to a lack of production facilities. The existence of both behavioral and structural VHDL models for ASICs means that ECAD capabilities can be used either to reengineer the function of the ASIC for new fabrication technologies or to transfer the design automatically to a new manufacturer's production line.

Subpar. 4.5.1 of MIL-HDBK-454 (Ref. 1) recommends that the circuit design of digital microelectronic ASICs developed be documented with behavioral and structural VHDL descriptions. The behavioral VHDL description must model both the function and timing of the microcircuit at the ports of the model. The behavioral VHDL model must be sufficiently detailed to permit its use within a larger VHDL model for test generation and fault grading of the larger model.

Subpar. 4.5.4 of MIL-HDBK-454 (Ref. 1) recommends that the test vectors and test waveforms for digital ASICs be documented and delivered to the Government in Waveform and Vector Exchange Specification (WAVES) format.

In an information section par. 5.6 of MIL-HDBK-454 (Ref. 1) references the VHDL DID (Ref. 2) as a guideline to be used to prepare the VHDL documentation of an ASIC.

Although MIL-HDBK-454 does not provide specific guidance on structural models, the information can be inferred from the VHDL DID (Ref. 2). As a guideline, the structural model of digital microcircuits should be sufficiently detailed to support fault coverage analysis based on the equivalence classes of single, permanent, stuck-at-zero, and stuck-at-one faults on all lines (i.e., interconnects). In general, this requirement implies a structural model that is decomposed into gate-level primitive modules and atomic storage functions, such as flip-flops. However, large regular structures, such as read-only memories (ROMs) and random-access memories (RAMs), can be treated as atomic structures provided they are tested using the appropriate algorithms.

Subpar. 4.5.4 of Guideline 64 of MIL-HDBK-454 recommends that the ASIC test stimuli be written and documented in Waveform and Vector Exchange Specification (WAVES) (Ref. 9).

Chapter 7 describes the WAVES standard and how to implement a VHDL test bench using WAVES.

4-2.2 DOCUMENTATION OF QUALIFIED DIGITAL INTEGRATED CIRCUITS WITH VHDL

Subpar. 4.5.3 of Guideline 64 of MIL-HDBK-454 recommends documentation with VHDL descriptions of microelectronic circuits used in board-level designs. These descriptions must fully define the functions of the device and must include timing of the device at the input/output (I/O) ports in sufficient detail to support test generation, fault detection, and fault isolation to the device when board or subsystem simulation is performed. The behavioral VHDL model recommended by MIL-HDBK-454 should be suitable for use as a leaf module in a VHDL model of a system using the modeled device.

4-2.3 THE LIBRARY OF VHDL DESCRIPTIONS OF STANDARD DIGITAL PARTS

Under the auspices of the Defense Electronics Supply Center (DESC), the Department of Defense (DoD) has started building a library of interoperable VHDL descriptions of microelectronic circuits. This VHDL model library (VML) acts as a standardization vehicle and is available to Government contractors to enable them to design systems quickly. As a result, the DoD will receive more VHDL designs for future use. Validated models placed in the VML will be a resource to aid design engineers in system upgrades or to provide logistical support after system delivery. The VML can be accessed through the World Wide Web at http://kirk.desc.dla.mil or via anonymous ftp at kirk.desc.dla.mil.

DoD project managers who are receiving VHDL models should contact DESC to alert them to the existence of the models, they should work with DESC on the specification and validation of these models, and they should send a copy of the VHDL models to DESC. DoD project managers who are tailoring the VHDL DID and defining acceptable leaf-level modules should contact DESC to find out whether the VML has VHDL models that could be used by the program. The VHDL DID (Ref. 2) requires Government approval of leaf-level models used in higher level VHDL models. In the future the VML may provide a source for such leaf-level models.

The models in such a library must have sufficient accuracy and quality to allow their use as formal models for parts of Government-procured systems. To deal with this issue, the US Air Force has published a procedure for validating VHDL models (Ref. 10). DESC is evaluating these validation techniques in order to use them to screen models delivered to the DESC VHDL model library. The validation process traces its requirements back to MIL-HDBK-454 and to the VHDL DID.

4-2.4 TEST BENCH REQUIREMENTS FOR VHDL DESCRIPTIONS

VHDL has popularized the concept of a test bench, a collection of VHDL modules that apply stimuli to a module under test (MUT). Test benches may also compare the response of the MUT with the expected output and report any differences between observed and expected responses. WAVES provides mechanisms for generating VHDL test benches and for using a standard format for the external files. WAVES is described in more detail in Chapter 7.

4-3 OVERVIEW OF THE VHDL DATA ITEM DESCRIPTION

The VHDL Data Item Description, DI-EGDS-80811 (Ref. 2), provides a definition of the Department of Defense requirements for a delivered VHDL model. The DID can be tailored for particular contracts to meet the unique requirements of a specified program. This tailoring specifies the models to be developed and delivered, and it may further define some of the basic terms used in the DID, such as "stand-alone modules". The DID can be tailored by rewriting its sections.

Appendix B contains an example of a tailored VHDL DID, including both the text of the initial DID and the changes that were made to it.

The VHDL DID requires delivery of a hierarchy of VHDL module descriptions. This hierarchy must be consistent with the hierarchy of the physical hardware (Ref. 2, subpar. 10.2.1), as described in Chapter 2. A VHDL module is defined by the DID as a deliverable item that includes several files and VHDL design units. The DID requires one VHDL module to be defined for the entire system and one for each physical electronic unit, such as an assembly, subassembly, or integrated circuit. VHDL modules should also be defined for important subsections or groupings of complex physical units. For each VHDL module of the design, the VHDL DID requires an associated VHDL entity interface, one or more behavioral bodies, and (except for leaf modules) a structural body. Furthermore, the VHDL DID requires a VHDL test bench for each stand-alone module.

An important aspect of tailoring the VHDL DID to a specific project is specifying the hierarchy of VHDL modules that will be delivered. Each of these VHDL modules requires its own test bench and its own structural and behavioral models. Within the VHDL modules the contractor is encouraged to use VHDL hierarchies to clarify the design.

Subpar. 10.2.2.3 of the VHDL DID requires that operating conditions for the physical hardware module be characterized in the corresponding VHDL entity interface. Operating characteristics include temperature range, logic level definitions (which relate the logic values used in the simulation to voltage levels in the physical design), power and heat dissipation, and radiation hardness. The VHDL DID also requires that VHDL packages be used to encapsulate this information when it can be reused across multiple VHDL modules. This use of packages is consistent with standards such as Institute of Electrical and Electronics Engineers (IEEE) 1164 (Ref. 11), WAVES (IEEE 1029.1) (Ref. 9), EIA-567 (Ref. 8), and VITAL (Ref. 12). One area of tailoring of the DID relates to the use of these standards. MIL-HDBK-454 (Ref. 1) specifies the use of WAVES. The computer-aided acquisition and logistics support (CALS) standard, MIL-STD-1840 (Ref. 3), specifies the use of EIA-567. A tailored DID can refer to other standards, such as IEEE 1164 (Ref. 11), IEEE 1149.1 (Ref. 13), and VITAL (Ref. 12).

Development of models without the use of standards runs the risk of reducing the interoperability and reuse potential of the models. Requiring the use of standards after model development has begun is very expensive. Also model developers should use standard packages so that models will work together when they are integrated.

4-3.1 ENTITY INTERFACE REQUIREMENTS

Subpar. 10.2.2 of the VHDL DID (Ref. 2) defines the requirements for the declaration of design entities as follows: "The entity declaration shall include an interface declaration which describes the input and output ports of the system. The entity declaration shall also describe timing and electrical requirements for the behavior of the device and allowable operating conditions. The entity declaration shall also include explanatory comments.".

These comments should identify the corporate and individual authors of the entity interface, the date and time of the last revision of the design interface, and identification of the device being modeled. The entity interface declaration should include references to VHDL libraries and packages that are required by every body for the interface. Libraries and packages specific to a particular architecture should not be included.

The entity interface can also include assertions about the interface, including relationships between the input and output ports and conditions on the value and timing of the entry and exit of input and output data. Assertions should be used to describe requirements on the module, and the timing delays in the behavioral bodies should capture the actual behavior of the physical device. If a behavioral body is used to describe a design for which no corresponding physical hardware exists, the behavioral body must be clearly commented to indicate the source of the timings.

4-3.1.1 Entity Names

Subpar. 10.2.2.4 of the VHDL DID (Ref. 2) requires that names for the VHDL entities be traceable to the names of the corresponding physical electronic components whenever such a correspondence exists. Similarly, the names of architecture bodies for a design entity should reflect a distinguishing implementation characteristic of that architecture body, such as the level of abstraction, the technology used to implement the component, or the manufacturer. This traceability is important for verification that the model is complete, i.e., each physical hardware component is instantiated in the VHDL design. Appropriate naming also aids verification that the VHDL model design hierarchy is consistent with the physical design hierarchy. Appropriate names for entity interfaces also aid maintenance of the model, particularly when upgrades are made to physical components. A well-structured VHDL model of a system allows changes to be isolated to those parts of the model that correspond to the physical components being upgraded and perhaps to configurations of those components.

4-3.1.2 Input and Output Definitions

Subpar. 10.2.2.1 of the VHDL DID (Ref. 2) requires that each entity interface shall describe all input and output ports. In particular for very large-scale integrated (VLSI) circuits, there should be a port declared for each pin of the circuit. This requirement is driven by the needs of WAVES (Ref. 9).

To use WAVES Level 1 to describe the test bench for a model, each input and output port must correspond to a single pin on the physical component. To support traceability between a VHDL model of the hardware and the physical hardware module, the labels used for the ports should support traceability to the corresponding bus, connector, or pins in the physical hardware whenever such a correspondence exists. These labels may be augmented by attributes, comments, or port maps. A port map can be used to link specific pins to an element of a bit vector. This link is particularly valuable for circuit pins to busses.

Building models with close relationships between VHDL ports and physical hardware pins is essential if the WAVES test bench is to be used for both testing the VHDL models and driving automatic test equipment (ATE). This traceability between the VHDL model and the physical hardware also allows better verification of the completeness of the VHDL model, either through manual review of the VHDL source code or through partially automated verification matching VHDL port names or port attributes against the net list of the physical hardware. When VHDL-based synthesis tools are used, this verification may be unnecessary. The synthesized VHDL model then becomes the standard description of the net list.

A set of standards that defines the possible types for all input, bidirectional, and output ports of the system should be set up for the entire system model. These standards should be consistent with the WAVES logic values and value dictionary for the entire system. This approach supports the interoperability of structural and behavioral models of different hardware modules and thus allows the Government and the contractor to build and simulate mixed abstraction models of the system.

4-3.2 BEHAVIORAL DESCRIPTIONS

Subpar. 10.2.1 of the VHDL DID (Ref. 2) requires delivery of a behavioral VHDL model of every physical electronic unit of the hardware system. As required by subpar. 10.2.3 of the VHDL DID, these behavioral models are required to express the timing and functional characteristics of the corresponding physical unit. Behavioral models are intended to serve several purposes: to provide simulation facilities for testing software written to execute on the hardware, to provide executable specifications for different physical implementations of the same hardware function, and to provide the reader of a VHDL model with a readable description of the system that reflects the partitioning decisions made during the design of the hardware module.

A behavioral VHDL model should accurately represent the visible interface, particularly for programmable devices. Thus behavioral models describing existing programmable hardware should accurately represent the instruction set and visible registers of the device being modeled. Furthermore, test and maintenance functions of the physical unit available to the user shall be included in the body. These requirements allow the model to be used to verify that system test programs and fault detection, isolation, and recovery algorithms are functioning. In general, these requirements do not permit the effectiveness of fault detection and isolation algorithms to be determined because evaluating effectiveness usually requires detailed structural information. Subpar. 10.2.3.3 of the VHDL DID states that structurally dependent signal values, such as scan path signatures, shall not be specified in behavioral bodies.

One important form of DID tailoring is the task of defining the set of modules for which behavioral models are to be delivered. Because the development costs of behavioral models can be a significant part of the entire cost of developing VHDL deliverables, the contractor and the contracting agency should decide early in the project on the set of behavioral models to be delivered. The contracting agency needs to ensure that the behavioral models represent those parts of the system for which different implementations may be needed due either to competing designs or to the evolution of technology over time. For example, because small-volume ASIC production poses a risk of obsolescence, subpar. 4.5.1 of Guideline 64 in MIL-HDBK-454 (Ref. 1) recommends both structural and behavioral models of all ASICs. Similarly, if the contracting agency expects that part of a system now implemented with several circuits is likely to be implemented in the future with a single circuit, it may insist on a behavioral model of that part of the system. If the contracting agency has plans to allow competitive bidding for a subsystem as part of a later stage in development, it should require a behavioral model of the subsystem since that behavioral model can be used as an executable specification of the subsystem.

The contracting agency should also verify that the combination of behavioral and structural models provides enough options for mixed abstraction models. These models allow detailed but acceptably rapid simulation of designated portions of the system. Thus part of the process of tailoring the VHDL DID for a specific program should include definition of scenarios used to simulate the system. Each scenario should identify the purpose of the scenario and the structure of the mixed abstraction model to be simulated. For example, a gate-level model of an entire multiprocessor is not an appropriate mechanism to use to debug software. In this case, a high-level behavioral model of the entire system or a high-level structural model that uses behavioral models of all of the processing elements, busses, and memories is more appropriate.

4-3.2.1 Functional Decomposition

Subpar. 10.2.3.1 of the VHDL DID (Ref. 2) allows decomposition of a behavioral model to ease simulation and increase the clarity of the model. Structural decomposition of behavioral bodies shall be used only to show functional partitions that are not represented in the physical partitioning of the hardware. For example, a behavioral body of a central processing unit (CPU) can be structurally partitioned into an arithmetic and logic unit (ALU) and several registers. However, the physical design may not follow this logical partitioning; instead it may partition the CPU into several bit slices, each containing one or more bits of the ALU and the same number of bits of each of the registers. In this case, decomposition into an ALU and several registers is an appropriate functional decomposition of the CPU because partitioning the functions of the ALU and the registers adds to the clarity of the model. Furthermore, execution of a single ALU process may be significantly faster than performing several operations on signals and ALU bit slices.

Subpar. 10.2.3.1 of the VHDL DID (Ref. 2) discourages delivering structural models at the Boolean logic level as behavioral models. Thus the use of structural VHDL models generated as output from schematic capture systems may be inappropriate as a behavioral model, as is the output of synthesis tools. These structural models do not serve any of the purposes of a behavioral model. In particular, they do not provide fast simulation or technology independence; they are not as readable as more abstract behavioral models, which are often maintained to document system design decisions.

4-3.2.2 Timing Descriptions

Subpar. 10.2.3.2 of the VHDL DID (Ref. 2) requires that signal delays at the output ports of VHDL modules accurately model the timing behavior of the physical units corresponding to the VHDL modules. The VHDL DID also requires that best-case, worst-case, and nominal output delays be included in the model.

The VHDL DID also encourages the use of more elaborate timing models that, for example, consider environmental factors such as supply voltage, temperature, or output loading. A unified approach should be developed for the invocation of appropriate timing models during simulation. The electronic data sheet (EDS) approach to capturing this information has been included in the EIA-567 standard (Ref. 8). The VITAL initiative is also developing approaches to providing best-case, worst-case, and nominal timing models that take into account environmental factors. Approaches used to define such timing models are also discussed in pars. 5-4 and 6-5 and subpars. 6-3.3.3 and 6-6.1 of this handbook.

4-3.3 STRUCTURAL DESCRIPTIONS

Subpar. 10.2.4 of the VHDL DID (Ref. 2) requires structural VHDL models to be sufficiently detailed and accurate to permit logic-level fault modeling and test vector generation. For a hierarchy of VHDL design entities to correspond to the physical hierarchy of the modeled system as subpar. 10.2.1 of the VHDL DID requires, the structural partitioning of the model must correspond to the physical partitioning of the hardware. Additional structural partitioning may be used to aid understanding of the system or to support effective built-in test techniques. Subpar. 10.2.4.1 of the VHDL DID states, “The naming of components and signals in structural VHDL models shall be the same, or be traceable to, their electronic schematic counterparts.”.

4-3.3.1 Acceptable Primitive Elements

Subpar. 10.2.1.1 of the VHDL DID (Ref. 2) restricts the choice of primitive or leaf-level components in VHDL models to one of three alternatives:

1. “Modules selected from a Government list of leaf-level modules referenced or contained in the contract.”
2. “Modules corresponding to a collection of hardware elements which together exhibit a stimulus-response behavior, but whose interaction is best modeled at the electrical or physical level. Examples of such modules are digital logic gates, analog circuit blocks, and power supplies.”
3. “Modules whose detailed design has not yet been completed, but whose behavior is required as a delivery disclosure at specified times during the contract.”.

The first alternative allows use of a Government-approved library of reusable VHDL descriptions. It is also a mechanism to ensure consistent descriptions of the same physical hardware design, and it encourages reuse of standard models whenever possible and thus reduces the validation efforts and increases the reuse of models. This alternative is an important consideration in tailoring the DID for a specific contract. For example, if the hardware design is using a specific commercial off-the-shelf (COTS) hardware component that has been militarized, the contractor and the contracting agency can agree to use a commercially available VHDL model of that component in their VHDL model of the system should such a model exist.

The second alternative allows the use of standard descriptions at the gate level, and the use of a standard logic package such as IEEE Std 1164 (Ref. 11) is strongly recommended.

The third alternative is designed to allow top-down development of VHDL models in parallel with the design of the system. In this situation the contractor and the contracting agency should agree on what leaf-level modules are to be delivered at each milestone in the contract. These delivery milestones are usually scheduled to coincide with program reviews such as the Preliminary Design Review and the Critical Design Review.

4-3.3.2 Testability Requirements

Subpar. 10.2.4 of the VHDL DID (Ref. 2) requires that the structural models be sufficiently detailed and accurate to permit logic-level fault modeling and test vector generation. Subpar. 10.2.4 also requires any structure created to support testing and maintenance, such as scan paths (Ref. 13), to be included in such a VHDL description. Modern synthesis tools are becoming sophisticated enough to generate the built-in test circuitry when given logic-level models and some guidance (Ref. 14). Circuit designers still need to partition the logic into appropriate blocks for automatic test pattern generation. VHDL design hierarchies provide a mechanism for this partitioning if the synthesis tools are sophisticated enough to use the information.

4-3.4 TEST BENCH REQUIREMENTS

A key part of a VHDL simulation package is the test bench. A test bench provides stimuli to the hardware module being simulated, checks the responses generated by the hardware module, and reports any discrepancies between the expected responses and the actual responses.

The test bench is used for verification and assessment of the VHDL description; hence subpar. 10.2.5.1 of the VHDL DID defines requirements on the design and implementation of the test bench. It requires configuration information necessary to simulate the module under test (MUT) to be delivered with the test bench. The WAVES header file provides this kind of configuration information. A VHDL configuration declaration is also needed to link the appropriate architecture bodies with their entity interfaces and to specify values of generics. However, a VHDL configuration declaration does not specify which source code versions of the design units are stored in which design libraries. This information is included in a WAVES header file. The WAVES header file also relates a given external file, i.e., a file not containing VHDL source code, to the test it implements.

4-3.4.1 Test Bench Functions

Subpar. 10.2.5.1 of the VHDL DID requires test benches to apply stimuli to a MUT and to compare the responses generated by the MUT with expected responses. The test bench must also report any differences between observed and expected responses. The VHDL DID also requires that the test bench and VHDL configuration information needed to integrate and simulate the model of the MUT with its test bench be included in the delivered package. Subpar. 10.2.5.2 of the VHDL DID also requires VHDL test benches to be cross-referenced to the contractually required hardware test plans, specifications, and drawings. The WAVES header file can and should relate the VHDL test bench model, the VHDL model of the MUT, and the external files of test vectors to the test plan or test procedure. This cross-referencing is another important reason to tailor the VHDL DID. Each test planned for the actual hardware should have a corresponding test bench. The WAVES header file and the VHDL configuration declaration for the VHDL test bench provide the corresponding information. The same VHDL test bench and configuration may be used for several tests with different external files corresponding to different test vector sets planned for the hardware. The capability of WAVES to drive both the VHDL test bench and the ATE for the actual hardware makes management of configuration and correlation of tests easier.

Fig. 4-1 shows the organization of a typical test bench. Its components are labeled using the WAVES naming convention. The waveform generator procedure (WGP) produces the stimuli for the MUT. Fig. 4-1 also shows the use of an auxiliary file as a source of data for stimuli generation. Dif-

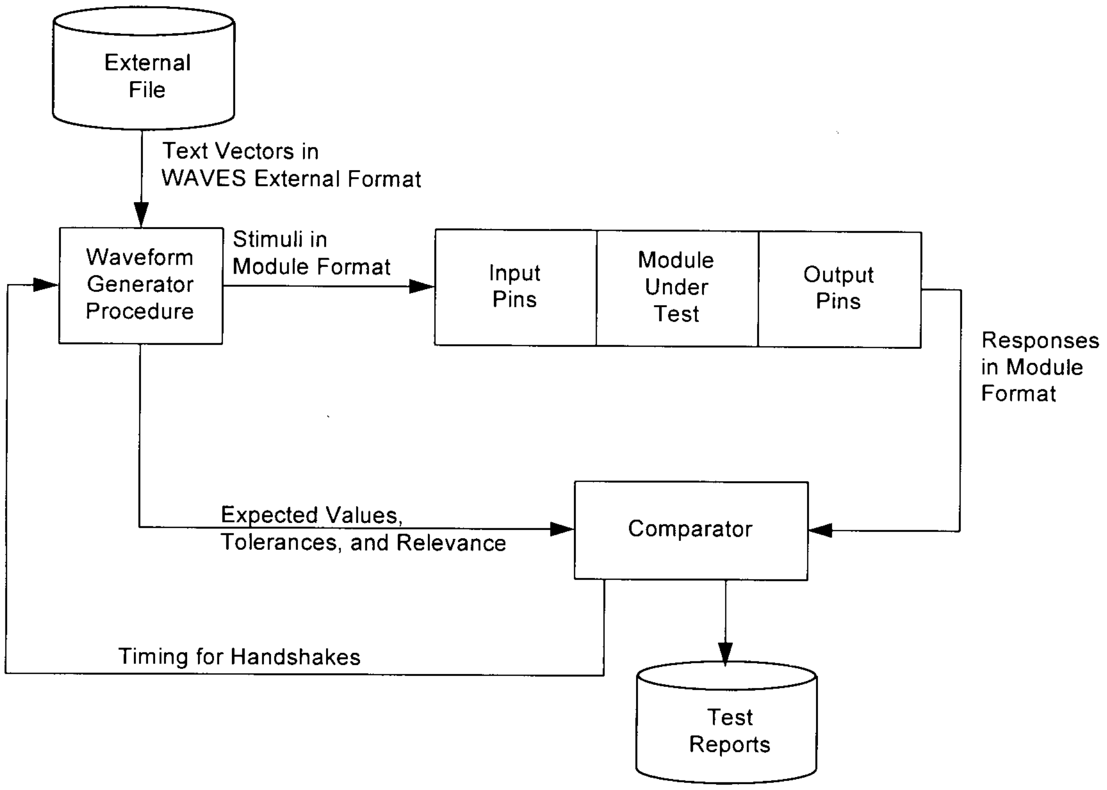

External File
Text Vectors in WAVES External Format
Waveform Generator Procedure
Stimuli in Module Format
Input Pins
Module Under Test
Output Pins
Responses in Module Format
Expected Values, Tolerances, and Relevance
Comparator
Timing for Handshakes
Test Reports

Figure 4-1. Logical Structure of a VHDL Test Bench Constructed Using WAVES

ferent tests can use different auxiliary files. Subpar. 10.2.5.1 of the VHDL DID requires that these auxiliary files be documented as part of the VHDL delivery item. The comparator shown in the figure compares the expected responses supplied by the WGP with the expected responses. In this example, the expected responses are stored in auxiliary files in a manner similar to the stimuli, but other organizations are possible. The comparator function must generate reports that indicate significant differences between the expected and the actual responses. The comparator must consider any tolerances or don't-care conditions for the output of the MUT and the timing of the expected values input.

4-3.4.2 Test Bench Relationships to Design Modules

Subpar. 10.2.5 of the VHDL DID requires that test benches be provided for "each VHDL module required by the contract to be simulatable as a stand-alone module". A tailored DID must specify which VHDL design entities represent stand-alone modules, which require their own test benches and a behavioral VHDL model. Subpar. 10.2.5.1 of the VHDL DID requires that every VHDL module of the hardware hierarchy be simulatable as a stand-alone module. Because the development of test bench software represents a significant cost, the choice of which VHDL modules must be simulatable as stand-alone modules is a critical tailoring of the DID for a specific contract. Candidates for these modules include hardware modules that are likely to be replaced as a result of preplanned program improvements or modules for which separate subcontracts are going to be let.

The hierarchy of test benches required by the VHDL DID provides a mechanism for the bottom-up validation of the model. Each low-level module can be tested individually with its own test bench. Then the low-level modules are integrated by a higher level structural design entity or hierarchy of design entities, and the resulting model is tested with its test bench. If necessary, the higher level test bench can use the corresponding behavioral model as the reference point to detect any differences between the behavioral and structural models.

Alternatively, the behavioral model can be used to generate the expected values for the output pins. These values, captured in an external file, can be combined with the inputs to form waveforms for both input and output pins of the MUT. In such an approach the behavioral model is not needed as part of the comparator function, but an external file combining stimuli and expected responses for the MUT is generated instead. An alternative format for a test bench has the behavioral model of the MUT running in parallel with a structural model of the MUT that is to be tested. The behavioral model of the MUT is combined with a compare function to serve as the comparator in Fig. 4-1.

Subpar. 10.2.5 of the VHDL DID also requires that the test bench design entities must be clearly distinguished from VHDL modules representing the MUT. The convention in WAVES is to use the suffix ".wav" for the test bench source code and to use distinct libraries for test bench entities and MUT entities. However, in some cases behavioral models of components of the system being designed are used as part of the test bench. For example, if a structural model of a signal processor using raw sensor data in external files is being tested, the designer may want to use behavioral models of the system bus interface unit (BIU) to handle the input of the sensor data into the signal processor local memory. Thus the behavioral model of the BIU is part of the MUT in one situation and is part of the test bench in another. In such cases the library organization should follow the physical organization of the entire hardware system. The configuration declaration for the test bench reveals which components are being used for the test bench and which are part of the MUT.

4-3.5 ERROR MESSAGES

Subpar. 10.2.6 of the VHDL DID requires that error messages generated either in the VHDL description of the MUT or in the test bench identify the requirement that has been violated together with the name of the VHDL design unit in which the error occurred. Subpar. 10.2.2.2 of the VHDL DID requires any violations of timing or electrical requirements, such as setup and hold times or power supply voltage extremes, to generate error messages. The VHDL assertion statement provides a means to add such conditions to a model.

4-3.6 DOCUMENTATION FORMAT

Par. 10.3 of the VHDL DID describes a set of at least eight files constituting a delivery to the Government. This description assumes that over the life of a system design several versions of the VHDL model for a system will be delivered to the Government.

The first file contains the names of all files of the deliverable VHDL documentation named in accordance with the originating host operating system (including the first file). Each record should contain exactly one file name.

The second file is a high-level prose overview of the VHDL deliverable. This file must cite contract, item number, and contract data requirements list (CDRL) sequence number and summarize the organization and content of the set of files.

The third file specifies the sequence used to analyze the VHDL design units delivered. The sequence must be consistent with the order of analysis rules in the *VHDL Language Reference Manual* (Ref. 5). A WAVES header file satisfies this requirement.

The fourth file is a list of the VHDL modules selected for use in the model and appearing in the Government-approved list of leaf-level modules. Because these files have already been approved by the Government, they do not need to be verified as part of the model acceptance procedure. Thus this list is used to reduce the workload of the Government reviewers of the models.

The fifth file is a list of VHDL modules that have been previously accepted by the Government but have been revised. Only those files that have been changed since the last delivery to the Government need to be identified in this list. For the first delivery of a model to the Government, this file should be empty.

The sixth file is a list of VHDL modules that originate with this delivery. For example, if this is the first time that a model is being delivered to the Government, all the VHDL modules are listed in this file. If since the last delivery of the model to the Government, the model hierarchy has been extended to include more detail, this file will identify the new VHDL modules. For example, if a behavioral model of a component has been augmented with a structural model that references several new behavioral models, these structural and behavioral models will be referenced in this list as will the new structural model.

The seventh file associates VHDL modules with their corresponding test benches. For each VHDL module there is a list of corresponding test benches. For each test bench there is a list of VHDL entities comprising that test bench. This list includes not only the VHDL entity for the MUT but also all of the VHDL entities that are components of the MUT and all of the VHDL entities that are part of the test bench external to the MUT.

The files after the seventh specified file contain VHDL design units and auxiliary files. The auxiliary files proceed VHDL design units. Auxiliary files include WAVES header files, WAVES external files, timing files (such as the standard delay format (SDF) files used by VITAL (Ref. 12)), and external environment parameter files (such as data files for behavioral models). VHDL MUT descriptions must be distinguished from VHDL test bench descriptions.

The delivery medium is another place where tailoring of the VHDL DID is important. Par. 7.3 of the VHDL DID defines the preferred media as nine-track magnetic tape, 1600 bits per inch, unlabeled, with 80-character records and 24 records per block. An identifying label must be attached to the tape reel, and a hardcopy of Files 1 and 2 must be included with the tape. Because of the wide variety of computer systems in existence, the Government may want to specify other magnetic media or other delivery format.

4-3.7 REQUIRED ANNOTATIONS OF VHDL MODULES

The VHDL DID requires explanatory comments to make the intent of a VHDL module clear. The following information is required:

1. Any factors restricting the general use of the VHDL module to represent the modeled hardware. For example, if nonstandard signal state/strength definitions are used, they should be noted in the explanatory comments.

2. General approaches taken to modeling, particularly decisions regarding model fidelity. Model fidelity information includes information about the timing models used and any variance in exact function from the subject hardware (such as the use of host-dependent floating point formats for calculations).

3. Any further information the originating organization considers vital to subsequent users of the descriptions. For example, if the source code for the VHDL module has been structured to support a particular synthesis tool, this fact should be noted with the version of the tool.

VHDL modules selected from the list of Government-approved modules and VHDL modules that have been previously accepted by the Government require documentation of the originator or source of the VHDL model, a DoD-approved identifier if such an identifier exists, and a design unit name or revision identifier.

A revision history must be maintained for design units that have been accepted by the Government. The design revision history is included as comments in the design unit. The revision history must include the dates of revisions, the performing individual and organization, the rationale for the revision, a description of what part of the design unit required revision, and the testing done to validate the revised model. Revision histories should also be maintained for auxiliary files using the same format as the VHDL design units where possible.

4-3.8 AN EXAMPLE OF A TAILORED DID

Appendix B, "Example of a Tailored DID", includes an example of a DD Form 1423 used to tailor the VHDL DID. This form describes a contract data requirement. Seven remarks are listed that specify the deliverables for the VHDL DID and reference the VHDL DID paragraph numbers. This modified form of the DID specifies a series of six versions of the model to be delivered. (These versions are called "levels" in DD Form 1423.)

Four behavioral model versions are required: an architectural level model, two application level models, and a bus functional model, which is a mixed abstraction level model.

Two structural model versions are required: a structural model whose leaf-level entities are integrated circuits and a structural model at the register transfer level of abstraction.

The tailoring also requires that the models of input stimuli and output results be specified in IEEE Std 1029.1 format (WAVES).

REFERENCES

1. MIL-HDBK-454M, *General Requirements for Electronic Equipment*, 28 April 1995.
2. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, Department of Defense, Washington, DC, 11 May 1989.
3. MIL-STD-1840B, *Automated Interchange of Technical Information*, 1992.
4. EIA-548, *Electronic Design Interchange Format (EDIF)*, Electronic Industries Association, Washington, DC, 1989.
5. IEEE Std 1076-1993, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, April 1994.

6. MIL-D-28000A, *Digital Representation for Communication of Product Data: IGES Application Subsets and IGES Application Protocols*, 14 December 1992.
7. IPC-D-351, *Printed Board Drawings in Digital Form*, Institute for Interconnecting and Packaging Electronic Circuits (IPC), 1989.
8. EIA-567-A, *VHDL Hardware Component Modeling and Interface Standard*, Electronic Industries Association, Washington, DC, March 1994.
9. IEEE Std 1029.1-1991, *Waveform and Vector Exchange Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1991.
10. *VHDL Model Verification and Acceptance Procedure*, Rome Laboratories/ERDD, Department of the Air Force, Griffiss Air Force Base, Rome, NY, March 1992.
11. IEEE Std 1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1993.
12. IEEE Std 1076.4-1995, *IEEE Standard for VITAL Application-Specific Integrated Circuits (ASIC)*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1996.
13. IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary-Scan Architecture*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1990.
14. J. Hallenbeck, J. Cybrynski, N. Kanopoulos, T. Markas, and N. Vasanthavada, *The Test Engineer's Assistant, A Support Environment for Hardware Design for Testability*, IEEE Computer Society Press, Los Alamitos, CA, April 1989.

BIBLIOGRAPHY

MIL-H-38534B, *General Specification for Hybrid Microcircuits*, 7 July 1993.

MIL-STD-883D, *Test Methods and Procedures for Microelectronics*, 15 November 1991.

MIL-M-38510J, *General Specification for Microcircuits*, 15 November 1991.

MIL-I-38535B, *General Specification for Integrated Circuits (Microcircuits) Manufacturing*, 1 June 1993.

MIL-HDBK-59A, *Computer-Aided Acquisition and Logistic Support*, 28 September 1990.

V. Berman, "An Analysis of the VITAL Initiative", *VHDL Boot Camp*, Proceedings of the VHDL International Users' Forum Fall Conference, 11-13 October 1993, San Jose, CA, VHDL International Users' Forum., c/o Conference Management Services, Menlo Park, CA.

O. Levia and F. Abramson, "ASIC Sign-Off in VHDL", *VHDL Boot Camp*, Proceedings of the VHDL International Users' Forum Fall Conference, 11-13 October 1993, San Jose, CA, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

CHAPTER 5
CONSTRUCTION OF BEHAVIORAL VHDL MODELS

The construction and use of behavioral VHDL models are described. Common techniques used to create behavioral VHDL models, develop timing specifications for behavioral models, and annotate behavioral models are presented. Also discussed is the usefulness of behavioral models in top-down design and mixed-abstraction-level model simulation.

5-1 INTRODUCTION

The very high-speed integrated circuit (VHSIC) hardware description language (VHDL) data item description (DID) (Ref. 1) describes two uses for behavioral models: (1) as abstract models for hardware modules that have not yet been completely designed and (2) as accurate documentation for hardware modules that already exist. For the first use the goal is to capture the functionality of a component in a manner that is as free of implementation decisions as possible. For the second use the goal is to develop an accurate model of the function and timing of the corresponding hardware module that can be clearly understood and rapidly simulated.

An implementation-free behavioral model is very useful, particularly in the early stages of a design process. Such a behavioral model can be used to validate internal and external interfaces, functional partitioning, and the synchronization of components. Behavioral models can be reviewed and simulated by independent validation and verification groups if required and can be used to develop functional tests for system integration. Behavioral models can also be used to verify the functionality of more detailed designs.

Behavioral models at different levels of abstraction are used in different ways. For example, an algorithmic model may be used as part of a system-level simulation to validate the partitioning and allocation of functions to the hardware modules, an instruction set architecture (ISA) model may be used to verify that the software will execute on the hardware, and a register-transfer-level model may be used as the starting point for the synthesis of an application-specific integrated circuit (ASIC).

In this chapter approaches to developing VHDL behavioral models at the algorithmic level, ISA level, and register-transfer level are discussed. This discussion focuses on methods used to capture functions and timing at these levels and to use VHDL models for validation and verification. Included in the discussion are common approaches to behavioral modeling in VHDL, the development of timing specifications for behavioral models, and the annotation of behavioral models.

5-2 CREATION OF VHDL BEHAVIORAL MODELS

The development of a behavioral VHDL model should be approached with its intended use in mind. The intended use of the model influences the level of abstraction of the model, which in turn affects the size, complexity, and cost of the resulting model. The intended use also influences the external support needed, such as test data generation and analysis. The intended use of the model should be examined to ensure that resources devoted to developing the model are expended as productively as possible.

The level of abstraction selected for the model should be the highest possible that meets the goals of the modeling effort. More abstract models generally are more concise, easier to read and comprehend, quicker to develop, and faster to simulate.

This paragraph discusses creating VHDL models for three levels of abstraction: algorithmic, instruction set architecture, and register transfer. It also discusses performance models, which are typically used to model real-time systems at very abstract levels for purposes of design tradeoffs.

5-2.1 CONSTRUCTING PERFORMANCE MODELS

Performance models are used at a high level to understand the timing requirements of a system. The system designer can use these models to estimate system response time and component utilization and to find potential performance bottlenecks in a design. The issues associated with constructing performance models are directly related to the issues of incorporating timing into functional models in order to create behavioral models as required by the VHDL DID. The most common reason to develop performance models or to include timing in algorithmic models is to support several types of analysis:

1. A performance- or algorithmic-level behavioral model can be used to detect inappropriate sequences of events. For example, a processor should not start sending messages across a bus before the bus interface unit (BIU) has been initialized.

2. An algorithmic model can be used to estimate the throughput of the hardware system if the throughput of a system is a measure of the number of output units generated in a given time unit. For example, an image-processing system may be required to generate 30 frames a second.

3. A performance- or algorithmic-level behavioral model can be used to compute the utilization of hardware modules. This computation can be used to determine potential bottlenecks in the design. These bottlenecks are high-risk areas of the design because if they do not meet their per-

formance requirements, the entire system will not be able to meet its performance goals. Measures of utilization include bus utilization, and estimates of bus utilization can be useful in determining the required bandwidth of the actual busses or networks of the system.

4. A performance- or algorithmic-level behavioral model can be used to estimate response time. For example, an electronic warfare (EW) system must be able to generate jamming emissions within a specific time upon detecting an enemy radar. An algorithmic VHDL model of an EW system provides information about the time interval between receiving an enemy radar pulse and generating jamming emissions.

5. A performance- or algorithmic-level behavioral model can be used to determine that the hardware system will keep up with the input data rate. For example, a radar-processing unit must be able to process radar frames at the pulse frequency rate of the transmitter; otherwise, it will lose input data.

This subparagraph describes techniques that use VHDL to model timing at the algorithmic level and to capture associated design metrics. The approaches described here are appropriate for VHDL performance models or algorithmic-level behavioral models created during top-down design.

5-2.1.2 Modeling Timing in Performance- and Algorithmic-Level Behavioral Models

In performance- and algorithmic-level behavioral models, "time" often refers to the time that a module is expected to be busy working on a specific function, not to the inertial delay of an electrical circuit or the setup and hold times required for a latch. For example, performance modeling early in a development cycle usually requires that estimates be made of the execution time of important system processes or functions. These estimates can be based on models of algorithmic complexity and the amount of data that must be processed, domain-specific knowledge of process timing characteristics, or timing budgets assigned by system engineers. A system function is modeled to use all of its input data, to process that data for some length of time, and then to produce its outputs. VHDL models can be constructed to reflect this pattern. Furthermore, the time a module is busy performing a function often can be computed from one or more of the parameters of the function. For example, the busy time for a fast Fourier transform on an ASIC with a single multiplier requires $4(N/2)\log_2(N)$ multiplier cycles. VHDL provides two mechanisms to introduce timing effects within a process: the delay clause on a signal assignment statement and the wait statement. Both of these mechanisms allow the delay to be parameterized, so the actual delay on a specific execution of the statement may vary. Signal assignment statements are discussed in subpar 3-2.3.1. Wait statements are discussed in subpar. 3-3.2. From the point of view of behavioral modeling, there are several issues to consider before choosing to use either a delay clause or a wait statement to introduce timing delays:

1. If a wait statement is used, the process will not respond to changes in its input signals during the interval the process is waiting. This unresponsiveness can cause unexpected and undetected losses of data, which are difficult to debug.

2. In a process use of signal assignment statements in a loop is risky unless there is a wait statement in the loop or the delay times vary on each execution of the loop. Otherwise, the output values on the signal will be overwritten.

3. A combination of wait statements and delays on signal assignment statements can be used in a behavioral model of a pipelined system to model the difference between latency and throughput. A system with a long pipeline (such as a pipelined floating point unit or a systolic array) may have a short time between inputs (a measure of throughput) but a very long latency. A wait statement can be used to define the minimum time between inputs, whereas the signal assignment delay can be used to capture the long latency time from the input of datum to the corresponding output.

If the delay expression in either the wait statement or the signal assignment statement is parameterized, the process is much easier to reuse. For example, the delay expression can be parameterized to adjust the timing delay on the size or value of an input or to allow the same model to be used to collect timing data about the best-, worst-, and average-case performance of the component being modeled.

There are three approaches used to collect statistics about algorithmic models:

1. A statistics package can be used to collect and reduce performance data during the simulation. This approach requires that calls to statistics-gathering functions be built into the algorithmic model.

2. Simulator controls can be used to produce trace data. This approach is likely to produce very large files of raw data that trace the necessary signal values and cause additional file input/output (I/O). Some VHDL simulation environments provide the postprocessing capabilities required to analyze trace data and produce statistics. However, if the simulation environment is used in this way, the underlying VHDL model may not be portable because not all simulation environments provide for the postprocessing of simulation trace data.

3. Statistical analysis functions can be built into the test bench for the module. The approach taken to statistics collection depends on exactly what information is needed and how much support the simulation environment provides for data collection and analysis. This approach requires that all of the information required for timing assessment is available to the test bench, such as internal signals that connect components.

5-2.1.3 Example of a Statistics Package and Its Use

As discussed in Chapter 2, the timing measures used at the algorithmic level are different than those used at the gate level. As a result, test bench or model infrastructure should include VHDL code that collects and condenses timing data into useful forms. This example describes a statistics package that collects and condenses information on-line and shows how to use data-dependent delays to model the execution time of an algorithm. It also shows how VHDL code can be developed and reused to collect statistics and provide data types for statistics that are shared between design entities. This sharing is necessary to describe the signals that provide the communication between design entities.

The example consists of two packages and a single design entity. The design entity calls a procedure to compute values using a fast Fourier transform (FFT).

The first package, of which the VHDL package interface is shown in Figs. 5-1 and the statistics package body is shown in Fig. 5-2, contains declarations and procedures used to collect statistics about a process during execution. The following statistics are included:

1. The current simulation time
2. The minimum busy time of the process over all invocations
3. The maximum busy time of the process over all invocations
4. The average busy time of the process across all invocations
5. The total busy time of the process
6. The utilization of the process.

This statistics package can be used by multiple design entities.

The second package, shown in Fig. 5-3, describes the data type for signals providing input into and output from the FFT design entity. The input to the VHDL design entity is assumed to be a record having fields that provide information on how much data are to be processed. This information

```
PACKAGE statistics IS

    -- Type for holding the statistics data as
    -- it is collected during execution

    TYPE stat_info IS RECORD
        max_time:         time;
        min_time:         time;
        average_time:     real;
        times_executed:   natural;
        total_time:       time;
        utilization:      real;
    END RECORD;

    CONSTANT STAT_INIT: stat_info := (0 ns, time'high, 0.0, 0, 0 ns, 0.0);

    -- Various string constants used for printing statistics data.

    CONSTANT CT: string := "    Current Time = ";
    CONSTANT DE: string := "    Delay = ";
    CONSTANT TE: string := "    Times executed = ";
    CONSTANT NT: string := "    Min Time = ";
    CONSTANT XT: string := "    Max Time = ";
    CONSTANT TT: string := "    Total Time = ";
    CONSTANT AT: string := "    Average Time = ";
    CONSTANT UT: string := "    Utilization = ";
    CONSTANT SP: string := "  ";

    -- The procedure for computing and printing statistics data

    PROCEDURE compute_stats( stat_data:       INOUT stat_info;
                             process_delay:   time;
                             process_id:      string);
END statistics;
```

Figure 5-1. VHDL Package Interface for Statistics for Performance and Algorithmic Models

```
USE std.textio.ALL;
PACKAGE BODY statistics IS
    PROCEDURE compute_stats(
               stat_data:     INOUT stat_info;
               process_delay:       time;
               process_id:    string) IS
        VARIABLE L: line;
    BEGIN
        IF process_delay < stat_data.min_time
            THEN stat_data.min_time := process_delay;
        END IF;
        IF process_delay > stat_data.max_time
            THEN stat_data.max_time := process_delay;
        END IF;
        stat_data.times_executed := stat_data.times_executed + 1;
        stat_data.total_time := stat_data.total_time + process_delay;
        stat_data.average_time := real(time'pos(stat_data.total_time))
                                 /real(stat_data.times_executed);
        stat_data.utilization := real(time'pos(stat_data.total_time))
                                 /real(time'pos(now));
        WRITE( L, PROCESS_ID);
        WRITE( L, CT);
        WRITE( L, now);
        WRITE( L, DE);
        WRITE( L, process_delay);
        WRITE( L, TE);
        WRITE( L, stat_data.times_executed);
        WRITE( L, NT);
        WRITE( L, stat_data.min_time);
        WRITELINE(output, L);
        WRITE( L, XT);
        WRITE( L, stat_data.max_time);
        WRITE( L, TT);
        WRITE( L, stat_data.total_time);
        WRITE( L, AT);
        WRITE( L, stat_data.average_time);
        WRITE( L, UT);
        WRITE( L, stat_data.utilization);
        WRITELINE(output, L);
    END compute_stats;
END statistics;
```

Figure 5-2. The Statistics Package Body for Performance and Algorithmic Models

```
PACKAGE sim_support IS
    -- Types that define a transaction record
    -- that is passed around the model.
    TYPE trans_vals IS ARRAY(0 TO 64) OF REAL;
    TYPE transaction IS RECORD
        values: trans_vals;
        data_size: positive;
        -- Add other fields as required
    END RECORD;

    PROCEDURE ComputeFFT(input: IN transaction; output: OUT transaction);

    FUNCTION NLogN(size: INTEGER) RETURN INTEGER;

END sim_support;
```

Figure 5-3. VHDL Data Type Definitions for a Performance and Algorithmic Model

is used to compute the execution time of the process.

Fig. 5-4 shows the entity interface for a hardware module FFT that uses both the data type and statistics packages. In addition, a generic scale factor `unit_delay` is declared. This scale factor is used to scale the performance of the algorithm when different implementations (with different clock speeds) are being evaluated.

The algorithm described in Fig. 5-5 is executed in four stages. In the first stage the data are input and the results computed. In the second stage the timing delay for the process is calculated based on the information in the input data. Next the process waits for the computed busy time. Finally, the process outputs its result data and calls the `compute_stats` procedure in the statistics package to compute and print certain statistics.

```
USE work.sim_support.ALL;
ENTITY FFT IS
        -- Generic unit_delay is the computation delay per unit of
        -- algorithmic complexity
        GENERIC (unit_delay: time := 25 ns );
        PORT (data_in: transaction; data_out: OUT transaction);

END FFT;
```

Figure 5-4. VHDL Entity Interface for a Performance and Algorithmic Model

```
USE work.sim_support.ALL;
USE work.statistics.ALL;
USE std.textio.ALL;
ARCHITECTURE algorithmic OF FFT IS
BEGIN
    implement_delay: PROCESS
        CONSTANT process_id:    string := "FFT.algorithmic.implement_delay";
        VARIABLE process_delay: time;
        VARIABLE stats:         stat_info := STAT_INIT;
                -- This variable holds the statistics for this process.
                -- Each process needs to declare its own stat_info variable
        VARIABLE results: transaction; -- Holder for the computed results
    BEGIN
        WAIT ON data_in; -- Wait for input data to begin processing

        ComputeFFT(data_in, results); -- Perform the algorithmic function

        -- Compute the process delay as a function of the size
        -- of the input and the generic unit delay.

        process_delay := 2 * unit_delay * NLogN( data_in.data_size );

        -- Wait for the process delay time. Note that process will not
        -- respond to input signals during wait interval.

        WAIT FOR process_delay;

        -- Collect process statistics

        compute_stats(stats, process_delay, process_id);
        data_out <= results; -- Assign output values

    END PROCESS;

END algorithmic;
```

Figure 5-5. VHDL Architecture Body for an Algorithmic Model

5-2.2 CONSTRUCTING ALGORITHMIC MODELS

Algorithmic models are models in which the function to be performed is described in a program-like manner independently of any particular hardware implementation. Details such as the timing and control of input/output operations or the specifics of internal data sequencing may not be specified.

In an algorithmic VHDL model the structure of the VHDL code may bear no relationship to the corresponding physical hardware. However, there should be a documented correspondence between the entity interface of the VHDL model and the corresponding physical hardware. For example, if the algorithmic model has ports with record data types, as is shown in Figs. 5-3 and 5-4, the correspondence between the pins of the physical hardware and the bit-level representation of the records may be fairly complex, and there may be other differences. For example, the algorithmic model of a microprocessor may not simulate the instruction-processing cycle; the "software" executed by the processor is part of the behavior of the microprocessor model. In this case the physical hardware pins required to fetch instructions may not be represented in the algorithmic model at all.

An algorithmic VHDL model can be used effectively in several ways:

1. To verify that the function required of the hardware module being designed has been completely and unambiguously specified
2. To help to relate system functional requirements to hardware design parameters. For example, algorithmic models can be used to evaluate word length effects on truncation errors and error propagation in mathematical processing functions such as matrix inversion or infinite impulse response (IIR) filters. An algorithmic model of a distributed

Kalman filter implemented with multiple ASICs could be extremely valuable in verifying that the multichip system architecture will provide the needed tracking accuracy.

3. To help to partition a hardware system into components, and allocate system functions to the hardware components. An algorithmic model can be used to verify that a given partitioning and allocation can collectively perform the functions required of the system. For example, an algorithmic model could be used to determine that the address generation function for the memory of a systolic array is consistent with the data addressing the requirements of the array.

4. To perform function vs throughput or latency tradeoffs for a system. For example, if an iterative refinement algorithm is used to compute the least squared error in a set of overdetermined equations, a behavioral model may be used to trade off between how many iterations are computed and how long these computations take.

5. To support testing of other models. For example, a detailed VHDL model may be built of a processor interface (PI) bus interface unit for a multiprocessing system. A VHDL model of the BIU may be tested by using algorithmic models for the processors that drive the BIU. This approach is the key concept behind the development of interface models; the interfaces and interconnections between the major components are modeled in detail, but the full functionality of the processors is not modeled. In this case, the processor models act as workload generators to produce realistic patterns of messages sent across the bus.

Algorithmic models are not usually sufficient as documentation of a developed hardware module because algorithmic models rarely provide both the complete bit-level accuracy on outputs and accuracy in timing required to document a completed hardware module.

5-2.2.1 Modeling Algorithms With VHDL Processes

The major VHDL constructs suitable for an algorithmic description include processes, functions, and procedures. These constructs form the basis for describing behavior independently of specific hardware details.

As discussed in subpar. 3-3.1, the process is the natural VHDL construct for algorithmic models. The VHDL constructs that can occur inside a process include all of the control structures of a modern programming language, e.g., C, Ada, and Pascal. These control structures include variable assignment statements, looping constructs, if-then-else constructs, and case statements. Other VHDL constructs that are legal inside a process include functions and procedures. Functions and procedures allow modelers to encapsulate behavior and reuse the same behavior in different places in the model. A VHDL function or procedure also can call other functions or procedures and result in even greater modularity. Functions and procedures can be declared and defined in packages, and the packages can then be referenced by any design unit. Library and use clauses are used in design units to access the package.

Signals are the interfaces between processes. Processes execute independently of each other. Thus multiple processes in an algorithmic model can be used to represent the parallelism in the hardware. A process communicates with other processes by writing to and reading from signals. A process is activated and starts execution when a signal to which it is sensitive changes value. A process suspends its execution by executing a wait statement. The process resumes when the condition on the wait statement has been met. Wait statements can also be used to synchronize processes. A signal can be used as a semaphore by having different processes write to the signal and by waiting on specific states of the signal.

5-2.2.2 An Example of an Algorithmic Model

In this subparagraph an algorithmic model of the FFT function described in the performance model shown in subpar. 5-2.1.3 is presented. The algorithmic model of the FFT function makes use of data abstraction to simplify the modeling of the system. The VHDL definitions of the data types and the procedures for this behavioral model are shown in Fig. 5-6. This VHDL package declaration describes the complex data type and one- and two-dimensional arrays of real and complex data. The package includes two versions of the FFT routine: one that operates on signals for use in structural models and one that uses variables and can be used in a behavioral model.

```
-- Title: PACKAGE dsp_prims
--
-- Purpose: VHDL Declarations for package dsp_primitives which contains
--          common real constants, common trigonometric functions,
--          primitive DSP procedures, and common complex arithmetic
--          functions
--
-- References: 1 PACKAGE math_real by IEEE VHDL Math Package Study Group
--          2 Signal Processing Algorithms in FORTRAN and C by
--            David Stearns
--          3 Discrete Time Signal Processing by Oppenheim and Schafer
--          4 Guidelines for VHDL Models in a Team Environment by
--            Janick Bergeron
------------------------------
PACKAGE dsp_prims_pkg IS

TYPE arr1_re_typ IS ARRAY(POSITIVE RANGE <>) OF real;
TYPE arr0_re_typ IS ARRAY(NATURAL  RANGE <>) OF real;
TYPE complx_typ IS RECORD
     re:   real;
     imag: real;
END RECORD;
TYPE arr1_complx_typ IS ARRAY(POSITIVE RANGE <>) OF complx_typ;
TYPE tod1_complx_typ IS ARRAY(POSITIVE RANGE <>, POSITIVE RANGE <>)
     OF complx_typ;

PROCEDURE fft_sig
    ( SIGNAL data:    IN  arr1_complx_typ;
      SIGNAL isi :    IN  integer;
      SIGNAL fft_out: OUT arr1_complx_typ);

PROCEDURE fft_var
    ( VARIABLE data:    INOUT  tod1_complx_typ;
      VARIABLE isign :  IN     integer;
      VARIABLE fft_out: OUT    tod1_complx_typ);
end  dsp_prims_pkg;
```

Reprinted with permission. Copyright © by Virginia Polytechnic Institute and State University.

Figure 5-6. Package Declaration for an Algorithmic Model of an FFT Processor (Ref. 2)

This VHDL package specifies the data types for the FFT procedures in terms of the built-in type `real`. The complex data type is defined as a VHDL record that has real and imaginary components. When the design is refined and the model is converted from the algorithmic level to ISA or register-transfer level, several issues must be addressed including (1) the number of bits in each word and (2) the organization of elements in a record, particularly the alignment of these elements. These issues can be resolved by changing the package declaration and package body without changing the code in the architecture bodies. Thus, by using packages to implement abstractions, the developer allows word size and alignment decisions to be abstracted out of the behavioral model.

Fig. 5-7 contains part of the package body for the package declaration of Fig. 5-6. The addition and multiplication operators are overloaded to define addition and multiplication operations for the complex data type. The package body also declares two constants, `math_pi` and `half_pi`, used to calculate the weighting factors. Because these two constants are declared in the package body rather than in the package declaration, they are not visible outside the package body.

```
PACKAGE BODY dsp_prims_pkg IS
--
-- Commonly used constants
--
CONSTANT math_pi: real := 3.14159_26535_89793_23846;
CONSTANT half_pi: real := math_pi / 2.0;

FUNCTION "*" (x: complx_typ;
              y: complx_typ )
         RETURN complx_typ IS
BEGIN
    RETURN ( x.re * y.re - x.imag * y.imag,
             x.imag * y.re + x.re * y.imag);
END "*";

FUNCTION "+" (x: complx_typ;
              y: complx_typ )
         RETURN complx_typ IS
BEGIN
    RETURN ( x.re + y.re, x.imag + y.imag);
END "+";

FUNCTION "-" (x: complx_typ;
              y: complx_typ )
         RETURN complx_typ IS
BEGIN
    RETURN ( x.re - y.re, x.imag - y.imag);
END "-";
```

Figure 5-7. Part of the Package Body for an Algorithmic Model of an FFT Processor (Ref. 2)

Fig. 5-8 contains the procedure body for the `fft_sig` procedure declared in Fig. 5-6. This code is part of the package body described in Fig. 5-7. The code uses a series of loops to rearrange the data, compute the weighting factors, and then compute the intermediate values. The code also allows a final pass that uses the second parameter of the procedure to normalize the output.

```
PROCEDURE fft_sig
    ( SIGNAL data:     IN  arr1_complx_typ;
      SIGNAL isi :     IN  integer;
      SIGNAL fft_out:  OUT arr1_complx_typ) IS

    VARIABLE w :     complx_typ;
    VARIABLE temp :  complx_typ;
    VARIABLE data1:  arr1_complx_typ(1 to data'HIGH);
    VARIABLE mmax :  integer := 1;
    VARIABLE istep : integer;
    VARIABLE x :     integer;
    VARIABLE j :     integer := 1;
    VARIABLE m :     integer;
    VARIABLE theta : real;
BEGIN
   data1 := data;
   FFT1: FOR i IN data1'RANGE LOOP
      IF (i < j) THEN
         temp := data1(j);
         data1(j) := data1(i);
         data1(i) := temp;
      END IF;
      m := data1'HIGH/2;
      FFT2: WHILE (j > m) LOOP
         j := j-m;
         m := ((m+1)/2);
      END LOOP FFT2;
      j := j + m;
   END LOOP FFT1;
   FFT3: WHILE (mmax < data1'HIGH) LOOP
      istep := 2 * mmax;
      FFT4: FOR m IN 1 TO mmax LOOP
         theta := math_pi * REAL(isi*(m-1))/REAL(mmax);
         w := (COS(theta),SIN(theta));
         x := m;
         FFT5: WHILE (X <= data'HIGH) LOOP
            j := x + mmax;
            temp := w * data1(j);
            data1(j) := data1(x) - temp;
            data1(x) := data1(x) + temp;
            x := x + istep;
         END LOOP FFT5;
      END LOOP FFT4;
      mmax := istep;
   END LOOP FFT3;
   IF (isi >= 0) THEN
      FFT6: FOR i IN data1'RANGE LOOP
         data1(i) := data1(i)/data1'HIGH
      END LOOP FFT6;
   END IF;
   fft_out <= data1;
END fft_sig;
```

Figure 5-8. The FFT Procedure in the Package Body for an Algorithmic Model of an FFT Processor (Ref. 2)

5-2.3 CONSTRUCTING INSTRUCTION-SET-ARCHITECTURE-LEVEL MODELS

ISA models accurately describe the functions, data types, and registers of a processor accessible to the programmer. An example is an ISA model of a microprocessor. As the microprocessor executes its instructions, the contents of its memory and registers change. The correct ISA modeling of such a programmable device requires that, given the same memory contents for both data and instruction as an actual device, the execution of the ISA model must accurately reproduce the same changes in memory contents as the actual device. ISA models usually represent data and instructions with bit-for-bit fidelity with respect to the data and instructions of the actual device. ISA models must also provide a higher level of timing fidelity than algorithmic or performance models. Usually, ISA models must provide accurate times to perform an instruction and update the associated registers and memories. ISA models may have to provide accurate timing at clock boundaries. This requirement is particularly true for pipelined programmable processors in which the execution of multiple instructions may be overlapped.

An ISA VHDL model can be used in several ways:

1. To document the functions and timing of an existing hardware module
2. To support verification of software through simulation of the hardware
3. To provide timing estimates for specific software workloads. Although an algorithmic-level model may be used to estimate software performance by counting operations, an ISA model provides specific information by interpreting the actual software instructions and adding up their execution times.
4. To support verification of a hardware implementation of a standard architecture. For example, an ISA model of the 1750A standard military computer architecture (Ref. 3) can be used in combination with a validation suite to test that a hardware design accurately implements the standard architecture (Ref. 4).

In many military electronic system development projects, the hardware is developed concurrently with the software. This concurrency of development means the hardware may not be available when software is ready for testing. An ISA model allows software developers to test portions of their code via simulation before the hardware is ready. Thus software errors can be found and corrected earlier in the design cycle, to save time and money.

The most common configuration for a programmable hardware system includes one or more processors, one or more memories (storing the program and its data), and one or more busses to provide the communication paths among the elements. A complete ISA model has all of these modules explicitly modeled at the ISA level, but simulations are often performed in which some of the modules are represented by algorithmic models. In the following subparagraphs issues relating to modeling these types of hardware modules are discussed.

5-2.3.1 Modeling Processors

An ISA model of a processor faithfully interprets the instruction stream input to the model. The ISA model of the processor must explicitly represent all of the internal registers of the processor accessible to any instruction. A register is accessible if an instruction can directly set or read the value of the register. For example, the instruction address register for a processor may be set by a program that executes a jump instruction. In contrast, a microcode address register is not set explicitly by any instruction and need not be included in an ISA model.

An ISA model must mimic the transformations of the processor to the accessible registers and the external interfaces at the bit level. The external interface of the processor, e.g., bus interface, I/O channel interface, etc., must be explicitly modeled at the bit level. The timing of an ISA model should be accurate at instruction boundaries or clock edges. In pipelined architectures, in which there is a significant amount of overlap between instruction execution, accurate representation of memory and registers may be required at the clock boundaries. This level of timing accuracy is not required in an algorithmic model.

The complexity of an ISA model is related to the number of components used in the VHDL model. More complex models take longer to build and verify. If a processing element including processor, memory, and bus interface is modeled as a single unit, the interface between the processor, memory, and bus interface unit can be modeled behaviorally. However, if a model separates the processor, memory, and bus interface into separate modules, their interfaces must be modeled accurately and explicitly. If the processing element uses virtual memory, the memory management process has to be modeled accurately as a separate function in the more fine grained model.

The code in Figs. 5-9, 5-10, 5-11, 5-12, 5-13, 5-14, and 5-15 constitutes an ISA model of a very simple processor. The processor has nine instructions and three registers accessible to the programmer: the program counter, the accumulator, and an index register. These registers are explicitly represented in the model as VHDL variables whose types specify the structure of the registers.

Fig. 5-9 contains the package declaration for a set of data types, constants, functions, and a procedure used to implement this ISA model. The first data type `word` specifies the word used by the processor. The functions `ToInt` and `ToWord` define the representation of integers in this processor as a sign-magnitude format. Integers are the only data type that this processor uses.

The “opcode” constants define the formats for the instruction set of the processor. There is also an enumerated data type for the instruction set `op_code`. One of the functions included in the package is `DeCode`, which decodes instructions read from memory into the `op_code` enumerated type.

```
PACKAGE isa_pkg IS
            -- The register datatype which defines the word size
            -- for this processor
    TYPE word  IS ARRAY(15 downto 0) OF bit;
           -- The memory array type
    TYPE mem IS ARRAY(natural RANGE <>) OF word;
            -- The external file for a program to be loaded into memory
    TYPE mem_file IS file OF word;
            -- The instruction set literals
    TYPE op_code IS (halt,load,store,add,subt,dec,inc,ldx,bne):
    CONSTANT halt_code:   word := X"0000";
    CONSTANT load_code:   word := X"0001";
    CONSTANT store_code:  word := X"0002";
    CONSTANT add_code:    word := X"0003";
    CONSTANT subt_code:   word := X"0004";
    CONSTANT dec_code:    word := X"0005";
    CONSTANT inc_code:    word := X"0006";
    CONSTANT ldx_code:    word := X"0007";
    CONSTANT bne_code:    word := X"0008";
            -- Literal constants
    CONSTANT ZERO:        word := X"0000";
    CONSTANT ONE:         word := X"0001";
    CONSTANT TWO:         word := X"0002";
            -- Utility routines for converting back and forth from
            -- word and integers
    FUNCTION ToInt( val: word ) RETURN integer;
    FUNCTION ToWord( val: integer ) RETURN word;
    FUNCTION DeCode( val: word ) RETURN op_code;
            -- Overloaded operators
    FUNCTION "+" (left: word; right: word) RETURN word;
    FUNCTION "-" (left: word; right: word) RETURN word;
    FUNCTION "*" (left: word; right: word) RETURN word;
            -- The procedure for loading a program into memory
    PROCEDURE load_program( VARIABLE memory: OUT mem );
END isa_pkg;
```

Figure 5-9. Package Declaration for an Instruction Set Architecture Processor Model

Figs. 5-10, 5-11, and 5-12 contain the contents of the package body associated with the package declaration shown in Fig. 5-9. Fig. 5-10 contains the type conversion functions `ToInt`, `ToWord`, and `DeCode`. To make these routines robust and thus easily able to deal with changes in the word size of the processor, extensive use has been made of built-in attributes associated with VHDL arrays. The loop ranges are all described in terms of the `RANGE` attribute. The `NEXT` statements are used to detect when the sign bit of the word has been detected and thus causes the body of the loop to be skipped. Note also that the `CASE` statement used in the `DeCode` function is not valid in VHDL 1987 (Ref. 5), because the constants are not locally static. An `IF` statement can be used instead.

```
-- Converts words (in sign-magnitude format) to integers
   FUNCTION ToInt( val: word ) RETURN integer IS
       VARIABLE result: integer := 0;
   BEGIN
       FOR i IN word'RANGE LOOP
           NEXT WHEN i = word'LEFT;
           IF val(i) = '1'
              THEN result := 2 * result + 1;
              ELSE result := 2 * result;
           END IF;
       END LOOP;
       IF val(word'LEFT) = '1'
          THEN result := - result;
       END IF;
       RETURN result;
   END;

   -- Converts integers to words in sign-magnitude format
   FUNCTION ToWord( val: integer ) RETURN word IS
       VARIABLE result: word := ZERO;
       VARIABLE temp: integer := val;
   BEGIN
       FOR i IN word'REVERSE_RANGE LOOP
           NEXT WHEN i = word'LEFT;
           IF temp mod 2 = 1
              THEN result(i) := '1';
           END IF;
           temp := temp/2;
           EXIT WHEN temp = 0;
       END LOOP;
       IF val < 0
          THEN result(word'LEFT) := '1';
       END IF;
       RETURN result;
   END;

   -- Converts words representing instructions to op_codes
   FUNCTION DeCode( val: word ) RETURN op_code IS
       VARIABLE result: op_code;
   BEGIN
       CASE val IS
          WHEN load_code  => result := load;
          WHEN store_code => result := store;
          WHEN add_code   => result := add;
          WHEN subt_code  => result := subt;
          WHEN dec_code   => result := dec;
          WHEN inc_code   => result := inc;
          WHEN ldx_code   => result := ldx;
          WHEN bne_code   => result := bne;
          WHEN OTHERS     => result := halt;
       END CASE;
       RETURN result;
   END;
```

Figure 5-10. Type Conversion Functions for an Instruction Set Architecture Processor Model

Fig. 5-11 contains the operator overloading functions for the ISA model. These functions overload the traditional addition and subtraction operators so that they are defined for the `word` format used in this processor. This implementation converts the inputs to these functions into `integer` type, performs the operation, and then converts the result back to `word` format. These functions could be made more robust by adding error handling for overflow and underflow conditions. An alternative implementation approach uses the operator overloading and the conversion functions in the proposed synthesis standard (Ref. 6).

Fig. 5-12 contains the procedure for loading the program into memory. This procedure uses the VHDL type of `FILE` and some of the built-in functions used to detect the end of file condition. In this model the memory is declared as a fixed size array. The size of the memory is specified by a generic whose value is set in the entity interface declaration shown in Fig. 5-13. The memory is initialized from an external file that contains the program to be executed.

Fig. 5-13 contains the entity declaration for the ISA model. There is no port clause in the entity interface for the ISA model because this is the highest level entity in the simulation. The generic associated with the entity specifies the memory size. The memory is not modeled as a separate de-

```
-- This function overloads the "+" (addition) operator.
-- It converts its arguments into the internal representation of
-- the host machine, adds them and then reconverts the result
-- back to sign-magnitude format. It does not check for possible
-- overflow of the result
FUNCTION "+" (left: word; right: word) RETURN word IS
BEGIN
    RETURN ToWord( ToInt(left) + ToInt(right) );
END;

-- This function overloads the "-" (subtraction) operator.
-- It converts its arguments into the internal representation of
-- the host machine, subtracts them and then reconverts the result
-- back to sign-magnitude format. It does not check for possible
-- overflow or underflow of the result
FUNCTION "-" (left: word; right: word) RETURN word IS
BEGIN
    RETURN ToWord( ToInt(left) - ToInt(right) );
END;

-- This function overloads the "*" (multiplication) operator.
-- It converts its arguments into the internal representation of
-- the host machine, multiplies them and then reconverts the result
-- back to sign-magnitude format. It does not check for possible
-- overflows or underflows
FUNCTION "*" (left: word; right: word) RETURN word IS
BEGIN
    RETURN ToWord( ToInt(left) * ToInt(right) );
END;
```

Figure 5-11. Operator Overloading Functions for an Instruction Set Architecture Processor Model

```
 -- This procedure loads a program into memory
PROCEDURE load_program( variable memory: out mem ) IS
    FILE pfile: mem_file IS IN "program";
 BEGIN
    FOR addr IN memory'RANGE LOOP
       EXIT WHEN endfile(pfile);
        READ(pfile, memory(addr));
    END LOOP;
END load_program;
```

Figure 5-12. Program Loading Procedure for an Instruction Set Architecture Processor Model

```
ENTITY isa_model IS

    GENERIC( MemSize: natural := 512);

END;
```

Figure 5-13. Entity Interface for an Instruction Set Architecture Processor Model

sign entity. If it were, it would need to be instantiated and connected to the processor by signals. This approach, in which the memory is not a separate component, simplifies the model and improves its simulation performance.

Fig. 5-14 contains the architecture body for the ISA model. At the start of simulation, the stored program is loaded into the simulated memory. Once the program has been loaded into memory, execution begins. Each instruction cy-

```
USE std.textio.ALL;
USE work.isa_pkg.ALL;
ARCHITECTURE isa OF isa_model IS
BEGIN
PROCESS
        -- Internal registers:
        -- pc = program counter,
    VARIABLE pc          : word := ZERO;
        -- acc = accumulator
    VARIABLE acc         : word := ZERO;
        -- index = index register.
    VARIABLE index       : word := ZERO;
        -- Internal variables
    VARIABLE data_addr   : word := ZERO;
    VARIABLE data_word   : word;
    VARIABLE stop        : boolean := FALSE;
    VARIABLE instr       : op_code;
        -- The memory array, constrained by the generic MemSize
    VARIABLE memory      : mem( 0 to MemSize -1 );
        -- String constants and variables for debugging output
    VARIABLE l           : line;
    CONSTANT PCS         : string := "  PC = ";
    CONSTANT INS         : string := "  Inst. = ";
    CONSTANT AC          : string := "  ACC = ";
    CONSTANT DX          : string := "  Index = ";
    CONSTANT errmsg      : string := "Invalid Instruction. PC = ";
        -- Writes out internal state of simulation
    PROCEDURE write_state IS
        BEGIN
            write(l, PCS);
            write(l, ToInt(pc));
            write(l, INS);
            write(l, ToInt(instr));
            write(l, AC);
            write(l, ToInt(acc));
            write(l, DX);
            write(l, ToInt(index));
            writeline(output, l);
        END write_state;

        -- Load the externally stored program into memory
    load_program( memory );
        -- Loop, fetching, decoding, and executing instructions until a halt
        -- instruction is received or an invalid instruction is processed.
```

(cont'd on next page)

Figure 5-14. Architecture Body for an ISA-Level Processor Model

```
    WHILE stop = FALSE LOOP
       data_word := memory( ToInt( pc ) );
       instr := DeCode(data_word);
       CASE instr IS
        -- Load accumulator from address specified in load instruction
       WHEN load=>   acc := memory(ToInt(memory(ToInt(pc + ONE))));
                     write_state;
                     pc := pc + TWO;
       -- Store accumulator into address specified in store instruction
       WHEN store=>  memory(ToInt(memory(ToInt(pc + ONE)))) := acc;
                     write_state;
                     pc := pc + TWO;
       -- Add contents of memory location to accumulator from
       -- location specified in add instruction, offset by value of index
       WHEN add=>    data_addr := memory(ToInt(pc + ONE)) + index;
                     acc := acc + memory(ToInt(data_addr));
                     write_state;
                     pc := pc + TWO;
       -- Subtract contents of memory location from accumulator from
       -- location specified in add instruction, offset by value of index
       WHEN subt=>   data_addr := memory(ToInt(pc + ONE)) + index;
                     acc := acc - memory(ToInt(data_addr));
                     write_state;
                     pc := pc + TWO;
       -- Decrement index register
       WHEN dec=>    index := index - ONE;
                     write_state;
                     pc := pc + ONE;
       -- Increment index register
       WHEN inc=>    index := index + ONE;
                     write_state;
                     pc := pc + ONE;
       -- Load index register from location specified in load instruction
       WHEN ldx=>    index := memory( ToInt(memory(ToInt(pc + ONE))));
                     write_state;
                     pc := pc + TWO;
       -- Branch to specified address if index not equal to zero
       WHEN bne =>   IF index /= ZERO
                        THEN pc := memory(ToInt(pc + ONE));
                        ELSE pc := pc + TWO;
                     END IF;
                     write_state;
       -- Halt simulation when halt instruction is received
       WHEN halt =>  stop := TRUE;
                     write_state;
       -- Print error msg and halt if a bad instruction is processed
       WHEN others =>write(l, errmsg);
                     write(l, ToInt(pc));
                     writeline(output, l);
                     stop := TRUE;
       END CASE;
    END LOOP;
    wait;
END PROCESS;
END;
```

Figure 5-14. (cont'd)

cle and instruction is read from memory, decoded, and executed. When a "halt" instruction is executed, the model stops simulating.

5-2.3.2 Modeling Memory

Memory, in the sense of a large, contiguous array of storage registers, can be modeled in several ways. In an algorithmic model the simplest way is to use an array of the appropriate size and type. Instructions and data are stored in the array and accessed using the appropriate index into the array. A read is modeled by indexing into the array and using the contents as required by the model; a write is modeled by assigning a new value to the location in the array indicated by the index.

Because modern computers have very large memories, this simple model can occupy very large amounts of the memory in the computer used to simulate the model without making efficient use of this memory. An alternative approach is to build a virtual memory model in which only the pages of memory that have been read or written are actually maintained by the model. VHDL access types provide a convenient mechanism to implement this type of virtual memory. The memory process maintains a list of pointers to pages of memory, maps the addresses received as part of read or write commands into references to specific pages, and then operates on the specific page. If an address is received that is not in the address space of any of the current pages, a new page is created and added to the list.

An instruction set architecture memory model includes explicit memory control signals, such as read and write lines and address and data lines. It also has timing delays and logic conventions appropriate to the model. An example of this type of memory model is shown in Fig. 5-15. This VHDL model makes use of an array to represent the memory but provides a faithful representation of the external memory in-

```
-- Memory entity has a generic size which defaults to 256 bits.
-- It also has address, data,  and read/write control signals.
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.isa_pkg.ALL;
ENTITY memory IS
   GENERIC( memsize:       integer := 256);
   PORT(    data:    INOUT word;
            address: IN    word;
            R_W:     IN    std_logic);
END memory;

ARCHITECTURE behavior OF memory IS
BEGIN
memory : PROCESS ( R_W, address, data )
        -- The memory array, constrained by the generic MemSize
    VARIABLE memarray   : mem( 0 to MemSize -1 );
    VARIABLE data_addr  : natural := 0;
    BEGIN
        data_addr := ToInt(address);
        IF data_addr <= memarray'HIGH
           THEN ASSERT FALSE
                   REPORT "Bad memory address"
                   SEVERITY Error;
        ELSIF R_W = '1'
           THEN data <= memarray(data_addr) after 10 ns;
        ELSIF R_W = '0'
           THEN memarray(data_addr) := data;
        ELSIF R_W = 'Z'
           THEN NULL;
        ELSE ASSERT FALSE
                REPORT "Bad memory read_write value"
                SEVERITY Error;
        END IF;
    END PROCESS;
END behavior;
```

Figure 5-15. Example Instruction Set Architecture Memory Model

terface. The entity interface declaration in Fig. 5-15 makes use of the IEEE standard logic package (Ref. 7) to model the read/write (`R_W`) pin and uses the definition of the word size and the memory array from the `isa_pkg` package.

In this memory model there is one control signal `R_W`, and it indicates whether the memory operation is a read (when `'1'`) or write (when `'0'`). The memory array is implemented as a variable by the declaration of memory. An additional feature of this model is its use of the generic parameter "memsize", which has a default value of 256. The use of a generic allows users to model a memory of any size without rewriting the memory model.

The "address" input signal is declared as type `word`, consistent with the ISA processor model. The memory model is made robust by use of an assertion statement to ensure that the integer used to index the memory array is within the range of the memory array. If an address value is received that is out of the specified range, the assertion violation is raised.

5-2.3.3 Modeling Busses and Bus Controllers

A VHDL bus has a meaning different than the hardware meaning of a bus as a collection of electrical signals, a protocol to acquire and release the bus resource, and a protocol to transfer data across the bus. A VHDL signal that has the signal kind of "bus" floats to a user-determined value when it is disconnected from all of its drivers, and thus it does not preserve the value last driven on it as a VHDL register signal does. This subparagraph describes how to model a bus used to transfer data among hardware components.

A traditional bus is often implemented with a component referred to as a bus interface unit, or bus controller. Hardware units that use the bus do so with the BIU. The BIU is responsible for implementing the details of the bus protocols and for transferring data and other information between the units on the bus.

The protocol for a traditional bus is often specified with a state transition diagram. Transitions in the state transition diagram represent actions on the bus such as data transfer, control signaling, and error conditions.

Fig. 5-16 shows the state transition diagram for the test access port (TAP) controller for the Institute of Electrical and Electronics Engineers (IEEE) 1149.1 boundary scan test bus (Ref. 8). The ellipses in the figure represent the states of the finite state machine. The lines represent transitions from one state to another. The value adjacent to each state transition line is the value present on the test mode select (TMS) input signal. Changes in these values cause the state machine to move to another state. A VHDL description of this BIU has been developed (Ref. 9).

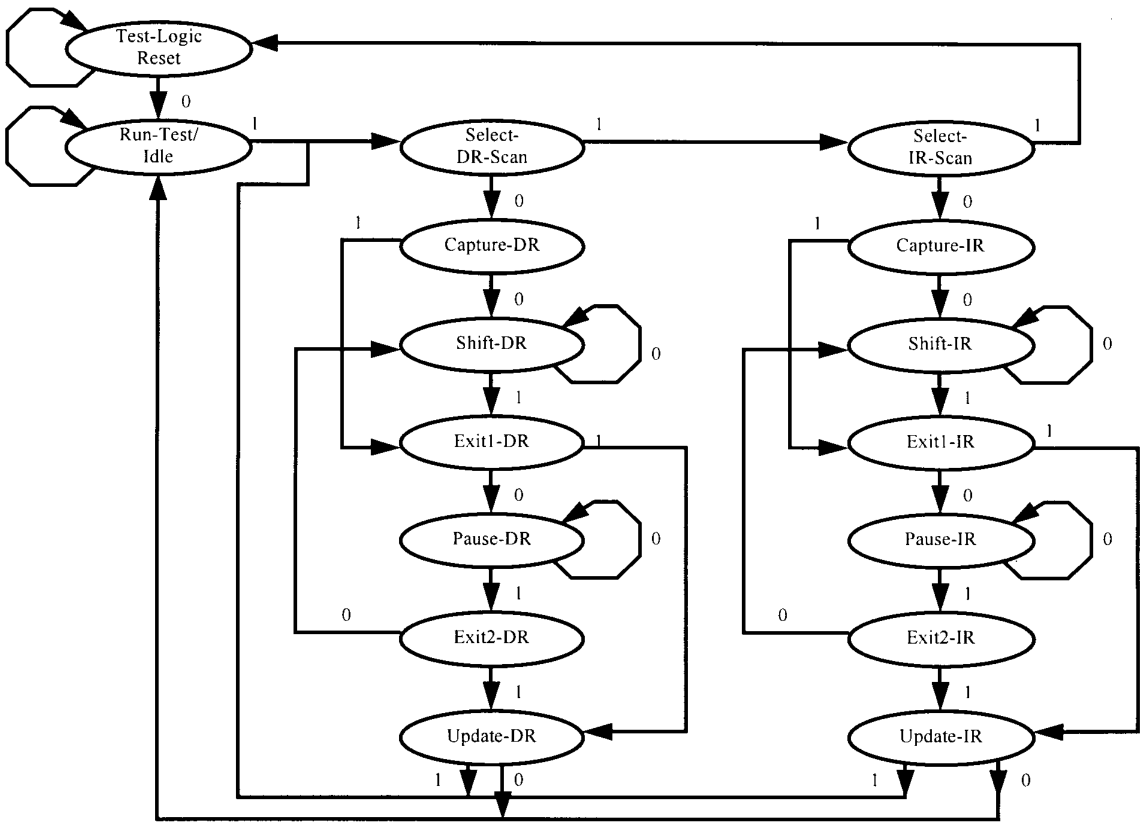


Figure 5-16. Example State Transition Diagram for a Bus Interface Unit Model (Ref. 8)

5-2.4 CONSTRUCTING REGISTER-TRANSFER-LEVEL MODELS

Register-transfer-level (RTL) models explicitly represent the internal registers, data path elements, and control mechanisms in a hardware component. Design decisions about the number and kinds of registers, the structure of internal data paths, and the data transformation functions are reflected in an RTL model.

Just as the major elements of an ISA model are processors, memories, and busses, the major elements of an RTL model are registers, combinational logic, internal busses, and clocks. Individual elements of an RTL model, such as an arithmetic and logic unit (ALU), may be modeled behaviorally. The scope of an RTL model is generally a chip.

A register transfer level model can be used effectively in several ways:

1. To support synthesis of very large-scale integrated (VLSI) circuit designs. This ability is particularly important because synthesis tools can adapt a design to different fabrication technologies. If synthesizable models are available for parts when they become obsolete, then new circuit designs can be created for current fabrication technologies.
2. To check that the logical decomposition of the hardware design into register-transfer-level elements is functionally consistent with a higher level behavioral design, such as an ISA model. For example, an RTL model can be used to decide whether a microcoded processor architecture will work or whether custom combinational logic is required.
3. To verify that the clocking scheme, sequencing, and control of a synchronous system work correctly together. Determining whether a single-phase clock is good enough (or that a two-phase clock is required) is one such question.
4. To analyze the performance of a particular register-transfer-level partitioning. For example, does a replicated bit-slice circuit have enough performance with its interchip carry lines, or is a family of custom chips required to reduce interchip communication?
5. To test microcode for microprogrammable processors. The existence of a VHDL simulation model allows microcode developers to debug and optimize their code before the actual hardware is available. If a VHDL model at the register-transfer level is used as the functional specification of a processor and is used to develop microcode, the microcode and the VHDL model can be used to help verify the correct functionality of the new hardware when it becomes available.

In an RTL model written in VHDL, signals are used to represent specific hardware registers. The types associated with the signals reflect the format of the data stored in the register. The flow of data through the system is represented by bit-level control signals acting on multiplexors. Functions are synchronized with clocks, and signals are used explicitly to distribute clocks to the components.

5-2.4.1 Synthesis of Designs From RTL Models

Register-transfer-level models are a particularly important class of models because commercially available hardware synthesis technology can be used to generate detailed integrated circuit designs from appropriate register-transfer-level models. Use of synthesis tools is an important method used to design application-specific integrated circuits.

Standards are being developed to support the use of VHDL for synthesis. Currently, each synthesis tool has its own packages and data types. The IEEE has a working group and a draft standard (Ref. 8), which is to be implemented through a set of guidelines for models and through a collection of VHDL packages.

The proposed standard includes two packages: `NUMERIC_BIT` and `NUMERIC_STD`. These packages have the same list of functions, but these functions differ in their target data types. The `NUMERIC_BIT` package defines functions for the `NUMERIC_BIT` data type; the `NUMERIC_STD` package defines functions for the IEEE `std_ulogic` data type. Each package defines two additional data types: one for vectors representing signed arithmetic values and one for vectors representing unsigned arithmetic values. Each package defines a comprehensive set of arithmetic, logical, relational, and shift functions that operate on these data types. The packages also include type conversion functions between the signed and unsigned data types and vectors of `BIT`, `BOOLEAN`, and IEEE Std 1164 `std_ulogic` data types.

The proposed standard defines functions used to detect rising and falling edges of signals that have a data type named `NUMERIC_BIT`. The `NUMERIC_BIT` data type describes vectors whose elements are of type `BIT`, a VHDL built-in type. Simulations based on the `NUMERIC_BIT` data type ordinarily require less execution time because they do not have to deal with operands containing metalogical values. The functions that detect rising and falling edges of `NUMERIC_BIT` signals are meant to be complementary to the functions that detect rising and falling edges in the IEEE standard logic package (Ref. 9).

The proposed standard provides an interpretation of the `BIT` and `BOOLEAN` data types of VHDL (Ref. 10). This interpretation defines how synthesis tools should handle the logic values of literals after named constants have been replaced by their values. The proposed standard describes how the metalogical values, i.e., the values `'U'`, `'X'`, `'Z'`, and `'-'`, of the IEEE `std_ulogic` data type should be handled by relational operators such as `'<'`, `'>'`, `'='`, `'/='`.

Additionally, the proposed standard defines a standard matching function that provides don't care or wild card testing of values based on the IEEE `std_ulogic` data type. This matching function returns `'FALSE'` whenever either of the arguments contains a metalogical value other than `'-'`, the don't care value. The function returns `'TRUE'` when `'H'` is compared with `'1'` or when `'L'` is compared with `'0'`.

5-2.4.2 An Example of a VHDL Register-Transfer-Level Model

Figs. 5-17 and 5-18 show a behavioral model of an Intel 8212 buffered latch described at a register-transfer level that has been used as a test input for a synthesis system (Ref. 11). The `I8212` has control inputs $\overline{DS1}$ (named `NDS1`), `DS2`, `MD`, and `STRB`. These inputs are used to control device selection, data latching, output buffer state, and service flip-flop. When `NDS1` is low and `DS2` is high, the device is selected. When `MD` is high, the chip is in output mode, and the output buffers are enabled. When `MD` is `'0'`, the chip is in input mode, and the `STRB` is used to latch data and to reset the service request flip-flop `SRQ`. `SRQ` is set when `NDS1` is low or the device is selected. When `MD` is `'0'`, the output buffers are enabled whenever the device is selected.

```
ENTITY I8212 IS
   PORT(DI   :  IN Bit_Vector(7 DOWNTO 0);
        DO   : OUT Bit_Vector(7 DOWNTO 0);
        NDS1,
        DS2,
        MD,
        STRB,
        NCLR :  IN Bit;
        NINT : OUT BIT
       );
END I8212;
```

Reprinted with permission. Copyright © by VHDL International

Figure 5-17. Entity Interface for an Intel Buffered Latch (Ref. 11)

```
USE work.functions.ALL;
ARCHITECTURE data_flow OF I8212 IS

   SIGNAL S0, S1, S2, S3 : Bit;
   SIGNAL SRQ: Bit;
   SIGNAL Q: Wired_OR_Bit_Vector(7 DOWNTO 0);

BEGIN
   C1: S0 <= NOT NDS1 AND DS2;
   C2: S1 <= (S0 AND MD) OR (STB AND NOT MD);
   C3: S2 <= S0 NOR NOT NCLR;
   C4: S3 <= S0 OR MD;
   B: BLOCK(S1 = '1' AND NCLR = '1')
      BEGIN
         C5: Q  <= GUARDED DI;
         C6: Q  <= "00000000" WHEN (NCLR = '1');
         C7: D0 <= Q WHEN (S3 = '1') ELSE "ZZZZZZZZ";
      END BLOCK B;
   P: PROCESS
      BEGIN
         IF (S2 = '0') THEN SRQ <= '1';
         ELSIF (STRB = '1') THEN SRQ <= '0';
         END IF;
         WAIT ON S2, STRB;
      END PROCESS P;
   C8: NINT <= NOT SRQ NOR S0;
END data_flow;
```

Reprinted with permission. Copyright © by VHDL International.

Figure 5-18. Synthesizable Architecture Body for the Intel Buffered Latch (Ref. 11)

5-3 VHDL DID SIMULATION REQUIREMENTS FOR BEHAVIORAL MODELS

The VHDL DID (Ref. 1) requires that all VHDL behavioral models exhibit certain responses to stimuli, timing, and error handling characteristics. These requirements are discussed in the next three subparagraphs.

5-3.1 CORRECT FUNCTIONAL RESPONSE TO STIMULI

The VHDL DID requires that behavioral models correctly express the function of their corresponding physical units. VHDL supports the verification of a model, typically via simulation. VHDL models are typically simulated in the context of a test bench.

Subpar. 10.2.5 of the VHDL DID (Ref. 1) requires VHDL test benches for each VHDL module. Part of the development of an effective test bench is creating test vectors to provide stimulus for the model. Another part of the development of an effective test bench is defining the correct responses to the test vectors. During the top-down development of a design, a high-level behavioral model can be used to generate the correct responses to a set of test vectors. When a less abstract model is subsequently developed, the responses of the low-level model can be verified by comparison with the results produced by the high-level behavioral model.

Some standard strategies are used to generate test vector sets for behavioral models. At the algorithmic level the following approaches can be used:

1. The test vectors should exercise all of the functions of the hardware module. For example, if the hardware module is programmable, each instruction should be tested.
2. The test vector set should include sequences of vectors that represent normal operational sequences of the system. If the model is general enough to accurately model system startup and shutdown, these transient modes of operation should also be tested.
3. The test vectors should include stress tests that reflect severe or unusual loads on the system but loads that are within the specifications for the hardware. These tests may include timing stress tests.
4. The test vectors should include invalid inputs that are "almost" valid inputs. These vectors test the error-handling ability of the model.
5. The input data should be divided into equivalence classes in which elements within each class are handled similarly. The simplest form of equivalence class partitioning on a data input is to divide it into classes of valid and invalid inputs. The test vector set should include representatives of each equivalence class. The number of equivalence classes may be chosen to reflect the time and resources available for testing. It is not practical to test most interesting functions exhaustively because of their internal complexity.

At the ISA level a similar set of guidelines for test vectors can be used. All of the instructions of a programmable processor should be tested. Normal sequences of instructions should also be tested. Equivalence classes for different input and output data formats also should be considered.

For VHDL behavioral models based on finite state machines, there are several more formal strategies used to generate test vectors. One strategy is based on testing the reachability of all of the states in the model. As a minimum, a sequence of test vectors should be defined that drives the model into every state.

Another strategy is to test the liveness of the model. This strategy says that except for certain specified terminal states, it should be possible to get from any given state to any other state. Another check for finite state machine-based models is a static check.

The code should be reviewed to make sure that any invalid states or invalid inputs are detected and appropriate error messages are generated. In the model in Fig. 5-15 whenever an out-of-range memory address is encountered, an error is asserted and a message is generated. Because the number of possible states of a processor may be very large, it may be necessary to define equivalence classes on states and test only some of the representative states in each class. For example, a tester may not distinguish between the data values in registers of a processor and instead may define only states based on the current instruction being processed.

When a hierarchical model is developed, a bottom-up testing strategy is recommended. Once the testing of individual submodels is complete, the entire model should be tested by executing tests designed to exercise all of the major functional groups as they work together. To assist in bottom-up testing, submodels should test their inputs and gracefully handle invalid inputs.

5-3.2 SIMULATION TIMING

Subpar. 10.2.3.2 of the VHDL DID requires that VHDL models exhibit correct timing behavior at the external interface including best, worst, and nominal output delays. Correct timing behavior should be monitored by individual VHDL modules, i.e., VHDL modules should test for invalid timing conditions on their inputs, such as inputs violating setup and hold conditions.

The VHDL language provides powerful facilities to detect improperly timed signals. These include assertion statements, passive processes and subprograms, and built-in attributes for signals. These mechanisms can be used to examine the timing relationships among the signals associated with an interface. If a timing violation is discovered by these checks, the module can indicate that the violation has occurred. Checks on these timing constraints can be implemented with passive processes (Ref. 12). This topic is discussed in detail in par. 5-4.

5-3.3 ERROR HANDLING

Subpar. 10.2.2.2 of the VHDL DID (Ref. 1) requires that "timing and electrical requirements (e.g., setup and hold times or power supply voltage extremes) shall be expressed in such a manner as to cause the simulator to generate error messages should the requirement be violated during a simulation.".

The assertion statement provides a way for model writers to issue messages when an error occurs. These messages should pinpoint the location of the error. The VHDL DID requires that these messages identify the design entity, process, procedure, or function in which the error occurred. In the case of components used many times within a model, the simulator must provide contextual information on exactly which component instance raised the exception.

Since not all errors require that the simulation be halted, some means must be available to categorize the severity of errors. VHDL provides this means with the severity clause of the assertion statement, which allows users to specify the severity of different errors. The action taken by the simulator can then depend on the severity of the error encountered.

The assertion statement may not be powerful enough for a model builder's needs; consequently, VHDL provides passive processes in which arbitrary computations can be performed. A passive process is a process that neither directly or indirectly, i.e., in a procedure called by the process, assigns to signals.

Passive processes, like any other process, can have a sensitivity list so more complex timing relations among signals can be examined. Appropriate messages can be written to external files, displayed immediately to users, or both.

```
ENTITY FPAdd IS
        PORT (  In1:  IN  real;
                In2:  IN  real;
                Out1: OUT real );
END FPAdd;

ARCHITECTURE Function OF FPAdd IS
BEGIN
   Out1 <= In1 + In2;
END Function;
```

Figure 5-19. Entity Interface and Architecture Body for a Functional Model Without Timing

5-4 TIMING IN BEHAVIORAL MODELS

5-4.1 TIMING SHELLS

It is often useful to separate the description of timing and function in behavioral models, especially in the early stages of the model development cycle. With this separation of concerns it is possible to modify timing and behavioral models (semi-)independently. This separation is particularly useful if the timing requirements external to an entity are known but the behavior is not finalized. This separation of concerns can be achieved in VHDL through the use of a timing shell.

A timing shell is used to define delays for signal assignments in behavioral models independently of the function-computing part of the model. Figs. 5-19, 5-20, 5-21, 5-22, and 5-23 describe a timing shell and a functional entity that when combined provide a behavioral model. This example shows how tradeoffs can be performed by using different timing shells with the same function to represent different implementations.

The example included here is a floating point adder. Fig. 5-19 shows the functional entity including both its entity interface declaration and a functional architecture body. It uses the built-in floating point addition function to compute the result, and it uses unit delay, i.e., the signal assignment statement in the architecture body has no delay clause.

Fig. 5-20 contains the package declaration for a set of functions that supports the computation of the delay for an implementation of the floating point adder. In this example, it is assumed that the floating point addition is done by an ALU with a simple shifter that can shift a word only one bit to the left or right per clock cycle. Therefore, the delay for a floating point addition is dependent upon the inputs, as well as on the time for a clock cycle.

Fig. 5-21 shows the body of the timing delay function. This timing function assumes that the ALU can perform a fixed point addition in one cycle, can compute the number of shifts required to align the inputs or to align the output in one cycle, and can shift a word one bit per clock cycle. The package includes functions to determine the amount of alignment

```
-- Example of a package that supports timing shell
PACKAGE Timing IS
       -- This function gets the exponent of its floating point argument
    FUNCTION GetExp( SIGNAL val: real ) RETURN integer;
       -- This function computes the maximum of two integers
    FUNCTION MaxInt (left: integer; right: integer) RETURN integer;
       -- This function computes the minimum of two integers
    FUNCTION MinInt (left: integer; right: integer) RETURN integer;
    FUNCTION SimpleShiftDelay(
                SIGNAL In1, In2, In3: real;
                CONSTANT ClockTime: time ) RETURN time;
END Timing;
```

Figure 5-20. Package Declaration for a Model That Uses a Timing Shell

```
FUNCTION SimpleShiftDelay(
   SIGNAL In1, In2, In3: real;
   CONSTANT ClockTime: time ) RETURN time IS

   VARIABLE Exp1:        integer;
   VARIABLE Exp2:        integer;
   VARIABLE Exp3:        integer;
   VARIABLE AlignedExp:  integer;
   VARIABLE ClockCount:  natural;
BEGIN
   Exp1 := GetExp(In1);
   Exp2 := GetExp(In2);
   Exp3 := GetExp(In3);
   -- Compute the exponent for the alignment
   AlignedExp := MaxInt(Exp1,Exp2);
   -- Initialize ClockCount with the time to compute the alignment
   ClockCount := 1;
   -- Add the number of clocks to align inputs
   ClockCount := AlignedExp - MinInt(Exp1,Exp2);
   -- Add the number of clocks to perform the fixed point addition
   ClockCount := ClockCount+ 1;
   -- Add the number of clocks to compute the result alignment
   ClockCount := ClockCount+ 1;
   -- Add the number of clocks to align the result
   ClockCount := ClockCount
              + MaxInt(AlignedExp,Exp3) - MinInt(AlignedExp,Exp3);
   -- Convert ClockCount to a delay time using ClockTime
   RETURN ClockTime * ClockCount;
END SimpleShiftDelay;
```

Figure 5-21. Function Definition for a Timing Function for a Floating Point Adder

required by computing the exponent (base 2) of the inputs and the result. The number of shifts required to align the inputs is the difference between the maximum input exponent and the minimum input exponent. The number of shifts required to align the output is the difference between the maximum exponent of the aligned input and the result and the minimum exponent of the aligned input and the result.

Fig. 5-22 shows the entity interface declaration of the behavioral model, i.e., the model that uses the timing shell in combination with the functional entity to model both function and timing. This entity declaration includes a generic that defines the clock cycle time.

Fig. 5-23 shows the architecture body of the timing shell entity. A configuration specification links the functional entity described in Fig. 5-19 to the component instance in this architecture. The architecture body contains a component instance whose port map links the functional output to the internal signal `OutVal`. A concurrent signal assignment statement is used to delay the output of the result by the time computed using the function `SimpleShiftDelay`.

```
USE work.Timing.ALL;
ENTITY TimedFPAdd IS
        GENERIC(ClockCycle : time := 500 ns);
        PORT (  Left:   IN  real;
                Right:  IN  real;
                Result: OUT real
             );
END TimedFPAdd;
```

Figure 5-22. Entity Interface for a Model That Uses a Timing Shell

```
ARCHITECTURE SimpleShift OF TimedFPAdd IS
   COMPONENT Adder
      PORT( In1:  IN  real;
            In2:  IN  real;
            Out1: OUT real );
   END COMPONENT;
   SIGNAL OutVal : real;
   FOR Comp1: Adder USE work.FPAdd(Function);
BEGIN
   Comp1: Adder PORT MAP ( Left, Right, OutVal );
   Result <= OutVal after SimpleShiftDelay(Left,Right,OutVal,ClockCycle);
END SimpleShift;
```

Figure 5-23. Timing Shell Architecture Body

An alternative implementation of the floating point adder that uses a barrel shifter would have a different delay function, but it would provide the same result value. This difference could be implemented with a different architecture body for `TimedFPAdd`, which uses a different delay function. A further refinement is to use the same name for delay functions for different implementations. The different delay functions are placed in different libraries. The same architecture body could be used for the different implementations, and the appropriate delay function would be selected using a configuration declaration, as described in par. 3-8.

5-4.2 CLOCK RATES

In algorithmic and ISA VHDL models, a common approach to specifying the timing of a synchronous hardware module is to specify the time for an operation in terms of the number of clock cycles required to complete the operation and then to measure the clock rate for the module. This approach has been taken in the architecture body in Fig. 5-5. More accurate timing may be available after the hardware has been designed and the microcode (if any) has been written.

5-4.3 CRITICAL PATH DELAY TIMES

Clock rates are critical to determining the performance of synchronous hardware modules. A register-transfer level or more detailed model can be used to predict the clock rate of a system by calculating the critical path times between registers or latches. The nature of semiconductor devices means that the critical path depends not only on the number of levels of logic between latches but also upon the data flowing through the network and the time it takes for devices to change output signal states and strengths. Thus, if the clock rate is being pushed to the limits of the technology, detailed analysis of the structure and physical layout of critical paths is needed.

In VHDL models error messages generated by setup or hold violations can be used to detect excessive clock rates. However, this approach requires a set of test vectors that force worst-case dynamic situations. Most computer-aided engineering (CAE) environments include tools that statically analyze worst-case timing. Generally, these tools are driven by a gate-level netlist and other data dependent on the implementation technology. The VITAL timing approach (Ref. 13) allows back annotation of worst-case timing information through the use of standard delay format (SDF) files.

5-4.4 BEST-CASE, WORST-CASE, AND NOMINAL DELAYS

The VHDL DID (Ref. 1) requires that all VHDL models incorporate worst-case, nominal-case and best-case timing delays at the model interface. Providing several delay models allows designers to evaluate the performance of the design under adverse as well as optimistic operating conditions. Simulating a system in which different components operate under different conditions may show anomalies that do not occur under more typical operating conditions. This kind of simulation provides useful information on the components likely to have problems when incorporated into a larger system.

Fig. 5-24 shows timing curves for a component as a function of either temperature or voltage. T_{min} and V_{min} are the minimum expected operating temperature and supply voltage, V_{nom} is the nominal operating voltage at the nominal operating temperature of 27°C, and T_{max} and V_{max} are the maximum expected operating temperature and supply voltage. t_{max}, t_{nom}, and t_{min} are the maximum, nominal, and minimum delays, respectively, that correspond to each of the temperature and voltage combinations. There is a range of delay times at any temperature and voltage combination because of slight variations in timing from component to component.

The diamonds and the ovals in the figure indicate measured times that are available for the component. The ovals indicate the timing measurements from the illustrated set of measurements that must be included in the VHDL timing model. The diamonds indicate optional timing values; better VHDL models consider a wider range of environmental factors.

5-4.5 PARAMETERIZED DELAY MODELS

Parameterized delay models permit more elaborate timing models to be constructed. In real hardware, timing delays are often functions of environmental factors such as supply voltage, output loading, or temperature. If these effects can be modeled, timing models can be constructed that

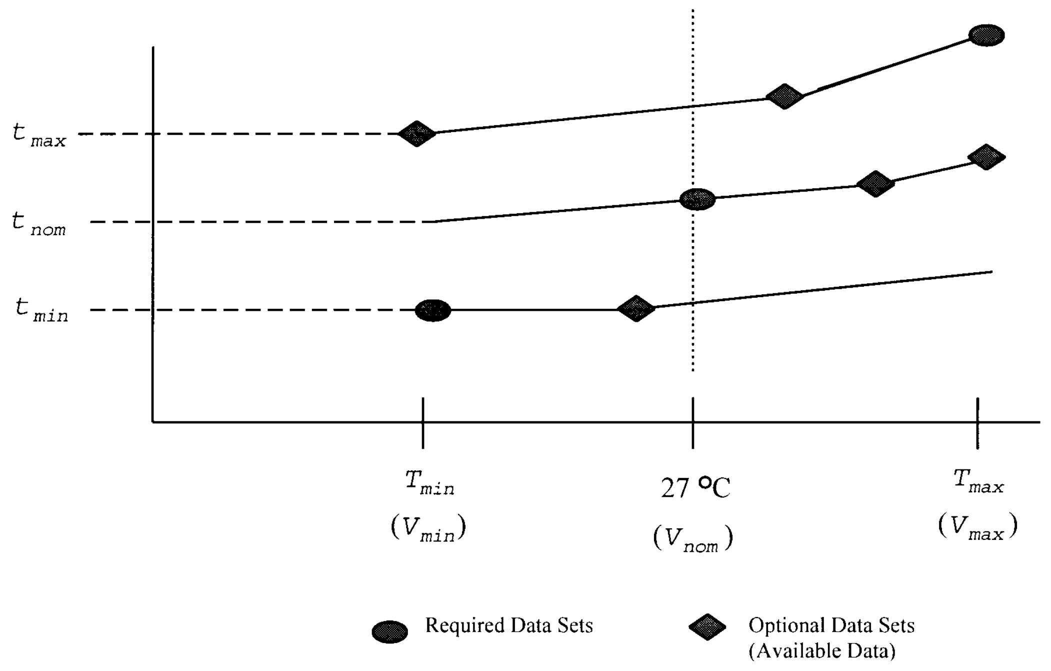


Figure 5-24. Best-, Nominal-, and Worst-Case Timing Curves (Ref. 14)

cover a broad range of environmental conditions rather than just those for which measurements have been taken.

These kinds of timing models can be constructed in VHDL by including environmental factors in delay calculations. Mechanisms used to supply environmental information include generics, special signals to represent environmental factors (e.g., temperature or radiation dosage), and computations performed by the design entities that evaluate these factors on an ongoing basis. An example shown in Figs. 5-25, 5-26, 5-27, and 5-28 adjusts the timing delay on an output signal as a function of the input voltage.

The package declaration for this example is Fig. 5-25. In the package declaration the voltage is declared as a physical type, and a subtype is declared that specifies the possible range of supply voltages. The three possible options for delay values are described in terms of an enumerated type `delay_case`.

```
PACKAGE reg_delay IS
      -- Declares a physical type for voltages
   TYPE voltage IS RANGE -50_000_000 TO 50_000_000
       units
          uv;
          mv    = 1000 uv;
          volt  = 1000 mv;
       END units;
   SUBTYPE supply_voltage_range IS voltage RANGE (0.0 volt TO 10.0 volt);
   TYPE delay_case IS (worst, nominal, best);
   TYPE delay IS ARRAY(delay_case) OF time RANGE (0 ns TO time'HIGH);
      -- Each signal for which a delay is specified should have its delay
      -- specified as below:
   CONSTANT out_delay: delay := (worst   => 15 ns,
                                 nominal => 10 ns,
                                 best    => 5 ns);
      -- This function computes an adjusted delay based on
      -- the current input voltage and a selected delay value.
   FUNCTION derated_delay(delay_val: time; supply_voltage: voltage)
            RETURN time;
END reg_delay;
```

Figure 5-25. Package Declaration for a Model That Uses Parameterized Timing

Fig. 5-26 shows the corresponding package body including the definition of the derating function. The derating function uses voltages within an acceptable range to compute a multiplicative factor that ranges from 0.8 to 1.2. This derating function can be reused in other parts of the model as required.

An acceptable range for the operating voltage is determined by an assertion statement in the entity interface shown in Fig. 5-27. Voltages outside this range cause an exception. This assertion statement ensures that the parameters for the derating function are within the acceptable range for the function. In this design entity the supply voltage is defined as a signal.

Fig. 5-28 shows the architecture body for the entity. The parameterized delay function is called in the AFTER clause of each of the signal assignment statements in the single process which comprises the architecture body.

5-4.6 TIMING DEFINITION PACKAGE

Since a simple model may be reused many times to construct a more complex model, it is desirable to provide the timing information so it can be shared by all instances of the

```
PACKAGE body reg_delay IS
      -- This function computes the actual delay based on the current
      -- voltage level and the delay selected by the delay model.
   FUNCTION derated_delay(delay_val: time;
                          supply_voltage: voltage)
                          RETURN time IS
       VARIABLE int_factor: integer;
       VARIABLE derate: real;
   BEGIN
      -- The first expression is a "universal integer"
      -- representing the number of micro volts by which the
      -- input voltage varies from a nominal 5.0 volts.
      -- The second factor converts it into a real value in the range
      -- 0.8 to 1.2. This could be done all in one expression, but was
      -- separated into two to enhance the clarity of the example. The
      -- input voltage can range from 4.8V to 5.2V.
      int_factor := (supply_voltage - 5.0 volt)/uv;
      derate := real(int_factor)/1.0e6;
      RETURN derate * delay_val;
   END derated_delay;
END reg_delay;
```

Figure 5-26. Package Body for a Model That Uses Parameterized Timing

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.reg_delay.ALL;
ENTITY PowLatch IS
          GENERIC( delay_model: delay_case := nominal);
          PORT   ( power:       supply_voltage_range;
                   input:       std_logic;
                   hold:        std_logic;
                   output: OUT  std_logic);
BEGIN
       -- This assertion ensures that the input voltage is within
       -- the allowable operating range.
    ASSERT (power >= 4.8 volt and power <= 5.2 volt)
        REPORT "Power Supply Voltage Is Out Of Range."
            SEVERITY ERROR;
END PowLatch;
```

Figure 5-27. Entity Interface for a Model That Uses Parameterized Timing

```
ARCHITECTURE ParmTime OF PowLatch IS
BEGIN
   LatchProc: PROCESS(hold,input)
   BEGIN
    -- These signal assignment statements call the derating function
    -- with the proper signal delay and the current input voltage.
      IF hold = '1' OR hold = 'H'
         THEN output <= input AFTER
                        derated_delay(out_delay(delay_model), power);
      ELSIF hold = 'X' OR hold = 'U'
         THEN output <= 'X' AFTER
                        derated_delay(out_delay(delay_model), power);
      END IF;
   END PROCESS;
END ParmTime;
```

Figure 5-28. Architecture Body for a Model That Uses Parameterized Timing

model. This approach ensures that all instances are operating with the same information and simplifies changing the timing information for all instances, if necessary. The VHDL package provides this mechanism.

All timing information related to a particular model can be provided in a package (or packages), which is then used by the VHDL model. (This approach is described in subpar. 3-8.3.2.) The information in this package is shared by all instances of the model that uses this package. Any changes to the timing of the model can be made in one place and automatically propagated to all instances of the model.

Although timing information could be hard-wired into the model itself, this practice is not desirable. Timing information is dependent on the details of specific implementation of the function. If timing information is incorporated in a package, different timing values can be used with the same VHDL model to account for different implementation technologies.

Figs. 5-29, 5-30, and 5-31 show the use of a timing package to provide the timing for a simple ALU. The timing data are defined with deferred constants. This approach allows the timing of the behavioral model to be changed without changing the text of the architecture body for the behavioral model. In this case the timing package is specific to the behavioral model because the matrix of times is specified in terms of a specific set of signals in the model. Furthermore, this example is specific to a particular technology, namely, complementary metal-oxide semiconductor (CMOS).

Fig. 5-29 is the package declaration. In this package a four-dimensional matrix of delays is declared. The four dimensions are

1. The mode of the chip (which is determined from the input data to the chip)
2. The signal name
3. The supply voltage
4. The operating temperature.

The use of the signal name as a dimension of the table makes the table specific to a particular implementation of the chip in terms of the internal interconnections. The voltage levels and operating temperatures are declared as enumerated types. Thus they can take on a small number of discrete values. This approach is in contrast with the derating function described in Fig. 5-28, which provides timings for a continuous range of voltages.

A function is declared in the package that computes the mode of the chip from the input values. The package also includes a declaration of the possible modes of the chip as an enumerated type. Thus all the dimensions of the delay matrix are defined by enumerated types.

Fig. 5-30 shows the corresponding package body. The timing information shown in this package is implemented as a deferred constant. To change technologies, only the package body needs to be changed and reanalyzed. No other component of the model needs to be changed to include new timing information.

The Electronic Industries Association (EIA) has developed a more general table (Ref. 16) for output timings of single-level logic where the times depend upon the time required for the signal to change strengths and states.

```
--------------------------------------------------------------------------
-- Package : CMOS_181_PERFORMANCE_CHARACTERISTICS
--
-- Description:
--
-- This package exports types and constants required to represent the
-- performance characteristics for the CMOS 181 ALU.  The data in this
-- package is from the National Semiconductor MM54HC181 ALU specification.
--
--------------------------------------------------------------------------
--
PACKAGE cmos_181_performance_characteristics IS
    SUBTYPE Bit_Vector_0_to_3 IS Bit_Vector( 0 TO 3 );
    SUBTYPE Bit_Vector_3_to_0 IS Bit_Vector( 3 DOWNTO 0 );
    TYPE Volts IS ( LOW, MID, HIGH );
    --             2.0, 4.5, 6.0
    TYPE Temperature IS ( TYPICAL, G_LOW, G_MID, G_HIGH );
    --                     25C,     25C,   85C,   125C
    TYPE Mode IS ( SUM, DIF, LOGIC );
    TYPE Signal_Name IS ( A_EQ_B, C_N_4, NOT_F, NOT_G, NOT_P );
    TYPE Delay_matrix IS ARRAY(
             Mode'left TO Mode'right,
             Signal_name'left TO Signal_name'right,
             Volts'left TO Volts'right,
             Temperature'left TO Temperature'right ) OF Time;
    CONSTANT DELAY_TABLE;     -- deferred constant
    -- This function determines the chip mode based on the Mode
    -- bit and the select bits.  Valid modes are SUM, DIF, LOGIC,
    -- and OTHER.
    FUNCTION Get_Mode( M, S0, S1, S2, S3 : IN Bit ) RETURN Mode;
END cmos_181_performance_characteristics;
```

Reprinted with permission. Copyright © by James P. Hanna

Figure 5-29. Package Interface for a Model That Uses a Timing Package (Ref. 15)

```
PACKAGE body cmos_181_performance_characteristics IS

  CONSTANT DELAY_TABLE : Delay_matrix :=
        ( -- SUM

              ( -- AEQB
                --  Typical, G_low, G_mid, G_high
                ( ( 0 ns,    0 ns,  0 ns,  0 ns ),      -- Low
                  ( 0 ns,    0 ns,  0 ns,  0 ns ),      -- Mid
                  ( 0 ns,    0 ns,  0 ns,  0 ns ) ),    -- High
                -- CN_4
                --  Typical, G_low,  G_mid,  G_high
                ( ( 110 ns,  250 ns, 325 ns, 375 ns ),  -- Low
                  (  35 ns,   50 ns,  63 ns,  75 ns ),  -- Mid
                  (  30 ns,   43 ns,  53 ns,  65 ns ) ),-- High
                -- NOTF
                --  Typical, G_low,  G_mid,  G_high
                ( ( 115 ns,  240 ns, 300 ns, 360 ns ),  -- Low
                  (  35 ns,   48 ns,  60 ns,  72 ns ),  -- Mid
                  (  30 ns,   41 ns,  51 ns,  61 ns ) ),-- HIgh
                -- NOTG
                --  Typical, G_low, G_mid, G_high
                ( ( 55 ns, 120 ns, 160 ns, 200 ns ),    -- Low
                  ( 17 ns,  24 ns,  30 ns,  36 ns ),    -- Mid
                  ( 14 ns,  20 ns,  25 ns,  30 ns ) ),  -- High
                -- NOTP
                --  Typical, G_low, G_mid, G_high
                ( ( 70 ns, 150 ns, 189 ns, 224 ns ),    -- Low
                  ( 20 ns,  30 ns,  38 ns,  45 ns ),    -- Mid
                  ( 17 ns,  26 ns,  32 ns,  38 ns ) ) ),-- High
          -- DIF

              ( -- AEQB
                --  Typical, G_low,  G_mid,  G_high
                ( ( 120 ns,  280 ns, 350 ns, 420 ns ),  -- Low
                  (  40 ns,   56 ns,  70 ns,  84 ns ),  -- Mid
                  (  35 ns,   48 ns,  60 ns,  72 ns ) ),-- High
                -- CN_4
                --  Typical, G_low,  G_mid,  G_high
                ( ( 120 ns,  280 ns, 350 ns, 420 ns ),  -- Low
                  (  40 ns,   56 ns,  70 ns,  84 ns ),  -- Mid
                  (  35 ns,   48 ns,  60 ns,  72 ns ) ),-- High
                -- NOTF
                --  Typical, G_low,  G_mid,  G_high
                ( ( 120 ns,  275 ns, 344 ns, 344 ns ),  -- Low
                  (  40 ns,   55 ns,  69 ns,  83 ns ),  -- Mid
                  (  34 ns,   47 ns,  59 ns,  69 ns ) ),-- High
                -- NOTG
                --  Typical, G_low,  G_mid,  G_high
                ( ( 70 ns,   150 ns, 189 ns, 224 ns ),  -- Low
                  ( 20 ns,    30 ns,  38 ns,  45 ns ),  -- Mid
                  ( 17 ns,    26 ns,  32 ns,  38 ns ) ),-- High
                -- NOTP
                --  Typical, G_low,  G_mid,  G_high
```

(cont'd on next page)

Figure 5-30. Package Body for a Model That Uses a Timing Package (Ref. 15)

```
                ( ( 70 ns,   150 ns, 189 ns, 224 ns ),  -- Low
                  ( 20 ns,    30 ns,  38 ns,  45 ns ),  -- Mid
                  ( 17 ns,    26 ns,  32 ns,  38 ns ))), -- High

          -- LOGIC
              ( -- AEQB
                --  Typical, G_low, G_mid, G_high
                ( ( 0 ns,    0 ns,  0 ns,  0 ns ),     -- Low
                  ( 0 ns,    0 ns,  0 ns,  0 ns ),       -- Mid
                  ( 0 ns,    0 ns,  0 ns,  0 ns ) ),     -- High
                -- CN_4
                --  Typical, G_low, G_mid, G_high
                ( ( 0 ns,    0 ns,  0 ns,  0 ns ),       -- Low
                  ( 0 ns,    0 ns,  0 ns,  0 ns ),       -- Mid
                  ( 0 ns,    0 ns,  0 ns,  0 ns ) ),     -- High
                -- NOTF
                --  Typical, G_low,  G_mid,  G_high
                ( ( 120 ns,  275 ns, 344 ns, 344 ns ),  -- Low
                  (  40 ns,   55 ns,  60 ns,  83 ns ),  -- Mid
                  (  34 ns,   47 ns,  59 ns,  69 ns ) ),-- High
                -- NOTG
                --  Typical, G_low, G_mid, G_high
                ( ( 0 ns,    0 ns,  0 ns,  0 ns ),       -- Low
                  ( 0 ns,    0 ns,  0 ns,  0 ns ),       -- Mid
                  ( 0 ns,    0 ns,  0 ns,  0 ns ) ),     -- High
                -- NOTP
                --  Typical, G_low, G_mid, G_high
                ( ( 0 ns,    0 ns,  0 ns,  0 ns ),       -- Low
                  ( 0 ns,    0 ns,  0 ns,  0 ns ),       -- Mid
                  ( 0 ns,    0 ns,  0 ns,  0 ns ))));    -- High

    --
    -- This function determines the chip mode based on the Mode
    -- bit and the select bits.  Valid modes are SUM, DIF, and LOGIC.
    --
    FUNCTION Get_Mode( M, S0, S1, S2, S3 : in Bit ) RETURN Mode IS

    BEGIN
        IF M = '1' THEN
            RETURN LOGIC;
        ELSIF ( S3 = '0' and S2 = '0' and S1 = '1' and S0 = '1' ) or
              ( S3 = '0' and S2 = '1' and S1 = '1' and S0 = '0' ) or
              ( S3 = '0' and S2 = '1' and S1 = '1' and S0 = '1' ) or
              ( S3 = '1' and S2 = '0' and S1 = '1' and S0 = '1' ) or
              ( S3 = '1' and S2 = '1' and S1 = '1' and S0 = '1' ) THEN
            RETURN DIF;
        ELSE
            RETURN SUM;
        END IF;
    END Get_Mode;

END cmos_181_performance_characteristics;
```

Figure 5-30. (cont'd)

5-4.7 TIMING THROUGH FILE INPUT

Another way to incorporate timing information into a simulation model is to read the timing information from external data files using the VHDL file I/O capability. This approach allows modification of the timing behavior at any time during a simulation. The example in Figs. 5-31, 5-32, and 5-33 shows the "latch" model described in Figs. 5-27 and 5-28, but its timing information now comes from an external file.

Fig. 5-31 shows a package declaration similar to that in Fig. 5-29. In this package a file is declared of records; each record contains a signal name and a delay value. An array of delays, similar to the matrix of delays defined in Fig 5-29, is also declared in this package, and this package declares the function that reads the file and fills the array of delays.

Fig. 5-32 shows the corresponding package body. It contains the definition of the function that reads the delay information from the file.

```
-- A package which defines the structure of the external timing data
-- file.
PACKAGE fio_delay IS
   TYPE delay_case IS (best, nominal, worst);
   TYPE signal_name IS (S1, S2, S3, S4, S5 );
   TYPE delay IS ARRAY(delay_case) OF time;
   TYPE delay_record IS RECORD
                            signal_id: signal_name;
                            delay_value: delay;
                        END RECORD;

   TYPE delay_file IS FILE OF delay_record;
   TYPE sig_delays IS ARRAY(Signal_name'left TO Signal_name'right) OF delay;
      -- This is the function which will read the file and initialize the
      -- signal delays.
   FUNCTION init_delays RETURN sig_delays;
      -- All signals in the model for which delays are to be initialized
      -- from a file are assumed to be members of this array. This
      -- function will be called when the model is elaborated.
   CONSTANT signal_delay: sig_delays := init_delays;
END fio_delay;
```

Figure 5-31. Package Declaration for a Model That Uses File I/O for Timing

```
PACKAGE body fio_delay IS

    FUNCTION init_delays RETURN sig_delays IS
        -- Delay information is kept in a file which is called
        -- "latchdel"
        FILE dfile: delay_file IS IN "latchdel";
        VARIABLE file_element: delay_record;
        VARIABLE ret_val: sig_delays := (others => (others => 0 ns));
        BEGIN
            -- Loop through the file, reading each file record in
            -- turn. Assign the delay data to the signal identified
            -- in the file record.
            WHILE not endfile(dfile) LOOP
                read(dfile, file_element);
                ret_val(file_element.signal_id) := file_element.delay_value;
            END LOOP;
            RETURN ret_val;
        END init_delays;
END fio_delay;
```

Figure 5-32. Package Body for a Model That Uses File I/O for Timing

Fig. 5-33 contains the entity interface and the architecture body for the latch model that uses the file I/O package to obtain its timing information. The generic `delay_model` declared in the entity interface is used to select the best-, nominal-, or worst-case timing. The code in the architecture body is identical to that in Fig. 5-28 except that the `AFTER` clauses now contain an array reference rather than a function call. In this example the timing information is loaded from an external file into an array at the start of the simulation.

5-4.8 MODELING ASYNCHRONOUS TIMING

VHDL provides for the creation of accurate timing models. In particular, small timing glitches can be modeled. Glitches are short-duration output pulses caused by rapid changes in input signal values. Including timing information with this level of detail is possible in VHDL, but it is usually inappropriate, particularly in behavioral models.

VHDL provides control of such timing details with two kinds of delay: transport and inertial signal. A signal assignment statement containing the word “transport” transmits the value of the input signal to the output signal regardless of its duration.

The use of transport delay is particularly inappropriate in behavioral models in which the glitches generated may not correspond to those in the actual hardware. These glitches are particularly risky in behavioral models if they are caused by rare timing conditions and are not revealed by standard behavioral test vectors.

Inertial delay can be used in signal assignment statements to prevent a model from generating glitches. Inertial signal assignment statements (the default in VHDL) do not transmit changes in signal values with a duration less than that specified by the time of the first waveform in the signal assignment statement. Thus glitches are prevented from propagating through the model. Details of the VHDL delay mechanism are discussed in subpar. 3-2.3.1.

Asynchronous timing constraints are timing constraints that are applied to single bit signals in isolation. Fig. 5-34 shows some general timing constraints on asynchronous timing of single bit signals. In the figure, th_{min} and th_{max} represent the minimum and maximum intervals a signal can be high, and tl_{min} and tl_{max} represent the minimum and maximum intervals a signal can be low.

The values for these constraints can be implemented as constants stored in a timing package. Checking of these constraints, required by the DID (Ref. 1), can be implemented through functions declared in a timing package and invoked by assertions associated with individual input ports or by passive process in the design entity if more sophisticated timing checks are needed.

Timing values may apply globally, may be associated with a specific technology, or may be associated with a specific hardware component. If the values are global or are associated with a specific technology, they can be defined in a global package. Generics that describe the technology can be used to select the appropriate constraints.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.fio_delay.ALL;
ENTITY Latch IS
       GENERIC( delay_model: delay_case := nominal);
       PORT( input:      std_logic;
             hold:       std_logic;
             output: OUT std_logic);

END;

ARCHITECTURE FileIOTime OF Latch IS
BEGIN
   LatchProc: PROCESS(hold,input)
   BEGIN
    -- These signal assignment statements call the derating function
    -- with the proper signal delay and the current input voltage.
      IF hold = '1' OR hold = 'H'
         THEN output <= input AFTER signal_delay(S4)(delay_model);
      ELSIF hold = 'X' OR hold = 'U'
         THEN output <= 'X' AFTER signal_delay(S4)(delay_model);
      END IF;
   END PROCESS;
END FileIOTime;
```

Figure 5-33. Entity for a Module That Uses File I/O for Timing

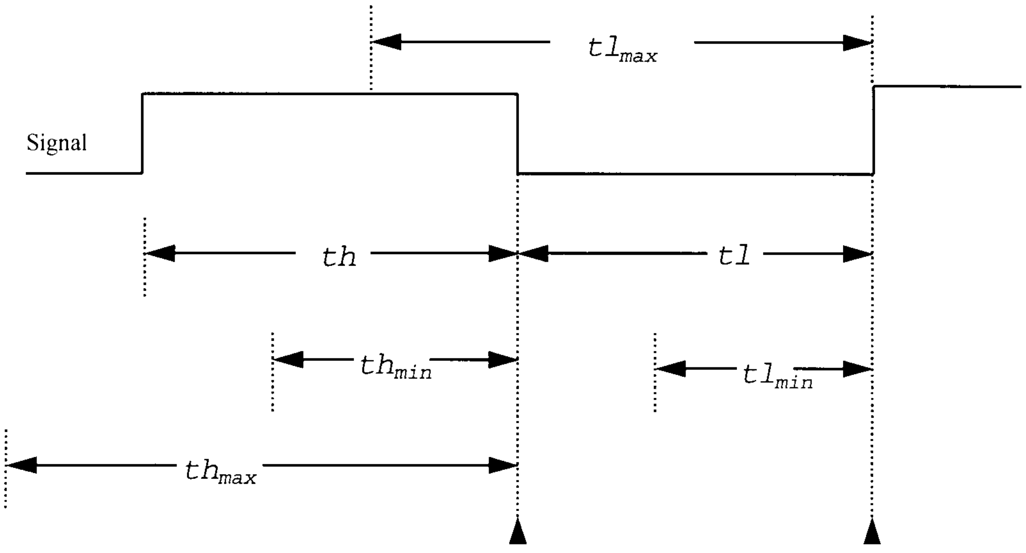


Figure 5-34. Potential Asynchronous Timing Constraints (Ref. 14)

5-4.9 MODELING SYNCHRONOUS TIMING

A synchronous timing constraint is a timing constraint that is applied to single-bit signals with respect to a second synchronizing signal. Fig. 5-35 shows some general constraints on synchronous timing of single-bit signals. The figure illustrates two types of synchronous timing constraints: setup time constraints and hold time constraints. These constraints are computed by comparing a synchronizing signal (`Clk` in Fig. 5-35) with another single-bit signal (`D` in Fig. 5-35). Checking of these constraints, as required by the DID (Ref. 1), can be implemented through functions declared in a timing package and invoked by assertions associated with clock inputs and individual data input ports.

The values for these constraints may apply globally, or they may be associated with a specific technology, or they may be measured for a specific hardware component. If the values are global or they are associated with a specific technology, they can be defined in a global package. Generics that describe the technology can be used to select the appropriate constraints. If they are measured for a specific hardware component, they may be defined as generics in the corresponding entity interfaces.

Figs. 5-36, 5-37, and 5-38 show a package containing reusable procedures used to check setup and hold times of a single-bit signal against a reference signal, such as a clock. Fig. 5-36 contains the package declaration, which declares

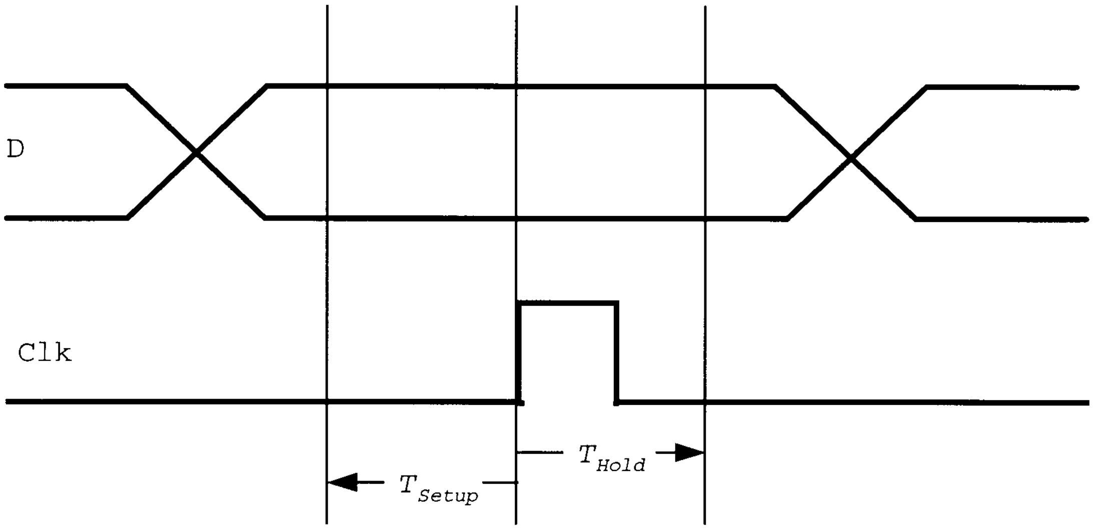


Figure 5-35. Potential Synchronous Timing Constraints (Ref. 17)

```
PACKAGE CheckTiming IS
   PROCEDURE CheckSetupTime( SIGNAL ref:         bit;
                                    ref_edge:    bit;
                             SIGNAL checked:     bit;
                                    setup:       time;
                                    signal_name: string );

   PROCEDURE CheckHoldTime(  SIGNAL ref_delayed_by_hold_time: bit;
                                    ref_edge:    bit;
                             SIGNAL checked:     bit;
                                    hold:        time;
                                    signal_name: string );
END CheckTiming;
```

Figure 5-36. Package Interface That Checks Synchronous Timing Constraints

two timing check functions: `CheckSetupTime` and `CheckHoldTime`. There is a subtle difference in the arguments for the two timing check functions. The reference signal input to `CheckHoldTime` is delayed by the hold time amount. This delay is the reason for the different argument sets in Fig. 5-37. Both of these procedures take the constraint value as their fourth parameter.

Fig. 5-37 shows the procedure body for the `CheckSetupTime` procedure declared in Fig. 5-36. This procedure uses the built-in `textio` package, in which the type of `line` is defined as an access type, which points to a string. This procedure uses the `write` function of `textio` to build up the `errmsg` (error message) string. This string is output by the assertion statement and then deallocated so that a new message can be constructed. The procedure also uses the built-in attribute `last_event` to get the time

```
PROCEDURE CheckSetupTime( SIGNAL ref:         bit;
                                 ref_edge:    bit;
                          SIGNAL checked:     bit;
                                 setup:       time;

   VARIABLE errmsg:   line;
   CONSTANT setupmsg: string    := "Setup Time Violation On: ";
   CONSTANT linefeed: character := LF;
BEGIN
   -- This section checks for set-up timing violations by checking to
   -- see if the difference in time between the last time the signal
   -- to be checked changed value and the time of current transition
   -- of the reference signal meets or exceeds the required set-up time.
   -- If not, then the assertion statement reports the timing violation
   -- with an appropriate error message.

   LOOP
      WAIT UNTIL ref = ref_edge;
      IF checked'last_event < setup THEN
         write(errmsg, setupmsg );
         write(errmsg, signal_name & linefeed);
         ASSERT FALSE
            REPORT errmsg.ALL
               SEVERITY warning;
         deallocate(errmsg);
      END IF;
   END LOOP;
END CheckSetupTime;
```

Figure 5-37. Procedure Body That Checks Setup Time Constraints

when the checked signal last changed state. This time is compared with the setup time to ensure that the reference signal allows enough setup time for the checked signal. The `wait` statement ensures that the comparison is made only when the reference signal transitions to the state specified by `ref_edge`.

Fig. 5-38 shows the procedure body for the `CheckHoldTime` procedure declared in Fig. 5-36. Similarly to `CheckSetupTime`, this procedure uses the `write` function of `textio` to build up the `errmsg` string and uses the built-in attribute `last_event` to get the time when the checked signal last changed state.

Fig. 5-39 shows an example of the entity interface for a simple multiplexor. The entity interface uses the timing

```
PROCEDURE CheckHoldTime(  SIGNAL ref_delayed_by_hold_time: bit;
                                 ref_edge:    bit;
                          SIGNAL checked:     bit;
                                 hold:        time;
                                 signal_name: string ) IS
   VARIABLE errmsg:   line;
   CONSTANT holdmsg : string := "Hold Time Violation On: ";
   CONSTANT linefeed: character := LF;
BEGIN
     -- This section checks for hold time violations by checking to
     -- see if the difference in time between a transition on the
     -- reference signal (delayed by thold time units) and a
     -- transition on the signal to be checked meets or exceeds
     -- the required hold time. If not, then the assertion statement
     -- reports the timing violation with an appropriate error message.

   LOOP
      WAIT UNTIL ref_delayed_by_hold_time = ref_edge;
      IF checked'last_event < hold THEN
         write(errmsg, holdmsg );
         write(errmsg, signal_name & linefeed);
         ASSERT FALSE
            REPORT errmsg.ALL
               SEVERITY warning;
         deallocate(errmsg);
      END IF;
    END LOOP;
 END CheckHoldTime;
```

Figure 5-38. Procedure Body That Checks Hold Time Constraints

```
USE work.CheckTiming.ALL;
ENTITY LS74151 IS
   PORT( A, B, C                        : IN  bit;
         STROBE                         : IN  bit;
         D0, D1, D2, D3, D4, D5, D6, D7 : IN  bit;
         Y, W                           : OUT bit );
BEGIN
    CheckSetupTime(strobe, '1', a, 10 ns, "A");
    CheckSetupTime(strobe, '1', b, 10 ns, "B");
    CheckSetupTime(strobe, '1', c, 10 ns, "C");

    CheckHoldTime(strobe'delayed(15 ns), '1', a, 15 ns, "A");
    CheckHoldTime(strobe'delayed(15 ns), '1', b, 15 ns, "B");
    CheckHoldTime(strobe'delayed(15 ns), '1', c, 15 ns, "C");

END LS74151;
```

Figure 5-39. Entity Interface That Checks Timing Constraints

checks provided in the package shown in Figs. 5-36, 5-37, and 5-38. The timing checks are included in the entity interface declaration, not in the architecture body. Thus these timing checks are applied to all of the multiplexor implementations.

5-5 ANNOTATION OF BEHAVIORAL MODELS

The VHDL DID requires that models include explanatory comments that clarify the intent of the model. These comments must also include the following (taken directly from the VHDL DID (Ref. 1) subpar. 10.2.7):

"a. Any factors restricting the general use of this description to represent the subject hardware.

b. General approaches taken to modeling and particularly decisions regarding modeling fidelity.

c. Any further information which the originating activity considers vital to subsequent users of the descriptions."

This kind of information can be included in a model by using VHDL comments.

Fig. 5-40 shows the header for a VHDL design unit that includes a comprehensive set of annotations (Ref. 18).

Further guidance on documenting revisions to models that have already been delivered to the Government is provided in Ref. 18, which is included as Appendix A.

5-5.1 DESCRIPTION OF FUNCTION

A description that clarifies the intent of the model should include a narrative discussion of the overall function of the model and a description of its inputs and outputs.

The description should also include information on any interface timing constraints and information on the proper sequencing of control and data signals, i.e., the interface protocol, needed to ensure proper operation.

Generics of the model also should be explained. The data sheets supplied with hardware components provide proper guidance for the documentation of a model.

If an understanding of the details of internal algorithms is important to the proper use of a model, these details should be explained. Examples in which the algorithm is important include numerical algorithms for which accuracy depends on input values.

5-5.2 DESCRIPTION OF RESTRICTIONS

Any restrictions on using a model should be explained in the comments. Restrictions include operating speeds, bounds of generics, and other limitations. To the extent possible, any limitations should be enforced by using appropriate language features such as subtypes and assertions. Using these language features makes the model self-checking.

5-5.3 MODELING APPROACH

The VHDL DID requires that the "general approaches taken to modeling, particularly decisions regarding modeling fidelity", be described in the comments. The general approaches to modeling should describe the level of abstraction of the model (ISA, RTL, etc.), the typical use of the model, the logic conventions used, any external components needed to use the model, any documents needed to supplement the model (such as an explanation of the instruction set), and any industrial or military standards the model is intended to meet. The intent of these comments is to pro-

```
-- Design Unit Identifier: LS74151
-- Identification of Originator: RTI
-- DoD Approved Identifier: None
-- Was Model Previously Delivered: No
-- General Approach To Modeling and Fidelity:
--      This model was developed as a behavioral data flow model.  It provides
--      for accurate output delays for best, worst, and nominal timing
--      conditions.  The model uses the simple unresolved "bit" data type
--      as the fundamental signal data type.
-- Further Information: None
-- Restrictions:
--      Since the model uses an unresolved data type, care must be taken
--      to ensure that any signal connected to a port of this model has
--      only one driver.  A more sophisticated model should use a resolved
--      data type for the signals in the model.
 -- Assumptions: None
-- Previously Approved by DoD: No
PACKAGE LS74151Timing IS
...
END LS74151Timing;
```

Figure 5-40. Annotation of a VHDL Package Using Header Comments

vide the user with the information needed to use the model effectively.

Modeling fidelity should also be addressed, particularly in the area of timing models. The types of timing errors that are checked should be described, as well as the time scales associated with events within the model. Variations of internal timing with external conditions, if any, or the provision and use of different timing models should also be described.

Any other information the model builder considers useful should also be included. Such information includes any assumptions made about the simulation environment, such as the location and names of data files needed for the simulation of the model. Other information also includes version numbers of design entities used in the model that were supplied by outside vendors or information on the structure of the VHDL design library needed to compile the model successfully, and the compilation order for the library units of the model if this order is not obvious from the model itself.

5-5.4 REVISION HISTORY

A model may be revised or corrected over time. These changes should be documented in the model. This documentation should include the date the revision was made (as established by the revision control procedures of the developing organization), a brief description of the nature and purpose of the revision, and the organization and person responsible for the revision. This information should be included in one location in the module so that the entire revision history is available for review.

If the revision is a major change to the model and affects its externally visible functionality, the change should also be reflected in the module documentation.

5-5.5 BACK ANNOTATION OF TIMING INFORMATION

As a hardware design becomes increasingly detailed, increasingly accurate and detailed timing information becomes available either from simulation results or an analysis of the actual hardware. Because these values are usually not extracted from the VHDL model, it is often desirable to update the VHDL model with this more accurate timing information. This process is referred to as “back annotation”. This updated timing information can be incorporated into the VHDL model using any of the mechanisms that handle timing information discussed in earlier paragraphs.

A timing package can be produced that includes the new timing data. If the timing information is represented as a deferred constant, only the package body needs to be modified and reanalyzed. The new timing information takes effect the next time the model is elaborated. Alternatively, a file is constructed that contains the timing information needed by the model. This information is read into the simulation as required. This method has the advantage of not requiring that the model be reanalyzed or reelaborated when the timing information changes.

5-6 USE OF STRUCTURAL HIERARCHY IN BEHAVIORAL MODELS

As described in subpar. 5-2.2.1, VHDL provides functions and procedures as methods for the functional decomposition of behavioral models. As shown in subpar. 2-3.2 and illustrated in Fig. 2-3, the calling structure of functions and procedures defines a hierarchy for behavioral models.

VHDL also provides two ways to create hierarchies using structural decomposition:

1. Structural decomposition through the use of component instantiations in architecture bodies, which may in turn have architecture bodies that use component instantiations
2. Nested blocks, which, in concert with guard signals, let designers decompose and isolate specific behaviors to specific blocks.

The VHDL DID (Ref. 1) requires that the “structural decomposition of behavioral bodies shall be used only when necessary to show functional partitions which are not clear from the partitions of the corresponding structural body”. The DID discourages the use of structural decomposition in behavioral bodies in order to

1. Reduce the implementation bias in a behavioral model
2. Encourage delivery of behavioral models that simulate quickly
3. Prevent the delivery of a gate-level structural model to fulfill the requirement for a behavioral model.

The VHDL DID does not prohibit the use of structural decomposition. The use of structural decomposition is negotiable and is an important opportunity to tailor the DID. The issue of structural decomposition in behavioral models is directly related to the issue of specifying the VHDL modules that are delivered. Each VHDL module should be delivered with a behavioral model, a structural model, and a test bench. If the behavioral model of a VHDL module includes multiple design entities, structural decomposition has been used in it.

The use of structural decomposition where the decomposition is implementation dependent is discouraged. The VHDL DID cites the example of a processor that is implemented from bit-slice components, and the structural model has a design entity that represents a bit slice, and the behavioral architecture body has separate component instances for each individual slice that makes up the processor. This is an implementation-dependent decomposition because a different implementation that does not use bit-slice components would not have the same set of components, and this decomposition does not help the reader understand the functional partitioning of the processor, i.e., the instructions of the processor. On the other hand, partitioning of the architecture into a fixed point processor and a floating point coprocessor does assist the reader in understanding the functional partitioning of the processor and therefore might be acceptable to the Government.

REFERENCES

1. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, Department of Defense, Washington, DC, 11 May 1989.
2. Ram Gummaddi, *Methodology for Structured VHDL Model Development*, Master's Thesis, Bradley Department of Electrical Engineering, Virginia Polytechnical and State University, Blacksburg, VA, April 1995.
3. MIL-STD-1750A, *Sixteen-Bit Computer Instruction Set Architecture*, 15 December 1989.
4. P. J. Hayes and A. M. Andrews, "Multiprocessor Performance Modeling With ADAS", *Proceedings of AIAA Computers in Aerospace VII Conference*, pp. 335-40, Burlingame, CA, October 1989, American Institute of Aeronautics and Astronautics, Washington, DC.
5. IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, March 1988.
6. IEEE Std 1076.3 (Draft), *IEEE Standard for VHDL Language Synthesis Package*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, September 1995.
7. IEEE Std 1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1993.
8. IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary Scan Architecture*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, © May 1990.
9. P. M. Campbell, M. Vai, and Z. Navabi, "Implementation of the IEEE Std 1149.1-1990 in VHDL", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring VIUF*, pp. 151-60, Scottsdale, AZ, May 1992, VHDL International, Santa Clara, CA.
10. ANSI/IEEE Std 1076-1993, *IEEE Standard VHDL Language Reference Manua*l, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, April 1994.
11. J. Roy and R. Vemuri, "DSS: A Distributed Synthesis System for VHDL Specifications", *Using VHDL for Electronic Product Design*, Proceedings of the VHDL Users' Group, Spring 1991 Conference, Cincinnati, OH, April 1991, VHDL International, Santa Clara, CA.
12. R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.
13. IEEE Std 1076.4-1995, *IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, December 1995.
14. L. Feingold, *F-22 Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) Model Specification*, Document No. 5PTA3009, General Dynamics Corporation, San Diego, CA, March 1992.
15. J. Hanna, Rome Laboratory, Griffiss Air Force Base, Rome, NY, Private communication, May 1992.
16. EIA 567-A, *VHDL Hardware Component Modeling and Interface Standard*, Electronic Industries Association, Washington, DC, May 1994.
17. P. Menchini, *Top-Down Design With VHDL*, Tutorial, First Annual Rapid Prototyping of Application-Specific Signal Processors (RASSP) Conference, Arlington, VA, August 1994, ARPA Electronic Systems Technology Office, Arlington, VA.
18. Rome Laboratories/ERDD, *VHDL Model Verification and Acceptance Procedure*, Technical Report, Department of the Air Force, Griffiss Air Force Base, Rome, NY, March 1992.

BIBLIOGRAPHY

R. E. Anderson, A. Coppola, J. S. Freedman, and M. A. Perkowski, "VHDL Synthesis of Concurrent State Machines to a Programmable Logic Device", *Using VHDL in System Design, Test, and Manufacturing*, Proceedings of the Spring 1992 VHDL International Users' Forum, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

J. R. Armstrong and F. G. Gray, *Structured Logic Design Using VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

J. Bhasker, *A VHDL Synthesis Primer*, Star Galaxy Publishing, Allentown, PA, 1996.

S. Carlson, *Introduction to HDL-Based Design Using VHDL*, Synopsis, Inc., Mountain View, CA.

M. Cohen, "Graphical Behavior Capture to VHDL", *Using VHDL in System Design, Test, and Manufacturing*, Proceedings of the Spring 1992 VHDL International Users' Forum, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

A. Dewey, *Analysis and Design of Digital Systems With VHDL*, Addison-Wesley, Piscataway, NJ, 1992.

R. Lipsett, C. Schaefer, and C. Ussery, *VHDL Modeling for Digital Design Synthesis*, Kluwer Academic Publishers, Norwell, MA, 1989.

Y. Hsu, K. F. Tsai, J. T. Liu, and E. S. Lin, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1995.

R. A. MacDonald and R. Waxman, “Operational Specification of the SINCGARS Radio in VHDL”, *Proceedings of the Tactical Communications Conference, Vol. 1, Tactical Communications: Challenges of the 1990s,* pp. 415-33, 1990, Piscataway, NJ, The Institute of Electrical and Electronics Engineers, Inc., New York, NY.

Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill Book Company, Inc., New York, NY, 1993.

M. S. Romdhane, V. K. Madisetti, and J. W. Hines, *Quick-Turnaround ASIC Design in VHDL: Core-Based Behavioral Synthesis*, Kluwer Academic Publishers, Norwell, MA, 1996.

A. Sama and J. Armstrong, “Behavioral Modeling of RF Systems With VHDL”, *Using VHDL for Electronic Product Design*, Proceedings of the Spring 1991 VHDL Users’ Group Meeting, Cincinnati, OH, 1991, VHDL International Users’ Forum, c/o Conference Management Services, Menlo Park, CA.

J. Schoen, *Performance and Fault Modeling With VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

Enabling Design Creativity, Proceedings of the VHDL Fall 1991 International Users’ Forum, Newport Beach, CA, October 1991, VHDL International Users’ Forum, c/o Conference Management Services, Menlo Park, CA.

Using VHDL in System Design, Test, and Manufacturing, Proceedings of the Spring 1992 VHDL International Users’ Forum, Scottsdale, AZ, May 1992, VHDL International Users’ Forum, c/o Conference Management Services, Menlo Park, CA.

VHDL Boot Camp, Proceedings of the Fall 1993 VHDL International Users’ Forum, San Jose, CA, October 1993, VHDL International Users’ Forum, c/o Conference Management Services, Menlo Park, CA.

VHDL: Windows of Opportunity, Proceedings of the VHDL Users’ Group Fall 1990 Meeting, Oakland, CA, October 1990, VHDL International Users’ Forum, c/o Conference Management Services, Menlo Park, CA.

Using VHDL for Electronic Product Design, Proceedings of the VHDL Users’ Group Spring 1991 Meeting, Cincinnati, OH, April 1991, VHDL International Users’ Forum, c/o Conference Management Services, Menlo Park, CA.

CHAPTER 6
CONSTRUCTION OF STRUCTURAL VHDL MODELS

This chapter discusses common approaches to creating structural models, VHDL DID requirements on structural models, timing specifications for detailed gate-level structural models, and annotation of structural models based on the physical measurements of existing hardware, on switch level or analog analysis, or on simulation of a component design. Common techniques used to create structural VHDL models, including automatic synthesis and schematic capture, are discussed. Applications of structural models for physical design and testability analysis are described. Annotation of structural models with layout and testability information is described.

6-1 INTRODUCTION

Structural models are the preferred mechanism for hierarchical decomposition in very high-speed integrated circuit (VHSIC) hardware description language (VHDL). Structural models allow a design to be partitioned into different physical or functional groupings. As discussed in Chapter 3, structural models are used at any level of abstraction. Both structural and behavioral models are used early in the hardware design cycle (when physical design has not been completed) and after design and fabrication have been completed (when the model provides accurate, machine-readable documentation of the completed design). Furthermore, no structural model is complete without behavioral models as its leaf components. Thus there are both structural and behavioral aspects to any complete VHDL description. The focus of this chapter is on detailed gate-level structural VHDL models used to document existing hardware components. This approach complements the one used in Chapter 5, in which the focus is on abstract VHDL models used to document designs that are in progress. As discussed in pars. 2-5 and 3-4, structural models can support simulation of mixed-abstraction-level models. Detailed structural models support design techniques such as logic synthesis and testability analysis. Structural models also provide a mechanism for reusing VHDL models. Reuse is supported by component instantiation and binding.

6-2 CREATION OF STRUCTURAL VHDL MODELS

The VHDL structural model of a hardware module consists of

1. An interface description, which describes the externally accessible signals, generic constants, and timing requirements of the module
2. Component declarations, which identify the types of components used in the model. Each component may be described by either a behavioral VHDL model or another level of structural model
3. Signal declarations, which name all of the signals that interconnect the components of the module
4. Component instances, in which the ports of the components (each of which corresponds to a pin or a set of pins on the actual hardware component) are tied to the signals that connect the components, tied to external signals declared in the interface, or left open.

Although the components of a structural model can represent abstract functional blocks, the VHDL data item description (DID) (Ref. 1) requires that VHDL structural models represent the physical or logical organization of the hardware. During the early stages of design, different structural models may be generated and evaluated; each of these represents different partitioning of the model. However, when an existing design is documented, the structural decomposition of the VHDL model should match the physical or logical organization of the hardware.

VHDL structural models can be constructed in several ways. They can be developed by manually writing the appropriate VHDL description. This approach is very tedious and error-prone for all but the simplest models. A detailed gate-level VHDL structural model generated in this way should be checked very carefully to assure that it is an accurate representation of the hardware and that it is internally consistent. A common alternative is to use a schematic capture system to create diagrams of interconnected components and then generate a structural VHDL description from the netlist.

Gate-level models can also be created automatically. Logic synthesis tools are commercially available that generate gate-level models from behavioral models written in restricted forms of VHDL.

Another form of generation involves modification of an existing gate-level design by adding built-in test (BIT) circuitry. This approach allows the designer to focus on partitioning the design into testable islands of logic rather than working on the details of integrating BIT components into an existing design.

6-2.1 TRANSLATION OF SCHEMATIC CAPTURE MODELS

Modern computer-aided engineering (CAE) tools support the capture of hierarchical structural models as schematic diagrams and the translation of schematic diagrams into VHDL structural models. A schematic capture tool usually works from a library of primitive elements that serve as the leaf-level modules in the design. Such libraries usually include the basic logic gates and higher level entities. These entities represent standard macrocells used in chip designs.

The library may also contain existing chips for circuit board design. The designer can usually assemble structural models and add them to the library for reuse. Sophisticated schematic capture tools provide icons for the elements of the library and allow the user to define icons for his or her designs.

These schematics can then be translated into VHDL structural models for simulation or for export for other uses. Each of the primitive elements in the schematic capture tool library has an associated structural or behavioral VHDL model that implements the function of the primitive. The netlist for the schematic is converted into a structural VHDL model, and the components of the structural VHDL model are the library elements.

A key issue for schematic capture systems is the choice of signal states and strengths to be used. Most non-VHDL CAE tools use a state/strength model compatible with their particular simulator and analysis tools. Also many tools can export VHDL models that use Institute of Electrical and Electronics Engineers (IEEE) Std 1164 logic values. For the resulting models to be interoperable, schematic capture tools used to build structural VHDL models for delivery to the Government should support at least a subset of IEEE Std 1164 (Ref. 2). If the tool supports only a subset of IEEE Std 1164, a set of type conversion functions should be provided to map the IEEE Std 1164 logic values onto that standard subset. The IEEE standard logic package contains definitions of several subtypes, such as `X01`, `X01Z`, and `UX01`. It also defines type conversion functions for these subtypes. Use of type conversion functions for interoperability is discussed in subpar. 7-2.2.

Although use of schematic capture tools provides greater productivity for engineers generating gate-level VHDL models and eliminates syntax errors in the models, it still requires human interaction to place every instance of a component in the model. Furthermore, these models must be verified against more abstract functional or behavioral models to ensure that the logic does implement the intended function.

6-2.2 SYNTHESIS OF STRUCTURAL MODELS FROM REGISTER-TRANSFER-LEVEL MODELS

Logic synthesis uses abstract VHDL descriptions to generate lower level, functionally equivalent structural descriptions that can be implemented directly as very large-scale integrated (VLSI) circuits. Logic synthesis saves a substantial amount of design time and effort and reduces the risk of design errors introduced through manual translation of an abstract design to a detailed design. Logic synthesis tools are now available in many CAE environments.

The VHDL features in models used for logic synthesis are restricted. These restrictions are tool specific and change as synthesis technology improves. Most synthesis tools accept as input a register-transfer-level model, as described in subpar. 5-2.4. Other restrictions may include limited data types, stereotypical use of processes and other constructs to define finite state machines (FSMs) and registers, or the required use of explicit configuration information. Because subsets vary, the documentation of the particular tool must be consulted for more specific information. Most tools also use comments of special form to guide synthesis. A draft standard (Ref. 3) is emerging that defines a standard set of data types and functions for use by synthesis tools. This standard is discussed in subpar. 5-2.4.1.

6-2.3 SYNTHESIS OF STRUCTURAL MODELS FROM FINITE STATE MACHINES

The finite state machine is another abstract functional hardware representation commonly used to describe behavior. Finite state machines are useful to model control sequencers and communication protocols. FSMs can be used in VHDL at several levels of abstraction from high-level abstract behavioral models to register-transfer models. The complexity of large FSMs can be controlled through the use of hierarchical models (Ref. 4) or through the use of communicating sequential processes (CSPs) (Refs. 5 and 6).

Because the mathematical attributes of FSMs are well understood, they are a natural starting point for logic synthesis. Synthesis tools can take advantage of the mathematical nature of FSMs to produce very compact and fast circuits. Certain forms of VLSI circuits are naturally suited to the implementation of FSMs, such as programmable logic arrays (PLAs).

Some CAE systems provide graphical tools for the definition and simulation of FSMs (Refs. 7 and 8). FSMs are easily translated into VHDL, and many CAE systems perform this translation. CAE vendors are beginning to link tools for the construction, debugging, and simulation of FSMs to tools that synthesize circuit designs from VHDL descriptions of the FSMs. In these integrated tool sets VHDL plays a key role as an intermediate form between the FSM and the circuit layout.

6-2.4 ENHANCEMENT OF GATE-LEVEL MODELS WITH GENERATED STRUCTURE

The use of built-in test circuitry is essential to achieving the testability of both military circuit boards and VLSI circuits. When a test operation is required for a hardware component, normal interconnects are disabled, and the BIT circuitry provides the control and observation of the signals to be tested. Some CAE tools provide a mechanism to augment a logic design BIT circuitry automatically. Thus a designer can focus on the development of a functional design, then partition the design into appropriate islands of logic for testability purposes, and have the additional structure automatically generated. Par. 8-4 describes approaches to BIT and discusses related IEEE standards, and par. 8-5 describes an approach used to enhance structural models with BIT.

An important part of accurately modeling existing hardware is representation of its BIT circuitry. Subpar. 10.2.4 of the VHDL DID (Ref. 1) requires that structural models include the physical implementation accurately enough to permit logic fault modeling and test vector generation. It also requires that structural models represent structures created to support testing and maintenance, such as scan paths. As a result, CAE tools should be chosen that generate the BIT circuitry and include the generated BIT circuitry in the VHDL models produced by the tool.

6-3 VHDL DID ORGANIZATIONAL REQUIREMENTS FOR STRUCTURAL MODELS

6-3.1 HIERARCHICAL ORGANIZATION OF STRUCTURAL MODELS

The VHDL DID (Ref. 1) requires that the structural hierarchy of VHDL modules be "analogous to the physical hierarchy of the hardware being documented". The VHDL DID also states, "One VHDL module shall be defined for the entire system and one for each physical electronic unit (assembly, subassembly, integrated circuit, etc.) of the hardware system. VHDL modules should also be defined for important subsections or groupings of complex physical units (e.g., macrocells of a chip or boards defining a processor).". For this correspondence to be traceable, the VHDL DID requires that the entity interface modeling the hardware component include a component identification based on the part number of the corresponding hardware component. In addition, the ports of the VHDL design entities must correspond to pins or connectors of the physical hardware.

Fig. 6-1 shows a typical physical design hierarchy for an embedded electronic system such as is used by the Army.

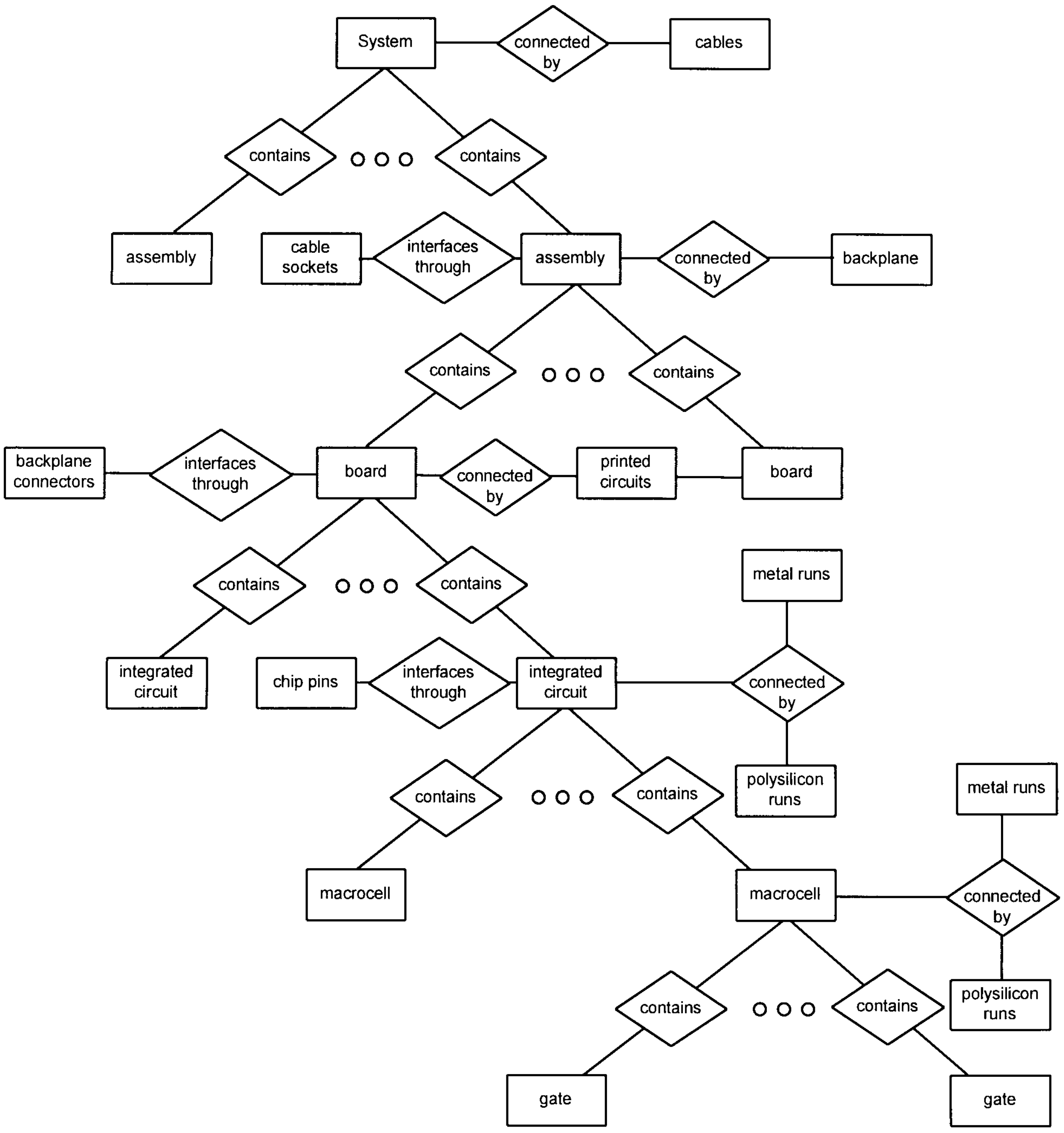


Figure 6-1. Typical Physical Hierarchy of an Embedded Electronic System

The system consists of a number of assemblies that can be removed for repair. The assemblies are connected by cables. A VHDL description of this system written to conform with the VHDL DID includes design entities for the assemblies. The ports of each of these design entities describe cable connections required to connect the assembly to the other assemblies in the system.

Within an assembly are a set of boards that can be removed and either replaced or repaired at a second level of maintenance. The boards are connected by a backplane that is internal to the assembly. A VHDL description of an assembly written to conform to the VHDL DID includes design entities for the boards. The ports of these design entities describe the types of connectors that connect the boards to the backplane.

Attached to the board is a set of chips that can be removed and replaced at a third level of maintenance. The chips are connected by metal traces etched into the boards. A VHDL description of a board written to conform with the VHDL DID includes design entities for each of the chips. The ports of these design entities describe the pins on the chip. For back-annotation purposes each pin may require a separate port. A "shell" design entity may be required in order to convert the connector ports into the wires for the pins. In this entity, signals connect the connector port bit vector to the single-bit ports of the chip model.

The chips may be further partitioned into macrocells connected by runs of metal or polysilicon. A VHDL description of a chip written to conform to the VHDL DID could include design entities describing the macrocells or islands of logic within the chip. At the next level of detail are logic gates, which are the lowest level of detail that can be represented reasonably in VHDL.

6-3.2 ALLOWABLE LEAF-LEVEL MODULES

The VHDL DID specifies the following as leaf-level modules for which no VHDL structural body is required:

1. Government-approved leaf modules
2. Modules that exhibit a stimulus-response behavior but whose internal structure is not properly modeled in a digital format
3. Modules whose detailed design has not yet been completed when the VHDL model is required to be delivered.

These three cases are discussed in the following subparagraphs.

6-3.2.1 Government-Approved Models

The VHDL DID allows VHDL modules selected from a Government list of VHDL modules to be used as leaf-level modules. The DID also requires that the contract include a list of Government-approved leaf-level modules. One mechanism used to approve multiple modules is to approve the use of all VHDL modules in a given model library. Model libraries facilitate the hardware design process by providing reusable, pretested components from which new hardware designs can be built. Many commercially available model libraries exist that provide functionally complete, fully timed simulation models of existing components. Typically, these models have been developed or approved by the manufacturer of the component. An important aspect of tailoring the VHDL DID for a specific program is specifying the models and libraries the contractor can use to develop VHDL descriptions. Also the Defense Electronics Supply Center (DESC) of the Defense Logistics Agency is collecting VHDL descriptions in its VHDL Model Library (See subpar. 4-2.3.).

The use of Government-approved high-level leaf modules serves many purposes. Use of previously developed high-level leaf modules can dramatically reduce the time to build and validate models of existing parts. Use of approved models also eliminates differences in simulation results due solely to differences in the VHDL models of the components. This similarity is particularly important with respect to timing, i.e., differences in the timing from one model to another may change the outcome of system race conditions.

For different models to interoperate they must be written with the same logic-level conventions or have translation routines to convert between the different conventions. Interoperability is an important consideration when the list of Government-approved, leaf-level models is generated. Interoperability issues and approaches are described in par. 7-2.

6-3.2.2 Modules With Stimulus-Response Behavior

Subpar. 10.2.1.1, Item (b) of the VHDL DID (Ref. 1) allows the use of behavioral models for "...a collection of hardware elements which together exhibit a stimulus-response behavior, but whose interaction is best modeled at an electrical or physical level.". The DID gives as examples digital logic gates, analog circuit blocks, and power supplies. Depending upon the complexity of a memory chip (in terms of fault tolerance and testability), high-level models may also be appropriate for random access and read-only memory circuits.

Behavioral models are appropriate to model analog devices, where necessary, because the discrete event approach of VHDL is inappropriate. Research is continuing on integrating analog circuit models into VHDL (Refs. 9 and 10). If this work is successful, the DID may, in the future, require use of VHDL or extensions to VHDL when analog systems or hybrid analog-digital systems are modeled.

6-3.2.3 Modules Without Detailed Designs

An important aspect of the use of VHDL during the design of a new hardware system is documentation of the design during the early stages of the life cycle of the system. VHDL behavioral models can be used to document design requirements and expected performance as a system is being developed. Behavioral models can also be used as simulatable specifications for more detailed designs.

VHDL models may be required at the Preliminary Design Review (PDR) and the Critical Design Review (CDR) as simulatable documentation of the design. For these early milestones the Government may want to specify that behavioral models are acceptable leaf modules, even though they do not support the logic-level fault modeling or automatic test vector generation. This approach encourages top-down design by the contractors and gives the Government simulatable documentation of a design as the work progresses.

For example, a program is developing a multiprocessor architecture using off-the-shelf 1750A processors, PIbus interface modules (PIbus BIMs), and high-speed data bus interface modules (HSDB BIMs), and a to-be-developed special-purpose signal processor (SPSP). VHDL models are to be delivered at the PDR, CDR, and Test Readiness Review (TRR). By the time of the PDR the architecture of the multiprocessor to the level of the number of busses, the number of data processors, and the number of signal processors should be known. The architecture can be defined with a structural model that uses Government-approved models for the existing components. The model for the SPSP delivered at the PDR is a model for a part without a detailed design, i.e., a high-level behavioral model. The model includes the interface to the rest of the system and communicates with the rest of the system through the detailed bus and BIM models. This level of model is appropriate for interface simulation, which is an appropriate verification step for the PDR. At this stage the VHDL model of the SPSP may also include some high-level timing estimates for critical functions and thus could be used as evidence that the resulting multiprocessor system will meet its timing requirements, at least for some critical subset of the system applications. At this point the entire SPSP may be represented by a single behavioral body. During the design process, changes in the number of components and the network topology must be reflected in the structural model of the multiprocessor.

The model delivered at the CDR extends the PDR model. The SPSP model should be extended to an instruction set architecture (ISA) or register-transfer-level (RTL) model. This VHDL model can be used for software design and verification; therefore, software and hardware design can continue in parallel. The CDR model allows more accurate timing analysis than the earlier version and supports complete functional verification, particularly if the entire instruction set of the SPSP is modeled. At this point the VHDL model of the SPSP should be extended to provide some internal structure. The VHDL model delivered at the start of fabrication should be a register-transfer model suitable for synthesis of the SPSP logic design.

The results of simulating both the CDR VHDL model and the PDR VHDL model on the same test sets should be compared to verify that the CDR VHDL model is a correct implementation of the PDR VHDL model. In practice, there may be so many design changes between the two reviews that comparisons may be very difficult to make. For example, refined area estimates for system ASICs may force a new partitioning of hardware, or they may force a change in algorithms. Either of these changes could make comparisons between the models difficult. However, the CDR VHDL model should be simulated with the same test sets used to simulate the PDR VHDL model. Also the contract should specify which VHDL models are to be maintained throughout the life of the project. If the PDR VHDL model is selected to be maintained, changes in the design should be reflected in both the PDR VHDL model and the CDR VHDL model so comparisons between the models should be straightforward. This technique is regression testing and is very valuable in ensuring that later levels of design do not introduce new problems into the design.

The model delivered for the TRR reflects the detailed gate-level design of the SPSP. The Test Readiness Review verifies that the model is complete and detailed enough to support analysis of test vectors. The model is now complete except possibly for timing information. Timing information should be provided through analysis and testing of the actual hardware and should be accumulated during the integration of the system.

6-3.3 VHDL DID ANNOTATION REQUIREMENTS FOR STRUCTURAL MODELS

The VHDL DID (Ref. 1) requires that structural models be annotated for three reasons:

1. To provide traceability between the physical hardware and the VHDL model (DID subpar. 10.2.2.4)
2. To capture timing and electrical requirements for the hardware in the model (DID subpar. 10.2.2.2)
3. To capture the acceptable operating conditions of the system (DID subpar. 10.2.2.3).

Traceability ensures that the VHDL model accurately documents the actual hardware. Without traceability it is difficult to use the VHDL model to evaluate possible upgrades or changes to the system because it is difficult to relate those components of the hardware to the corresponding components of the VHDL model. Similarly, traceability allows analysis of simulation results (such as the utilization of design entities) to be related back to the hardware components.

Timing and electrical requirements document the acceptable range of timing and electrical parameters, e.g., clock frequency and pin voltage levels, for the components of the hardware system. According to subpar. 10.2.2.2 of the VHDL DID (Ref. 1), these documented ranges must interact with the simulation in the sense that if during a simulation the operating conditions of a component go outside the acceptable range, an error message is generated. The operating conditions interact with simulation in that operating condition parameters are used to calculate timing values for the components.

A mechanism useful to organizing the annotation of detailed models of physical components is an electronic data sheet (EDS). This is the approach taken by the Electronic Industries Association (EIA) in EIA-567 (Ref. 11). The electronic data sheet consists of several views of a hardware system. A view of a hardware module is a set of logically related data representing the significant characteristics of the module within the scope of the data. The EIA approach fully documents the relationships among the physical design, the electrical characteristics, and the system timing directly in the VHDL model. These relationships are captured in VHDL and in three interdependent views in the EDS for any hardware module: a physical view, an electrical view, and a timing view.

The VHDL initiative toward ASIC libraries (VITAL) standard (Ref. 12) uses a different approach in which the timing information is generated by external tools in the form of a timing file that is used to generate generics. This approach is described in subpar. 6-6.1.

These two standards are being made compatible. They both use generics to put the timing information into the models. Different configuration declarations can be used to define the values for generics. An EIA configuration declaration references the information in the EDS packages; a VITAL configuration declaration is generated using data in an standard delay format (SDF) file.

The purpose of the EDS is to capture information traditionally supplied by a manufacturer but in a manner that is more easily created, verified, and used. Data missing from the manufacturer's data sheet should be calculated and inserted into the EDS and then annotated to distinguish it from data supplied by the manufacturer. Equations used to calculate the missing data must be included in the package.

In the EIA approach the three views of an EDS are represented as a collection of VHDL packages. Each view has a primary package containing declarations of data characterizing the view. These packages are used throughout a VHDL model. There may be other packages in a view that are specific to a technology but used by all models using that technology. Technology-specific packages are used particularly in the timing view.

There are specific packages for each component in the VHDL design library. These packages are used, for example, in the physical view to provide traceability between the VHDL component and the corresponding physical component.

6-3.3.1 Physical View Requirements

In the EIA approach to defining an EDS, each VHDL module has a collection of constants describing the physical view of the system. Fig. 6-2 shows the hierarchical organization of constants characterizing the physical view of a VHDL module and providing traceable linkages between a VHDL model and its corresponding hardware component and interconnections. As shown in Fig. 6-2, the constants are divided into two categories: the electrostatic discharge (ESD) limit and the pin to signal mapping. The `ESD_LIMIT` constant has the type `VOLTAGE`. Component identification is handled through entity-naming conventions and through header comments. The pin to signal mapping information is defined through two data structures: an enumerated type `PIN_LIST_PV`, which lists the pins, and an array of records `PIN_TO_SIGNAL_RECORDS`, which is indexed by `PINT_LIST_PV`. Each element of `PIN_TO_SIGNAL_RECORDS` is a record containing two strings: one containing the name of the pin and one containing the name of the signal. These strings are both deferred constants, so if different packaging options exist for the component, different package bodies can be used to define the mapping.

This physical view information is described in a VHDL model using one package for the entire model and another package for each component.

Pin-out constants are associated with the port declarations in the entity declaration. These constants are sufficient to satisfy the VHDL DID requirements in subpar. 10.2.2.1 that the VHDL entity declaration port declarations "...shall include information which relates each input or output port to a package pin number or connector pin number whenever such a correspondence exists.".

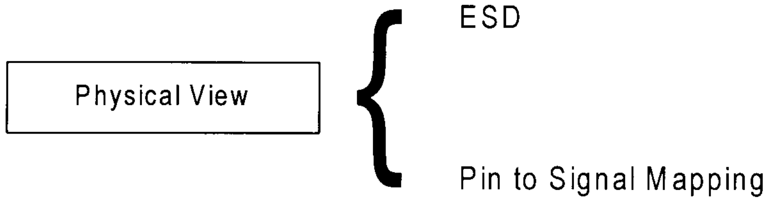
The package defining the timing view may depend upon the packages defining the electrical and the physical views. The combination of electrical, timing, and physical views constitutes the electronic data sheet for the physical component. The VHDL structural model then uses these constants to define the timing and error handling characteristics of the models.

6-3.3.2 Electrical View Requirements

As shown in Fig. 6-3, the electrical view of a component consists of two parts:

1. The signal characteristics, which characterize each input port of the component by its input threshold voltages and leakage currents, each output port by its output drive voltage and current and alternating current (AC) test load, and all ports by their capacitive loads

2. The power characteristics, which describe the maximum and minimum operating voltages and the maximum



Reprinted with permission. Copyright © by Electronic Industries Association.

Figure 6-2. EIA 567 Physical View Organization (Ref. 11)

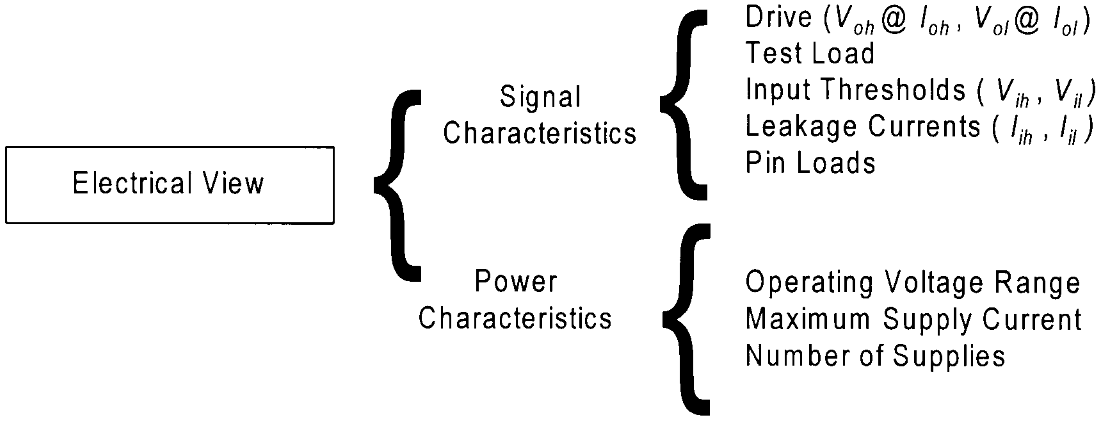


Figure 6-3. EIA 567 Electrical View Organization (Ref. 11)

power supply current for each power pin of the component.

The drive capabilities of each output pin are described in terms of two pairs, each consisting of a voltage and a current. The first pair V_{oh} and I_{oh} is the voltage and current generated when the output port is sustaining a high signal value. The second pair V_{ol} and I_{ol} is the voltage and current generated when the output port is sustaining a low signal value.

Similarly, the input pin threshold voltages and leakage currents are specified in terms of two pairs, each also consisting of a voltage and a current. The first pair V_{ih} and I_{ih} is the threshold voltage and the leakage current received when the input pin is presented with a high signal value. The second pair V_{il} and I_{il} is the threshold voltage and the leakage current received when the input pin is presented with a low signal value. The pin load is used to calculate the net dependent load due to the number of receivers and drivers.

6-3.3.3 Timing View Requirements

As shown in Fig. 6.4, the EIA timing view for a component consists of two parts: a set of timing constraints that are defined for each input pin and a set of parameters that defines both internal and external delays. The external delays are further subdivided into input wire and output load delays.

The constraints are used to generate timing error messages and actions. According to the EIA guidelines, each input pin should have a subset of the four timing constraints defined. If there is no additional signal that acts as a clocking signal, then asynchronous timing constraints are specified. As described in subpar. 5-4.8, the EIA pulse width and cycle time define how long a signal can stay at a particular signal state. If that time is exceeded, the VHDL model should generate an error message, and if a timing flag has been set, the

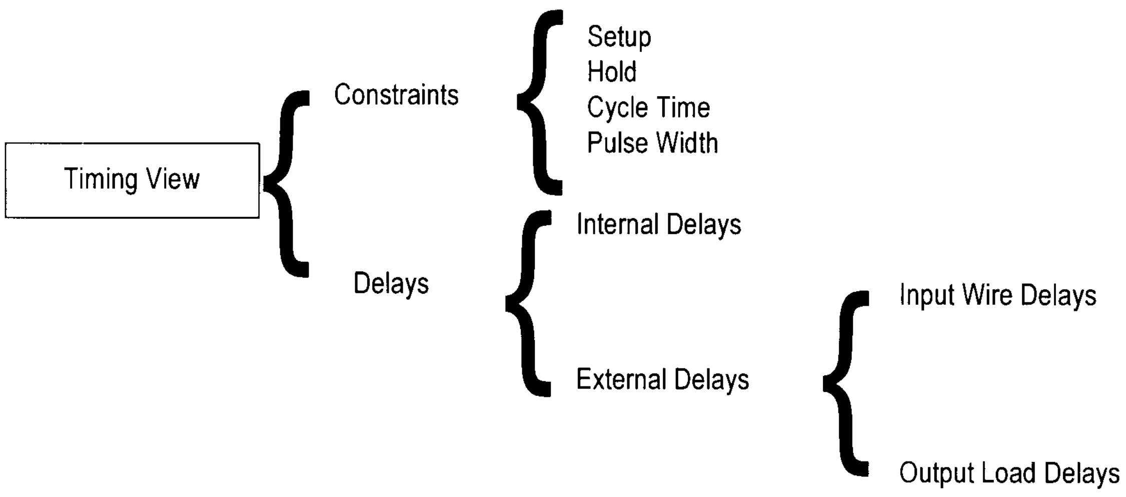


Figure 6-4. EIA Timing View Organization (Ref. 11)

signal should be set to 'X'. If there is a clocking signal associated with the input pin, the setup and hold times for the input pin with respect to the clocking signal should be specified as constraints, as described in subpar. 5-4.9. Again, if a setup or hold timing constraint is violated, the VHDL model should generate an error message, and if a timing flag has been set, the appropriate signal should be set to 'X'. Timing flags are described in subpar. 7-6.3.

The timing delay parameters are used to capture internal pin-to-pin delays, input wire delays, and output capacitance loading delays.

Another approach to documenting the timing behavior of a VHDL model is provided by the VITAL specification (Ref. 12). A VITAL-compliant model uses generics to document the timing data and a set of VITAL timing functions and model primitives to implement the timing behavior. The actual timing data can be imported into the model from an external file, which complies with the VITAL standard delay file format.

The VITAL specification provides a detailed procedure used to develop models suitable for hardware acceleration and back annotation of timing information. VITAL uses generic parameters to pass timing information into the VHDL model. VITAL-compliant models perform no environmentally dependent delay calculations themselves. All such delay information is calculated outside the model and passed into the model using an SDF file.

The timing information in the SDF file is used to set the generic parameter values prior to simulation. The simulation environment is responsible for inserting the SDF timing information into the model.

VITAL-compliant models require strict adherence to naming conventions, model organization, and use of the VITAL primitive library. The advantage of using VITAL is that compliant models produce the same result regardless of the simulator on which they are executed, particularly if the simulator has hardware acceleration. Gate-level models to be used for final timing verification should be VITAL compliant.

The timing behavior can be documented either by producing a VHDL source code model that has been back annotated from the SDF file or by including the SDF file itself as part of the documentation package. A self-contained VHDL model has the benefit of not requiring external files and, therefore, the resulting configuration management issues of linking versions of external files with versions of design entities using those files. Including the SDF file and the default generic VHDL model has the advantage of easier incorporation of different timing behaviors into the same basic model.

Fig. 6-5 illustrates the two alternative approaches used in VITAL-compliant architecture bodies. The VITAL Level 1 architecture uses either a VITAL process model or a VITAL primitive concurrent procedure call approach. Both approaches use a wire delay block and a negative constraint block. The VITAL process is a timing shell approach which separates the function (in the functionality section) from the timing (in the path delay section). The VITAL primitive concurrent procedure call approach allows the delays to be distributed across multiple components of the model.

A detailed structural model constructed in compliance with the VITAL specification primarily documents the detailed internal timing of the component. It does not provide a simpler external “data sheet” view of the timing requirements of the interface. However, a data sheet view of timing could be developed from a VITAL-compliant timing model.

A model that documents the low-level behavior of an ASIC will most likely use the VITAL approach to timing. Such models benefit from hardware acceleration and simulator optimization because they are usually complex and are used for final timing verification prior to fabrication. VITAL was developed specifically to address the acceleration and verification needs of ASIC designs.

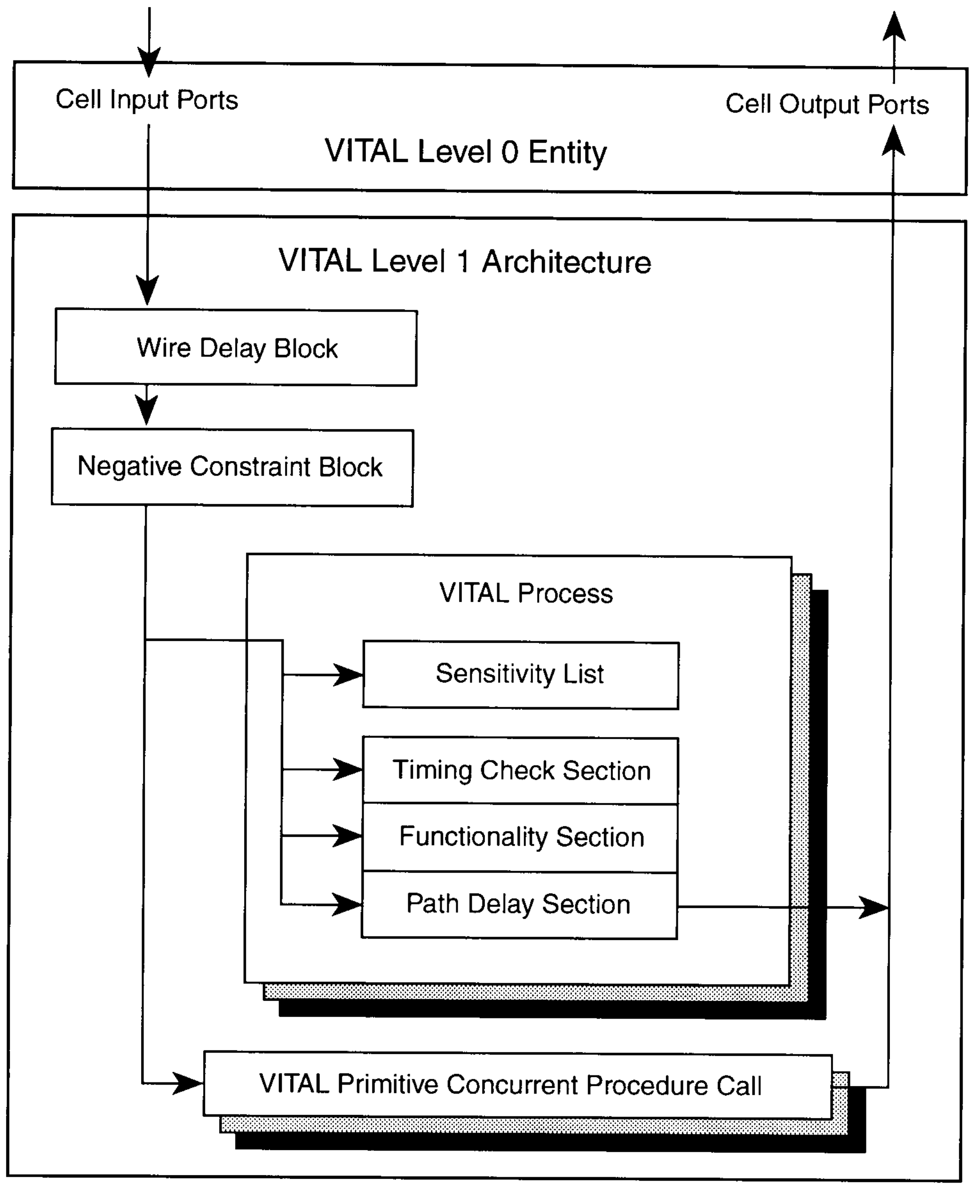


Figure 6-5. VITAL Model Organization (Ref. 12)

6-4 VHDL DID SIMULATION REQUIREMENTS FOR STRUCTURAL MODELS

The VHDL DID specifies the level of fidelity required in gate-level structural models in terms of two requirements on the functionality of the models: models must support logic-level fault modeling and test vector generation. These requirements, combined with the requirement that the structural model accurately represents the physical hardware, both in its hierarchy of components and in its interconnections, specify a level of fidelity that is required in the model. These requirements are driven by a need to support the maintenance of hardware through the development of diagnostic aids such as test vectors.

6-4.1 SUPPORT FOR LOGIC-LEVEL FAULT MODELING

The VHDL DID requires gate-level structural models to support logic-level fault modeling. The common approach to logic-level fault modeling inserts faults into the VHDL model of the circuit. These inserted faults represent failures in the gates or their interconnections. For fault modeling to represent physical faults effectively in particular interconnects of a device, there must be a one-to-one correspondence between the internal signals in the VHDL model and the physical wires on the modeled printed circuit board or the polysilicon or metal run in a VLSI circuit.

6-4.2 SUPPORT FOR TEST VECTOR GENERATION

The VHDL DID requires gate-level structural models to support test vector generation. Typically, a test vector generation tool either works with a static representation of the circuit and creates test vectors from the structure of the circuit or the test vector generator uses fault simulation to grade the test vectors and assure that the test vector set is complementary (i.e., each test vector detects at least one fault not detected by any others) and complete (i.e., the test vectors detect a sufficient percentage of the faults that have been modeled). In either case the tool must have a detailed and accurate model of the physical interconnection of gate-level primitives.

To make use of test vectors generated using a VHDL structural model, the test vectors should be compatible with Waveform and Vector Exchange Specification (WAVES) (Ref. 13). Test vectors generated in the WAVES format can be used directly by WAVES-compatible automatic test equipment (ATE).

Test vectors are an important part of the documentation of a hardware component. The VHDL DID requires test benches for each VHDL module. One of the essential components of a test bench is a collection of test vectors that apply stimuli to the circuit and define the expected response.

6-5 TIMING SPECIFICATIONS FOR STRUCTURAL MODELS

The EIA approach to timing specifications allows three types of delays in a model: input wire delay, output load delay, and internal (pin-to-pin) delays. The input wire delay and the output load delay are functions of the component layout. Input wire delay is implemented using VHDL transport delays. An input wire delay is associated with each input port. Output load delays are associated with `output`, `buffer`, and `inout` ports. Internal pin-to-pin delays are defined in terms of the transitions in the state of output signals. Table 6-1 shows the names of the constants for the different possible transitions between signal strengths. The values of these timing constants may vary among various semiconductor fabrication technologies. In some cases a specific port will not support particular transitions; thus a subset of all of these constants would be applied.

Fig. 6-6 shows a VHDL implementation of a primitive OR function that uses the input timing delays. Each input port has its own process and a corresponding internal signal. The process delays the input according to an input wire delay function, which is based on the current and new values of the input signal. The function computing this delay uses atable lookup scheme to provide the delay values; the tables are defined in the package body of the package containing the function. As a result, it is easy to substitute different package bodies that provide different times without reanalyzing the entire model. This approach has the problem that the delays are not specific to the circuit design, but all signals using the same technology get the same delay. An alternative approach uses generics to describe the delays and has the local processes select the appropriate generic delay using a sequence of if statements (Ref. 14). This approach is well suited to back annotation.

TABLE 6-1. INTERNAL (PIN-TO-PIN) DELAY SPECIFICATIONS (Ref. 11)

TO / FROM	'U'	'X'	'0'	'1'	'Z'
'U'	NA*	NA	txl	txh	txz
'X'	NA	NA	txl	txh	txz
'0'	NA	tlx	NA	tlh	tlz
'1'	NA	thx	thl	NA	thz
'Z'	NA	tzx	tzl	tzh	NA

*NA = not applicable

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY timing;
USE timing.in_wire_delay;
USE timing.internal_delay;
ENTITY ORXT IS
   PORT (IN1: IN  std_ulogic;
         IN2: IN  std_ulogic;
         OUT: OUT std_ulogic);
END ORXT;

ARCHITECTURE Externally_Timed OF ORXT IS
   SIGNAL Internal_IN1: std_ulogic;
   SIGNAL Internal_IN2: std_ulogic;
BEGIN
   PROCESS (IN1)
      VARIABLE old_IN1: std_ulogic;
   BEGIN
      Internal_In1 <= transport IN1 AFTER in_wire_delay(old_IN1,IN1);
      old_IN1 := IN1;
   END PROCESS;
   PROCESS (IN2)
      VARIABLE old_IN2: std_ulogic;
   BEGIN
      Internal_In2 <= transport IN2 AFTER in_wire_delay(old_IN2,IN2);
      old_IN2 := IN2;
   END PROCESS;
   PROCESS (Internal_In1, Internal_In2)
      VARIABLE new_OUT std_ulogic;
      VARIABLE old_OUT std_ulogic;
   BEGIN
      new_OUT := Internal_In1 OR Internal_IN2;
      OUT <= transport new_OUT AFTER internal_delay(old_OUT, new_OUT);
      old_OUT := new_OUT;
   END PROCESS;
END Externally_Timed;
```

Figure 6-6. Extrinsic Timing Delay VHDL Model

6-6 BACK ANNOTATION OF STRUCTURAL MODELS

Detailed gate-level models are natural targets for back annotation. The extraction of a netlist from a gate-level VHDL model is straightforward. The analysis of the netlist and calculation of timing and electrical information from the netlist, perhaps augmented with layout information, is a common capability in modern CAE tools.

The primary emphasis of back annotation is on providing more accurate timing and electrical information. The focus for electrical information has been on the computation of capacitive loads on the output ports of components. This information is then used to compute timing information.

6-6.1 BACK ANNOTATION OF TIMING INFORMATION

Back annotation of timing information is one of the most common ways to provide detailed timing for models, and it can take advantage of the many timing analysis capabilities of modern CAE tools.

Two methods of back annotation have been used for VHDL models: external input files (Ref. 15) and generation of configuration declaration combined with generics (Ref. 14). The VITAL specification (Ref. 12) uses an external file, the standard delay file, to provide timing data generated externally to the model. VITAL-compliant models provide a rigid naming convention for timing-related generics in the model. This convention allows the information in the SDF file to be properly associated with the corresponding generic.

6-6.2 BACK ANNOTATION OF LAYOUT INFORMATION

Back annotation of layout information can be particularly important when VHDL models are used as the interface between high-level synthesis tools and layout tools (Ref. 16). Typical layout information for VLSI circuits includes lengths of runs, numbers of metal and polysilicon levels, and area requirements. Similar information, such as the number of levels of printed circuits, the number of chips on the board, and the required board size, is used for printed circuit board models. The layout tool can annotate the VHDL model with layout information. Synthesis tools can then use this information. The designer can explore different ways of expressing the behavior in VHDL, which may result in different synthesized models.

6-6.3 BACK ANNOTATION OF TESTABILITY INFORMATION

Back annotation can be used to support testability analysis. This form of back annotation may be useful for the design and optimization of the built-in test capabilities of a hardware system. The appropriate metrics and BIT techniques are discussed in par. 8-3 of this handbook. As part of the work on WAVES, a general fault dictionary language is being developed (Ref. 17). This language can be translated into annotations for VHDL models. With this fault dictionary defined, individual signals can be labeled with the appropriate set of tests.

REFERENCES

1. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, Department of Defense, Washington, DC, 11 May 1989.
2. IEEE Std 1164-1991, *IEEE Standard Logic Package*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, October 1991.
3. IEEE Std 1076.3 (Draft), *IEEE Standard for VHDL Language Synthesis Package*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, September 1995.
4. D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming **8**, 231-74 (August 1987).
5. C. Hoare, "Communicating Sequential Processes", Comm. of ACM 12 **10**, 666-77 (August 1978).
6. P. M. Campbell, M. Vai, and Z. Navabi, "Implementation of IEEE Std 1149.1-1990 in VHDL", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
7. P. Clemente, P. Runstadler, L. Specter, and K. Walsh, "From Statecharts to Hardware FPGA and ASIC Synthesis", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, Scottsdale, AZ, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
8. M. Cohen, "Graphical Behavior Capture to VHDL", *Using VHDL in Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
9. H. Tahawy, D. Rouquier, and D. Rodriguez, "Toward Analog-VHDL: Some of the Problems for Mixed Simulation", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
10. J. Dube and Z. Navabi, "Behavioral VHDL Transistor Models", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
11. EIA 567-A, *VHDL Hardware Component Modeling and Interface Standard*, Electronic Industries Association, Washington, DC, March 1994.
12. IEEE Std 1076.4-1995, *IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, © December 1995.
13. IEEE Std 1029.1-1991, *Waveform and Vector Exchange Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1991.
14. S. Turner, "Back Annotation for VHDL Structural Models", *VHDL Windows of Opportunity: Proceedings of Fall 1990 VHDL Users' Group Forum*, Oakland, CA, October 1990, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
15. V. Berman and C. Ussery, "A Proposed Back Annotation File Format for VHDL", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
16. K. Kumar, M. Tovey, S. Sawant, and P. George, "Using VHDL Beyond Synthesis", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

17. K. J. Parella and A. Wilmot, "Fault Detection and Localization", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring 1992 VHDL International Users' Forum*, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

BIBLIOGRAPHY

J. R. Armstrong, *Chip-Level Modeling With VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

J. R. Armstrong and F. Gail Gray, *Structured Logic Design With VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

J. Hallenbeck, J. Cybrynski, N. Kanopoulos, T. Markas, and N. Vasanthavada, "The Test Engineer's Assistant, A Support Environment for Hardware Design for Testability", *IEEE Design & Test of Computers*, IEEE Computer Society Press, Los Alamitos, CA, April 1989.

R. E. Harr and A. G. Stanculescu, Eds., *Applications of VHDL to Circuit Design*, Kluwer Academic Publishers, Norwell, MA, 1991.

IEEE Std 1149.1-1990, *IEEE Standard Test Access Port and Boundary Scan Architecture*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1990.

S. S. Leung and M. A. Shanblatt, *ASIC System Design With VHDL: A Paradigm*, Kluwer Academic Publishers, Norwell, MA, 1989.

O. Levia and F. Abramson, "ASIC Sign-Off in VHDL", *VHDL Boot Camp*, Proceedings of the Fall VIUF, San Jose, CA, October 1993, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.

Z. Navabi, S. Day, and M. Massoumi, "Investigating Back Annotation of Timing Information into Data Flow Descriptions", *Using VHDL in System Design, Test, and Manufacturing: Proceedings of the Spring VHDL International Users' Forum*, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

A. Rushton, *VHDL for Logic Synthesis*, McGraw-Hill Book Co., Inc., New York, NY, 1994.

CHAPTER 7
PREPARATION OF VHDL MODELS FOR SIMULATION

In this chapter the preparation of VHDL models for simulation is described, as is the process of configuring a model from libraries of component descriptions. Emphasized are techniques that support the interoperability of models in component libraries so they can be combined freely to provide mixed-abstraction-level models. The development of test benches and test vectors to check the correctness and completeness of the model are discussed. Also discussed are the use of parameterized timing models and the selection of timing options for simulation.

7-1 INTRODUCTION

A key advantage of documenting hardware with very high-speed integrated circuit (VHSIC) hardware design language (VHDL) is that VHDL models of hardware systems can be simulated. Previous chapters have discussed model development; the emphasis of this chapter is on preparation of VHDL models for simulation. The target audience of this chapter is the user of a VHDL model, i.e., the person responsible for verifying, validating, and using the model to support decision making. This chapter discusses five aspects of the preparation of VHDL models for simulation:

1. The assembly and integration of a complete test bench, including the test bench components and the unit under test (UUT). This aspect includes the steps necessary to ensure the interoperability (par. 7-2) of all of the components of the test bench, particularly the UUT.
2. The development of test benches (par. 7-3) that provide the stimulation for the UUT and check to ensure that the results produced by the UUT are correct
3. The development of test vectors (par. 7-4), which are the stimulation data for the UUT and also may specify the correct result values
4. The configuration of a complete test bench (par. 7-5), including the test bench components and the UUT
5. The definition of simulator options (par. 7-6), which control the execution of the simulation and the trace data generated as a side effect. The choice of simulator options can have a very significant effect on the time required for the simulation, the amount of disk space required for the simulation, and the kind of data available to support decision making after the simulation has run, including decisions about the validity of the model.

The VHDL data item description (DID) (Ref. 1) refers to the required simulation capabilities and constraints that must be considered when preparing a model for simulation. Subpar. 10.2.2 of the DID requires VHDL modules to produce error messages detecting timing and electrical faults. Subpar. 10.2.5 of the DID requires VHDL modules to be accompanied by appropriate test benches. Subpar. 10.2.5.2 of the DID requires test benches to be traceable to test plans for the physical hardware, where possible. Subpar. 10.2.5.3 of the DID requires test benches to be supplied for each VHDL module of the hardware hierarchy.

7-2 INTEROPERABILITY OF MODELS

The preparation of a VHDL model for simulation may involve assembling several component models so that when the assembled model is simulated, it will produce valid outputs. This preparation requires that the component models be interoperable. In this paragraph preventative methods used to ensure interoperability and methods used to combine components that are not directly interoperable are discussed. Interoperability involves ensuring that component models can be connected together through common type definitions for signals connecting components and that components operate in a common semantic environment. Essential to ensuring a common semantic environment is ensuring a consistent timing model for the entire environment.

Two scenarios for the use of VHDL models illustrate this need. In the first scenario a prime contractor for a hardware system has its subcontractors design the components of the system and produce gate-level VHDL models of the components as documentation of the designs. This prime contractor also develops test benches and test vector sets to validate the subcontractors' designs and to ensure that the system works properly as a whole. All of the models developed by the subcontractors must work together correctly. The component models at the same level of abstraction must interoperate, and the best approach to ensuring this interoperability is to use a standard signal definition as described in subpar. 7-2.1. If this is not possible, type conversion functions may be used in some situations to provide interoperability. Use of type conversion functions is described in subpar. 7-2.2.

For the second scenario the Government has developed (or had developed for it) a high-level-of-abstraction test bench and high-level-of-abstraction models of all of the components of an existing hardware system that is going to have some of its components upgraded. This test bench and the models of the components that are not going to be upgraded will be used for functional testing of gate-level designs of the components that are going to be upgraded. This test bench and the high level models of the components that are not being upgraded must be interoperable with the gate-level models of the components being upgraded. This scenario is an example of a situation in which component models at different levels of abstraction must interoperate.

To allow for configuration of mixed-abstraction-level models, the following approach is recommended:

1. Define a standard set of data types for signals at the lowest common level of abstraction needed. Standards for the data types of signals are discussed in subpar. 7-2.1.

2. Define and acquire or implement a set of type conversion functions that allow higher level of abstraction components to communicate with lower level of abstraction components and vice versa. Type conversion functions are discussed in subpar. 7-2.2.

3. Even though the port data types for models of the same component at different levels of abstraction are different, maintain a one-to-one correspondence between the number of ports in models at different levels of abstraction. An example of this problem is discussed in subpar. 7-2.2.

4. When configuring the mixed-abstraction-level models, use the type conversion components in configuration declarations to remap interfaces at different levels of abstraction so that the entire mixed-abstraction-level model can be compiled and simulated without recompiling the component models.

Configuring some mixed-level-of-abstraction models may require more than just the use of type conversion in port maps. For example, embedding a functional model with no timing in a behavioral system model that is modeling the timing may require construction of a timing shell. (Timing shells are described in subpar. 5-4.1.) The concept of a shell may be required to configure a system model when the VHDL component models at different levels of abstraction do not have the same number of ports. For example, if a high-level component model has a single port for the address bus but a gate-level model of the same component has separate ports for each of the bits of the address bus, then a shell will be needed to interface the gate-level model of this component with high-level models of other components.

7-2.1 USE OF STANDARD SIGNAL DATA TYPES

The VHDL language requires that the type of a port be consistent with the type of the signal connected to the port, i.e., all of the ports connected to a signal must be consistent. A minimal requirement for interoperability between two component models is the ability to connect the ports of the two components with signals. Use of standard data types for signals and ports is the most common approach to achieving interoperability of VHDL models at the same level of abstraction. These standard data types are supported by standard semantics as implemented by resolution functions and (for logical types) by definitions of the basic logic functions. The standard for models at the gate level of abstraction is Institute of Electrical and Electronics Engineers (IEEE) Standard 1164, the standard logic package (Ref. 2). The IEEE 1164 standard logic data type uses a VHDL package to encapsulate a data type definition, a resolution function, and several common type conversion functions. The emerging standard for register-transfer-level models is IEEE Std 1076.3 (Ref. 3), the logic synthesis package. These packages provide type conversion functions for related data types and thus support interoperability with other similar models. These standards are described in more detail in Chapters 5 and 6.

7-2.2 TYPE CONVERSION FOR DIFFERENT SIGNAL DATA TYPES

Type conversion functions provide a way to connect VHDL models of components whose ports are not syntactically consistent. Type conversion functions are often needed in mixed-level-of-abstraction models because the data types for signals at different levels of abstraction are usually different.

Packages provide a natural mechanism for collecting type conversion functions. An early step in the top-down design of a computer is definition of the basic data types, e.g., character, integer, floating point, instruction, and address, that are supported by the computer. The definitions of these data types can be formalized in VHDL by creating a package defining the formats of these data types and including the corresponding type conversion functions that convert these data types into bit arrays. The IEEE synthesis package, IEEE Std 1076.3 (Ref. 3), provides a generic set of such definitions including signed, sign-magnitude, and twos-complement formats whose word size is parameterized.

Different types in the same network can be converted by using type conversion functions in the port maps of component instantiation statements or binding indications. This technique is particularly useful for connecting structural models whose components use different signal types. Type conversion functions can also be used for variables in the parameter association lists of subprogram calls. This technique allows a user to assemble high-level behavioral models from subprograms that use different interface types.

To make use of type conversion functions in port-mapping statements, there must be a one-to-one correspondence between the signals and the ports, even though the data types of the signals and the ports are different. One interoperability problem for mixed-level-of-abstraction models is representation of busses at different levels of abstraction. High-level models typically represent an entire bus as a single signal. At the highest level such a bus may resemble a VHDL composite data type with fields for control, address, and data. These formats vary from one bus to another. Conversion routines can be developed to convert the bus data type into an array of standard logic values and to convert from an array of standard logic values back into the bus data type. Fig. 5-10 shows a VHDL package body that includes conversion functions that convert an integer value into an array of bit-level values and back.

If simulating VHDL models in combination with other, independently developed VHDL models is to be worthwhile, each model must process the full range of possible input values including error states. If a model does not process all possible inputs appropriately, the simulation of the whole system will fail, or worse, the results of system simulation will not be accurate. Therefore, if it is necessary to develop type conversion functions, these functions should be tested to ensure that if they aggregate lower level of abstraction data types, they accurately handle cases in which the value of a lower level of abstraction data type is an error state. For example, if one bit of an address field of a bus has an `'X'` value, then what is the aggregate value of the bus? If the definition of the bus includes parity bits, then the existence of an `'X'` value may not propagate through the type conversion. In an algorithmic-level model the parity decoding may not be modeled. If this algorithmic-level model of one component is connected to a register-transfer-level model of a bus interface unit (BIU), the effects of parity decoding would have to be handled by the type conversion function. The purpose of this example is to point out the potential difficulties of error propagation in mixed-level-of-abstraction models and the roles that type conversion functions may have to play in such models, not to encourage the use of type conversion functions to model hardware functions.

7-2.3 INTEROPERABILITY OF TIMING MODELS

Subpar. 10.2.3.2 of the VHDL DID requires that VHDL models provide accurate timing information in the form of signal delays at the output ports of all VHDL entities. This requirement implies that VHDL models must have a common timing framework. If there is a common timing framework, different architecture bodies for components of a structural model can be interchanged and still provide accurate timing information for the structural model.

VHDL includes a predefined set of time units as part of a built-in physical data type, which is discussed in subpar. 3-6.2. VHDL converts time units to a common base so that different time units can be intermixed freely. In high-level models the time delay for an operation may be a complex computation based on parameters such as clock speed, clock cycles per instruction, and word size. If models are being combined into a system model for simulation, it is important to verify that the same parameters are being used in the same way in all the components. For example, are word sizes stated in terms of bits or bytes?

As described in Chapter 6, the VHDL initiative toward application-specific integrated circuit (ASIC) libraries (VITAL) package (Ref. 4) provides a standard representation for timing and back annotation for gate-level models. VITAL is consistent with the IEEE Std 1164 standard logic package (Ref. 2). Electronic Industries Association (EIA) 567 (Ref. 5) includes a VHDL package that describes a standard timing view. This timing view relates the physical characteristics of the hardware and the implementation technology of the hardware to delays for logic functions. This standard is consistent with IEEE Std 1164.

7-2.4 PORTABILITY REQUIREMENTS FOR INTEROPERABLE VHDL MODELS

A critical Government requirement is that VHDL models be portable to different simulation environments. This requirement allows contractors to select the most competitive computer-aided engineering (CAE) system for their needs and allows the Government to simulate models that may have been developed on a number of CAE systems. Also this requirement allows the Government to provide selected VHDL models to contractors as leaf modules or as specifications for redesign of existing systems. Sharing of models will be practical only if the models are portable to any environment that may be used by a contractor.

Although the VHDL DID does not have any explicit portability requirements, several of its requirements are designed to ensure portability of models from one simulation environment to another. Par. 10.3 of the DID requires that files containing VHDL source code delivered to the Government be in full compliance with the current IEEE VHDL standard (Ref. 6). Most CAE systems support VHDL IEEE Std 1076 fully. Some CAE systems have specific subsets that can be mapped to their proprietary, high-performance simulator or silicon compiler. A few vendors have even extended VHDL to add features or capabilities. These extensions, however, are not portable and must not be used in documentation delivered to the Government. A contractor may wish to restrict a model to a specific subset of the standard that is sufficiently expressive and is compatible with the subsets required for proprietary tools in the contractor's tool suite. If the contractor is importing VHDL models, the design engineer needs tools that support the IEEE VHDL standard fully. A superset of standard IEEE VHDL should never be used.

One portability issue is data. The most portable way to represent data is to use American standard code for information interchange (ASCII) numbers and use the functions in the VHDL `TEXTIO` package for input/output (I/O). However, this approach may require much more space and execution time than formatted I/O of VHDL composite data types. Therefore, to achieve portability of the ASCII representation and the speed and space reductions of formatted I/O, the ASCII data may be provided along with a conversion function that produces the equivalent data ready for formatted I/O.

The VHDL model verification procedure (Ref. 7) has been developed to establish guidelines and procedures to ensure that VHDL models are compliant with the VHDL DID. These procedures involve inspection of the package received from the contractor as well as compilation and simulation of the model. (Ref. 8)

7-3 TEST BENCH DEVELOPMENT

The VHDL DID uses a test bench to provide the external stimuli for a model and to collect and evaluate the responses generated by the model.

"A VHDL test bench is a collection of VHDL modules which apply stimuli to a module under test (MUT), compare the MUT's response with an expected output, and report any differences between observed and expected responses during simulation." (subpar. 10.2.5.1 of Ref. 1)

A VHDL model delivered to the Government must be accompanied by a test suite, i.e., a collection of operating condition specifications, test benches, and associated test vectors that, taken together, test the VHDL model of the hardware under a variety of conditions. The same test bench can be configured to use different sets of test vectors or operating conditions to achieve different test purposes. The test plan for the model should specify these configurations.

The VHDL DID contains the requirements of the contents of the test suite. Subpar. 10.2.5.3 of the DID requires test benches to be provided for each VHDL module in the model hierarchy. Subpar. 10.2.5 of the DID requires that VHDL modules written and delivered to serve as part of one or more test benches be clearly distinguished from VHDL modules that represent part of the hardware design. Guidelines for tailoring the VHDL DID are discussed in subpar. 4-3.4.2.

MIL-HDBK-454 (Ref. 9) recommends additional requirements for the test benches and test vectors delivered to the Government. The use of VHDL to document ASICs in accordance with the DID is recommended in subpar. 4.5.1. These models should allow test vector generation and fault isolation to the integrated circuit pins. Subpar. 4.5.3 states that the same level of VHDL modeling for qualified integrated circuits in board-level applications should be used. Subpar. 4.5.4 of Guideline 64 of MIL-HDBK-454 advises the use of the Waveform and Vector Exchange Specification (WAVES) (Ref. 10) for documentation of test vectors.

The VHDL model verification procedure (Ref. 7) describes the procedure the Government can use to verify that a model meets all requirements.

7-3.1 WAVES

WAVES (Ref. 10) is designed to describe highly structured sets of test vectors, discrete event simulator trace output, and automatic test equipment (ATE) input. WAVES is designed to facilitate exchange of this information between design environments and automatic test equipment. Thus test vectors developed to validate VHDL models can also be used to drive the test equipment used to validate the hardware. WAVES is a subset of VHDL and uses only the sequential statements. WAVES has standardized on two levels of test benches: Level I and Level II. WAVES Level II is a subset of VHDL in that it does not allow arbitrarily complex types as the types of signals. WAVES Level I is a subset of WAVES Level II. Although WAVES is a subset of VHDL, the value of an event is given more structure in WAVES than in VHDL, so WAVES is more restrictive.

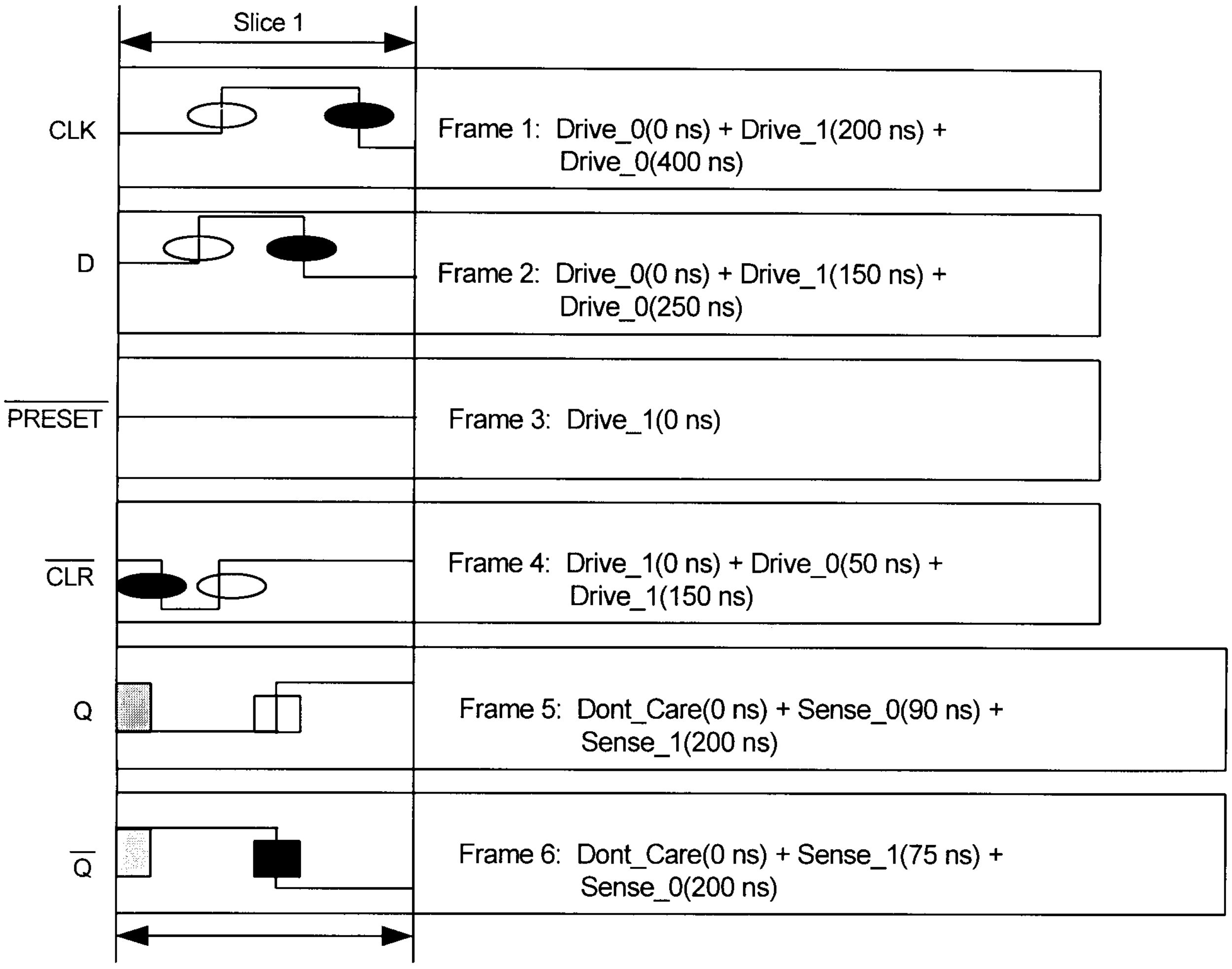
WAVES is built around two key concepts: the concept of an event and the concept of a waveform. As in the VHDL concept of events, a WAVES event has an associated time, signal, and value. The time of a WAVES event is the time at which the value of the signal changes.

The value of a WAVES Level I event has four separate components: a state, a strength, a direction, and a relevance. The state is the logic value of the signal; it is either low, midband, or high. The strength is the ability of a driver to force the signal to be resolved to the driver's state regardless of conflicting states from other drivers. The possible values for the strength of an event are disconnected, capacitive, resistive, drive, and supply. These values are ordered—disconnected is the weakest, and supply, the strongest. WAVES does not specify a physical interpretation of the strength values, i.e., WAVES does not define the impedance levels associated with particular strengths. The direction of an event denotes whether the event represents a stimulus to the MUT or an expected response from the MUT. The relevance is used to indicate the significance of the event to the simulation. The possible values for the relevance of an event are required, predicted, observed, and unknown. A required event is one that is part of the specification and that the MUT must match in order to meet the specifications. A predicted event is a response that has been calculated or expected as part of the specification, but is not required of the MUT. An observed event is a response from a MUT that is not part of the specification for the behavior of the MUT; it is not a predicted or required event. Other events are unknown events; typically, an unknown relevance would be associated with a don't care event. Each of the four components of the value of a WAVES Level I event may have one of two additional values: unspecified or unknown. Unspecified is typically used to indicate that the value of an event is missing from the waveform specification but could be determined from the MUT, whereas unknown is used to indicate that the value cannot be determined, e.g., an unstable output from a MUT. Unspecified may be used to indicate that an input to a MUT has not been initialized.

WAVES uses two different methods to specify times: delay time and event time. Event time is defined as an offset from either the current time or some specified event, a delay time is defined as an offset from the previous event on a specified pin, such as the clock. In WAVES the time of a response event may have an associated tolerance. A response time may be specified as a nominal response time, a lower tolerance on a response time, or an upper tolerance on a response time. Just as it allows a tolerance in the timing of a response event, WAVES allows a tolerance in the value of a response. In particular, WAVES allows a set of values to be associated with a single response event. Sets of event values may be used to represent uncertainty and to aid mapping between different state/strength systems.

A WAVES waveform is a sequence of time-ordered events across a set of signals. A waveform can specify both stimulus and response values. A waveform describes the testing of a MUT in that if the stimuli in the waveform are applied to the MUT, the responses generated by the MUT are associated with the responses of the waveform. A VHDL implementation of a WAVES test bench supports simulation of a VHDL model of the waveform. Such a simulation can verify that the MUT responds to the stimuli as the waveform predicts.

A waveform in WAVES is usually organized in terms of slices, as shown in Fig. 7-1. A WAVES slice is a specifica-


Slice 1
CLK
Frame 1: Drive_0(0 ns) + Drive_1(200 ns) + Drive_0(400 ns)
D
Frame 2: Drive_0(0 ns) + Drive_1(150 ns) + Drive_0(250 ns)
$\overline{\text{PRESET}}$
Frame 3: Drive_1(0 ns)
$\overline{\text{CLR}}$
Frame 4: Drive_1(0 ns) + Drive_0(50 ns) + Drive_1(150 ns)
Q
Frame 5: Dont_Care(0 ns) + Sense_0(90 ns) + Sense_1(200 ns)
$\overline{\text{Q}}$
Frame 6: Dont_Care(0 ns) + Sense_1(75 ns) + Sense_0(200 ns)

C. R. Unkle and W. G. Swavely © 1994 IIT Research Institute

Figure 7-1. Slice and Frames of a Waveform (Ref. 11)

tion of a portion of a waveform that occurs in a fixed period of time across all signals of the MUT. Different slices of a waveform may have different periods of time, but the period of time during a slice is the same for all signals of the MUT. In Fig. 7-1 the first slice has a period of 450 ns. A frame is the set of events defined within a slice for a single signal. Frames may be used like macros to capture multiple events and assign them to a single pin code. Six frames are shown in Fig. 7-1 for the six pins on the MUT. Five of the frames have two events defined in them, each of which specifies a new logic value and a time (in terms of the offset from the start of the frame).

The period of a slice is the time from the beginning of the slice to the end of the slice. The times for events occurring within a slice are defined as offsets from the starting time of the slice. A slice may contain events that are defined after the end of its period but never before the beginning of its period. A waveform is constructed of concatenated slices so that the end of the period of one slice is the beginning of the period for the succeeding slice.

WAVES provides VHDL procedures and types used to build events, frames from events, and slices from frames. These procedures and types are specified in the WAVES standard package.

A WAVES data set consists of a header file, WAVES files (which contain WAVES-specific VHDL design units), and external files (which contain test vector data in ASCII format). A WAVES data set includes a package that defines a procedure which generates a waveform. A waveform is generated by building slices, applying those portions of slices with direction `stimulus` to the MUT, and then sampling the MUT responses to those portions of the slices with direction `response`. Slices provide a way to build hierarchical test pattern descriptions that are consistent with modern ATE.

Fig. 7-2 shows how the three components of a WAVES data set interact with the module under test. The WAVES comparator function, part of the WAVES data set, is implemented in VHDL and is executed under the control of the waveform generator procedure (WGP). WAVES uses three operations in the WGP to interact with the waveform: apply, tag, and match. The apply operation adds events to the wave-

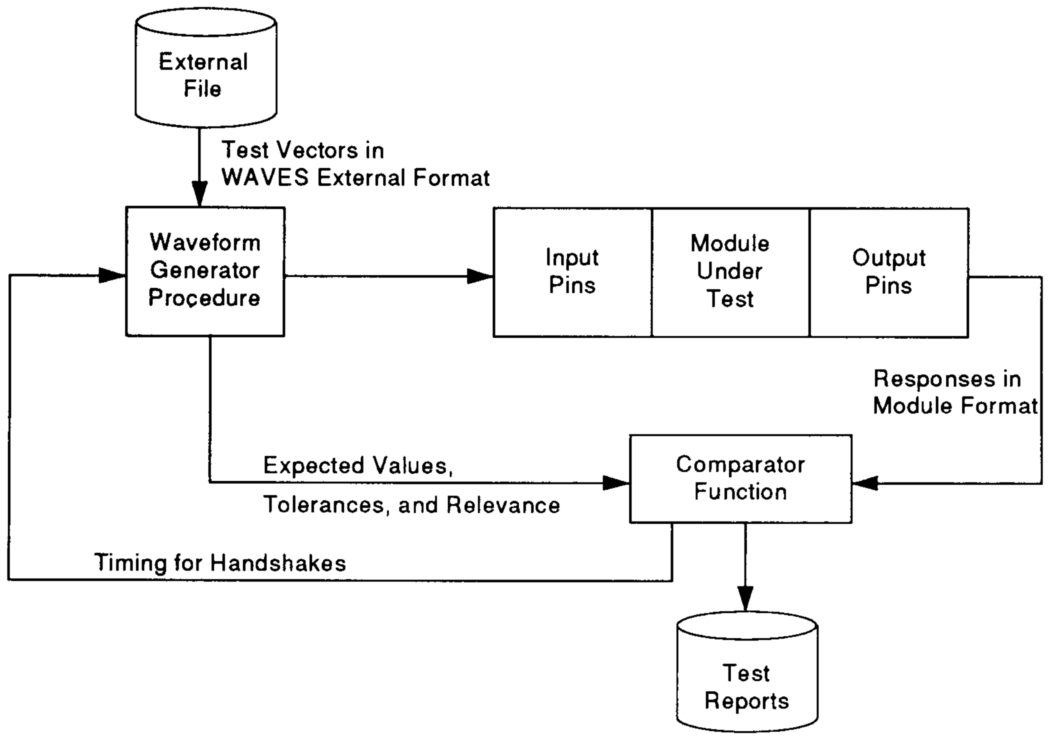

External File
Test Vectors in WAVES External Format
Waveform Generator Procedure
Input Pins
Module Under Test
Output Pins
Responses in Module Format
Expected Values, Tolerances, and Relevance
Comparator Function
Timing for Handshakes
Test Reports

Figure 7-2. Dependencies Between WAVES Packages

form and advances the current time. Events in the waveform with times less than the current time are unchangeable, but events with times in the future are considered pending and will be superseded if an event is applied with an earlier time. The tag operation adds a textual annotation to the waveform at the current time. The match operation samples the actual response of the MUT, compares it with the expected response (as specified in the slice by events with direction response), and produces a flag value of true or false depending upon whether the response exactly matched one of the values in the value set for the event labeled `response`.

A primary function of WAVES is to provide a standard format for describing all information relevant to test patterns and to extract appropriate views of that information in forms that can be used either by the model of the MUT or by ATE. As shown in Fig. 7-2, the WAVES packages can import test vectors in a common format from one or more optional external files and then translate this information for use as stimuli for the MUT and for comparison with MUT responses. The external files are optional because a waveform generator procedure does not have to read its data from an external file. A major part of the design of a WAVES data set is determining the VHDL type definitions and function definitions that correctly implement these views, and much of this effort is reusable if the definitions are packaged appropriately.

A WAVES data set is implemented as a collection of VHDL packages, VHDL design entities, and external files. To support portability of a WAVES data set, a data set is organized as a collection of files. Each WAVES file contains one or more packages, but all packages in the same file are analyzed into the same library. In general, WAVES declarations may be split up into multiple files. In this way, different declarations can be analyzed and stored in different libraries.

Every WAVES data set must contain at least five declarations:

1. *Logic Value*. An enumerated type naming all possible signal values that can appear in a waveform, i.e., can be either applied to the external inputs of the MUT or sensed as external outputs of the MUT. In Level I WAVES each port of the MUT is assumed to have values that range across an enumerated type.
2. *Value Dictionary***.** A function that translates the logic value names into event values
3. *Pin Codes***.** All of the characters used in the MUT to describe the value of any signal in the test bench at any time and contained in a string. Pin codes are used in external WAVES files and in table lookups in the WAVES functions and procedures.
4. *Test Pins*. Values of an enumerated type naming all signals to which the waveform will be applied. In WAVES the term "pin" is used to refer to any external signal of the MUT that is to be stimulated or compared with known outputs.
5. *Waveform Generator Procedure***.** The procedure that generates a waveform, annotates the waveform, and monitors the response of the MUT to the waveform. A WAVES data set may have more than one waveform generator procedure.

A WAVES data set must contain a minimum of three files: a header file and two files containing declarations and function definitions. Within a single WAVES data set there may be multiple files, each containing a different waveform generator procedure. The names of the WGPs may be the same, but in this event the WGPs must be placed in different libraries. There may also be one or more external files containing test vector sets. These vectors are stored in a common format, as defined by WAVES.

The order of analysis of the WAVES packages is determined by the dependencies of the objects in the packages. Fig. 7-3 shows dependencies among and between the packages. Packages are indicated by rectangles, and the name of the package appears in a box in the upper left-hand corner. Other names in the rectangle are names of objects or types declared in the package. These latter declarations are the source of the dependencies. The lists of objects in packages shown in Fig. 7-3 are not necessarily complete; other objects, subprograms, or types may be declared in these packages. A trapezoid represents a VHDL procedure; the name is inside the trapezoid.

7-3.1.1 Standard WAVES Packages

The standard WAVES packages provide a collection of functions used to construct waveforms. They also include the different variations on the apply, tag, and match functions. There are three standard WAVES packages: `WAVES_STANDARD`, `WAVES_INTERFACES`, and `WAVES_OBJECTS`.

`WAVES_STANDARD` is stored in its own library, which is also called `WAVES_STANDARD`. This package provides definitions for WAVES constants, data types, and functions that do not change between WAVES data sets.

The `WAVES_OBJECTS` package contains declarations of the some of the basic objects used by WAVES, including `delay_time`, `time_data`, `file_slice`, `pinset`, and `pin_code_string`. The `time_data` object holds a frame set array and is a parameter to the apply operation used by WGPs.

The `WAVES_INTERFACES` package contains declarations of the other objects used by WAVES. Its declarations include events and event values.

The `WAVES_OBJECTS` and `WAVES_INTERFACES` packages must be analyzed using information that is specific to both the MUT and the required test outputs, e.g., to the types of operations supported by a particular ATE. The user must edit these design units to include the appropriate context clauses, i.e., `library` and `use` clauses. Dependencies of these design units are shown in Fig. 7-3; a library structure is shown in Fig. 7-4.

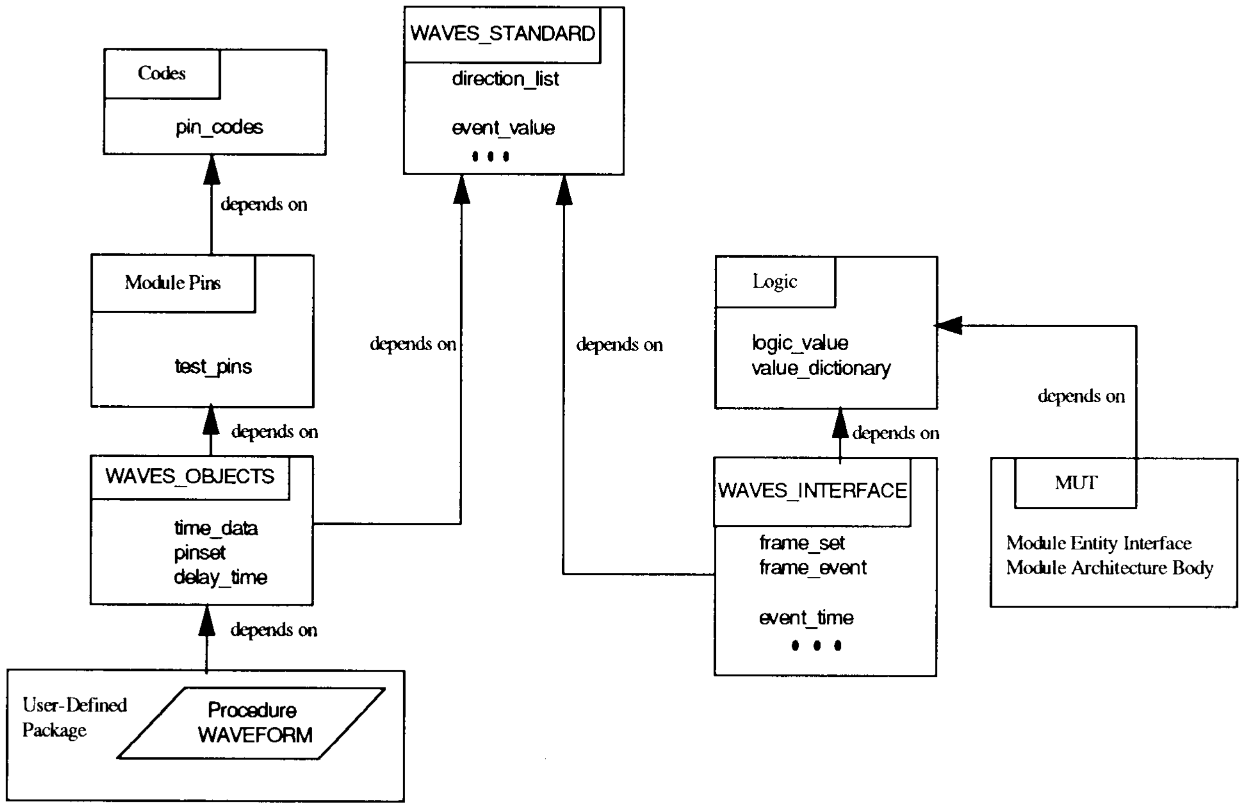


Figure 7-3. Partitioning of WAVES Packages into Libraries

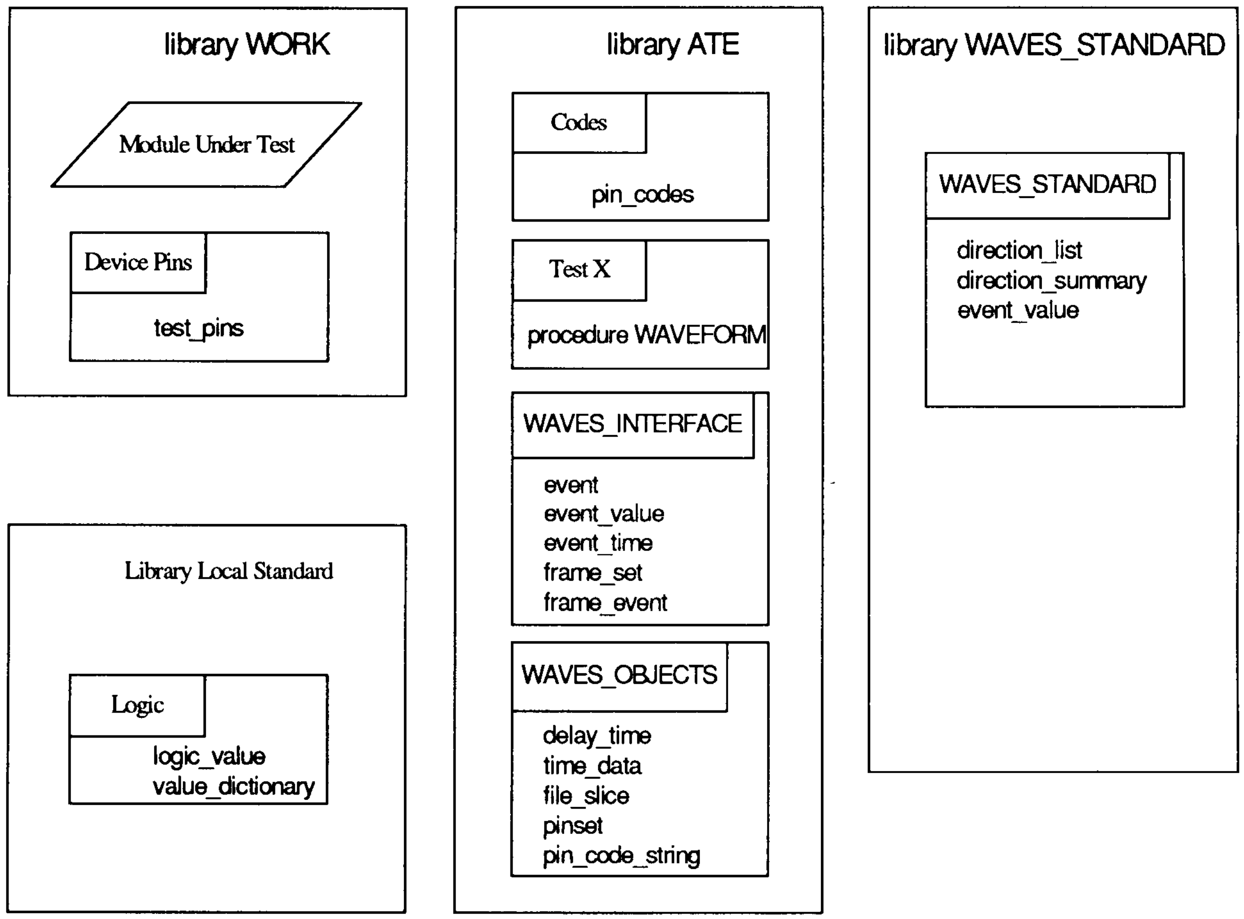


Figure 7-4. Library Structure of WAVES Packages

7-3.1.2 Local WAVES Packages

WAVES files may be reused at two different levels: the source code form of a WAVES package may be reused, or a WAVES package may be analyzed once and then referenced by multiple WAVES header files. The `WAVES_OBJECTS` package must be reanalyzed whenever the MUT module description changes because it depends upon the test pins description. However, this package may be reused without reanalysis if the external interface of the MUT does not change. The `WAVES_STANDARDS` package is an example of a package that is analyzed once and then reused without reanalysis.

To make the most efficient use of a WAVES test bench, the user must carefully plan the partitioning of the declarations into packages and WAVES files in a way that minimizes the amount of reanalysis. Fig. 7-4 shows a partitioning of these declarations into libraries and packages. This partitioning assigns packages with similar reasons for change to the same library. The ATE library contains the pin codes definition, which is derived from the requirements of the ATE equipment. The ATE library also contains the WPG specific to that ATE structure and the WAVES packages that are dependent upon the pin code information. Different ATE systems require different ATE libraries if the pin codes for the ATE systems are different. If no ATE device has been selected, an ATE library should be constructed using whatever pin codes make sense to the user. When an ATE system is chosen, this library should be updated. The `Local_Standard` library contains the logic value and value dictionary declarations because those are likely to be used across several MUTs. For example, the local standard library could include logic value and value dictionary definitions that are appropriate to the IEEE Std 1164 logic package.

7-3.1.3 WAVES Test Suites

Several different WAVES packages must be developed for a specific module and a specific type of test equipment. The module-specific declarations include the test pins and the logic values. These declarations and their partitioning into packages are shown in Fig. 7-4.

Two sets of files are specific to a particular MUT and to a particular test case: the WAVES header file and the external test vector files. Each WAVES data set has a unique header file. The header file identifies the data set, describes the other files in the data set and their intended use (including the target library for VHDL packages), identifies VHDL libraries and packages that already have been analyzed for use in the test bench, and defines the order of analysis of the VHDL source code files comprising the WAVES test suite. Fig. 7-5 shows an example header file.

```
-- Example WAVES Header File
--
    Title                   WAVES Example
    Author                  Research Triangle Institute
    Author                  Geoffrey Frank
    Date                    April 3, 1992
    Origin                  ieee.std_logic_1164
    Device_Id               Multiple XOR chip

    -- The file names are case sensitive.  The originating environment
    -- is UNIX, specifically Ultrix on a DEC Workstation.  These
    -- files are being ported to an MS/DOS environment, where the
    -- Model Technology VHDL analyzer and simulator are being used
    -- for validation.

    Waves_Filename    waves_logic.wav  Local_Std
    Waves_Unit        Waves_Interface  ATE
    Waves_Filename    waves_codes.wav  Local_Std
    Waves_Filename    device_pins.wav  Work
    Waves_Unit        Waves_Objects    ATE
    Waves_Filename    seq_wgp.wav      ATE

    External_Filename pattern          seq_pat.dat

    -- The waveform generator procedure is contained in seq_wgp.wav

    Waveform_Generator_Procedure       waves_logic.wav  Local_Std
```

Figure 7-5. Example WAVES Header File

The first part of the header file identifies the data set by brief text strings associated with the following WAVES keywords:

1. *Title*. This field provides a brief description of the data set. Different releases of the same data set normally keep the same title.

2. *Author*. This field contains the name of the organization and the name of the individual or individuals within the organization who developed the data set. It defines the responsible organization and specifies one or more points of contact who are the best sources of technical information about the data set.

3. *Date*. This field contains release information for configuration management purposes. The data should uniquely distinguish this release from all other releases of the same data set.

4. *Origin***.** This field is used to indicate the source of the model of the MUT and also "soft standards" that have been agreed upon by the producers and users of a WAVES data set. The DoD contracting agency and the contractor should agree in advance on what information is allowed in the origin field of a WAVES data set delivered to the Government.

5. *Device Id*. The device identification field identifies the target MUT of the data set.

These fields are required. After the identification section of the header file, the WAVES files are specified in the order in which they are to be analyzed. A WAVES header file can specify several different types of files and different uses of those files. WAVES file name commands are included in the header file to associate the name of a WAVES file in the host operating system and the target VHDL library for all packages contained in the WAVES file. WAVES units are used to refer to standard WAVES packages. WAVES unit commands are included in the header file to indicate the order in which standard WAVES packages are to be analyzed and to indicate the VHDL library in which the analyzed package should be stored. A WAVES header file may also include VHDL `library` and VHDL `use` clauses. These clauses provide the context for analyzing later packages, and reduce the time and effort required to prepare the data set for simulation. A WAVES header file may also include references to external files. External file name commands associate a logical name for the file used in the WAVES packages with the host operating system name for the file.

A WAVES external file uses a standard format for test vectors to be stored as data files. The WAVES external file format is very flexible. Comments in an external file are delimited at the start by a '%' and at the end by the end of the line. An external file contains a sequence of WAVES slices. WAVES slices are separated by semicolons. A WAVES slice contains four fields. The description of a WAVES slice in an external file consists of a subset of the first three of these fields. In an external file, the fields are separated by a colon. These four fields are possible:

1. *codes*. The codes field is typically used to hold a pin code string for an apply operation.

2. *fs time*. The file slice time is typically used to define the period for a slice.

3. *fs integer*. The file slice integer is typically used to select a slice period from an array of possible slice periods. For example, the file slice integer can be used to index into an array of slice times with tolerances for output ports.

4. *end of file*. The end of file field is a Boolean flag that is set to true if the last attempt to read a slice from the external file caused an end of file condition. When the end of file flag is true, all other fields are invalid.

Typically, a WAVES slice is used to drive all of the MUT input ports at the same time. This is the approach that is taken in the external file shown in Fig. 7-6.

7-3.2 DOCUMENTATION OF TEST BENCHES

Subpar. 10.2.5.2 of the VHDL DID (Ref. 1) requires all test benches to be cross-referenced to physical hardware test plans, specifications, and drawings, when possible.

The WAVES header file format, as discussed in subpar. 7-3.1.3, specifically addresses several of these DID requirements. The device ID field links the test bench to the identification of the physical device whose model is being tested. If the waveform generator procedure is designed for a specific ATE system and the WGP is stored in an ATE-specific library, the header file listing the file dependencies will indicate the ATE system used to test the physical device. Furthermore, WAVES allows the same external file to drive both the VHDL test bench and the test of the physical device by the ATE. Thus configuration management of the external files and appropriate naming conventions for external WAVES files provide another link between VHDL simulations and physical device simulations. The origin field may be used to link a specific test as defined by the WAVES header file to a specific physical hardware test plan. The reference to the MUT VHDL description should be specific enough to link the VHDL test to a specific version of the physical hardware, as modeled by a specific MUT VHDL description.

Subpar. 10.2.5 of the VHDL DID (Ref. 1) requires all VHDL test benches to be clearly distinguished from VHDL modules that represent the hardware design. Recommended file-naming conventions are provided in Chapter 9.

A hierarchical directory structure should be used to organize the source code and auxiliary files. There should be a separate directory for each component, and all files relating to a specific component, such as alternative architecture bodies, test benches, auxiliary files of test vectors, and configuration declarations, should be stored in the directory for that component.

7-4 TEST VECTOR DEVELOPMENT

Test cases for VHDL models fit into two categories. The first category, which is supported by WAVES, specifies the tests needed to demonstrate the functionality of the physical device represented by the VHDL model. The second category of tests demonstrates that the VHDL model also meets the requirements of the DID, e.g., by providing error messages upon detection of timing and electrical faults. This category of test is not supported by WAVES. Although the contractor is responsible for test vectors that test both the functionality of the model and the physical hardware, the contractor, an independent model verifier, or the Government is responsible for creating test vectors that verify that the VHDL model complies with the properties required by the VHDL DID.

7-4.1 BEHAVIOR TESTS

Subpar. 10.2.5 of the VHDL DID (Ref. 1) requires that the test vectors supplied with the test bench test the intended behavior of the MUT. These functional tests must be equivalent to the physical tests performed on the physical device. These behavior tests should verify that the models act the same as the physical device on startup, on recovery from er-

```
%   Example WAVES External File
%   Author              Geoffrey Frank, Research Triangle Institute
%   Version Date        April 3, 1992
%   Intended for        Waves example data set, 4 input chip
%
%   The pin codes are '1', '0', 'X', 'Z'
%   The device has 4 pins.
%   Pins 1 and 2 are scan path inputs,
%   Pins 0 and 3 are parallel inputs.
%   This driver file updates every pin in every file slice
%  P P P P
%  0 1 2 3
   0 1 0 1 : 5 ns ;
   0 1 0 1 : 5 ns ;
   0 0 0 1 : 5 ns ;
   0 0 0 0 : 5 ns ;
   0 1 0 1 : 5 ns ;
   0 0 0 1 : 5 ns ;
   0 0 0 0 : 5 ns ;
   1 1 1 1 : 5 ns ;
```

Figure 7-6. Example WAVES External File

rors, prior to an enable, and during a restart. The test vectors developed to test the behavior of a VHDL model should determine whether the VHDL model omits any functions specified for the hardware. The test vectors should also determine whether the VHDL model incorrectly implements any specified functions. These tests verify that the results produced by the model are consistent with the specification for the physical hardware and should not differ significantly from the results produced by the physical hardware. The acceptable tolerance for deviation between the timing and values of responses generated by the MUT and the expected responses should be specified as part of the test sets. For example, WAVES allows don't care as the value for expected responses and allows tolerances to be specified for the timing.

Subpar. 10.2.3.3 of the VHDL DID requires that the values of signals depending upon the structure of the hardware, e.g., a scan path, be absent from behavioral bodies. A recommended approach to describing such functions in a behavioral model is that the behavioral body should respond with a note-level assertion indicating that the structure determining the signal value is not implemented in the model when the system is placed in a mode in which valid output is expected on a structurally dependent output signal and that any valid signal value should be generated. This approach allows functional tests of the processing of structurally specific BIT data but does not provide feedback on quality measures of the BIT implementation, such as coverage. This approach also allows tests of the diagnostic modes of a system to be developed early in a top-down development process and refined as the design detail is added.

For structural models the behavior tests should test that the diagnostic and test modes of the system, such as scan paths, have been implemented correctly. If structural models are available, it should be possible to verify estimates of BIT, such as coverage. In a top-down development process the structural models are not available until late in the development process.

7-4.2 PROPAGATION DELAY TESTS

It is strongly recommended that a test bench submitted to the Government include test vectors that test the intended propagation delay from the inputs to the outputs of the MUT. Propagation delay tests must provide information on best-, worst-case, and nominal delays to ensure the design will operate as required throughout the operating range. These test vectors and their associated test benches should provide mechanisms to compare the delays reported by a VHDL simulation with the delays measured on a physical device. For simple devices these test vectors can be created by hand. For example, the worst-case timing for a ripple-carry adder occurs when the carry signal ripples through all stages. Similarly, the best-case timing occurs when no carry occurs between any stages. Critical path analysis may be used for more complex systems to determine the best- and worst-case times. Sensitivity analysis may be required to determine how changes in the input test vectors change best- or worst-case delay times. Nominal delays may require some statistical analysis of input signal patterns.

As noted in subpar. 5-4.4, variations in environmental factors such as power, voltage, and temperature should be considered when defining best- and worst-case delay times. If best-case, nominal, and worst-case delay times are defined for a range of environmental conditions, the tests should check the extremes of the ranges. If best- and worst-case delay times are dependent upon specific environmental conditions, the test bench that tests these times must specify the appropriate environmental factors. To obtain tests that reflect best- and worst-case delay times, it may be necessary to run tests on several combinations of environmental conditions. For example, the best-case times may occur when the component is run in a low-temperature, high-voltage environment. The worst-case times may occur when the component is run in a high-temperature, low-voltage environment. From these tests, test vectors for best- and worse-case timing that reflect a range of environmental conditions can be extracted.

Propagation delay tests for high-level models of programmable electronic systems depend not only on the physical environment but also on the software executing on the hardware (Ref. 12). Performance models, described in subpar. 2-2.2.1, are often used to compute propagation delays in these types of systems. Very often mixed-level-of-abstraction models are used. (The processors are modeled at a very abstract level, and the system interconnection hardware is modeled at a detailed level.) It is very difficult to obtain test data to drive propagation delay tests for high-level models, and it is difficult to verify that the best- and worst-case times are what they are claimed to be. Annotation of test data explaining why a certain test configuration and input data produce worst-case timing is an important part of the documentation. An important source of such test data is tests of previous systems or regression test data that capture sequences of events that have caused trouble during integration and test of the system. If such data is obtained, the source of the data and the reason for inclusion in the test set, e.g., reference to a specific problem report, should be documented.

7-4.3 ERROR CONDITION TESTS

A complete test suite must test that the model responds appropriately to invalid usage. Error condition tests promote effective reuse of the components by warning the user of the model if the planned use violates the operating constraints of the model.

Subpar. 10.2.6 of the VHDL DID (Ref. 1) specifies the minimum contents but not the format of error messages. Each error message must identify the requirement or constraint that has been violated and the name of the VHDL design unit in which the error occurred, i.e., where the violation was detected. In some cases static analysis of the VHDL code may be sufficient to determine whether a model meets these conditions. In other cases, tests may be required to determine whether error messages provide this information. For example, a static analysis can determine whether an assertion statement in a design entity interface identifies its enclosing design entity. However, a test may also be required to determine whether the assertion statement correctly identifies the component instance in which the error occurred.

7-4.3.1 Invalid Operating Condition Tests

Subpar. 10.2.2.2 of the VHDL DID (Ref. 1) requires that external error conditions on a MUT, such as electrical constraint violations and timing violations, should be reported. If the operating conditions are static and are not changed by elaboration of the VHDL model, a static analysis should be sufficient. However, if the operating conditions for an entity are determined by calculations made during elaboration, e.g., if the operating conditions are computed from generics, tests may have to be performed to determine whether invalid operating conditions are checked by the model. If, however, the operating conditions vary as a function of the test inputs, the test vectors will have to be more complex. In either of these cases it is important that the test vectors used to produce reports of invalid operating conditions be linked to the configuration declaration (or equivalent documentation), which specifies the generics that cause the error message.

7-4.3.2 Invalid Input State Tests

VHDL models should be robust enough to respond appropriately to invalid states of input signals. The VHDL model verification procedure (Ref. 7) recommends tests of the ability of the VHDL model of the MUT to respond to invalid inputs. These tests must be either delivered with the model or developed by the verifier. Invalid states of input signals would include error values associated with the input data types. For example, the `'U'` and `'X'` values of the IEEE 1164 (Ref. 2) standard logic package can be considered invalid input states. A set of test vectors that contains invalid input values for the MUT should be provided. These tests are part of ensuring that a VHDL model of a component is interoperable with VHDL models of other components. They are also used to ensure that invalid inputs are properly propagated through the model.

Invalid input state tests are particularly important if the model of the component will be used in a mixed-level-of-abstraction model of a system, in which these tests can be used to verify the robustness of the type conversion functions. In particular, if the type for a port can express unknown, uninitialized, or don't care values, the model must react appropriately to those values either by issuing a warning or an error message or by propagating the error value through the system. If error values are not properly propagated, the user may not be able to isolate design faults in systems using the MUT as a component.

It is a good design practice to provide models with the capability to warn the user when invalid inputs occur. If the invalid input indicates a design error, these warnings can be used to isolate the error to another component. However, sometimes an assertion statement on an input port may produce a false alarm. If an invalid value occurs on an input port that is currently a don't care input to the function of the component, then an error message produced at the input port would be a false alarm. In such situations the user should be able to mask these messages by setting a simulation option.

IEEE Std 1164 (Ref. 2) extends the basic logic values of `'0'` and `'1'` to include additional signal states, such as `'X'` for unknown and `'U'` for uninitialized. The definition of this set of logic values is shown in Fig. 3-7. The standard also includes definitions for all of the basic logical operators that cover propagation of error inputs including unknown, uninitialized, and don't care values. Fig. 3-12 shows how the logical `AND` function has been extended to handle invalid signal states. A logic state of `'U'` for one input propagates to the output in every case except when the other input is `'0'`. Similarly, a logic state of `'X'` for one input propagates to the output in every case except when the other input is `'0'` or `'U'`. Models built using these logical operation definitions handle any standard logic input consistently. The IEEE Std 1164 also includes a definition for a resolution function that handles error propagation as described in subpar. 3-2.3.2.

Using IEEE Std 1164 types is an effective approach to detect invalid input states in gate-level models. To support use in mixed-level-of-abstraction models, high-level models must also be able to deal with unknown, uninitialized, and don't care inputs by either generating an error message or propagating appropriate values to the output ports. For example, consider a test bench for a bus interface module (BIM). Suppose that the bus consists of three subsignals: a control signal, an address signal, and a data signal. During arbitration for control of the bus (which involves exchanges of information on the control signal), the BIM may receive and generate don't care values on the data signal. When an input data signal arrives with an unknown value for one bit, the BIM should not propagate this value to the control signal, but it may generate an error message to indicate that it has received a bad input signal. However, this unknown value may not represent a design error, so it should be possible for the user to turn off the error message. On the other hand, during data transmission the occurrence of an unknown value on the data signal does represent an invalid input and should cause an error message and propagation of the error value across the bus. This example indicates that error propagation is data dependent, which means that test data are required to ensure that invalid values are propagated when they should be propagated and that the model does not generate false alarms by propagating invalid values when it should not.

7-4.3.3 Timing Constraint Violation Tests

VHDL models delivered to the Government should be tested to ensure that they create error messages whenever timing violations occur. This testing can be performed at two levels: by statically checking that timing constraints have been declared and are used in the model and by creating test vectors that force a range of timing violations. One simple form of such a test is to increase the setup and hold times until errors occur. A more interesting test is to increase the clock speed. (This test can, of course, be performed only if the clock is a signal external to the hardware component being modeled or if the clock generator can be externally controlled.) This test may cause a range of timing errors that should be reported by the model. The VHDL model verification procedure recommends that if the developer does not supply test vectors that

1. Violate the timing and voltage specifications
2. Attempt to perform illegal model operations
3. Test the functionality of the unit at its operational limits,

the verifier should develop and apply them.

Both the EIA 567 standard (Ref. 5) and the VITAL standard (Ref. 4) include timing violation checks in the VHDL packages that implement their standards.

The EIA 567 timing violation works with two generics required for each entity: `MGENERATION`, which is used to determine whether messages are to be generated whenever violations are detected, and `XGENERATION`, which is used to determine whether the unknown value `'X'` should be assigned to any signal at which a timing violation is detected. When `MGENERATION` is `TRUE`, messages are generated through assertion statements or are written to files using `TEXTIO` functions. If both `MGENERATION` and `XGENERATION` are `FALSE`, no timing checks are performed. These two generics should be set at simulation time. The VITAL package also provides simulation options to control the generation of messages and the performance of timing checks.

7-4.4 INTEROPERABILITY TESTS

Tests to ensure interoperability should verify that all VHDL design entities correctly process test vectors containing representative combinations of data input values, including error values, uninitialized values, and don't care values. Tests to ensure interoperability should also verify that all VHDL design entities correctly process any environmental and physical data including data inside and outside the acceptable ranges for the components of the system; the models should produce appropriate timing and delay values dependent on these environmental conditions. Finally, tests to ensure interoperability should verify that the model is portable over some range of simulation environments. Portability test values may be selected based on differences in representation of data types in different simulation environments. Portability tests may also include tests of simulation control options. Portability issues include directory, path, and file-naming conventions. If model portability is important, the range of target environments on which the model needs to be tested should be specified. If model portability is not a primary concern, verification that the model is written in standard VHDL may suffice. For example, if the model is to be archived when delivered, the model should operate correctly at least in the archive environment. Similarly, if the model is going to be validated through analysis and simulation (as opposed to just inspected), the validation environment should be specified in a tailored DID. In general, ensuring portability to future environments is not possible. However, ensuring that the model has been written in IEEE Std 1076 VHDL (Ref. 6) provides a good basis for portability.

7-4.5 ORGANIZATION AND DOCUMENTATION OF TEST VECTORS

To obviate documentation and traceability problems, the test vectors should be divided into separate files, and each file should address a specific set of requirements. The test bench should label the output so that examination of the reports generated by the test bench indicates the environment assumptions, the hardware model configuration, and the test vectors used in the run. This can be accomplished if the environmental assumptions, the hardware model configuration, and the test vector file name(s) are defined by a configuration declaration and the test bench prints the name of the configuration declaration. This use of a configuration declaration is discussed in par. 7-5.

The test bench and the test vector files should be commented to show their purpose. As shown in Fig. 7-5, WAVES (Ref. 10) allows comments in external files. VHDL model files, including design entities and test benches, should have header comments at the front of the file. The following format for header comments has been tailored to deal with test vector sets:

1. *Design Unit Name Identifier*. This should indicate the corresponding test bench and the VHDL name of the MUT.
2. *Identification of Originator or Source*. This should name the person responsible for creating the test vector set and also the corporate source if the originating person is an employee.
3. *DoD-Approved Identifier*. If an identifier for the physical hardware exists, it should be used for this field. If the physical hardware does not have an identifier, the sponsoring agency for the VHDL code development should create an identifier.
4. *Revision History*. At a minimum, indicate whether the test vector set has been delivered before. If it has, indicate
   a. The dates of the revisions
   b. The individual and organization making the revision
   c. The purpose of the revision
   d. The part of the test vector file modified.
5. *Test Vector Set Purpose*. This part of the header comment should relate the test vectors back to requirements stated in the VHDL DID or to functional or timing tests specified for the physical hardware.

6. *General Approach*. This part of the header comment should deal with issues such as timing tolerances allowed for the comparison of MUT responses and expected responses. This section should also indicate how the test vectors were generated: Are they randomly generated according to a distribution, hand coded, or automatically generated by an automatic test pattern generator (ATPG)?

7. *Additional Information for Users*. This part of the header comment should include information about how to modify the set of test vectors in response to changes in the module design or how to adapt the test vectors to different environments.

8. *Restrictions*. This part of the header comment should describe restrictions on the use of the test vector set and identify any part of the test bench or module description that, if changed, would make the set of test vectors ineffective.

9. *Assumptions*. This part of the header comment should describe assumptions made during creation of the tests.

10. *Previous Approval*. This part of the header comment should indicate whether the test vector set has been accepted previously by the DoD.

7-5 USE OF CONFIGURATION DECLARATIONS TO INSTANTIATE THE TEST BENCH FOR A MODEL

To be prepared for simulation, the model must be configured from a potentially complex database that contains these items:

1. At least one test bench for the model
2. One or more sets of test vectors
3. At least one architecture body for the model
4. At least one design entity for each component instantiated in the model
5. Optional packages specifying global data types, constants, and subprograms
6. One or more configuration declarations configuring the model or pieces of it.

VHDL allows a great deal of flexibility in configuring a specific model for simulation. A configuration declaration can be used to

1. Select libraries to be used as sources for packages and design entities. This choice allows the compilation units to be collected into libraries.
2. Select architecture bodies for components. VHDL allows several architecture bodies to be associated with a single entity interface. Through the use of configuration declarations or configuration specifications, an appropriate architecture body can be selected for each component instance.
3. Specify values for generics. Defining the values of key generics in a configuration declaration provides an audit trail to the values and makes the simulation repeatable.
4. Define port maps, particularly to specify type conversion functions. The combination of the selection of architecture bodies and the choice of type conversion functions is a key part of the construction of a mixed-level-of-abstraction model from a design database in which components are modeled at multiple levels.

Effective use of configuration declarations can provide significant assistance in the configuration management of models. However, care must be taken to centralize the late binding decisions in the configuration declaration rather than distribute this information throughout the different compilation units. For example, if the same test bench can be used with different external data files, the names of the data files should be defined in the configuration declaration rather than in the architecture body of the design entity that reads the file.

The purpose of the model affects decisions about what aspects of the model can be changed in a configuration declaration. For example, during the design of an ASIC, when the timing parameters are changing frequently as the layout of the circuit changes, the VITAL (Ref. 4) approach of using generics for the timing is appropriate. If a VHDL model of the ASIC is built as documentation after the design is complete, then the validated timings may be permanently bound into the VHDL architecture body of the ASIC model. This is a situation in which configuration specifications may be made in an architecture body rather than in a configuration declaration. Configuration specifications are typically used in an architecture body to specify a more permanent binding.

During the specification and design of a VHDL model, decisions must be made about what aspects of the model are likely to change and what aspects are not likely to change, and these decisions should be reflected in the organization of the VHDL code. These decisions should be made by considering the way the model will be configured for simulation. For example, if the model is to be simulated several times during testing, the different configurations required for testing should be considered.

7-5.1 SELECTION OF ALTERNATIVE DESIGN LIBRARIES

Selection of appropriate design libraries is a key step in preparing a model for simulation. Alternative design libraries provide a mechanism to encapsulate information about technology or common data types, constants (such as timing constants), and functions (such as derating functions) in packages. Alternative design libraries also provide a way to describe alternative implementations of design entities together with the packages of data types and constants appropriate for the elements of that library. For example, in subpar. 7-3.1 different libraries for different ATE systems are suggested for WAVES (Ref. 10).

VHDL constrains the way deferred constants can be used to define values for global parameters. Within a single library each package declaration has at most one body. Thus, if different values are required for deferred constants, packages with the same interface but different bodies must be installed in different libraries. Alternatively, the package body can be reanalyzed between simulations. This approach has the disadvantage that the different sets of values for the constants are not maintained in the VHDL libraries but must be maintained as source code files. Thus they create another level of configuration management.

7-5.2 SELECTION OF ALTERNATIVE ARCHITECTURES

Selection of appropriate architectures is a key step in configuring a VHDL model. A single entity interface can have several associated architecture bodies. Different architectures can represent different implementations of the same entity interface, so selecting an architecture is a means of trading off or evaluating alternative implementations. Different architectures may represent different levels of abstraction of a design entity, so selecting an architecture determines the level of abstractions to be used for a particular component. Subpar. 10.2.1 of the VHDL DID (Ref. 1) requires both behavioral and structural models for all modules that are not leaf modules. Therefore, selecting an architecture for each component is an essential step in configuring a DID-compliant model for simulation.

7-5.3 BINDING OF GENERICS

Defining the value of generics is another important step in preparing a model for simulation. Generics can be assigned values hierarchically: a parent structural architecture receives values of generics, computes the values of the generics of its components, and then assigns those values to the components through a generic map. This process can be continued down the hierarchy so that values trickle down through the design hierarchy. Generics can be assigned values in architectures or in entity interfaces, which is an early binding of values.

Generic constants provide a way for the late binding of parameter values through the use of configuration declarations. These constants are particularly valuable for setting the value of tradeoff parameters or parameters used in sensitivity analyses. Generic constants can be used in combination with constants declared in packages, and a few may be used to determine the values of many model characteristics. For example, in EIA 567 (Ref. 5) a single generic constant is used to select among minimum, nominal, and maximum timing options. The value of this constant can be used to look up timing information stored in a table or tables; the values selected from these tables may set many timing values. Alternatively, a generic constant may be a complex structure containing the values for many model parameters. The latter approach is the preferred mechanism for back annotation, particularly if the tables of parameters cannot be precalculated.

EIA 567 (Ref. 5) requires that at least a minimal set of generics be provided for each design entity. VITAL (Ref. 4) also uses generics to provide control. The parameter `TimingChecksOn` activates timing checks and the VITAL parameter `XGenerationOn` performs a similar function to the EIA `XGENERATION` parameter. A VITAL-compliant model uses generics to implement back-annotation of timing information. VITAL has established a mapping mechanism between standard delay format (SDF) information and the names of ports and generics of a design entity. This mapping assumes a specific naming convention consistent with the SDF file.

7-5.4 PORT MAPPING

As discussed in subpar. 7-2.2, the use of type conversion functions may be required to construct mixed-level-of-abstraction models and also to configure models that are not immediately interoperable. The port-mapping capabilities of VHDL configuration declarations allow these conversions to be specified without rewriting and reanalyzing the VHDL source code for the two components being connected.

Type conversion functions may be specified in port maps in architecture bodies as well as in configuration declarations. The use of type conversion functions in the configuration declarations separates the configuration issues from the interconnect issues and allows more reuse of structural architecture bodies without reanalysis.

7-6 DEFINITION OF SIMULATOR OPTIONS

A VHDL model has been configured when all component instances are bound to specific design entities. During the binding of VHDL component instances, the modeler has also selected test bench design entities that will drive the simulation. Selection of the test bench design entities determines some of the simulation parameters, but other decisions still need to be made.

The mechanisms used to control simulation are not specified in the *VHDL Language Reference Manual* (Ref. 6). Some simulators allow these parameters to be set as a part of the simulation. In other cases these simulation parameters may have to be included in a package body dedicated to simulation control, and others may have to be specified as generics in configuration declarations. The option to set simulation parameter values in configuration declarations is strongly preferred over entering them manually during elaboration of the model. Simulations requiring manually entered parameters are not automatically reproducible and are, therefore, less valuable as elements of a test suite. Manual entry of parameters can be very time-consuming when a battery of tests is conducted. The following subparagraphs describe some of the parameters that need to be set and some possible ways of controlling them.

7-6.1 CONTROL OVER ENVIRONMENTAL PARAMETERS

Subpar. 10.2.2.3 of the VHDL DID (Ref. 1) mandates capture of physical and electronic parameters of the hardware, such as temperature range, power dissipation, and radiation. The same subparagraph encourages the use of packages to describe common operating conditions across components of the hardware system. Subpar. 10.2.3.2 of the DID recommends the use of timing models that consider environmental parameters, and subpar. 10.2.2.2 requires error messages to be generated when the values of these parameters are outside their acceptable range.

Each simulation has a particular value for each of these environmental parameters. These parameters are best declared in packages, such as the physical and electrical views described in EIA 567 (Ref. 5). The values of these parameters should be defined in the corresponding package bodies. Thus the parameters become deferred constants. When the values of deferred constants are changed for a particular simulation, only the body of the package needs to be reanalyzed; the VHDL design units that use the package need not be reanalyzed. Thus the use of deferred constants for simulation parameters is strongly recommended.

The VITAL standard (Ref. 4) does not include environmental parameters in its models. Instead, it requires that all computations of delay times be performed externally to the model and imported either as generics or SDF files.

7-6.2 SELECTION OF DELAY TYPES

One important simulation option is selection of the type of delay to be used in the simulation. Subpar. 10.2.3.2 of the VHDL DID (Ref. 1) requires VHDL models delivered to the Government to support at least three timing delay options: worst-case, best-case, and nominal. The VITAL standard (Ref. 4) and EIA 567 (Ref. 5) have different approaches to the selection of timing information.

EIA 567 defines a data type, called `operating_point_type`, which is an enumerated type with three values: `minimum`, `nominal`, and `maximum`. Each entity interface must have a generic constant of type `operating_point_type`. The value of this generic is used to select the timing model used within the model.

The VITAL standard requires all timing calculations be performed “off-chip”. All load-dependent or environmentally dependent timing data are calculated outside the VITAL model and then provided to it as actual values to the model via generic maps or a standard delay format. The DID requirement can be satisfied for a VITAL model with three separate configuration declarations, each including generic maps with different values, or through three separate SDFs.

Generics should be used when different values of simulation options are used in different parts of the model, particularly when different instances of the same component model require different values. These simulation options can be used directly in the behavioral architecture of the leaf modules of the simulation, or they may be passed down to the leaf modules through generics.

7-6.3 CONTROL OVER EXECUTION OF ASSERTIONS

Subpar.10.2.2.2 of the VHDL DID (Ref. 1) requires error messages to be generated by models when conditions such as violations of setup and hold time constraints occur. Concurrent assertion statements are a natural mechanism to support such tests. However, because these assertions are checked whenever any signal referenced in the assertion condition changes value, simulation of a model with assertions can be much slower than simulation without assertions.

Both VITAL (Ref. 4) and EIA 567 (Ref. 5) use multiple generics to control error handling. In EIA 567 these are called `MGENERATION` and `XGENERATION` (See subpar. 7-4.3.3.). The VITAL standard uses two generics, `TimingChecksOn` and `XGenerationOn`.

For models at higher levels of abstraction, either new generics must be defined for control of assertions or the existing generics must be reinterpreted for the needs of more abstract models. The latter approach is probably better suited to modeling at mixed levels of abstraction.

7-6.4 CONTROL OVER PROPAGATION OF UNKNOWN SIGNAL STATES

IEEE Std 1164 (Ref. 2) defines the propagation of unknown values in its definition of its logic functions. Both VITAL (Ref. 4) and EIA 567 (Ref. 5) standards follow IEEE Std 1164 in supporting the propagation of unknown signal states. However, the VITAL and EIA 567 standards differ in the approaches they take to convert the detection of timing errors into reports and to generate unknowns.

The VITAL standard includes a timing check procedure called `VitalTimingCheck`. This routine is overloaded so that the output variable `Violation` is either of type `BOOLEAN` or of type `X01`. If the `Violation` variable is of type `BOOLEAN`, then it can be used in an `IF` statement or an assertion statement. If the `Violation` variable is of type `X01`, then it can be `or`ed with a variable of the `std_ulogic` type to propagate the error. VITAL does not support generation of unknown signal states without the associated generation of error messages, so it should always be possible to determine where an unknown value is generated. VITAL addresses the issues of simulation speed through the use of native mode implementations of logic primitives with timing checks rather than through the elimination of timing checks. Although VITAL includes VHDL source code descriptions of its primitive library and timing check functions, it assumes that the simulation vendors will implement these primitives as an integral part of their simulators. Thus the VHDL source code is intended as an executable specification.

EIA 567 does not include any timing check routines as part of the standard package; definition of these routines is left to the user. EIA 567 does require that error checks be included in EIA-567-compliant models. If the EIA 567 generic `XGENERATION` is `TRUE`, then when a timing error is detected, an unknown signal state is output. EIA 567 requires timing checks to be suppressed during simulation when both the `XGENERATION` and the `MGENERATION` generics are `FALSE`.

As in the case of control over the execution of assertions, models at higher levels of abstraction should define new generics for control over error propagation or reinterpret the VITAL and EIA 567 generics. In subpar. 7-2.2 support for error propagation through the use of type conversion functions in mixed-level-of-abstraction models is discussed. Those type conversion functions could be adapted to take into account the `XGENERATION` generic. A configuration declaration could select the appropriate type conversion function based on the value of the `XGENERATION` generic.

REFERENCES

1. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, Department of Defense, 11 May 1989.
2. IEEE Std 1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1993.
3. IEEE Std 1076.3 (Draft), *IEEE Standard for VHDL Language Synthesis Package*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, September 1995.
4. IEEE Std 1076.4-1995, *IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, December 1995.
5. EIA 567-A, *VHDL Hardware Component Modeling and Interface Standard*, Electronics Industry Association, Washington, DC, March 1994.
6. ANSI/IEEE Std 1076-1993, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, September 1993.
7. Rome Laboratories/ERDD, *VHDL Model Verification and Acceptance Procedure*, Technical Report, Department of the Air Force, Griffiss Air Force Base, Rome, NY, March 1992.
8. M. T. Pronobis, "VHDL—Windows of Opportunity", *VHDL Model Verification*, VHDL Users' Group, Oakland, CA, October 1990.
9. MIL-HDBK-454, *General Guidelines for Electronic Equipment*, 28 April 1995.
10. IEEE Std 1029.1-1991, *Waveform and Vector Exchange Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1991.
11. B. Johanson and J. P. Hanna, *Learning to Use WAVES by Example*, Tutorial C, VHDL International Users' Forum, VHDL International, San Jose, CA, October 1993.
12. B. E. Clark, and G. A. Frank, "System Modeling for V&V", *Proceedings of the 2nd Annual International NCOSE Symposium*, Seattle, WA, July 1992, National Council on System Engineering, Seattle, WA.

BIBLIOGRAPHY

J. R. Armstrong, *Chip-Level Modeling With VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

J. R. Armstrong and F. G. Gray, *Structured Logic Design Using VHDL*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.

O. Levia, F. Abramson, "ASIC Sign-Off in VHDL", *VHDL Boot Camp*, VHDL International Users' Forum, San Jose, CA, October 1993.

S. Turner, "Back Annotation for VHDL Structural Models", *VHDL—Windows of Opportunity*, VHDL Users' Group, Oakland, CA, October 1990.

V. Berman, "An Analysis of the VITAL Initiative", *VHDL Boot Camp*, VHDL International Users' Forum, San Jose, CA, October 1993.

V. Berman and C. Ussery, "A Proposed Back Annotation File Format for VHDL", *Using VHDL in System Design, Test, and Manufacturing*, VHDL International Users' Forum, Scottsdale, AZ, May 1992.

Enabling Design Creativity, VHDL International Users' Forum, Newport Beach, CA, October 1991.

Using VHDL in System Design, Test, and Manufacturing, VHDL International Users' Forum, Scottsdale, AZ, May 1992.

VHDL Boot Camp, Proceedings of the Fall 1993 Conference, San Jose, CA, October 1993, VHDL International, Santa Clara, CA.

VHDL Users' Group Fall 1990 Meeting, VHDL Users' Group, Oakland, CA, October 1990.

Using VHDL for Electronic Product Design, VHDL Users' Group, Cincinnati, OH, April 1991.

Z. Navabi, S. Day, and M. Massoumi, "Investigating Back Annotation of Timing Information into Data Flow Descriptions", *Using VHDL in System Design, Test, and Manufacturing*, VHDL International Users' Forum, Scottsdale, AZ, May 1992.

CHAPTER 8
MODELING TESTABILITY WITH VHDL MODELS

The modeling of testability information using VHDL, testability measures, and techniques is described. A hierarchy and the functions of test components are described, and the IEEE Stds 1149.5 and 1149.1 test interfaces are discussed. The use of behavioral modeling is recommended to verify that the test bus and test controller systems interface correctly without regard for the contents of the information sent across the bus. The use of detailed structural models is recommended as the starting point for generation of built-in test structures, such as boundary scan. This chapter emphasizes that detailed structural models are necessary to evaluate many testability measures.

8-1 INTRODUCTION

The very high-speed integrated circuit (VHSIC) hardware description language (VHDL) Data Item Description (DID) (Ref. 1) requires two kinds of models: behavioral and structural. The VHDL DID also states specific requirements for the modeling of testability in both of these models. Testability is any design characteristic that contributes to fault masking, detection, isolation, or containment. Subpar. 10.2.3 of the VHDL DID requires, "Test and maintenance functions which are part of the physical unit and are available to the user shall be included in the behavioral body.". However, the VHDL DID states in subpar. 10.2.3.3, "Signal values which are dependent upon a particular structural implementation, such as scan path signatures, shall not be specified in the behavioral module.". One of the purposes of this chapter is to recommend methods to meet both of these requirements, which may at first seem to be contradictory. The crux of the issue is the extent to which test and maintenance features can be described in a manner that is implementation independent because the goal of a VHDL behavioral model is to provide the user with a model that is free of implementation dependencies. Such models can be used as the specification for multiple implementations.

Test and maintenance issues are critical to specifying how detailed a VHDL structural model must be to meet VHDL DID requirements. Subpar. 10.2.4 of the VHDL DID requires, "Structural bodies shall represent the physical implementation accurately enough to permit logic fault modeling and test vector generation. Structure which is created to support testing and maintenance such as scan paths shall be included in the VHDL structural description.". This chapter discusses the level of detail required in a structural model to support this requirement based on the standard fault models used for test vector generation.

8-2 PURPOSE AND SCOPE OF DESIGN FOR TESTABILITY

Testability features of electronic systems include fault masking, detection, containment, or isolation. Fault detection and isolation are critical to maintenance of military electronic systems. A major function of hardware maintenance is the detection and isolation of a fault to a line-replaceable unit (LRU) and subsequent replacement of the LRU. From the Government point of view, the primary goal for modeling testability and fault tolerance circuitry in VHDL models is to validate that a design provides the testability required to maintain fielded systems. Testability is an aspect of design for maintainability.

In a fault-tolerant system either faults are automatically masked so that they cannot corrupt the system or the system provides automatic fault detection, isolation, and recovery (FDIR). FDIR reconfigures the system by switching in a built-in spare in place of the faulty subsystem. Design for testability of a fault-tolerant system must support FDIR. Fault containment circuitry is used to prevent corruption of data outside the LRU, especially outside the digital part of the electronics. For example, failure of flight control circuitry should not cause sudden, drastic changes in the control surfaces of a wing.

The scope of design for testability includes fault models, hardware design models, test strategies, and test techniques. This chapter focuses on the stuck-at-zero (SA/0) and the stuck-at-one (SA/1) fault models, particularly for use with logic-level structural models. Other, more complex models are appropriate for more detailed electronic hardware models, such as switch-level models. High-level behavioral VHDL models and logic-level structural VHDL models of the hardware are considered.

8-3 TESTABILITY DESIGN ISSUES

During design for maintainability the system must be partitioned into physical components referred to as LRUs, which can be tested and replaced as necessary at appropriate logistical levels, e.g., in the field or at the depot.

Once the replaceable units have been identified, appropriate fault detection and isolation strategies must be selected and techniques chosen to implement those strategies. The testability design must provide the fault detection and isolation strategies required by the maintainability design. VHDL can provide the simulation capabilities necessary to assess the success of the fault detection and isolation strategies.

A critical issue of the testability design is what are the acceptable measures of cost and effectiveness, both for the tests themselves and the additional circuitry and equipment necessary to implement the tests. Subpar. 8-3.2 describes measures of cost and effectiveness and indicates the information needed to estimate those measurements. The resulting cost information may be back annotated into the VHDL model to allow reassessment of design decisions made as part of design for testability (See par. 8-6 for annotation.).

8-3.1 TEST STRATEGIES AND TECHNIQUES FOR MAINTENANCE AND FAULT TOLERANCE

Fig. 8-1 shows a taxonomy of design for testability strategies. This taxonomy can also be used as a decision tree for the selection of test strategies. Test strategies can be divided into on-line and off-line approaches. On-line test strategies include all strategies that allow the system to provide its normal services while testing occurs. Off-line test strategies include all strategies that require the system to suspend its normal services in order to perform test functions. On-line approaches can be divided into concurrent and background processing. Concurrent strategies are those for which the test functions are carried out concurrently with the normal functions of the test component. Background strategies are those that require the component being tested to suspend its normal processing, even though the system as a whole continues to provide its normal services. Background tests are typically scheduled when the component is idle, or they are scheduled as background tasks to be performed by the processor and compete at a low priority with other tasks assigned to the processor. Because on-line processing occurs while the system is in service, it uses primarily built-in test techniques, in which the test functions are provided by the system.

Off-line test strategies are designed to support fault detection and isolation while the system is not in service. These strategies may either employ built-in test techniques or external test equipment. Off-line, built-in test techniques are the same as those used in nonconcurrent background testing. Off-line, built-in test features of a very large-scale integrated (VLSI) circuit may be used both for off-line testing of the complete LRU and for nonconcurrent, background testing of the circuit while the rest of the LRU is still on-line. Design of test strategies for external testing depends upon the selection of the interface with external test equipment. External testing strategies also emphasize how the information collected during testing is presented to the field or depot maintenance technician.

External test strategies focus on the choice of automatic test equipment, on the partitioning of responsibilities between built-in test equipment and external test equipment, and on the interface between the built-in test (BIT) and au-

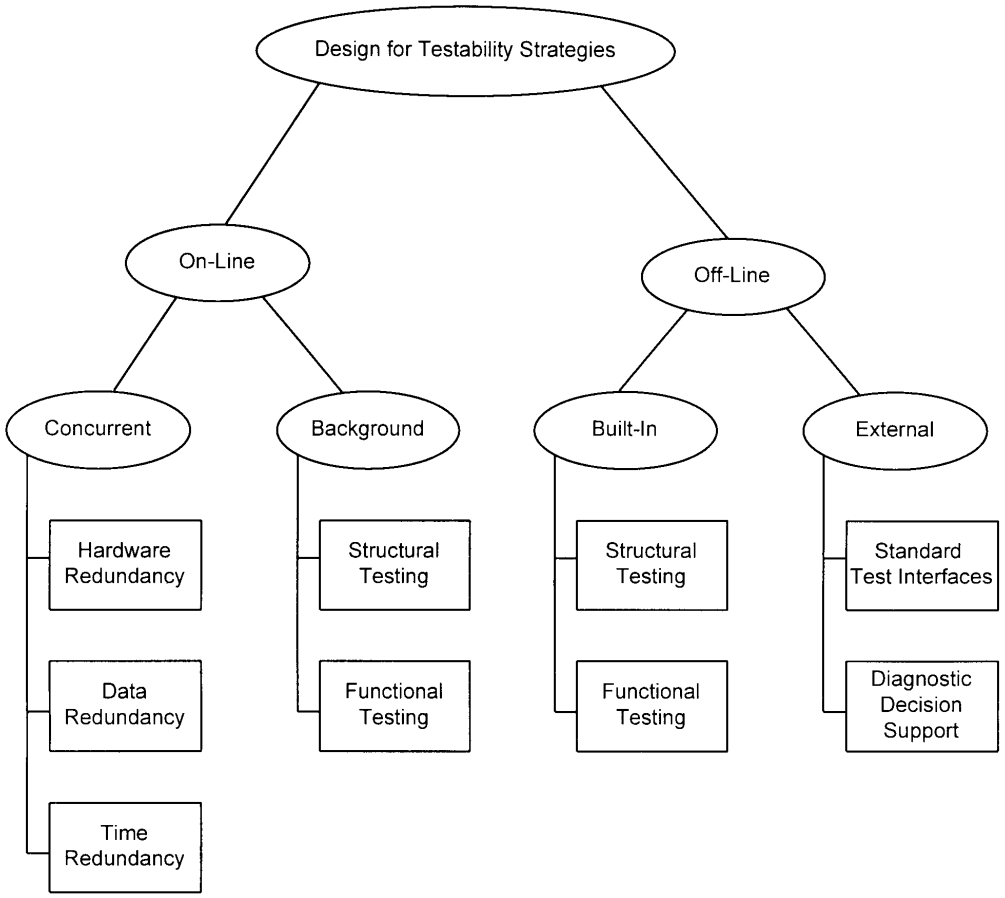


Figure 8-1. A Taxonomy of Design for Testability Strategies (Ref. 16)

tomatic test equipment (ATE). WAVES (Ref. 3) provides a means of expressing test vectors in a form that can be used by multiple ATE vendors. WAVES is also used to design VHDL test benches, and it is recommended by MIL-HDBK-454 (Ref. 4).

Diagnostic decision support strategies relate to the partitioning of responsibilities between BIT and ATE. One important aspect of this partitioning is deciding which equipment is going to log the test information and how much information is to be preserved. At the lowest level, BIT techniques such as signature analysis and circular self-test reduce the amount of data that needs to be stored. With or without data compression, a lot of information has to be maintained. Ideally, maintenance personnel would like to know everything about the state of the hardware just before the fault occurred. During design for maintainability, however, tradeoffs must be made to balance the need for this information with the space, sensors, and circuitry required to capture the information.

Different test strategies are needed for different components of a hardware system. For example, a concurrent test strategy, e.g., a coding technique such as parity or Hamming code, is typically used to test the busses and memories of an electronic system. A nonconcurrent strategy, e.g., a scan path technique such as level-sensitive scan design (LSSD), may be used for the arithmetic and logic unit (ALU) of the same system. The testability design of the system must address how these component (e.g., bus, memory, or ALU) testing techniques can be integrated into the test strategy for an entire system consisting of many components.

8-3.2 TESTABILITY MEASURES

Measures of the testability of a design are critical to ensure that cost-effective testability features have been introduced into the design. Testability measures must be assessed in terms of the following issues:

1. What is the cost of evaluating the measure?
2. What system testability techniques and strategies are effectively evaluated by the measure?
3. What types of models (particularly structural or behavioral) are required to allow accurate estimates of a measure?
4. What types of tools and procedures are currently available to evaluate a measure? What VHDL models can be used to provide this information, or which VHDL models should be annotated with analysis results from external tools?

Fig. 8-2 shows a taxonomy of testability measures. The top-level division is between performance and cost measures. The second-level split is between spatial and temporal measures, with the concept of fault isolation measures also being applied to the performance measures. Watterson et al (Ref. 5) provide more detail on the techniques used to estimate these measures.

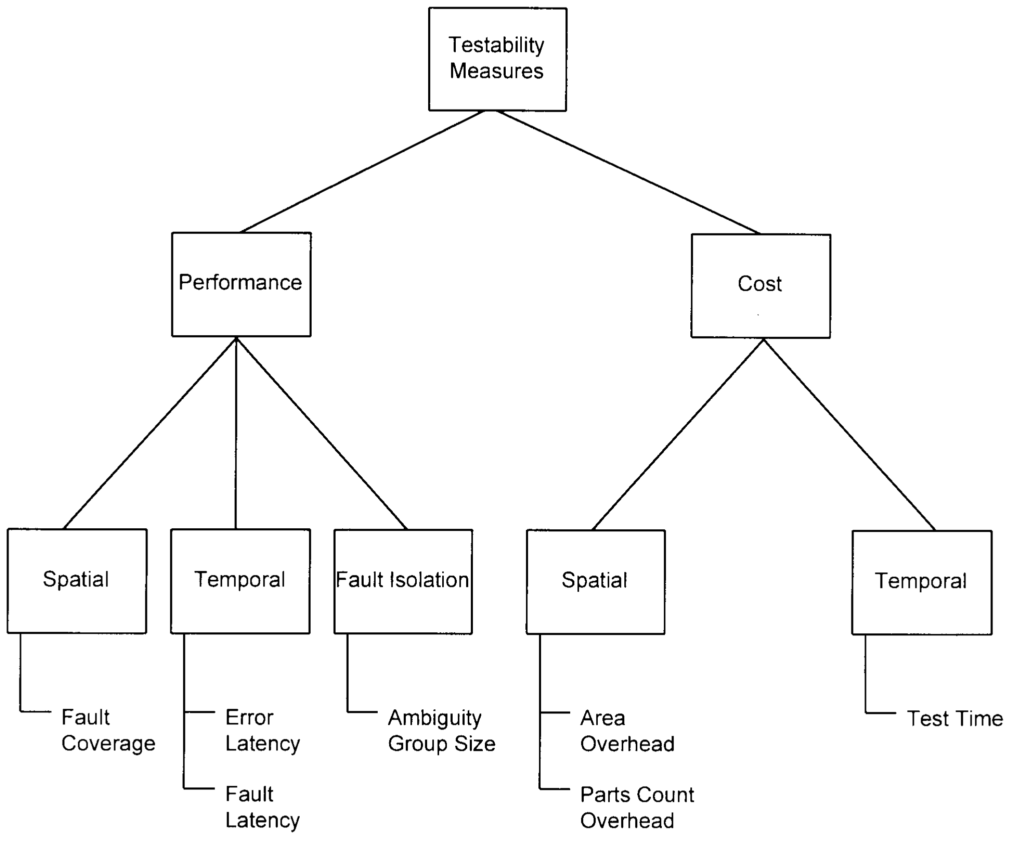


Figure 8-2. A Taxonomy of Test Measures

8-3.3 TEST STRUCTURE BOUNDARIES

Subpar. 10.2.1 of the VHDL DID (Ref. 1) requires the hierarchy of VHDL design entities in a model to reflect the physical design hierarchy of the hardware being modeled. Five levels in a typical physical design hierarchy are shown in Table 8-1: system/subsystem, LRU, fault containment region (FCR), board, and integrated circuit. The LRU level is the critical physical partitioning from the maintenance point of view. Levels above the LRU are involved in diagnosing problems within LRUs and with isolating faults to specific LRUs. Fault containment regions are not necessarily physical boundaries, but they are important electrical boundaries for fault-tolerant systems. Table 8-1 describes the test functions associated with each of these levels and the corresponding test components and their interfaces and provides a list of references for information on test techniques used at these levels and information on corresponding test components and interfaces. The test components and interfaces described in Table 8-1 are also shown in Fig. 8-3.

Because fault containment regions require a concurrent testing or fault-masking strategy, fault containment is typically implemented in the hardware with software support. As a result of this implementation the design of the hardware requires tradeoffs between the size of the fault containment region, the test time, and the area overhead for fault containment. High-level structural VHDL models can be used to capture the data required for such analyses. High-level behavioral models that provide timing information can be used to assess the concurrent testing or fault-masking overhead and its impact on system performance.

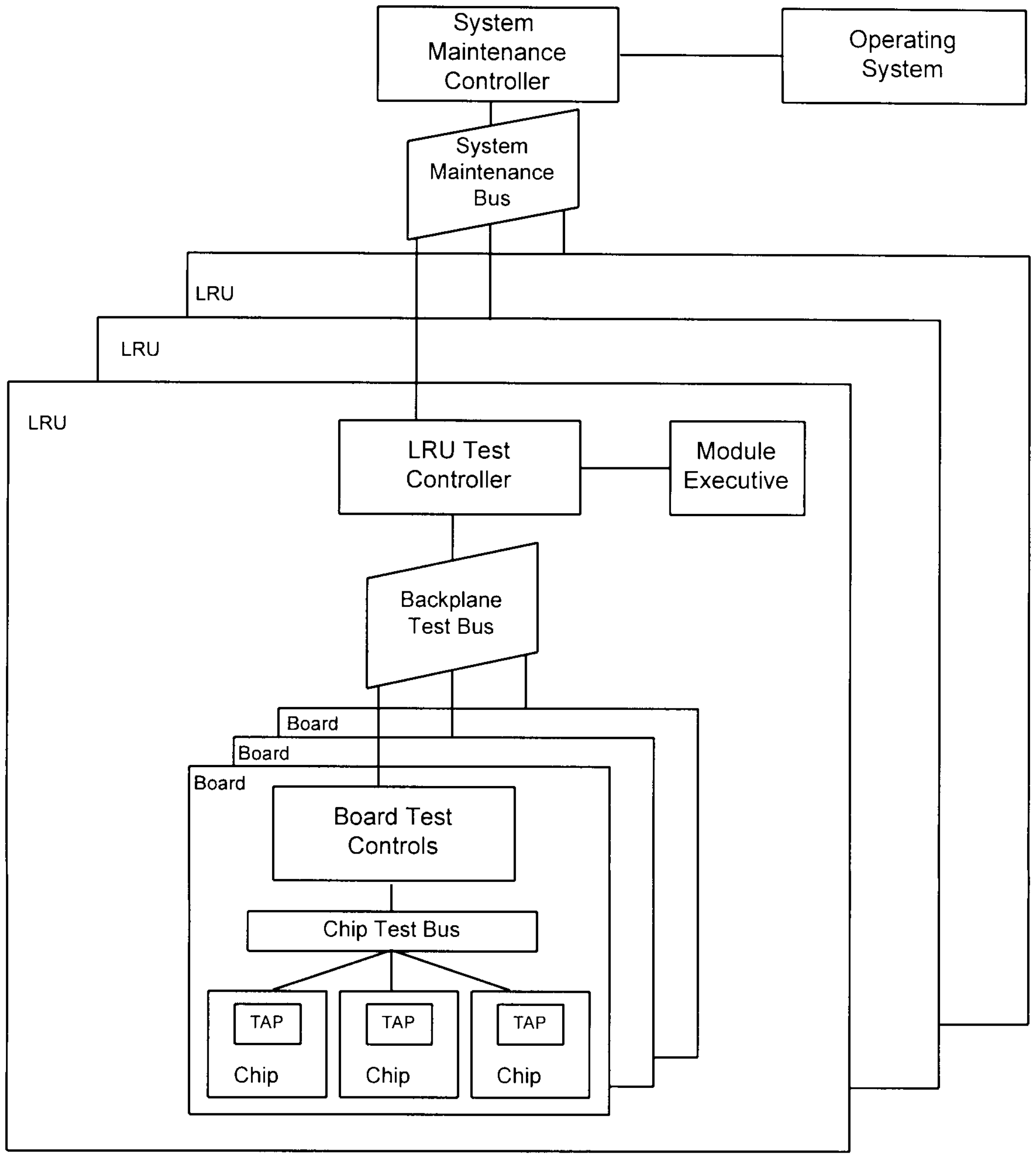


Figure 8-3. A Hierarchy of Test Controllers and Busses

Table 8-1. TESTABILITY FUNCTIONS, COMPONENTS, AND INTERFACES FOR A PHYSICAL DESIGN HIERARCHY

PHYSICAL DESIGN HIERARCHY LEVEL	FUNCTIONS	TYPICAL IMPLEMENTATION APPROACH (SW vs HW)*	RELATED TEST COMPONENTS AND TEST INTERFACES	REFERENCES
System and/or Sub-system	1. Error logging 2. Communication with external ATE 3. Support for system reconfiguration 4. Management of spare LRUs 5. Scheduling of test functions 6. Management of LRU test sets 7. Interpretation of LRU test results	High-level software on a general-purpose processor	System maintenance controller System maintenance bus	(Ref. 6) (Ref. 7) (Ref. 8) (Ref. 9) (Ref. 2)
Line-Replaceable Unit (LRU)	1. Communication with system maintenance controller 2. Storage of LRU status 3. Management of spare components 4. Fault isolation to LRU components 5. Management of component tests 6. Management of component test data 7. Interpretation of component test results	Microcode software on dedicated hardware (e.g., microcontroller)	LRU test controller Backplane test bus controller	(Ref. 10) (Ref. 11)
Fault Containment Region (FCR)	1. Fault containment 2. Error reporting	Dedicated hardware with software monitoring	Voter Error-correcting coder/decoder Parity/error code lines on busses	(Ref. 7) (Ref. 9) (Ref. 2)
Board	1. Communication with LRU controller 2. Management of spare ICs on board 3. Isolation of faulty ICs 4. Management of IC tests 5. Management of IC test data 6. Interpretation of IC test results	Microcode software on a dedicated controller Boundary scan paths	Board test controller Backplane test bus (1149.5) Chip test bus (1149.1)	(Ref. 12) (Ref. 13) (Ref. 14) (Ref. 10)
Integrated Circuit	1. Communication with board controller 2. Fault detection 3. Fault containment 4. Fault masking	Dedicated controller Boundary scan paths signature analysis	Test access port (TAP) Chip test bus (1149.1)	(Ref. 15) (Ref. 16) (Ref. 5) (Ref. 17)

*SW = software
HW = hardware

8-3.4 TEST COMPONENTS AND INTERFACES

Fig. 8-3 shows a typical test system hierarchy. This hierarchy follows the format introduced by Lien and Breuer (Ref. 11). It addresses four levels in the physical design hierarchy: system/subsystem, LRU, board, and integrated circuit. Three levels of test busses are identified: system maintenance bus, backplane test bus, and chip test bus. The software/hardware interaction at the system maintenance controller is indicated by the connection between the operating system and the system maintenance controller. A similar relationship is shown between the LRU module executive and the LRU test controller.

8-4 MODELING TESTABILITY USING VHDL BEHAVIORAL MODELS

The Department of Defense (DoD) requires VHDL models to be delivered with certain hardware components. This requirement encourages effective reuse of those components in DoD systems and allows replacement of those components as technology advances. The DoD also views VHDL as a formal description of the design of a hardware system. This paragraph discusses the possible roles of VHDL models in supporting design for testability as the design evolves.

Subpar. 10.2.3 of the VHDL DID requires behavioral models of hardware to model the diagnostic and test functions of the hardware. This paragraph describes capabilities that can be included in behavioral VHDL models responsive to the DID.

8-4.1 EVALUATING TEST STRATEGIES

Use of VHDL to explore design for testability tradeoffs is still a research area. However, many VHDL capabilities make it an attractive tool for supporting such tradeoffs. During the hardware design process, VHDL models can be used to explore possible partitionings of systems into LRUs and fault containment regions. A scenario for this use of VHDL is to generate a VHDL model reflecting a partitioning and then to evaluate the testability effectiveness measures (such as coverage) and the testability cost measures (such as area overhead, interconnect overhead, and test time) for that model. If the evaluation produces results that do not meet the effectiveness or cost requirements, the partitioning is revised and the process is repeated. Evaluation of coverage may not be possible during the early stages of design. However, the time required to execute the tests could be evaluated with high-level behavioral VHDL models if the number and lengths of test vectors are known or estimated. Interconnect overhead and parts count overhead can be extracted from a board-level VHDL structural model; area overhead is more difficult. If the hardware system consists primarily of existing components, information about coverage for the components, information about number and sizes of test vectors for the components, and information about area overhead for testing of the components can be used to evaluate different ways of organizing the testing process.

Developing a test strategy involves dividing the system into partitions, each of which uses a single test strategy. The appropriate test strategy for each partition is then selected. A decision process for determining a test strategy can be described in terms of answering a series of questions:

1. How much fault tolerance is required in the partition? (The answer will determine the minimum level of concurrent BIT required for the partition.)
2. Which of the test and diagnosis functions of the partition will be performed internally by BIT, and which of these functions will be performed externally by ATE?
3. What test mode will be used: concurrent or nonconcurrent?
4. What type of redundancy will be used for a concurrent mode partition: data, hardware, or temporal?
5. What type of test set will be used for a nonconcurrent mode partition: functional, deterministic, or pseudorandom?
6. Which specific test techniques will be used to implement the strategy that has been selected?

The results of these decisions can be captured in a VHDL model, and tradeoffs can be made by analysis of information in the VHDL model and by simulation.

Test functions at the system/subsystem level are usually performed by a combination of software and hardware. The need to use both software and hardware together poses a problem for behavioral VHDL models. Algorithmic VHDL models can be used to demonstrate system FDIR concepts during the early stages of system design. Performance models for reconfiguration may be needed for fault-tolerant systems (Refs. 8 and 18). Algorithmic VHDL models can be used as part of system interface models to demonstrate that

1. Faults detected by BIT can be logged.
2. Faults detected by BIT can be used by FDIR.
3. FDIR can be performed within the time constraints imposed by fault tolerance requirements.
4. Hardware fault containment functions provide the necessary warnings about errors and perform appropriate fault-masking functions.
5. System controllers can configure subsystems for nonconcurrent background tests. When background tests are run on a module or subsystem, their execution should not corrupt the state of the system, i.e., the module under test (MUT) should be isolated from the rest of the system for testing. System controllers are also responsible for resetting the MUT to an appropriate state to continue processing after nonconcurrent background tests.

Building test benches that demonstrate these fault detection and fault tolerance functions also clearly defines the role the executive software of the system must play in providing system-level fault tolerance, testability, and error containment.

An important benefit achieved using behavioral VHDL models and simulation is verification that the testability concept of a system is correct, i.e., that

1. Faulty components can be isolated and their faults contained.
2. Spare components can be configured into the system.
3. Test vectors can be stored in the system as required and can be distributed to the components that use those tests.
4. Test results are communicated to the system maintenance controller or to ATE.
5. Appropriate error logs are maintained.

For VHDL behavioral models of existing hardware, this handbook recommends that the model support this functionality.

8-4.2 MODELING TEST INTERFACES IN VHDL

Behavioral models should be able to verify that when properly interconnected, their test busses and interfaces are able to communicate correctly. For example, test busses such as the Institute of Electrical and Electronics Engineers (IEEE) 1149.1 and 1149.5 specify a set of instructions to be communicated through those busses. Behavioral models of modules should be able to interpret those instructions and respond appropriately.

A chip test bus is the interface between the board or LRU controller and the VLSI circuits. IEEE Std P1149.1 (Ref. 19) defines this interface, as well as a standard set of components. The bus provides a serial port for loading test vectors from the controller into a chip and for unloading test results from the chip to the controller. The serial path minimizes the number of pins on the chip dedicated to test functions. Behavioral models of the IEEE Std 1149.1 test access port (TAP) controller exist (Ref. 20), and models are being developed for the IEEE 1149.5 backplane test bus (Ref. 10). Both of these bus structures define a set of test instructions supported by the bus and its controllers.

VHDL models can be used to evaluate different strategies for interconnecting test subsystems proposed in a hardware design. If the VHDL models of the test and maintenance busses and their interfaces include timing information, the VHDL model can be used to explore test times for different configurations of modules and test busses.

8-4.3 MODELING TEST CONTROLLER FUNCTIONS

An example of a test controller in a VLSI circuit is the IEEE Std 1149.1 TAP controller. An example of a board-level test controller is the controller for the IEEE 1149.5 backplane test bus. Both of these controllers execute a required set of functions as well as some additional functions specific to the structural implementation of the module. A behavioral model of the test controller must be able to interpret the required functions and respond appropriately.

A behavioral model should produce results that represent both correct and incorrect functioning of the model. The former is used to verify that the system-level diagnostic and maintenance functions operate correctly with a fault-free module. The latter is used to verify that the system responds with appropriate fault isolation and reconfiguration commands to a faulty module. One mechanism that can be used is to load test responses from an auxiliary file. The response of the MUT to a specific test is ascertained by table lookup.

At the highest levels of the hardware design hierarchy, the test control functions may be implemented by software that is part of the normal function of the module. Modeling these functions in VHDL is beyond the scope of the VHDL DID (Ref. 1). In this case, it is very valuable for a VHDL model of the fault detection, isolation, reconfiguration, and recovery process to be generated from a software or system-level description to verify the system has the desired fault tolerance.

Behavioral VHDL models of the IEEE Std 1149.1 TAP exist (Ref. 20), and computer-aided design (CAD) tool vendors are beginning to generate test structures that are compatible with the 1149.1 standard (Refs. 21, 22, 23, and 24).

8-4.4 EVALUATION OF TEST COMMUNICATION AND STORAGE REQUIRE MENTS FOR BIT

One important aspect of design for testability that is measurable using behavioral VHDL models is the storage requirements for test vectors and error logs that must be maintained by the hardware system. Error logs are registers accessible to the user and therefore must be represented in the behavioral model of the component. If test vectors are downloaded and stored inside the hardware system, memories used to store the test vectors must also be modeled.

If test vectors and error logs are included in the behavioral models and the test access ports on the devices are also modeled, the times required to load and run the tests and extract the results can be analyzed. This analysis is useful when evaluating proposed test systems.

8-5 MODELING TESTABILITY USING VHDL STRUCTURAL MODELS

8-5.1 DESCRIPTION OF TEST CIRCUITRY GENERATED FROM STRUCTURAL INFORMATION

Subpar. 10.2.4 of the VHDL DID (Ref. 1) requires that structural VHDL models delivered to the Government must include test circuitry. This circuitry includes scan paths such as the data registers in an IEEE-Std-1149.1-compatible design. Several commercial computer-aided engineering (CAE) tools now automatically generate scan path circuitry when given a gate-level circuit and partitioning assistance. Other forms of generated test circuitry include built-in logic blocks, pseudorandom test vector generators, and test signature analyzers. Description of such circuitry is essential for logic-level fault modeling and for test vector generation.

8-5.2 SUPPORT FOR FAULT DICTIONARY GENERATION

A fault dictionary is a collection of information about the potential electrical faults in a module and includes

1. The type of fault, e.g., stuck-at-zero, stuck-at-one, short, or open
2. The location of fault in the module
3. The origin of fault, e.g., predicted by the fault simulator or discovered by the test engineer.

Fault dictionaries are used to evaluate the quality of test vectors. During this evaluation process, additional information may be attached to a fault dictionary. For example, information about the test vectors that detect a given fault may be attached to the fault dictionary, e.g.,

1. The identity of the test vectors that discover the fault
2. The way that the fault manifests itself, e.g., the pin that contains the error or the change in the compressed test signature that results from the error
3. The ambiguity group associated with the particular fault and the particular test vector.

The IEEE P1029.2 committee is developing a standard fault dictionary language (Ref. 25). This language will be compatible with VHDL and with WAVES. This proposed standard is described in terms of VHDL packages and syntactic and semantic rules for analyzing such packages.

The VHDL DID requires that VHDL structural models model the physical implementation closely enough to support gate-level fault modeling. The creation and maintenance of a fault dictionary for the fault universe of the component is essential for fault modeling. A fault dictionary is separate from the VHDL structural model but uses the VHDL structural model to define the locations of the fault. The structural model must be detailed enough so each fault can be precisely located and the effects of the faults can be simulated.

8-5.3 SUPPORT FOR AUTOMATIC TEST GENERATION

In order to generate test vectors automatically, an automatic test pattern generator (ATPG) must define a fault universe and then construct test vectors covering that universe. To define a fault universe, the ATPG must have a detailed (at least gate-level) structural model of the hardware to be tested. The faults are defined as failures of the components connecting signals in the structural model.

8-5.4 SUPPORT FOR COVERAGE ANALYSIS

Coverage of a test vector set is defined as the probability that at least one vector in the set will discover a fault given that some fault has occurred. For deterministic test vector sets a test vector t is said to detect a fault f in a circuit c if when t is applied to c with fault f, the output is different from the output when t is applied to c without fault f. Under the assumptions that all faults are considered equally likely for deterministic test vector sets, the coverage of a test vector set can be estimated for a particular fault universe as the number of distinct faults detected by the test vector set divided by the number of distinct faults in the fault universe.

There are several fault simulation tools available that work with VHDL models. These tools generally work with models that assume unit delays, specific logic primitives, and specific signal values. The tool sets that contain these fault simulation tools convert the VHDL gate-level structural models into simple gate-level models using special formats and then perform fault simulation on the internal models. For example, parallel fault simulation runs many test vectors through the same circuit at the same time. By packing 32 cases into a single word and using the wordwide bit vector logic available in most ALUs parallel simulation can provide speedups of close to 32 over sequential fault simulation.

8-5.5 SUPPORT FOR TEST TIME COMPUTATION

Test time is a critical factor in real-time, fault-tolerant systems and is a significant factor in system availability. VHDL gate-level structural models can provide important information related to computing the test time. For example, the test time for a component using scan paths is a function of the product of the number of test vectors and the lengths of the scan paths. If VHDL structural models are used to create the test vectors for the system, the auxiliary files containing test vectors should be included with the VHDL deliverables. These auxiliary files can be used to determine the number of test vectors. The VHDL structural models contain the scan paths, so they can be used to determine the number of cells in the scan paths. Also VHDL gate-level structural models can be simulated to provide information about the test time for a hardware module.

8-6 ANNOTATION OF VHDL MODELS WITH TESTABILITY INFORMATION

8-6.1 ANNOTATION OF STRUCTURAL MODELS TO IDENTIFY LRUs

The concept of a line-replaceable module is a critical element of a system logistics strategy. An LRU is a physically separate element; therefore, as required by the VHDL DID (in subpar. 10.2.1), each LRU must be represented as a separate design entity. This handbook recommends that each design entity that represents an LRU be so annotated. This annotation could be a comment in the entity interface, but making the annotation into an attribute could provide future support for computer-assisted analysis such as LRU counts or costing analysis. Also this annotation provides important design information and should discourage anyone redesigning or modifying the system from making changes that could compromise the ability of the test system to isolate faults to the LRU.

8-6.2 ANNOTATION OF STRUCTURAL MODELS TO IDENTIFY FCRs

The concept of fault containment regions is critical to fault-tolerant systems since they mark boundaries where mechanisms have been placed to prevent errors from propagating out of the FCR. Unlike LRUs, there is no DID requirement that an FCR represent a separate physical entity.

Because fault containment regions require a concurrent testing or fault-masking strategy, fault containment is typically implemented in the hardware with software support. This means that the design of the hardware requires tradeoffs between the size of the fault containment region, the test time, and the area overhead for fault containment. High-level structural VHDL models can be used to capture the data required for such analyses. High-level behavioral models that provide timing information can be used to assess the concurrent testing or fault-masking overhead and its impact on system performance.

This handbook recommends that signals or entities representing the boundary of an FCR be annotated to specify that the signal or entity represents an FCR boundary. It may also be possible to define assertions to verify that faults are contained. These tests provide important design information that can reduce the risk that a redesign will inadvertently damage the fault containment capabilities of the system.

8-6.3 BACK ANNOTATION WITH COVERAGE INFORMATION

The engineer using a VHDL model as the starting point for the redesign of a system requires information about the cost and effectiveness of testability.

Because coverage information is a combination of information about the module under test and the test vectors, it should be included in the fault dictionary. The VHDL DID does not specifically require delivery of a fault dictionary. However, it does require delivery of both the VHDL model that describes the MUT and the test bench needed to drive the VHDL model, including the test vectors. This handbook recommends that the fault dictionary, including information about test vector coverage, be included in the delivery package. This information should include the target fault models and coverage information for each LRU.

REFERENCES

1. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, Department of Defense, Washington, DC, 11 May 1989.
2. D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Equipment Corporation, Bedford, MA, 1982.
3. IEEE Std 1029.1-1991, *Waveform and Vector Exchange Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1991.
4. MIL-HDBK-454, *General Guidelines for Electronic Equipment*, 28 April 1995.
5. J. W. Watterson, M. Royals, and N. Kanopoulos, *Chip-Level Testability Requirements Guidelines*, Technical Report RTI/4968/01F prepared for the US Air Force, Rome Air Development Center, Rome, NY, by Research Triangle Institute, Research Triangle Park, NC, November 1991.
6. R. S. Mejzak and Lt K. T. Schmierer, "JIAWG Module Fault Coverage Metrics Methodology", *Proceedings of the IEEE/AIAA 10th Digital Avionics Systems Conference*, Los Angeles, CA, October 1991, Institute of Electrical and Electronics Engineers, Inc., New York, NY, and American Institute of Aeronautics and Astronautics, Washington, DC.
7. R. E. Harper, *Critical Issues in Ultrareliable Parallel Processing*, Technical Report CSDL-T-944, Charles Stark Draper Laboratory, Inc., Cambridge, MA, June 1987.
8. B. E. Clark, J. T. Morrison, F. G. Gray, and T. White, *A Model of the Ada Avionics Real-Time System: An Example of the Benefits of the Hardware/Software Codesign Approach in Development of Real-Time Systems*, Technical Report WL-TR-92-1022, US Air Force, Wright Laboratory, Dayton, OH, March 1992.
9. D. A. Rennels and J. A. Rohr, "Fault-Tolerant Parallel Processors for Avionics With Reduced Maintenance", *Proceedings of the IEEE/AIAA/NASA 9th Digital Avionics Systems Conference*, Virginia Beach, VA, October 1990, Institute of Electrical and Electronics Engineers, Inc., New York, NY, and American Institute of Aeronautics and Astronautics, Washington, DC.
10. P. McHugh, "IEEE P1149.5, Module Test and Maintenance Bus", *IEEE Design and Test of Computers* (December 1992).
11. J. C. Lein and M. A. Breuer, "A Universal Test and Maintenance Controller for Modules and Boards", *IEEE Transactions on Industrial Electronics* **36**, 231-40 (May 1989).
12. M. M. Pradhan, R. E. Tulloss, H. Bleeker, and F. P. M. Beenker, "Developing a Standard for Boundary Scan Implementation", *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE Computer Society Press, Los Alamitos, CA, 1987.
13. IBM, Honeywell, and TRW, "Element Test and Maintenance Bus (ETM-Bus), Preliminary Specification", *VHSIC Phase 2 Interoperability Standards*, VHSIC Program Office, Washington, DC, September 1985.
14. L. Avra, "A VHSIC ETM-Bus-Compatible Test and Maintenance Interface", *Proceedings of the IEEE International Test Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1987.
15. C. M. Maunder and R. E. Tulloss, Eds., Chapter 3, "The Test Access Port and Boundary-Scan Architecture",

The Development of IEEE Std 1149.1, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 23-30.

16. J. W. Watterson, J. J. Hallenbeck, G. A. Frank, D. L. Franke, and J. B. Clary, *Tools and Techniques for Assessment of VHSIC On-Chip Self-Test and Self-Repair*, Technical Report RTI/2086/01-1F, by Research Triangle Institute, Research Triangle Park, NC, for US Air Force, Rome Air Development Center, Griffiss Air Force Base, NY, February 1985.
17. IBM, Honeywell, and TRW, "TM-Bus Specification," *VHSIC Phase 2 Interoperability Standards*, V. 3.0, VHSIC Program Office, Washington, DC, November 1987.
18. M. J. Strickland and D. L. Palumbo, "Fault Tolerant System Performance Modeling," *AIAA/AA/ASEE Aircraft Design, Systems and Operations Conference*, American Institute of Aeronautics and Astronautics, Washington, DC, 1988.
19. IEEE Std. 1149.1-1990, *IEEE Standard Test Access Port and Boundary-Scan Architecture*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1990.
20. P. M. Campbell, M. Vai, and Z. Navabi, "Implementation of IEEE Std 1149-1-1990 in VHDL", *Using VHDL in System Design, Test, and Manufacturing*, Proceedings of the VHDL International Users' Forum Spring Conference, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.
21. Mentor Graphics Corporation, "Mentor Graphics Systems—1076", *Supplier Directory*, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA, May 1992.
22. Gen Rad, Inc., "System HILO 4: HiDesignA", *Supplier Directory*, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA, May 1992.
23. DAZIX, An Intergraph Company, "DAZIX Synthesis Toolset: High-Level Implementation Tools for Today's Top-Down Design", *Supplier Directory*, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA, May 1992.
24. Racal-Redac, "SilcSyn VHDL Synthesis", *Supplier Directory*, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA, May 1992.
25. K. J. Parella and A. Wilmot, "Fault Detection and Localization", *Using VHDL in System Design, Test, and Manufacturing*, Proceedings of the VHDL International Users' Forum Spring Conference, Scottsdale, AZ, May 1992, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

BIBLIOGRAPHY

W. C. Carter, J. R. Dunham, J. Laprie, T. Williams, W. E. Howden, and B. Smith, *Design for Validation: An Approach to Systems Validation*, Final Report, Task Assignment No. 7 NAS1 17964, Research Triangle Institute, Research Triangle Park, NC, 1987.

J. B. Clary, R. K. Joobbani, and F. M. Smith, *Development of a Methodology for Verifying Military Computer Family Built-In Test Performance Specifications*, Research Triangle Institute, Research Triangle Park, NC, September 1980.

SDIO BM/C^3 Processor and Algorithm Working Group, *Application of Fault Tolerance Technology; Volume I: Design of Fault-Tolerant Systems; Volume II: Management Issues, Contractor Milestones and Evaluation; Volume III: Tools for Design and Evaluation of Fault-Tolerant Systems; Volume IV: System Security and Its Relationship to Fault Tolerance*, Rome Air Development Center, Rome, NY, October 1987.

L. S. DeBrunner, F. G. Gray, and R. L. Baker, "A Methodology for Comparing Fault-Tolerant Computers", *Proceedings of the AIAA/IEEE 8th Digital Systems Avionics Conference*, San Jose, CA, 17-20 October 1988, American Institute of Aeronautics and Astronautics, Washington, DC, and The Institute of Electrical and Electronics Engineers, Inc., New York, NY.

"Test: Faster, Better, Sooner", *Proceedings of the 1991 IEEE International Test Conference*, Nashville, TN, October 1991, IEEE Computer Society Press, Los Alamitos, CA.

E. Yourdan, *Design of On-Line Computing Systems*, Prentice-Hall International, Englewood Cliffs, NJ, 1972.

CHAPTER 9
PREPARATION OF VHDL MODELS FOR DELIVERY TO THE DoD

This chapter describes the preparation of a VHDL model for delivery to the Government. The contents and organization of the files delivered to the Government as specified in the VHDL DID are described. The files that must be delivered include not only the VHDL source programs but also test vectors, annotations, other external files, and documentation. This chapter also recommends VHDL naming conventions.

9-1 INTRODUCTION

If very high-speed integrated circuit (VHSIC) hardware description language (VHDL) models are to serve their intended purpose, they must be delivered to the Government in a consistent form and with all supporting documents and files. As a design progresses, a series of models is delivered to the Government; each model represents a refinement of the previous version. The final, delivered model represents the final hardware design. Each of these models is delivered to the Government in accordance with the data item description (DID).

Par. 10.3 of the VHDL DID (Ref. 1) requires that all information required to document and test a model must be delivered with the model. This information includes textual documentation, supporting VHDL model libraries, test benches, test data, and the model itself. The DID requires that these items be packaged into text files and delivered on magnetic tape. Because of the variety of magnetic tapes in existence, the Government may wish to tailor the DID to specify an alternate tape size and format, e.g., 8-mm Unix tar tape, from that stated in the DID.

The DID specifies the contents of the files on the tape, but it does not specify which, if any, higher level file structure (such as file names and directory structures) should be included. Thus, if the files are delivered on unlabeled ASCII magnetic tapes as recommended by the DID, no file- or directory-name information is included. Therefore, it is not possible to connect the contents of a file directly to its original host file name. Even though the VHDL DID requires that a list of host file names be provided, without some convention it is not possible to associate this host name with a given VHDL source file on the tape, as par. 10.3 of the VHDL DID also requires.

Par. 7.3 of the VHDL DID allows the Government agency receiving the models to specify the machine format of the tape for the deliverables so directory structures and names can be negotiated. Because of the generality of the DID format specification, it is recommended that each VHDL source file contain its host file name as a comment at the top of the file. The host file name should specify the directory hierarchy above the file to the level of directories included in the tape. For example, consider a directory (called `model`) containing four subdirectories:

1. A utilities directory, `utilities`
2. A structure directory, `structure`
3. A leaf cell directory, `leaf_cells`
4. A component specification directory, `comp_lib`.

Then a file named `memory_arch.vhdl`, which contains the architecture body for a memory component (one of the leaf cells), would have the comment

```
File: model/leaf_cells/memory_arch.vhdl
```

VHDL specifies the library as the unit of organization for VHDL models; VHDL source files are analyzed into these libraries. Although the VHDL DID does not explicitly require specification of the library structure, the names of the libraries and the library units they contain need to be provided in the documentation so that a model can be successfully recovered from the contents of the tape. It is recommended that the directory structure reflect the VHDL library structure, so all files containing VHDL source code for units in a particular library should be in the same directory. It is also recommended that each VHDL source file contain the name of the library into which it should be analyzed as a comment at the top of the file. To continue the example, suppose that the directory structure described reflects the target library structure; the VHDL model includes four libraries:

`Utilities, Structure, LeafCells,` and `CompLib`

Therefore, the source code file named `memory_arch.vhdl` would have the comment

```
Library: LeafCells
```

The Waveform and Vector Exchange Specification (WAVES) header file provides information about the relationship between VHDL units, the files that contain their source code, and the VHDL libraries in which they reside. For each file delivered with the model, there is a specification of the type of the file (either external, WAVES standard, or VHDL) and a specification of the associated library. An example of a WAVES header file is shown in Fig. 7-3.

These recommendations require tailoring of the VHDL DID. Additional recommendations relating to file-, entity-, signal-, port-, and package-naming conventions are included in this chapter and also require tailoring of the VHDL DID should the Government want to specify these details. If the Government anticipates a need for interoperability between components of the model or use of packages in other models, these issues should be spelled out in the tailored DID.

9-2 FILES TO BE INCLUDED IN DELIVERY TAPE

Par. 10.3 of the VHDL DID requires files to be partitioned into two basic types: VHDL design files conforming in all respects to the *VHDL Language Reference Manual* (Ref. 2) and other files containing auxiliary information needed to document various aspects of the design and its operation. Par. 7.3 of the VHDL DID also describes the technical requirements for the layout of the delivery tape.

In general, it is recommended that only one design unit be included in each VHDL file; however, there are exceptions that should be made. For small packages the package declaration and the package body may be included in the same file, particularly if few changes in the package body without corresponding changes in the package declaration are anticipated. All design units in the same file should belong in the same VHDL library.

Par. 10.3 of the VHDL DID requires that design units new to a delivery not be contained in a file that has been previously delivered to the Government. The VHDL DID requires that the files be delivered in a specific order as described in the following subparagraphs.

The Navy Technology Independent Representation of Electronic Products (TIREP) project (Ref. 3) has produced a complete example of a VHDL model and its supporting information. The TIREP VHDL model conforms to the requirements of the VHDL DID and makes use of WAVES (Ref. 4). It also applies an early version of Electronic Industries Association (EIA) 567 (Ref. 5), and makes recommendations for changes in EIA-567 (Ref. 6).

9-2.1 LIST OF FILES

Subpar. 10.3.a of the VHDL DID requires that the first file included on the delivery tape contain, in order, the names of all of the files included in this delivery. Each file name should be followed by the name of the design unit contained in the file, and there should be one record or line for each file. It is recommended that the order in which the files are listed match the order in which the files occur on the tape.

This handbook recommends that this file be given an easily recognized name when practical, e.g., `<model_name>.toc`, for this table of contents. This convention makes it easy to find the file after it has been copied from the tape into the host file system.

9-2.2 DID OVERVIEW FILE

Subpar. 10.3.b of the VHDL DID requires the second file on the tape contain a high-level, prose overview of the nature of the VHDL model. This overview should cite (1) the contract number that required the development of the model, (2) the contract line item, and (3) the contract data requirements list (CDRL) sequence number. The overview should also summarize the organization and content of the set of files.

Information on the purpose, level of abstraction, or any issues related to fidelity or other special considerations or limitations of the model, should also be included in this file. If the models use special or unusual algorithms, either a discussion of their operation should be included or a reference should be made to a readily available report or other document describing the algorithm.

It is recommended that information about the library structure be included in this overview to explain the rationale behind the partitioning of design units into libraries. This handbook also recommends that this file be given a name that reflects the purpose of the file. For example, the UNIX tradition is to call such a file `README`. Other choices for this file name are `<model_name>.cdrl` or `<model_name>.readme`.

9-2.3 VHDL ANALYSIS ORDER SPECIFICATION

Subpar. 10.3.c of the VHDL DID requires that the third file on the tape describe the required order of analysis of the files included in the delivery.

This handbook recommends that when practical, the file specify the name of the library in which a VHDL design unit should be stored. The WAVES header file (Ref. 4) is an example of a file format that meets this recommendation. An example of a WAVES header file is shown in Fig. 7-5. The WAVES header file has three columns. The first column specifies the type of file: `WAVES_FILENAME` (i.e., a test bench component), `WAVES_UNIT` (i.e., a WAVES standard file), or `EXTERNAL_FILENAME` (i.e., an auxiliary file). The second column gives the file name. The third column specifies the VHDL library into which the design unit contained in the file is to be analyzed.

A good naming convention helps to verify that the analysis order is correct, e.g., that package declarations are analyzed before package bodies. A suggested naming convention is described in par. 9-3.

It is also recommended that the file describing the order of analysis be given a name that reflects the purpose of the file, e.g., `<model_name>.order`.

9-2.4 GOVERNMENT-APPROVED LEAF MODULE VHDL DESCRIPTIONS

Subpar. 10.3.d of the VHDL DID requires that the fourth file on the tape contain a list of the VHDL leaf-level design entities used in the model that are supplied by or approved by the Government. This list should include the name of the Government organization supplying (or authorizing the use of) each design entity. This name should be first in the file and should include enough information to enable a future user to contact the supplying organization, if necessary. The supplying organization may authorize the use of files without actually supplying the VHDL source code; in such cases the supplying Government organization should be listed first in the file, and the sources of the files used as leaf modules should be specified in the actual files.

The VHDL Model Library being developed by the Defense Electronics Supply Center (DESC) is a potential source for leaf-level design entities. It should be contacted for information on available models and the standards required to support interoperability with models in their library. Subpar. 4-2.3 describes the DESC efforts to develop this library. Par. 4-2 provides more information on the types of models required by MIL-HDBK-454 (Ref. 7). These models are candidates for leaf-level entities to be supplied by the Government.

Subpar. 10.2.8 of the VHDL DID requires the following documentation for design units used in the model but not developed as part of the effort:

1. Identification of originator or source
2. Department of Defense (DoD)-approved identifier (if one exists)
3. The design unit name
4. The design unit revision identifier.

The primary consideration for the definition of libraries should be configuration management, particularly in terms of who has read and write privileges for it. Each library should have a single organization responsible for it. If a VHDL model is being developed by a team, each library under construction should have one person responsible for its contents. A secondary consideration is the cohesiveness of the units in a library. The goal is to limit the number of design units that have to access the library, to ensure that the design units that access a library use most of the units in the library, and to be able to describe the contents of the library succinctly.

A Government agency should supply acceptable leaf-level design entities and any appropriate VHDL packages already organized into libraries. The agency may specify commercial standard packages, e.g., Institute of Electrical and Electronics Engineers (IEEE) 1164, that are provided as part of commercial tool sets. The description should include the host file name if the models were supplied in source form. The description should also include the name of the library unit contained in the file, its classification (i.e., package declaration, package body, entity declaration, architecture body, configuration declaration), and some indication of its revision level, if available.

It is recommended that some file, preferably this one, include descriptions of all VHDL libraries containing library units either not developed by the contractor or maintained by some organization other than the developer. Such libraries include libraries containing standards such as IEEE Std 1164 (Ref. 8), WAVES (Ref. 4), or EIA-567 (Ref. 6).

It is also recommended that this file be given a name that reflects its purpose, e.g., `<model_name>.leaf`.

9-2.5 REVISED VHDL MODULE LIST

Subpar. 10.3.e of the VHDL DID requires that the fifth file on the tape contain a list of the VHDL design units that are revisions of design units previously delivered to and accepted by the Government.

As a model is refined during the design cycle, it is necessary to deliver revisions of design units previously accepted by the Government. Whenever possible, the file name for these revised design units should be the same as the name of the previously delivered version. For example, if an architecture body is modified but the entity interface is not changed, the body retains the same design unit name and should keep the same file name that was used when it was delivered previously. The text of the modified design unit should include additional information about the revision, as described in subpar. 9-2.9.

It is also recommended that the file describing the revised design units be given a name that reflects the purpose of the file, e.g., `<model_name>.revised`.

9-2.6 ORIGINAL VHDL MODULE LIST

Subpar. 10.3.f of the VHDL DID requires that the sixth file on the tape contain a list of VHDL source code design units newly created for this delivery. Par. 10.3 of the VHDL DID also requires that these VHDL design units be placed in files other than those containing design units previously accepted by the Government.

It is recommended that the list of design units be grouped by libraries. It is also recommended that the list of VHDL design units have a description for each design unit. This description should include the host file name, the name of the design unit, the class of the design unit (i.e., package declaration, package body, entity declaration, architecture body, or configuration declaration), and some indication of its revision level or history. Further, it is recommended that the file describing the original design units be given a name that reflects the purpose of the file, e.g., `<model_name>.original`.

9-2.7 TEST BENCH CORRELATION LIST

Subpar. 10.2.5.3 of the VHDL DID requires that every design entity be accompanied by an associated test bench. This association may not necessarily be one-to-one. The same test bench may be used to test several hardware module design entities, and several test benches may be required to test a single design entity fully. A test bench may consist of a hierarchy of VHDL design entities. Configuration declarations may be used to combine design entities into a test bench or to specify generic constant values. For example, a test bench may have as a generic constant the file name for the external file containing the test vectors. The same test bench runs different tests merely by changing the value of the generic constant. The different values of the generic constant may be defined in different configuration declarations.

The configuration declarations may also be used to select the architecture body for the module under test (MUT). This way, the same test bench can be used to test both the behavioral and structural models of the MUT using the same test vectors. Alternatively, the test bench may have several different architecture bodies that are used for different tests. In this case, a configuration specification in the top-level architecture body of the test bench defines the value of the generic constant. The same configuration specification can specify which architecture body is to be used for the MUT.

A VHDL model (often a behavioral model) of a system component may be used as part of a test bench used to test another system component. For example, a test bench may use both a behavioral and a structural model in back-to-back configurations to verify that the structural model produces the same results as the behavioral model.

Subpar. 10.3.g of the VHDL DID requires that the seventh file on the tape indicate which test bench is associated with which VHDL model. This file should contain a list of pairs of names; each name should specify both a design entity and either an architecture body or a configuration declaration. The name of the test bench should specify a configuration of the root entity interface of the test bench, and the name of the VHDL model should specify a configuration of the root entity interface of the MUT model hierarchy. For example, consider a test bench for Test A of the behavioral model of a board maintenance controller for which the root entity of the test bench hierarchy is called `TestBench`, and the configuration declaration associated with the `TestBench` entity is called `TestA`. The root entity of the board maintenance controller is called `BoardMaintenanceController`, and the behavioral architecture body for this entity is called `Behavior`. A line in the association file states

```
   TestBench(TestA) tests
   BoardMaintenanceController(Behavior).
```

It is recommended that each of these pairs be accompanied by a description of how the test data for the test bench were generated (e.g., internally generated, WAVES test data, or other external files), how test bench options and parameters affect the nature and scope of testing, and any assumptions or requirements needed to operate the test benches. This description might include assumptions on the location of test data files or other operating-system-specific requirements.

It is also recommended that the seventh file be given a name that reflects the purpose of the file, e.g., `<model_name>.test`.

9-2.8 AUXILIARY INFORMATION FILES

Subpar. 10.3.h of the VHDL DID requires that the files following the seventh file be the auxiliary files and VHDL source files. The auxiliary files should precede the VHDL files. The auxiliary files include supporting files such as test data machine code for programmable processor models, other memory initialization data, and environmental parameter data. Fig. 7-5 shows an example of a WAVES external file, which is an auxiliary file. It is recommended that the file name for an auxiliary file indicate that the file is an auxiliary file, not a VHDL source file. Use of an appropriate file name suffix, such as `<file name>.dat` or `<file name>.ext`, is recommended. In particular, the extension `vhdl` or `vhd` should not be used for auxiliary files.

ASCII format auxiliary files are preferred because these files are portable from one VHDL environment to another. If the model developer is creating a new format for an external file, (particularly ASCII files, which can be read using TEXTIO), the formats should allow comments such as the header comments in the WAVES external file shown in Fig. 7-5. Performance reasons and file sizes may force the use of non-ASCII files. For example, synthetic aperture radar files can require several hundred megabytes of data for a small number of frames. Using the TEXTIO capabilities of VHDL may pose a considerable performance burden for large data files; therefore, the model developer may prefer to use the implementation-dependent binary file I/O built into the language. Use of this may save time and space at the cost of portability. Thus the model developer must supply a mechanism to convert the non-ASCII files into a usable format. The VHDL DID does not require ASCII auxiliary files. The VHDL DID must be tailored to specify a mechanism that ensures the portability of auxiliary data.

9-2.9 VHDL DESIGN UNIT FILES

Subpar. 10.3.h of the VHDL DID requires that VHDL source code files follow the auxiliary files on the tape. These files should contain all the new and revised VHDL design units as identified in the fifth and sixth files described in subpars. 9-2.5 and 9-2.6.

The VHDL model verification procedure (Appendix B and Ref. 9) recommends that each design unit, i.e., entity declaration, architecture body, package declaration, package body, and configuration declaration, contain a header file including the following information:

1. The design unit name
2. The design unit revision identifier (e.g., Version 2.3)
3. The design unit file name
4. Identification of the originator or source of the VHDL including both individual and organization
5. DoD-approved identifier for the design unit, if one exists (e.g., the contract data requirements list (CDRL) data item number).

These recommendations are consistent with the requirements stated in subpar. 10.2.8 of the VHDL DID.

Subpar. 10.2.7 of the VHDL DID requires that the following documentation be included in explanatory comments augmenting the formal VHDL text:

1. Any factors restricting the general use of this description to represent the subject hardware
2. General approaches taken to modeling, particularly decisions regarding model fidelity
3. Any additional information the originating organization considers vital to subsequent users of the descriptions.

These comments are intended to clarify the intent of the VHDL model.

Subpar. 10.2.8.1 of the VHDL DID requires that each revised design unit have comments including the following information, which must be included for each revision:

1. The date of the revision
2. The individual and organization making the revision
3. The reason for the revision

4. Identification of the part or parts of the original design unit that changed

5. A description of the testing done to verify that the revised design unit is correct.

9-3 FILE NAMING CONVENTIONS

The purpose of file-naming conventions is to aid a user in selecting a file. In particular, the file name should give an indication of the type of VHDL design unit contained in the file and an indication of the purpose of the design unit, e.g., modeling a particular physical hardware component, standard package, or test bench module. File-naming conventions must be considered in the context of directory structures. Much of the information that could be put in the file name can be inferred from the directory path to the file; redundant information should be avoided. Some popular operating systems place limits on the length of the file name (e.g., 8 characters) and on the length of the suffix (e.g. 3 characters) and may not discriminate between upper- and lowercase letters in a file name, or they may allow only uppercase letters. In the interest of portability it is recommended that file names should not use mixes of upper- and lowercase letters. It is also recommended that directory structures be used to keep file names short.

Thus a directory structure starts with the directory for the entire library. This directory has subdirectories for packages and entities, and each design entity has its own subdirectory. The design entity subdirectory contains the entity interface declaration and multiple architecture bodies for the entities. The WAVES standard places the test bench components in different libraries than the design entity for the module under test (MUT), so test bench components are stored in a test bench library subdirectory of the design entity directory. Fig. 9-1 shows a directory structure and file names for the algorithm library (which holds an algorithmic-level model) for the sobel edge detector described in Chapter 2. The path name for the horizontal filter test bench entity interface declaration is `alg_lib/entities/h_filt/t_b_lib/iface.vhd`. The path name contains the information needed to identify the file and to work out where the unit should be placed in the VHDL library structure.

9-3.1 NAMING VHDL DESIGN UNIT FILES

The syntax for file-naming conventions used in this subparagraph is to surround the part of a name that changes on instantiation with left and right brackets. Thus the specification of `<package_name>.vhd` for a package declaration could be instantiated as `std_logic_1164.vhd`, which indicates a package declaration named `std_logic_1164`.

The VHDL model verification procedure (Ref. 9) recommends the following naming convention for files containing individual VHDL design units:

1. `<package_name>.vhd` for package declarations

```
alg_lib
        pkgs
                image.vhd
                image_b.vhd
        entities
                edge_det
                        iface.vhd
                        behave_a.vhd
                        struct_a.vhd
                        t_b_lib
                                iface.vhd
                                behave_a.vhd
                mem_proc
                        iface.vhd
                        behave_a.vhd
                        t_b_lib
                                iface.vhd
                                behave_a.vhd
                win_proc
                        iface.vhd
                        behave_a.vhd
                        struct_a.vhd
                        t_b_lib
                                iface.vhd
                                behave_a.vhd
                dir_mag
                        iface.vhd
                        behave_a.vhd
                        t_b_lib
                                iface.vhd
                                behave_a.vhd
                h_filt
                        iface.vhd
                        behave_a.vhd
                        t_b_lib
                                iface.vhd
                                behave_a.vhd
                v_filt
                        iface.vhd
                        behave_a.vhd
                        t_b_lib
                                iface.vhd
                                behave_a.vhd
                ld_filt
                        iface.vhd
                        behave_a.vhd
                        t_b_lib
                                iface.vhd
                                behave_a.vhd
                rd_filt
                        iface.vhd
                        behave_a.vhd
                        t_b_lib
                                iface.vhd
                                behave_a.vhd
```

Figure 9-1. Directory Structure and File Names for Sobel Algorithm Library

2. `<package_name>_body.vhd` for package bodies
3. `<model_name>_e.vhd` for entity declarations
4. `<model_name>_a_str.vhd` for structural architecture bodies
5. `<model_name>_a_beh.vhd` for behavioral architecture bodies

6. `<model_name>_testbench.vhd` for test bench models.

Because of the previous discussions about directory structures, it is recommended that the file names be further defined as follows:

```
1. <library_dir>/pkgs/
     <package_abrv>_p.vhd
```
for general-purpose package declarations that are applied to multiple MUTs and test benches

```
2. <library_dir>/pkgs/
     <package_abrv>_b.vhd
```
for bodies of general-purpose packages that are applied to multiple MUTs and test benches

```
3. <library_dir>/entities/
     <entity_dir>/iface.vhd
```
for entity interface declarations

```
4. <library_dir>/entities/
     <entity_dir>/<arch>_a.vhd
```
for architecture bodies

```
5. <library_dir>/entities/
     <entity_dir>/t_b_lib/<test>_i.vhd
```
for test bench entity interface declarations

```
6. <library_dir>/entities/
     <entity_dir>/t_b_lib/<test>_a.vhd
```
for test bench architecture bodies

```
7. <library_dir>/entities/
     <entity_dir>/t_b_lib/<test>_p.vhd
```
for test bench package declarations that are specific to a particular entity and test

```
8. <library_dir>/entities/
     <entity_dir>/t_b_lib/<test>_b.vhd
```
for test bench package bodies that are specific to a particular entity and test.

This handbook recommends that configuration declarations be given names of the form

```
<library_dir>/entities/<entity_dir>/
     <config>_c.vhd
```
and that test bench configuration declarations be given names of the form

```
<library_dir>/entities/<entity_dir>/
     t_b_lib/<test>_c.vhd.
```

This handbook also recommends that configuration declarations be given names of the form `<entity_name>_config_<config_name>.vhd`.

9-3.2 NAMING AUXILIARY FILES

Specific environments may have special naming conventions for simulation output files. If the contractor and the receiving Government agency have the same VHDL environment, they can simply use the naming conventions of the environment. Otherwise, it is recommended that the name `<model_name>_<test_set_name>.trace` be used for trace files. If the VHDL models use file input/output (I/O) to load tables or initialize memories, these files should be given explanatory names, such as `<model_name>_<function_name>.program` for the machine language instructions for a programmable hardware system or `<model_name>_<function_name>.table` for the table of coefficients for a function implemented by table lookup.

9-4 SUGGESTED CODING CONVENTIONS FOR VHDL MODELS

Because VHDL is a programming language intended to communicate design information, it is important that this information be presented as clearly as possible. Clarity can be improved by establishing certain programming conventions. These conventions can also help to establish that a VHDL description accurately reflects the function and structure of a hardware system.

9-4.1 DESIGN ENTITY NAMING CONVENTIONS

VHDL entity interfaces intended to represent the final form of an actual hardware system should have the same names as the actual hardware components. The VHDL name for components that have names composed of more than one word should be constructed by substituting underscores ("_") for the spaces between the individual words of the hardware component name. Alternatively, uppercase letters can be used to mark the beginnings of words. For example, a model of the board maintenance controller could be named `board_maintenance_controller` or `BoardMaintenanceController`. The VHDL initiative toward ASIC libraries (VITAL) (Ref. 10) encourages the latter approach. Different styles may be used in different parts of the model as long as a clearly defined protocol for naming is specified. For example, all uppercase letters may be used for VHDL-reserved words, underscores may be used in external standards such as IEEE Std 1164 (Ref. 8), and capitalization may be used for names created by the model builder.

VHDL entity interfaces that do not represent actual hardware should have names that are descriptive of the functions they perform.

Architecture bodies should also be descriptively named. It should be possible to tell from the name whether the body is behavioral or structural, and it should be possible to tell which implementation of the entity has been modeled, particularly if more than one implementation will reside in the library. From a configuration management viewpoint, implementation from different vendors may need to reside in separate directories.

Component instance labels should be chosen to show the connection with the parent component and should describe the role the component is playing. For example, if an architecture always uses a particular I860 for an address generator and other I860s for other purposes, the address generator role should be indicated by giving the address generator component instance the label `I860AG`. If the component instances are generated by a schematic capture tool, the schematic capture tool may have its own rules for defining component instance labels. In this case, the roles of the component instances need to be indicated for diagnostic reasons and to improve readability. This information can be documented with generic constants that specify attribute values associated with the component instances.

9-4.2 PORT-NAMING CONVENTIONS

Port names for entity interfaces that represent hardware should have the same names as the pins on the actual hardware. If a pin name starts with a digit, the port name should consist of the interconnect name prefixed with a letter. For example, if the pin name is 123, the corresponding signal may be named P123. If the hardware pin names have more than one word, the corresponding VHDL name should be formed by concatenation and capitalization of the words as described in subpar. 9-4.1.

A major style issue arises around the declaration of ports that represent a multiline bus. From the viewpoint of top-down design, there may be a desire to have a single port declared a composite or abstract type. To support back annotation, there may be a need to have a separate port for each bit of the multiline bus. It is possible to create a “wrapper” entity that converts a model using separate ports for each bit into a single port with a composite or abstract type. Names for separate single bit ports that make up a multiline bus should be chosen to indicate their role in the bus. For example, the port name for address bus bit 3 could be `AddressBus3`.

Port names that do not represent actual hardware should be descriptive of their function or usage.

9-4.3 SIGNAL-NAMING CONVENTIONS

Signal names for models that represent hardware should be the same as the names of the electrical interconnections in the hardware. If an interconnect name starts with a digit, the signal name should consist of the interconnect name prefixed with a letter, as described in subpar. 9-4.2. If an interconnect name consists of more than one word, the corresponding VHDL name should be formed by concatenation and capitalization of the words, as described in subpar. 9-4.1.

Names of signals that do not represent hardware should be descriptive of their function or usage. For example, a performance model may have a globally declared signal called `statistics`, which is used by the test bench to collect statistics.

9-4.4 PROCESS AND SUBPROGRAM NAMING CONVENTIONS

Process labels and subprogram names should be descriptive of the function performed by the process or subprogram. Labels should be active instead of passive, e.g., `generate_next_address` instead of `address_generator`. The names of conversion functions should indicate both the source and result types of the function as is done in IEEE Std 1164 (Ref. 8).

9-4.5 COMMENTING CONVENTIONS FOR VHDL

The inclusion of comments in a VHDL description can enhance understanding of the model. Comments should describe the contents of files.

Because different organizations will have different coding conventions, it is not the intent of this subparagraph to give detailed guidance on the exact format of comments. The guidelines that follow are intended to be suggestive of the kinds of information that should be included in comments. Other information can be included as required.

More detailed requirements can be specified either by tailoring the VHDL DID or by requiring that VHDL models be developed in accordance with a development plan approved by the Government.

9-4.5.1 Files

VHDL source files should not contain more than one library unit, although it may be desirable to combine a package declaration and its package body when the combination is relatively short. Files containing VHDL source code should have the following comments at the beginning of the text file:

1. A brief description of the overall purpose of the library units contained within the file
2. A list of the library units, by name, contained within the file
3. The date the final form of the file was created for delivery to the Government
4. The name of the organization that created the file
5. The contract number(s) under which the contents of the file were created.

9-4.5.2 Packages

Each package declaration should have a brief description of the purpose and contents of the package. This description should include the following information:

1. A functional description of the package contents
2. The date the final form of the package was created for delivery to the Government
3. The name of the organization that created the package.

This information should be included at the beginning of the package text file.

9-4.5.3 Entity Interfaces

Each entity interface should have brief description that contains the following information:

1. A description of the function the design entity performs
2. Description of the generic constants: names, meanings, and ranges
3. Description of the ports
4. Description of errors checked for by the entity interface

5. The date the final form of the entity interface was created for delivery to the Government

6. The name of the organization that created the entity interface.

9-4.5.4 Architecture Bodies

Each architecture body should have a brief description that contains the following information:

1. A description of the style of architecture body: structural, behavioral, data flow, or a mixture

2. An indication of the level of abstraction employed: algorithmic, instruction set architecture, register-transfer level, or gate level

3. An indication of the complexity of the timing model employed: zero delay, fixed delays, parameterized delays, etc.

4. Description of any internal error checking

5. A reference to the corresponding CDRL number and the intention of the body to serve the requirements of the VHDL DID (Ref. 1), e.g., does the body serve as a behavioral or structural body in the sense of the DID

6. The date the final form of the architecture body was created for delivery to the Government

7. The name of the organization that created the architecture body.

This description should come immediately before the declaration of the architecture body.

9-4.5.5 Configuration Declarations

Each configuration declaration should have a brief description that contains the following information:

1. A description of the purpose of this particular configuration declaration

2. A description of any specific operating conditions for which this configuration declaration is intended

3. The date the final form of the configuration declaration was created for delivery to the Government

4. The name of the organization that created the configuration declaration.

This description should come immediately before the declaration of the configuration declaration.

9-4.5.6 Internal Comments

In addition to the comments previously mentioned, each VHDL description should include comments to help users understand the internal operation of the model. These comments can be either in-line or block.

Each subprogram and process should have comments that describe the operation, expected inputs and outputs, and error conditions associated with the process or subprogram.

Assertion statements should have comments that describe the error conditions detected.

All type and object declarations or groups of related declarations should have comments explaining the purpose of the declarations.

Finally, executable sections of a model should have comments explaining the operation of the model.

REFERENCES

1. DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Documentation*, Department of Defense, Washington, DC, 11 May 1989.
2. IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 31 March 1988.
3. C. Rogers et al, *A VHDL Modeling Guide*, Draft Report TP-804, Technology Independent Representation of Electronic Products (TIREP) Project, NAWC-AD, NSWC, Navy Research Laboratory, Indianapolis, IN, May 1994.
4. IEEE Std 1029.1-1991, *Waveform and Vector Exchange Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1991.
5. *F-22 Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) Model Specification*, Technical Report 5PTA3009, General Dynamics Corporation, San Diego, CA, March 1992.
6. EIA 567-A, *VHDL Hardware Component Modeling and Interface Standard*, Electronic Industries Association, Washington, DC, March 1994.
7. MIL-HDBK-454, *General Guidelines for Electronic Equipment*, 28 April 1995.
8. IEEE Std 1164-1993, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, May 1993.
9. *VHDL Model Verification and Acceptance Procedure*, Technical Report, Rome Laboratories/ERDD, Griffiss Air Force Base, Rome, NY, March 1992.
10. IEEE Std 1076.4, *IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, December 1995.

BIBLIOGRAPHY

J. R. Armstrong, *Chip-Level Modeling With VHDL*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

J. R. Armstrong and F. G. Gray, *Structured Logic Design Using VHDL*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

P. J. Ashenden, *The VHDL Cookbook*, University of Adelaide, Adelaide, South Australia, 1992.

J. Bergeron, “Guidelines for Writing VHDL Models in a Team Environment”, *VHDL Boot Camp*, Proceedings of the Fall 1993 VIUF Conference, San Jose, CA, October 1993, VHDL International Users’ Forum, c/o Conference Management Services, Menlo Park, CA.

D. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, Norwell, MA, 1989.

B. Doray and P. Yousefpour, “Issues in Writing Large Models in VHDL", *VHDL Boot Camp*, Proceedings of the Fall 1993 VIUF Conference, San Jose, CA, October 1993, VHDL International Users' Forum, c/o Conference Management Services, Menlo Park, CA.

Randolph Harr and Alec Stancluescu, Eds., *Applications of VHDL to Circuit Design,* Kluwer Academic Publishers, Norwell, MA, 1989.

R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.

D. Perry, *VHDL*, McGraw-Hill Book Co., Inc., New York, NY, 1991.

J. Schoen, *Performance and Fault Modeling With VHDL*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

APPENDIX A
VHDL MODEL VERIFICATION PROCEDURE

A-0 PREFACE

This Appendix is a procedure developed by the Department of Defense (DoD) to verify VHDL models. This *VHDL Model Verification and Acceptance Procedure* was developed at the US Air Force Rome Laboratories, Griffiss Air Force Base, Rome, NY, through the coordinated efforts of a triservice working group and industry consultants. This Appendix is the procedure as originally published by Rome Laboratories in November 1992 as Version 1.0 and updated 3 February 1995; it is unchanged by the author or editors of this handbook. References in the procedure to documents or standards that have been updated or superseded as of the date of publication of this handbook are footnoted, and the current reference is given in the footnote.

The intent of use of the very high-speed integrated circuit (VHSIC) hardware description language (VHDL) and the *Waveform and Vector Exchange Specification* (WAVES) within the DoD is to reduce the life cycle cost associated with unit or device development, testing, maintenance, and reprocurement. To achieve this goal, the DoD now suggests, as specified in MIL-HDBK-454, that all electronic devices delivered be documented in VHDL and their test vectors be in WAVES format. The need to establish a single model as the simulatable representation of an electronic unit in order to eliminate duplication and assure common simulation results is recognized by DoD and industry. Procurement personnel need to verify that such a model accurately represents the unit or device specification. In addition, this document provides VHDL model developers a means by which to evaluate their models against a set of criteria that the Government will use to evaluate whether a VHDL model has captured the necessary design information.

The verification procedure consists of six paragraphs. The first paragraph, “Scope”, provides an overview of the motivation behind the DoD requirement for VHDL models. This paragraph also addresses models that will be archived and the minimal simulation environment needed for model evaluation. The second paragraph, “Referenced Documents”, explains the order of precedence for reference documents in determining the functionality, timing, and operation of the electronic device. Next is “Initial Inspection”, which is a visual examination of the delivered files for proper documentation and format. The final three paragraphs, “Detailed Inspection”, “Testing and Data Analysis”, and “The Final Report”, include a detailed examination and execution of VHDL source code, library components, header information, and a report that discusses the findings.

A-1.0 SCOPE

The Department of Defense (DoD) is engaged in a number of programs which require VHDL models of ASICs and systems. Specifically, the details of the deliverable VHDL models are expressed in a combination of documents such as MIL-STD-454*, the VHDL Data Item Description (VHDL-DID (DI-EGDS-80811)) and any additional requirements specified in any given Contract Deliverable Data Items (“CDRLs” or “data items”).

VHDL data items capture the behavior and structure of an electronic system, subsystem, or device. The primary purpose of these data items is to document hardware designs in a machine executable, simulatable, and hierarchical format. VHDL models themselves must be inspected to insure that they meet the requirements specified in the contract or VHDL DID, as applicable. The VHDL DID may be tailored by the contract requirements for some applications.

For acceptance, VHDL simulation models provided to the Government as CDRLs must satisfy some known acceptance and verification criteria and procedure. These criteria and procedures are the purpose of this document.

The verification procedure includes model evaluation for compliance with the VHDL DID, inspection and testing of the code for VHDL correctness, verification of models against the supplied WAVES test vectors, verification of the models against the functionality of the described part, and verification of the model against the part specifications. Such verification methodologies require an in-depth knowledge of VHDL simulation, electronics hardware functionality, and electronics test.

This document shall be used as the procedure document for the verification of VHDL simulation models supplied to the Government under contract, for certification and qualification under the new Qualified Manufacturers List (QML), or as part of Line Replaceable Module (LRM) acceptance.

A-2.0 REFERENCED DOCUMENTS

The first step in model verification is to obtain a set of specifications and references concerning the device or system being modeled. This information is then used by the individuals performing this verification and acceptance procedure to educate themselves as to the functionality, timing and operation of the electronic system. Expert level understanding of the system’s design, functionality and timing are essential prerequisites of the verifier.

*MIL-STD-454 has been superseded by MIL-HDBK-454, *General Guidelines for Electronic Equipment,* 28 April 1995.

A-2.1 Order of Precedence

The order in which the publications are listed below shall be the order of precedence in the event that one publication modifies the specifications or statements of a document at a higher level of precedence, i.e., requirements under par. A-2.2.2 override conflicting requirements at par. A-2.2.3 and so on).

A-2.2 System Specifications

Any or all of the foregoing specifications may contain block diagrams, timing charts, truth tables, stimulus response vectors, schematics and any additional information.

A-2.2.1 Standard IC Data Books/Specifications

The verifier shall obtain a copy of the commercially available device or system data book or specification if one is available from the manufacturer.

A-2.2.2 ASIC Design Specification

For each ASIC undergoing verification and acceptance under this procedure, a detailed design specification shall be obtained.

A-2.2.3 System Level Specifications

For systems incorporating more than one of any combination of ASICs or standard ICs, a detailed system specification shall be obtained.

A-2.2.4 Hardware Test Plan

For any system, subsystem, board or ASIC undergoing verification under this procedure, a detailed test plan shall be obtained for each design unit undergoing verification.

A-2.3 IEEE Publications

The following Institute of Electrical and Electronics Engineers (IEEE) publications are referenced either explicitly or implicitly within this document. The verifiers should make each of these documents available to themselves for reference. Copies of the standards may be obtained from IEEE Standards Sales, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331.

A-2.3.1 IEEE Std 1076-1987*

IEEE Standard VHDL Language Reference Manual (VHDL-LRM), 1988, The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY.

A-2.3.2 IEEE Std 1029.1-1992

IEEE Standard (Waveform and Vector Exchange Specification) Language Reference Manual, 1991, The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY.

A-2.4 Government Documents

The following US Government standards are referenced either explicitly or implicitly within this document. The verifiers shall make each of these documents available to themselves for reference. Copies of the standards may be obtained from the US Government by contacting:

Naval Publications and Printing Service Office
700 Robbins Ave.
Philadelphia, PA 19111-5094

A-2.4.1 Military or Contract Specification

The verifier shall obtain a copy of any applicable military or contract specification in addition to those mentioned above under which the VHDL model(s) have been developed. For some items, more than one specification (SMD, SCD, commercial, etc.), may be required, in which case it must be determined which specification the model is supposed to represent. Also, many specifications may differ by timing alone so that one model with different timing packages may satisfy several different military specifications (or "dash numbers"). In addition, the same part may occur in different packages.

A-2.4.2 MIL-STD-454M, Requirement 64 to be published April 1991**

A-2.4.3 DI-EGDS-80811, *VHSIC Hardware Description Language (VHDL) Data Item Description*

A-2.5 Verification Procedure

To facilitate the performance of this verification procedure, the verifier shall follow the instructions provided in par A-6.0.

Note 1: Hereafter MODEL shall refer to the VHDL code delivered to represent a digital electronic unit, device, or component. The MODEL may be described as a high level (behavioral) model or as a gate level model or as a combination of behavior and structure (mixed model).

Note 2: The intended unit, device, or component for which the MODEL was developed will hereafter be called the REFERENCE. The REFERENCE item may be hardware, or if no hardware exists it will be the specifications for the intended unit, device, or component.

A-3.0 INITIAL INSPECTION

A-3.1 Documentation Files Required Under DID DI-EGDS-80811

The verifier shall determine that the contractor has provided all of the system specifications, hardware test plans, and any additional documentation required for the verifier to determine the functionality and timing of the system, sub-

*IEEE Std 1076-1987 has been superseded by ANSI/IEEE Std 1076-1993, *IEEE Standard VHDL Language Reference Manual,* September 1993.

**MIL-STD-454 has been superseded by MIL-HDBK-454, *General Guidelines for Electronic Equipment*, 28 April 1995.

system or device undergoing verification.

This section explains which items are to be “visually” inspected to determine whether all the required deliverables are present, in the proper order, and that they meet certain criteria specified in the VHDL DID.

The following eight auxiliary information files shall precede VHDL design files.

A-3.1.1 Table of Contents File

Inspect that this file contains the names of each of the VHDL files delivered; one file name per record and nothing else (pad with trailing blanks). (DID requirement 10.3a)

The file should be an ASCII file. Comments should begin with the character #. ONLY one file per line.

Note 3: From a model style guide perspective, the developers should be encouraged to deliver the models in the following format: `<package>.vhd` for package declarations,`<package>_body.vhd` for the corresponding package body, `<modelname>_e.vhd` for model entities, `<modelname>_a_str.vhd` for structural architectures, `<modelname>_a_beh.vhd` for behavioral architectures, `<modelname>_c_str.vhd` for structural configurations, `<modelname>_c_beh.vhd` for behavioral configurations, and `<modelname>_testbench.vhd` for test benches. This information is provided as a guide, not a requirement. The developer is encouraged to follow a similar file naming methodology.

Note 4: In addition, it is suggested that each model’s entity, architectures, test benches and configurations be located in its own subdirectory under the name of the model. The same holds true for packages. In addition, all simulation scripts and results can be located in the same subdirectories as the models to which they pertain.

A-3.1.2 CDRL File

Inspect that this file contains a high-level prose description of the VHDL deliverables. The description shall contain the following: (a) contract, (b)line item, (c) Contract Data Requirements list sequence number, and (d) a summary of the organization and content of the set of files. (DID requirement 10.3b)

A-3.1.3 Analysis File

Inspect that this file contains a specification of the order of analysis of the VHDL design units. Verify that the order of analysis is consistent with the rules of VHDL, (DID requirement 10.3c) (i.e., Packages compiled before their corresponding bodies, which in turn are compiled before the entities/architectures, and configurations which reference them).

A-3.1.4 Leaves File

Inspect that this file contains the list of unmodified VHDL leaf-level models that have been provided by the Government, and referenced within any VHDL files. (DID requirement 10.3c)

A-3.1.5 Modifications File

Inspect that this file contains the list of modules previously accepted by the Government and subsequently modified. (DID requirement 10.3e)

A-3.1.6 Deliverables Files

Inspect that this file contains a list of VHDL modules that originate with this VHDL delivery. (DID requirement 10.3f)

A-3.1.7 Test Bench Association File

Inspect that this file contains a list that associates VHDL modules with their corresponding test benches. (DID requirement 10.3g)

A-3.1.8 Auxiliary Information File(s)

Inspect that this file(s) contains any additional information concerning the VHDL descriptions and VHDL design files. Inspect that the contents of the auxiliary files do not contain any complete VHDL design units. (DID requirement 10.3h)

A-3.2 Conformance to IEEE VHDL-1076

The verifier is instructed to compile (analyze) the VHDL files on a fully compliant VHDL IEEE-1076 analyzer, in the order specified in the Analysis File delivered under par. A-3.1.3. Each of the files shall analyze with no errors. Certain analyzers will issue warnings. The verifier shall make a record of the execution of the analyzer and specifically note any errors or warnings indicated.

A-4.0 DETAILED INSPECTION

The second phase of the verification process is a detailed inspection of entities, architectures, configurations and other support modules delivered.

A-4.1 Comment Banner

In order to assist the model verifier, a comment section is required to precede each VHDL module. The comment section should contain the following information:

1. Design unit name identifier
2. Identification of originator or source
3. DoD approved identifier (if one exists)
4. Whether model has been previously delivered
5. General approaches taken to modeling, and particular decisions regarding Modeling fidelity
6. Any further information vital to subsequent users of the descriptions
7. Any factors restricting the general use of this description to represent the actual hardware
8. Any assumptions taken in developing the model
9. Previous approval of the module by the DoD.

A-4.1.1 Comment Banner With Revision Information

If the module is a previously approved module and has been revised with this delivery, the following information shall also be included:

1. The date of revisions
2. The performing individual and organization
3. The rationale for the revision
4. A description of which part of the original design unit which required modification
5. A description of the testing done to verify the revised model.

A-4.1.2 Comments

While it is difficult to determine quantitatively that the model author has sufficiently commented the VHDL code, a usual rule of thumb is to have an approximate 20% comment overhead.

A-4.1.3 Inspection for Orthogonality

The files shall be inspected to ensure that each file is either a VHDL design file, whose entire contents conform to the requirements of the VHDL Language Reference Manual, or an auxiliary information file containing no VHDL design units. (DID requirement 10.3)

A-4.1.4 Inspection for Incremental Information

The files shall be inspected to ensure that new design units are not contained in the same file as design units that have been previously accepted by the Government. (DID requirement 10.3)

Note 5: A previously accepted module can be checked to ensure that it has not been altered by using a text comparison to discover any differences between the archived module and the delivered module. Differences other than variable names and comments should be examined for their effect on module functionality, these differences should be noted in the final report.

A-4.2 Model Evaluation and Inspection

The procedure described in this section should apply to entities, architectures, and configurations of the model. All material in par. A-4.1 pertains to each module of the VHDL deliverables.

A-4.2.1 Entity Declaration (DID Conformance)

Each entity declaration shall be inspected for the specifications listed in this section. In addition, if the entity is contained in a separate file, then the procedures pertaining to revision information and comments (par. A-4.1) shall also apply.

A-4.2.1.1 Entity Declaration

The entity declaration for each entity shall include:

1. An interface declaration
2. Timing and electrical requirements for the behavior of the device
3. Allowable operating conditions
4. Component identification
5. Explanatory comments.

(DID requirement 10.2.2.)

A-4.2.1.2 Entity Interface Declaration

The interface declaration for each entity shall be inspected to assure:

1. That a description has been included for every port that exists on the device
2. The inclusion of information relating each input and output port to a package pin number or connector pin number whenever such a correspondence exits.

(DID requirement 10.2.2.1.)

Note 6: If a condition should arise such that the name of the port violates the rules of VHDL, an appropriate alternative name should have been selected and commented as such.

Note 7: There are a number of ways in which this information may be obtained including (1) comments, (2) port attributes, or even (3) the instantiation of a "packaging" entity whose port names correspond to the pin numbers of the packaging of the device, i.e., `Pin_23` as a port name (DID requirement 10.2.2.1).

A-4.2.1.3 Entity Naming Conventions

The entity declaration shall be inspected to ensure that the names for VHDL entities are traceable to the names of their physical electronic counterparts whenever such a correlation exists. (DID requirement 10.2.2.4)

A-4.2.1.4 Timing Electrical Requirements

The model shall be inspected to ensure that timing and electrical requirements are expressed in such a manner as to cause the simulator to generate error messages upon violation of a specification during simulation. (DID requirement 10.2.2.2)

The specifications may include the following:

1. Timing specifications such as setup, hold, pulse width, periodicity, and release or recovery times, among others.
2. Electrical specifications such as maximum fanout DC load, maximum fanout capacitive load, maximum drive current limits, voltage range, temperature range.
3. Additional timing considerations such as required number of clock cycles for correct reset to occur.

A-4.3 Architectures

Each architecture shall be inspected for the specifications listed in this section. In addition, if the architecture is contained in a separate file, then the procedures pertaining to revision information and comments shall apply as well.

A-4.3.1 Hierarchy

Inspect that the models delivered are written with a "reasonable" level of hierarchy. The model shall be inspected to ensure that structural decomposition of behavioral bodies is used only when necessary to show functional partitions of the corresponding structural body. Ease of simulation and clarity of behavior shall be considered when determining the appropriate level of hierarchical decomposition. (DID requirement 10.2.3.1)

A-4.3.1.1

The model shall be inspected to ensure that the hierarchy of VHDL modules is analogous to the physical hierarchy of the hardware being documented. The model shall be inspected to ensure that one VHDL module is defined for the entire system, and one for each physical electronic unit (assembly, subassembly, integrated circuit, etc.) of the hardware system, and that VHDL modules are defined for important subsections or groupings of complex physical units (e.g., macrocells of a chip or boards defining a processor). (DID requirement 10.2.1)

Note 8: As a guide, an ASIC should have a minimum of three (3) levels of hierarchy: (1) A behavioral model of the ASIC at the pin boundary level (no structural subarchitectures), (2) a level representing the ASIC's block diagram (which includes structural subarchitectures), where the structural subarchitectures are written as behavioral models, and lastly (3) a detailed, gate-level architecture, where all of the components are leaf-level models.

A-4.3.2 Physical Correspondence

Inspect that the model's architecture is written and commented sufficiently well such that the internal signal names and hierarchical component names reasonably match the names of the physical implementation. (DID requirement 10.2)

A-4.3.3 Signal Delays

Inspect that all signal delays accurately model the behavior of the device specification. At a minimum, the models shall be coded to incorporate a means of evaluating minimum, typical, and maximum timing delays. More elaborate timing models which take into account other variables such as supply voltage or output loading may also be used. (DID requirement 10.2.3.2)

Note 9: Determining that the model "accurately" models the timing of a specification is a difficult task. Certain areas to look for include:

1. All possible input to output pin asynchronous "cause-effect" paths have a corresponding delay path. This delay path may, in addition, be level sensitive.
2. Determine how the model should respond under conditions when simultaneous events could trigger events that preempt previous timing events causing the output to change to a new state at the wrong time.
3. Normally, inertial delay should be used. However, certain conditions, such as glitch detection, require a transport delay mechanism.

A-4.4 Behavioral Subarchitecture

Each behavioral subarchitecture shall be inspected to ensure that it meets the specifications listed in this section.

A-4.4.1 Visibility of Internal Registers

The model shall be inspected to ensure that all user programmable operations and registers are clearly identifiable in the simulation model. The model verifier shall make a checklist of the programmable operations and registers for later use. (DID requirement 10.2.3)

A-4.4.2 Test and Maintenance Functions

Inspect that, if test or maintenance functions are available to the user of the actual component, the model includes a description of the test functions. (DID requirement 10.2.3)

A-4.4.2.1 Test and Maintenance Functions for Behavioral Models

Detailed structural scan signature paths shall not be specified. However, the entity interface of the device should include the scan test port declarations. The model shall be inspected to ensure that signal values which are dependent on a particular structural implementation, such as scan path signatures, are not specified in the behavioral body. (DID requirement 10.2.3.3)

Note 10: In addition, the behavioral model, when placed into a test mode, should respond with a NOTE level assertion stating that the scan structure has not been implemented in the model.

A-4.5 Structural Subarchitecture

Each structural subarchitecture shall be inspected to ensure that it meets the specifications listed in this section.

A-4.5.1 Test and Maintenance Functions

Inspect that, if test or maintenance functions are available to the user of the actual component, the model includes a description of the test functions. (DID requirement 10.2.3)

A-4.5.2 Test and Maintenance Functions for Structural Models

The model shall be inspected to ensure that structure which is created to support testing and maintenance (such as scan path signatures) is included in the VHDL structural description. (DID requirement 10.2.4)

Detailed structural scan path signatures shall be specified.

A-4.5.3 Correspondence to Actual Implementation

The model shall be inspected to ensure that the structural bodies represent the physical implementation. The details of the model at this level should enable logic fault modeling and test vector generation to be performed, not necessarily within a VHDL environment (DID requirement 10.2.4)

A-4.5.4 Traceability

The model shall be inspected to ensure that the names of components and signals are the same as, or traceable to, their electrical schematic counterparts, for ease of schematic drawing correlation, and within the constraints of the lexical rules of VHDL. (DID requirement 10.2.4.1)

A-4.5.5 Leaf-Level Modules

The model shall be inspected to ensure that each leaf level module can be classified in one of the following categories:

1. Modules selected from a Government list of leaf level modules

2. Modules corresponding to a collection of hardware elements which together exhibit a stimulus-response behavior, but whose interaction is best modeled at the electrical or physical level. Examples of such modules are digital logic gates, analog circuit blocks, and power supplies.

3. Modules whose detailed design has not yet been completed, but whose behavior is required as an interim contractual deliverable. (DID requirement 10.2.1.1)

A-4.6 Dataflow Subarchitecture

Each of the procedures defined in and above shall be applied to dataflow modeling subarchitectures as well.

A-4.7 Inclusion of Packages

The model shall be inspected to ensure that VHDL package declarations are used whenever operating conditions are common across a class of similar components. (DID requirement 10.2.2.3).

Note 11: Operating conditions are the physical and electronic environment in which components are designed to operate, such as temperature range, signal excursions, logic level definitions, maximum power dissipation, and radiation hardness.

A-4.7.1 Traceability

Inspect that all such specifications are traceable back to the physical device specifications. (DID requirement 10.2.2.3)

A-4.8 Test Benches

A-4.8.1 Check for Existence of Corresponding Test Bench

Every VHDL module shall be simulatable as a standalone model and hence a corresponding VHDL test bench is required for every VHDL module of the hierarchy. (DID requirement 10.2.5.3)

A-4.8.2 WAVES Conformance Requirements

The test vectors shall be inspected to determine that they have been written in the WAVES format.

A-4.8.3 Distinguishable from the Module

The test benches shall be inspected to ensure that they are clearly distinguishable from the VHDL modules representing the design itself. (DID requirement 10.2.5)

A-4.8.4 Test Bench Comments

The test bench shall be inspected for explanatory comments. Refer to pars. A-4.1 and A-4.1.2. (DID requirement 10.2.7)

A-4.8.5 Test Vector(s) Description

A detailed description of the purpose of each test bench shall be included. (DID requirement 10.2.7)

A-4.8.6 Assertion Reports

The test bench shall be inspected to assure that it stimulates the Module Under Test (MUT) and reports any discrepancies with expected response during simulation. (DID requirement 10.2.5.1)

A-4.8.6.1 Assertion Messages

For each error message, inspect that it identifies the requirement that has failed, and that the error message includes the name of the violating VHDL design entity. (DID requirement 10.2.6)

A-4.8.7 Sufficiency of Configuration Information

Inspect the test bench to assure that sufficient configuration information is present to facilitate the test.

A-4.8.8 Test Requirements Correlation

The VHDL test benches, the hardware test drawings, and test plans shall be inspected to ensure that they are cross-referenced to any required hardware test plans as necessary. (DID requirement 10.2.5.2)

A-4.8.9 Necessary Tests

Each test bench shall be inspected to ensure that the WAVES test vectors used within it are necessary to simulate the correct behavior of the VHDL module to which it corresponds. (DID requirement 10.2.5)

A-4.8.10 Sufficient Tests

Each test bench shall be inspected to ensure that the WAVES test vectors used within it are sufficient to simulate the correct behavior of the VHDL module to which it corresponds. This includes a sufficient set of test vectors to violate all timing constraints. (DID requirement 10.2.5)

A-4.9 Configurations

Note 12: While there are no specific procedures to verify configurations, the following issues should be pointed out:

1. Default Bindings. When default bindings are used, a comment stating such is useful.

2. Open Associations. When open ports or design units are encountered in the configuration, a comment should be made as to the purpose of the open association.

Note 13: Type Conversions. When type conversion functions are used to map data from one VHDL model to another, a comment should state if the mapping is identical or not. If the mapping is not identical, then the comment should state whether any unmapped signal values are likely and to which state they are being mapped.

A-5.0 TESTING AND DATA ANALYSIS

A-5.0.1

Verifying the correct behavior of a VHDL simulation model is a complex undertaking. The verifier needs a detailed understanding of the device that has been modeled. All sources of information concerning the device's operation shall be obtained and used to determine what testing is required to verify that the delivered VHDL simulation model operates correctly and if the delivered test suite is sufficient to meet those requirements. The intent of the verification is to detect errors or omissions in the functionality, timing, style or content of the VHDL simulation model. The mechanics of the verification procedure are dependent on the amount of design detail available for the REFERENCE and the amount of design detail available for the MODEL.

A-5.0.2 Execution of the Test Suite

Simulation results must match those indicated by the specification for items such as best-case, worst-case, and nominal output delays versus temperature and voltage ranges. Error messages caused by timing violations shall be inspected to ensure that they correctly identify the requirement which has been violated and the name of the VHDL design unit in which the error has occurred.

A-5.0.3 The Testing Procedure

The test procedure consists of:

1. Comparing the operation of the MODEL to the specifications of the REFERENCE
2. Comparing the operation of the MODEL to the operation of the intended REFERENCE.

A-5.1 Definitions for Testing

A-5.1.1 Test Bench

A VHDL module that applies WAVES stimuli to a module under test (MUT) compares the MUTs response and the WAVES expected output, and reports any differences between observed and expected responses during simulation. Each configuration should have a corresponding test bench which is clearly distinguishable from the MUT modules.

A-5.1.2 Test Suite

A collection of one or more test benches to which is associated a corresponding MODEL.

A-5.1.3 Test Bench Configuration Sets

Determine that a test bench configuration set has been made available for every combination of entity, architecture, and test bench such that a test bench configuration exists for each pair, i.e., MODEL-structural view + Test 1, MODEL-behavioral view + Test 1, etc. (DID requirement 10.2.5.1)

A-5.1.4 Commercial Model REFERENCE

For commercially available integrated circuits, the DID specifies that the VHDL test case shall correspond to an equivalent hardware test plan. If no such test plan exists, as in the case of a model of a standard IC device, then the REFERENCE shall be the actual device.

A-5.2 Execution of Test Suite

Each test bench shall be executed and the results of the simulation runs recorded. (DID requirement 10.2.5.1)

A-5.2.1 Behavioral and Structural Verification

Run every test bench configuration set and record the results. (DID requirement 10.2.5.1)

A-5.2.2 Automatic Checking

The simulation results shall be analyzed to ensure that each VHDL test bench does correctly apply stimuli to the MODEL, compares the MODEL's response to an expected WAVES output, and reports any differences between observed and expected responses during simulation. (DID requirement 10.2.5.1)

This involves monitoring the test bench and the MODEL during simulation to insure that the proper functions in the MODEL are activated by the test bench and that the proper responses occur in the MODEL and are properly monitored by the test bench. The comments provided with the test bench defines what should happen with each test bench.

A-5.2.4 Augmentation of Test Vectors

The model developer shall provide WAVES test vectors designed to check functionality and timing with a comment provided for each vector or set of vectors describing the associated function being tested. The model verifier shall develop a set of test vectors that violate the timing and voltage specifications, attempt to perform illegal model operations and test its functionality at its operational limits should such test not be provided by the developer. Any additional vectors developed to augment the test vector set shall be documented in the final report. The MUT shall then be simulated and the results analyzed.

A-5.2.3 Determination of REFERENCE Test Goodness

Referring back to the REFERENCE specifications and the hardware test plan, perform the following tasks:

A-5.2.5 REFERENCE Test Coverage Determination

Determine if there exists any REFERENCE behavior specified in the functional specification that is not tested by a test bench. This involves comparing the functional specification with the test bench descriptions to identify test bench omissions. (DID requirement 10.2.5.3).

A-5.2.9 Augmentation of Test Benches

If in the opinion of the verifier, additional test benches are warranted, then the verifier may write those test benches and document the purpose of each test.

A-5.2.10 Simulation

The VHDL modules shall be simulated on any available IEEE-1076 compliant simulator using the supplied test vector set in the WAVES format.

A-5.3 Results Analysis

A-5.3.1 Result Documentation

Document every test performed under this section. Note any errors or omissions and write additional test benches as deemed necessary.

A-5.3.2 MODEL Functionality Omissions

Look for REFERENCE behavior specified in the functional specification that is not modeled at all in the MODEL. This involves comparing the functional specification with the MODEL to identify MODEL omissions.

A-5.3.3 MODEL Functionality Errors

Look for REFERENCE behavior specified in the functional specification that has been modeled in error in the MODEL. This involves comparing the functional specification with the MODEL to identify MODEL errors. REFERENCE behavior includes timing behavior and functional behavior.

A-5.3.4 MODEL Timing Performance

Check for proper modeling and testing of best, worst and nominal output delays. (DID requirement 10.2.2.2).

A-5.3.4.1

Among other timing tests situations to look for, the following is a list of timing conditions commonly found in commercial device models:

1. Setup, hold, recovery, and release time specifications
2. Periodicity, pulse-width and cycle-count specifications
3. Timing variations due to voltage, temperature or loading.

(DID requirement 10.2.3.2)

Note 14: Performing this procedure involves monitoring the MODEL during simulation to insure that the proper timing relationships exist in the MODEL and that they are activated by the test bench. The comments provided with the test bench define what should happen with each test bench.

A-5.3.5 Timing Violation Error Reports

The error messages caused by the timing violation shall be inspected to ensure that they correctly identify the requirement which has been violated and the name of the VHDL design unit in which the error occurred. Applicable VHDL design units include: entity declarations, architectures, package declarations, package bodies, and configurations. (DID requirement 10.2.6).

OPEN ISSUE: An alternative to verifying timing with simulation is to verify timing with a timing analysis tool.

A-5.3.6 MODEL Programmable Operations Performance

Check for proper operation of all user programmable operations (instructions, registers, etc.) (DID requirement 10.2.3).

Note 15: This check involves comparing the REFERENCE specification with the MODEL to identify MODEL errors or omissions. In addition, common areas to investigate include instruction operation in all addressing modes, explicit use of illegal opcodes, and determination that instructions execute in proper time sequence with the correct cycle count among other test.

A-5.3.7 MODEL Test and Maintenance Functions

Check for proper operation of all test and maintenance functions that are available to the user. (DID requirement 10.2.3)

A-5.4 REFERENCE Implemented in Hardware

A correlation between the actual hardware and the VHDL model to ensure correctness is the next step of the testing process. The same WAVES test vectors used to stimulate the MODEL are used to test the corresponding hardware. At this level, discrepancies indicate a failure in the model's description, an incorrect test vector set, or hardware that fails to meet the specification.

This procedure shall be applied when the actual hardware component is considered to be of higher quality than the VHDL model. This is normally the case whenever a third-party develops a behavioral model of a commercial digital integrated circuit from the description contained in a nonproprietary data book or data sheet.

A-5.4.1 Hardware Test Fixture

Construct or mount the REFERENCE into a hardware test fixture. (For a commercial component, this is usually a hardware modeler interfaced to a digital simulator.) Develop a means of applying the test patterns generated by the test suite to the REFERENCE. (In a typical VHDL-based simulator with a hardware modeler interface, this step requires the writing of a configuration design unit binding the formal component instantiation to the physical device through the hardware modeler software interface protocol.)

A-5.4.2 Verification Procedure

Repeat par. A-5.2 through par. A-5.3.7 with the physical REFERENCE.

A-5.4.3 Test Response Comparisons

Compare the responses of the REFERENCE against the response of the MODEL.

A-5.4.3.1 Comparison Considerations

The intent of comparing the responses of the MODEL with the responses of a REFERENCE is to insure the MODEL reflects the behavior of the REFERENCE both functionally and to some allowable timing tolerance. Because many differences may exist between the two, special care needs to be taken to insure a valid comparison.

A-5.4.4 Different Levels of Abstraction

By itself, different levels of modeling abstraction between the REFERENCE and the MODEL should not present additional problems. The REFERENCE operation may be encapsulated in a written specification, a high level (behavioral) model, a gate level model or by the hardware; the MODEL may be described as a high level (behavioral) model or as a gate level model. But the VHDL DID only mandates that physical I/O pins, timing characteristics on I/O pins, and user accessible hardware objects be clearly identifiable and hence comparable. Other internal objects may or may not match across the two models and certainly should not be used as a basis for comparison.

A-5.4.5 Data Sampling

While there will be differences in the details, both models are intended to represent a common behavior. Keeping in mind the issues presented below, a reasonable comparison is possible.

The comparison is only as good as the data being compared. Getting good data is a function of proper sampling. Proper sampling is determined by the amount of timing detail incorporated into the REFERENCE and the MODEL. Sampling rates, times and locations are determined by the REFERENCE or MODEL with the least accurate timing for pin to pin comparisons. For internal comparisons, sampling rates, times and locations are determined by the REFERENCE or MODEL with the most abstract description.

A-5.4.6 Strobing Intervals/Time Offset

If both models contain accurate timing information, sampling can occur at regular timed intervals, for instance: every 5 ns. This interval is determined by the level of accuracy required in the comparison and the allowable timing tolerances. Data collection must be offset far enough from sample initiation to guarantee valid data in the presence of any modeled or physical delays and any timing ambiguity due to timing tolerances.

If one of the models does not contain detailed timing information, then sampling must be initiated with system clocks (synchronous design) or control signals (asynchronous designs). Data collection must be offset far enough from sample initiation to guarantee valid data in the presence of any modeled or physical delays and any timing ambiguity due to timing tolerances.

If both models contain accurate functional information, sampling can also occur at any internal location; for instance at every register. These locations are determined by the level of accuracy required in the comparison. If one of the models does not contain detailed functional information, then sampling is only useful where there are common objects.

A-5.4.7 Cycle Count

Certain processors require a number of clock cycles in order to perform a given function (i.e., multiplication in 154 clocks on a MC68000). Check that the MODEL matches the REFERENCE with respect to cycle counts.

A-5.4.8 Timing Tolerance Windows

Certain REFERENCE specifications indicate that the MODEL shall respond to a stimulus within a certain relative time interval with respect to the stimulus or a gating clock signal. Check that the test bench has been written in such a way as to determine that the response transition occurs within that timing window and that the test bench issues an error if the response (a) fails to occur, (b) fails to provide the correct value during the time window, or (c) occurs outside of the time window.

A-5.4.9 Discrepancies

Document any errors or omissions and write additional test benches as necessary.

A-5.4.10 Justifiable Discrepancies

Make a list of justifiable discrepancies indicating the discrepancy along with an explanation of why the discrepancy is acceptable.

A-5.5 Simulation Values

If the REFERENCE and the MODEL have different value systems, a mapping from one to the other must be defined. This mapping will be used during the comparison process to insure response equality.

Consider the case where the REFERENCE uses a three state (`'U', '0', '1'`), two strength (`'W', 'S'`) value system and the MODEL uses a five value system (`'U', '0', '1', 'Z', '-'`). The mapping might be something like (`'W', 'U'`) `-> 'U'`, (`'S', 'U'`) `-> 'U'`, (`'W', '0'`) `-> 'Z'`, (`'S', '0'`) `-> '0'`, (`'W', '1'`) `-> 'Z'`, (`'S', '1'`) `-> '1'`. (There is no mapping to `'-'`.)

Keeping in mind the issues of data sampling below, the comparison procedure uses the mapping defined above to determine when responses in the REFERENCE are equivalent to responses in the MODEL. If the MODEL ever exhibits a `'-'` during the comparison process, special analysis is probably called for to determine what is happening. The existence of the `'-'` does not automatically imply model differences.

A-5.6 Model Initialization

Situations may occur where it may appear that things can be more complicated than is actually the case. Because the models are supposed to be equivalent, the external stimulus that initializes the REFERENCE and the MODEL are identical. After that stimulus has been applied, the two models must be in identical states or the models aren't equivalent. The question remains: how to compare states for identity?

Consider the case where an initialization sequence consists of holding a reset line active and then applying 5 clock pulses. Upon completion of the initialization sequence, all state machines should be at their starting state, all registers should be cleared and all output busses should be tristated. A comparison of the two models is constrained in both space and time because of the way the specification is defined and because of what the DID allows.

The state machines in the two models cannot be observed and compared because they do not represent user-accessible features. The DID does not require that internal hardware objects be modeled to any standard or that they must behave in any set way. Only the subset of registers that is "user programmable" may be observed and compared for the same reason. Nothing in either of the models (registers, busses, etc.) may be compared during the reset sequence, only after it is completed. The specification and the DID make no statement about the behavior of the circuit or the models during the reset sequence. Because of the data sampling issues below, there may need to be some additional delay before the comparison sample is taken because of pin switching delays.

A-5.7 Unjustifiable Discrepancies

Make a list of unjustifiable discrepancies indicating the discrepancy along with an explanation of why the discrepancy is unacceptable.

A-5.8 I/O Pin Differences

There are two possibilities here. One possibility is that a pin is present in one model and absent in the other. For instance: a high level model has no scan out pin because that behavior wasn't modeled. An "equivalent" low level model has a scan out pin. The other possibility is that a pin is present in both models but behaves differently due to "level of abstraction" issues. While it may seem there are extenuating circumstance here, there really are not. In neither case are these models equivalent. The DID requires pin for pin compatibility.

A-6.0 VERIFICATION REPORT

A final report shall be written detailing the results of this Model Verification Procedure. The report shall contain the following sections.

A-6.1 Contents and Organization of the Report

A-6.1.1 Final Report Header

Contains essential information regarding the hardware being modeled and the modeling environment. (See Tab A.)

A-6.1.2 Verification Procedure Checklist

Assures that the model has been inspected against each item of the procedure. (See Tab B.)

A-6.1.3 Final Report Format

Explains the expected deliverable format of the final report. (See Tab C.)

Tab A: Final Report Header

Report Name:
Verificator Name:
Model Name:
Model Version:
Model Vendor:
Authorizing Requester:
Analyzer Vendor Name:
Analyzer Model:
Analyzer Version:
Simulator Vendor Name:
Simulator Version:
Hardware Modeler Vendor:
Hardware Modeler Model:
Hardware Modeler Version:
Source of REFERENCE data:
List of additional HW / SW used for this test:
List of auxiliary test benches:
Instructions to the Verificator:

Tab B: Verification Procedure Checklist

The verificator shall check that each task item has been completed as described in the Verification Procedure.

A-2.0 REFERENCED DOCUMENTS
A-2.2 System Specifications
  A-2.2.1 Standard IC Data Books/Specifications
  A-2.2.2 ASIC Design Specification
  A-2.2.3 System Level Specifications
  A-2.2.4 Hardware Test Plan
A-2.3 IEEE Publications
  A-2.3.1 IEEE Standard VHDL Language Reference Manual (VHDL-LRM) Exchange Specification
  A-2.3.2 IEEE Standard VHDL View of WAVES (Waveform and Vector Exchange Specification)
A-2.4 Government Documents
  A-2.4.1 Military or Contract Specification
  A-2.4.3 DI-EGDS-80811 VHSIC Hardware Description Language (VHDL) Data Item Description
  A-2.4.4 TISSS Specification

A-3.0 INITIAL INSPECTION
A-3.1 Documentation Files Required under DID DI-EGDS-80811
  A-3.1.1 Table of Contents File
  A-3.1.2 CDRL File
  A-3.1.3 Analysis File
  A-3.1.4 Leaves File
  A-3.1.5 Modifications File
  A-3.1.6 Deliverables Files
  A-3.1.7 Test Bench Association File
  A-3.1.8 Auxiliary Information File(s)

A-3.2 Conformance to IEEE VHDL-1076

A-4.0 DETAILED INSPECTION
A-4.1 Comment Banner
A-4.1.1 Comment Banner with Revision Information
A-4.1.2 Comments
A-4.1.3 Inspection for Orthogonality
A-4.1.4 Inspection for Incremental Information
A-4.2 Model Evaluation and Inspection
A-4.2.1 Entity Declaration DID Conformance
A-4.2.1.1 Entity Declaration
A-4.2.1.2 Entity Interface Declaration
A-4.2.1.3 Entity Naming Conventions
A-4.2.1.4 Timing Electrical Requirements
A-4.3 Architectures
A-4.3.1 Hierarchy
A-4.3.2 Physical Correspondence
A-4.3.3 Single Delays
A-4.4 Behavioral Subarchitecture
A-4.4.1 Visibility of Internal Registers
A-4.4.2 Test and Maintenance Functions
A-4.4.2.1 Test and Maintenance Functions for Behavioral Models
A-4.5 Structural Subarchitecture
A-4.5.1 Test and Maintenance Functions
A-4.5.2 Test and Maintenance Functions for Structural Models
A-4.5.3 Correspondence to Actual Implementation
A-4.5.4 Traceability
A-4.5.5 Leaf-Level Modules
A-4.6 Dataflow Subarchitecture
A-4.7 Inclusion of Packages
A-4.7.1 Traceability
A-4.8 Test Benches
A-4.8.1 Check for Existence of Corresponding Test Bench
A-4.8.3 Test Bench Comments
A-4.8.4 Test Vector(s) Description
A-4.8.5 Assertion Reports
A-4.8.5.1 Assertion Messages
A-4.8.6 Sufficiency of Configuration Information
A-4.8.7 Test Requirements Correlation
A-4.8.8 WAVES Conformance Requirements
A-4.8.9 Necessary Tests
A-4.8.10 Sufficient Tests
A-4.9 Configurations

A-5.0 TESTING AND DATA ANALYSIS
A-5.2 Execution of Test Suite
A-5.2.1 Behavioral and Structural Verification
A-5.2.2 Automatic Checking
A-5.2.3 Determination of REFERENCE Test Goodness
A-5.2.4 Augmentation of Test Vectors
A-5.2.5 REFERENCE Test Coverage Determination
A-5.2.6 Augmentation of Test Bench(es)
A-5.2.7 Simulation
A-5.3 Results Analysis
A-5.3 1 Result Documentation
A-5.3.2 MODEL Functionality Omissions
A-5.3.3 MODEL Functionality Errors
A-5.3.4 MODEL Timing Performance
A-5.3.5 Timing Violation Error Reports
A-5.3.6 MODEL Programmable Operations Performance
A-5.3.7 MODEL Test and Maintenance Functions
A-5.4 REFERENCE Implemented in Hardware
A-5.4.1 Hardware Test Fixture
A-5.4.2 Verification Procedure
A-5.4.3 Test Response Comparisons
A-5.4.4 Different Levels of Abstraction
A-5.4.5 Data Sampling
A-5.4.6 Strobing Intervals/Time Offset
A-5.4.7 Cycle Count
A-5.4.8 Timing Tolerance Windows
A-5.4.9 Discrepancies
A-5.4.10 Justifiable Discrepancies
A-5.5 Simulation Values
A-5.6 Model Initialization
A-5.7 Unjustifiable Discrepancies
A-5.8 I/O Pin Differences

Tab C: Delivery of the Final Report

The Verification Report, along with the Verification Checklist, shall be filed in ASCII version and be appended to the final tape provided for acceptance. In addition, a written copy of the following shall be provided to the Program Office requiring the acceptance.

Certification of Verification

THIS VERIFICATION PROCEDURE has hereby been performed in accordance with the Verification Procedure attached hereto.

WHEREAS, Verificator hereby certifies that the Model(s) under consideration have been evaluated in accordance with the verification procedures set forth in the Verification Procedure document; and

WHEREAS, the Verificator hereby represents that any discrepancies found have been indicated in an accompanying Verification Report attached to this Checklist; and

NOW THEREFORE, the verificator certifies that such tests were performed as required by affixing his or her signature below.

Verificator Signature:

Date:

APPENDIX B
CONTRACT DATA REQUIREMENTS LIST AND DATA ITEM DESCRIPTION

B-0 PREFACE

This Appendix contains examples of a completed contract data requirements list (CDRL), DD Form 1423-1, and a completed tailored data item description (DID), DI-EGDS-80811.

The CDRL is an actual CDRL developed by the US Army Research Laboratory and is the CDRL of the DID developed by the Naval Research Laboratory that is used as an example. The CDRL and the tailored DID are presented exactly as used in contractual requirements documents for the delivery of very high-speed integrated circuit (VHSIC) hardware description language (VHDL) documentation. Some documents referenced in the tailored DID have been updated or superseded. Anyone developing a DID and using this DID as an example must verify the current version of any document referenced in this example.

CONTRACT DATA REQUIREMENTS LIST
(1 Data Item)

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 220 hours per response, including the time for reviewing instructions, searchin existing data sources, gathering and maintaining the date needed, and completing and reviewing the collection of information. Send comments regarding the burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, paperwork Reduction project (0704-0188), Washington, DC 20503. Please DO NOT RETURN your form to either of thes addresses. Send completed form to the Government Issuing Contracting Officer for the Contract/PR No. listed in Block E.

A. CONTRACT LINE ITEM NO. **0009AA**	B. EXHIBIT **D**	C. CATEGORY: TDP ______ TM ______ Other **EGDS**
D. SYSTEM / ITEM **RASSP DEVELOPMENT**	E. CONTRACT / PR NO. **DAAL01-93-C-3380**	F. CONTRACTOR **MARTIN MARIETTA**
1. DATA ITEM NO. **D001**	2. TITLE OF DATA ITEM VHSIC Hardware Description Language (VHDL) Documentation	3. SUBTITLE VHDL Technical Reports
4. AUTHORITY (Data Acquisition Document No.) **DI-EGDS-80811**	5. CONTRACT REFERENCE **SOW PARA 3.1.3. & 3.4.2.2**	6. REQUESTING OFFICE **ARPA/ESTO**

7. DD 250 REQ **F**	9. DIST STATEMENT REQUESTED **A**	10. FREQUENCY **PER YEAR MODEL EACH DEMO**	12. DATE OF FIRST SUBMISSION **One (1) month after first model year demo**
8. APP CODE **A**		11. AS OF DATE **SEE BLOCK 12**	13. DATE OF SUBSEQUENT SUBMISSION **1 month after each year demo**

17. PRICE GROUP

18. ESTIMATED TOTAL PRICE

14. DISTRIBUTION

a. ADDRESSEE	b. COPIES Draft	Final Reg.	Final Repro
AMSRL-EP-I	0	1	0
NRL	0	1	0
WL/EL	0	1	0
DESC	0	1	0
15. TOTAL ------>	0	4	0

16. REMARKS

1. Performance/un-interpreted/architectural model views at the first level of VHDL module hierarchy (10.2.1) Decomposition. This view shall contain timing-only behavior for leaf level entities (10.2.1.1) such as processor nodes. Buses/interconnects, inputs, and outputs. This view documents the view required by a system engineer to make high level choices relative to the type of processor. Number of processors, and the type of network required.

2. Application model view at the second level of VHDL module hierarchy (10.2.1) decomposition. This view shall describe the full functional behavior (10.2.3) with multi-component electronic modules as the leaf level entities (10.2.1.1). This view documents the functionality of the module such that the system engineer can 1) choose from a model library the module with the appropriate functionality, 2.) integrate the described module's behavior with the behavioral descriptions of other modules in the system, 3.) can perform integrated hardware and software diagnostics of the system software, and 4.) can investigate and analyze the impact of replacing the module during a model year upgrade.

3. Application model view at the third level of VHDL module hierarchy (10.2.1) decomposition. This view shall describe the full functional behavior (10.2.3) with individual integrated circuits as the leaf level entities (10.2.1.1). This view documents the functionality of the integrated circuit that the application engineer can 1.) choose from a model library the integrated circuits with the appropriate functionality, 2.) combine individual integrated circuit models into a composite model of an electronic module, and 3.) perform integrated hardware and software diagnostics on programmable integrated circuits, and 4.) investigate and analyze the impact of replacing an individual integrated circuit during a model year upgrade.

4. Bus functional views shall be documented at the second and third level of decomposition (i.e., module and integrated circuit level) to include interface declaration, pin timing and electrical information, and operating conditions of all the interfaces (10.2.2.1, 10.2.2.2, and 10.2.2.3). This view documents the information required by the electronic module and/or board designer to determine if he has correctly interconnected the integrated circuits on the module. Continued....

G. PREPARED BY **Arnold Bard ARL. (908) 544-4469**	H. DATE **21 July 1992**	L. APPROVED BY **R. A. Reitmeyer ARL. (908) 544-3465**	J. DATE **21 July 1993**

DD FORM 1423-1, JUN 90 ***Previous editions are obsolete*** **Page 1 of 1 Pages**

007/183

DATA ITEM DESCRIPTION	*Form Approved OMB No. 0704-0188*

1. TITLE VHSIC Hardware Description Language (VHDL) Documentation	2. IDENTIFICATION NUMBER DI-EGDS-80811

3. DESCRIPTION / PURPOSE

3.1 VHDL documentation contains behavioral and structural descriptions of an electronic system, subsystem, or device. The primary purpose of these data items is to document hardware designs in a machine processable, simulatable, and hierarchical format.

4. APPROVAL DATE (*YYMMDD*) 890511	5. OFFICE OF PRIMARY RESPONSIBILITY (OPR) N SPAWAR-324	6a. DTIC APPLICABLE	6b. GIDEP APPLICABLE

7. APPLICATION / INTERRELATIONSHIP

7.1 This Data Item Description contains the format and content preparation instructions for the data product generated by the specific and discrete task requirement as delineated in the contract.
7.2 The contract should provide a list of Government approved leaf level modules and a list of VHDL language definitions. The Contracting Officer Technical Representative is responsible for monitoring the status of the list of Government leaf level modules and preparing contract modifications to reflect current leaf level module requirements in the contract.

(Continued on page 2).

8. APPROVAL LIMITATION	9a. APPLICABLE FORMS	9b. AMSC NUMBER N4694

10. PREPARATION INSTRUCTIONS

10.1 Reference documents. The applicable issue of the documents cited herein, including their approval date and dates of any applicable amendments, notices, and revisions, shall be as specified in the contract.

10.1.1 VHDL Manual.
(a) IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987
The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017, USA

10.2 VHDL Documentation Content. VHDL documentation contains behavioral and structural descriptions of the hardware being documented and behavioral descriptions of the VHDL test benches required to demonstrate their functionality.

10.2.1 VHDL module hierarchy. A VHDL description for the hardware shall be a hierarchy of VHDL modules, analogous to the physical hierarchy of the hardware being documented. A VHDL module consists of a VHDL entity declaration, one or more behavioral VHDL bodies, and except for allowable leaf level modules, a structural VHDL body. One VHDL module shall be defined for the entire system and one for each physical electronic unit (assembly, subassembly, integrated circuit, etc.) of the hardware system. VHDL modules should also be defined for important subsections or groupings of complex physical units (e.g., macrocells of a chip or boards defining a processor).

(Continued on page 2)

11. DISTRIBUTION STATEMENT

Block 7, Application/Interrelationship (Continued)

7.3 The DD Form 1423 (Block 16) should contain the requirements for preparation of the deliverable VHDL documentation. The preferred means of delivering all VHDL documentation should be machine readable ASCII files contained on the specific magnetic media and in the machine format required by the Government Activity user. ASCII files are defined as those satisfying character set requirements of the VHDL Language Reference Manual. Requirements for preparation of deliverable hard copy (printed-paper media) documentation should also be provided on the DD Form 1423 (Block 16). The preferred documentation shall be comprised of:

a. Nine-track magnetic tape, 1600 bits per inch, unlabeled, 80-character records, and a blocking factor of 1920 (i.e., 24, 80-character records per block). A label containing text identifying the tape contents shall be affixed to the tape reel.

b. Hard copy (printed on paper) containing the machine loading instructions and the contests of the file 1 and file 2 on the tape (Section 10.3).

Block 10, Preparation Instructions (Continued)

10.2.1.1. Allowable leaf level modules. Leaf level modules are VHDL modules for which no VHDL structural body is required. The only permitted leaf level modules are:

a. Modules selected from a Government list of leaf level modules referenced or contained in the contract.

b. Modules corresponding to a collection of hardware elements which together exhibit a stimulus-response behavior, but whose interaction is best modeled at the electrical or physical level. Examples of such modules are digital logic gates, analog circuit blocks, and power supplies.

c. Modules whose detailed design has not yet been completed but whose behavior is required as a delivery disclosure at specified times during the contract.

10.2.2 Entity declaration The entity declaration for each module shall include an interface declaration, timing and electrical requirements for the behavior of the device, allowable operating conditions, component identification, and explanatory comments.

10.2.2.1 Interface declaration The interface declaration for each entity shall describe all input and output ports. The interface description shall include information which relates each input and output port to a package pin number or connector pin number whenever such a correspondence exists. This information may be in the form of port attributes or port mapping statements which relate functional port names with connector pin numbers.

10.2.2.2 Timing and electrical requirements Timing and electrical requirements (e.g., setup and hold times or power supply voltage extremes) shall be expressed in such a manner as to cause the simulator to generate error messages should the requirement be violated during a simulation.

Block 10, Preparation Instructions (Continued)

10.2.2.3 Operating conditions Operating conditions are the physical and electronic environment in which physical components are designed to operate, such as temperature range, signal excursions, logic level definitions, maximum power dissipation, radiation hardness, etc. VHDL package declarations should be used whenever operating conditions are common across a class of similar components.

10.2.2.4 Entity naming conventions Names for VHDL entities shall be traceable to the names of physical electronic counterparts whenever such a correlation exists.

10.2.3 Behavioral body A behavioral body is an abstract, high-level, VHDL description which expresses the function and timing characteristics of the corresponding physical unit. All user programmable registers should be clearly identifiable in the simulation model. Test and maintenance functions which are part of the physical unit and are available to the user shall be included in the behavioral body. Data flow, procedural and structural constructs may be used for expressing behavior.

10.2.3.1 Decomposition of behavioral bodies Structural decomposition of behavioral bodies shall be used only to show functional partitions which are not clear from the partitions of the corresponding structural body. When determining the appropriate level of hierarchical decomposition, ease of simulation and clarity of behavior should be kept in mind. For example, it may be appropriate to decompose a computer which is made up of several bit-slice microprocessors into composite arithmetic logic units and register files which span portions of several chips. However, decomposing it into Boolean logic primitives (e.g., AND and OR operators) would neither clarify the behavior of the system nor make it easy to simulate.

10.2.3.2 Timing characteristics Signal delays at output ports of the VHDL modules shall accurately model the behavior of the physical units corresponding to the VHDL modules. Best, worst, and nominal outputs delays shall all be included. More elaborate timing models which take into account other variables such as supply voltage or output loading may also be used.

10.2.3.3 Structurally dependent signal values Signal values which are dependent on a particular structural implementation, such as scan path signatures, shall not be specified in the behavioral module.

10.2.4 Structural body A structural VHDL body is composed exclusively of interconnected lower level components. Structural bodies shall represent the physical implementation accurately enough to permit logic fault modeling and test vector generation. Structure which is created to support testing and maintenance such as scan paths shall be included in the VHDL structural description.

10.2.4.1 Structural naming conventions For ease of schematic drawing correlation, and within the constraints of the lexical rules of VHDL, names for components and signals shall be the same as, or traceable to, their electrical schematic counterparts.

10.2.5 VHDL simulation support VHDL test benches which simulate the correct behavior of each VHDL module required by the contract to be simulatable as a stand alone module shall be furnished and clearly distinguished from the VHDL modules representing the design itself.

10.2.5.1 VHDL test benches A VHDL test bench is a collection of VHDL modules which apply stimuli to a module under test (MUT), compare the MUT's response with an expected output, and report any differences between observed and expected responses during simulation. VHDL configuration information required to simulate the MUT shall be included with the test bench.

10.2.5.2 Test requirement correlation VHDL test benches shall be cross-referenced to the contractually required hardware test plans, specifications and drawings.

10.2.5.3 VHDL test bench completeness Every VHDL module of the hardware hierarchy shall be simulatable as a stand alone module and hence a corresponding VHDL test bench is required for every VHDL module of the hierarchy.

10.2.6 Error messages Error messages generated anywhere in either the VHDL description of the actual hardware or the test bench should identify the requirement which has been violated and the name of the VHDL design unit in which the error occurred. Applicable VHDL design units include: entity declarations, structural and behavioral bodies, package declarations, package bodies, and configurations.

10.2.7 Annotations VHDL design units shall include explanatory comments which augment the formal VHDL text to make the intent of the VHDL model clear. The following information is required:

a. Any factors restricting the general use of this description to represent the subject hardware.
b. General approaches taken to modeling and particularly decisions regarding modeling fidelity.
c. Any further information which the originating activity considers vital to subsequent users of the descriptions.

10.2.8 Reference to origin Included in the VHDL documentation shall be a list of VHDL modules new with this deliverable and a list of VHDL modules that have been used without change from VHDL documentation previously accepted by the Government under this contract or VHDL modules selected from the list of Government VHDL modules referenced in the contract. Those modules included from previously existing descriptions shall include:

a. identification of originator or source
b. DoD approved identifier (if one exists)
c. design unit name/revision identifier

10.2.8.1 Revision management VHDL design units, once accepted by the Government, shall be revised only with the approval of the Contracting Officer. A design unit revision history shall be included in comments in each revised design unit (Refer to 10.3, h). The revision history shall include: the date of revisions, the performing individual and organization, the rationale for the revision, a description of where the original design unit required modification and the testing done to validate the revised model.

10.3 VHDL documentation format Each file delivered under contract shall be either a VHDL design file, whose entire contents conform to the requirements of the VHDL Language Reference Manual (including the definition of comments), or an auxiliary information file, containing no VHDL design units. Design units which are new with this contractual deliverable shall not be contained in the same design file with design units which have been previously accepted by the Government. The sequential order of the files of the deliverable shall be:

a. File 1: Names of all files of the deliverable VHDL documentation, named in accordance with the originating host operating system; one file name per record and nothing else (pad with trailing blanks).
b. File 2: High-level prose overview of the VHDL description that cites contract, line item, Contract Data Requirements List sequence number, and summarizes the organization and content of the set of files.
c. File 3: Specification of a sequence for analyzing the VHDL design units of the deliverable that is consistent with the order of analysis rules in the VHDL Language Reference Manual.
d. File 4: List of VHDL modules which were selected

from the Government list of leaf level modules.

e. File 5: List of VHDL modules which are revisions of modules previously accepted by the Government.

f. File 6: List of VHDL modules which originate with this VHDL delivery.

g. File 7: List which associates VHDL modules with their corresponding test benches.

h. File 8 et seqq.: Auxiliary information files concerning the VHDL descriptions and VHDL design files. Auxiliary information files shall precede VHDL design files.

Modifications

1. Performance/un-interpreted/archictural model views at the first level of VHDL module hierarchy (10.2.1) decomposition. This view shall contain timing-only behavior for leaf level entities (10.2.1.1) such as processor nodes, buses/interconnects, inputs, and outputs. This view documents the view required by a system engineer to make high level choices relative to the type of processor, number of processors, and the type of network required.

2. Application model view at the second level of VHDL module hierarchy (10.2.1) decomposition. This view shall describe the full functional behavior (10.2.3) with multi-component electronic modules as the leaf level entities (10.2.1.1). This view documents the functionality of the module such that the system engineer can 1.) choose from a model library the module with the appropriate functionality, 2.) integrate the described module's behavior with the behavioral descriptions of other modules in the system, 3.) can perform integrated hardware and software diagnostics of the system software, and 4.) can investigate and analyze the impact of replacing the module during a model year upgrade.

3. Application model view at the third level of VHDL module hierarchy (10.2.1) decomposition. This view shall describe the full functional behavior (10.2.3) with multi-component electronic modules as the leaf level entities (10.2.1.1). This view documents the functionality of the integrated circuits such that the application engineer can 1.) choose from a model library the integrated circuits with the appropriate functionality, 2.) combine individual integrated circuit models into a composite model of an electric module, and 3.) perform integrated hardware and software diagnostics on programmable integrated circuits, and 4.) investigate and analyze the impact of replacing an individual integrated circuit during a model year upgrade.

4. Bus functional views shall be documented at the second and third level of decomposition (i.e., module and integrated circuit level) to include interface declaration, pin timing and electrical information, and operating conditions of all the interfaces (10.2.2.1, 10.2.2.2, and 10.2.2.3). This view documents the information required by the electronic module and/or board designer to determine if he has correctly interconnected the integrated circuits on the module.

5. Structural views (10.2.4) of multi-component electronic modules with individual integrated circuits as the leaf level entities. This view documents the information required to describe to the module's test engineers, model year upgrade engineers, and maintenance technicians how the components on the module are interconnected.

6. Structural views (10.2.4) of all integrated circuits designed by the RASSP program with register-transfer level cells as the leaf level entities. This view documents information that may be used by the model year upgrade engineers to re-implement the integrated circuit in a newer technology.

7. IEEE 1029.1 (Waveform and vector exchange specification) waves compatible views of input stimulus and output results (10.2.5) for all VHDL test benches at all levels of VHDL module decomposition. This view documents the information necessary to show the correct functionality of the models to anyone utilizing them.

GLOSSARY*

Algorithmic Model. A high-level behavioral model written as "a prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps; for example, a full statement of an arithmetic procedure for evaluating sin(x) to a stated precision.". [IEEE] The inputs and outputs of an algorithm model may not be exactly identical to the realized hardware at the bit level but will provide the same overall functionality as the final system. Moreover, an algorithmic model may not support all of the diagnostic functions of the final realized hardware.

Application-Specific Integrated Circuit (ASIC). A microelectronic device customized for a particular application. Customization of a microcircuit may include programming of programmable read-only memories (PROMs), electrically programmable read-only memories (EPROMs), electrically erasable programmable read-only memories (EEPROMs), and ultraviolet erasable programmable read-only memories (UVEPROMs). It also includes customized circuit designs such as sea of gates, programmable logic arrays (PLAs), programmable logic devices (PLDs), gate arrays, and microelectronic devices designed using standard cells or silicon compilation. [MIL-HDBK-454]

Apply. The WAVES operation that schedules events to a waveform and advances the current time for the waveform.

Architecture Body. A VHDL design unit used to define the behavior or structure of a design entity. There is only one entity declaration permitted for a given architecture body; however, multiple architecture bodies may be generated for a single entity declaration.

ASCII File. The American Standard Code for Information Interchange (ASCII) defines a character set that is used by VHDL'87 for source programs. Any file written in that character set is considered an ASCII file. VHDL'93 uses ISO 8859-1 as its character set.

Assertion Statement. A VHDL statement used to check that a specified condition in a VHDL model is true and reports an error if it is not. [VHDL'93 LRM]

Attribute. A named characteristic that can be associated with VHDL items including types, ranges, signals, and functions. Attributes are used to annotate designs with information in addition to timing, structure, and function. Some attributes are predefined. VHDL provides attribute specifications to specify the values of attributes, and user-defined attributes are supported by VHDL.

Back Annotation. The process of assigning values to attributes as the result of the use of an external assessment tool or when the parameters of an abstract model are updated with accurate values obtained from more detailed models. Back annotation is frequently used to refer to the process of refining delay values based on detailed calculations of fan-out, capacitive loading, and other physical factors. The term is from the process of creating the model and declaring its attributes, then exporting the model to the external assessment tool, and then replacing the attribute values of the model with the more accurate values obtained from the assessment tool.

Behavioral Model. An abstract, high-level VHDL description that expresses the function and timing characteristics of the corresponding physical unit independently of any particular implementation. A behavioral model is a model whose inputs, outputs, functional performance, and timing are known but whose internal implementation is not further defined. [IEEE] Behavioral models are also called black box models or input/output models. "Behavioral model" is also a general term for any VHDL model that is not a structural model.

Block Statement. A VHDL statement that defines an internal block representing a portion of a design. Blocks may be hierarchically nested to support design decomposition. [VHDL'93 LRM]

---

*Some definitions extracted from authoritative sources are annotated with an abbreviation of the source in brackets following the extracted portion. The following abbreviations are used:

[EIA] = EIA-567-A, *VHDL Hardware Component and Modeling Interface Standard*, Electronic Industries Association, Washington, DC, March 1994.

[IEEE] = IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, January 1993.

[VHDL'93 LRM] = IEEE Std 1076-1993, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, April 1994.

[MIL-HDBK-454] = MIL-HDBK-454, *General Guidelines for Electronic Equipment*, 28 April 1995.

[WAVES] = IEEE Std 1029.1-1991, *Waveform and Vector Exchange Specification*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1991.

[Lipsett] = R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Norwell, MA, 1989.

[VLSI] = Joseph DiGiacomo, *VLSI Handbook: Silicon, Gallium Arsenide, and Superconductor Circuits*, McGraw-Hill Book Co., Inc., New York, NY, 1989.

Boards. Physical electronic units designed for easy replacement in a system. Boards are typically the finest grained form of a line-replaceable unit (LRU) found in a military electronic system. An example of a board is the standard electronic module E (SEM-E) package, which is a 6-in. by 5.6-in. board that is approximately 0.10 in. thick.

Boundary Scan. A method of component testing that involves the inclusion of a shift register stage (contained in what is called a boundary scan cell) adjacent to each component pin. The boundary scan cells are connected into a serial shift register chain around the border of the design, and the path is provided with serial input and output connections and appropriate clock and control signals. [IEEE 1149.1] A component designed with boundary scan cells for each of its ports is tested by shifting test vectors into the boundary scan override input ports, executing the function of the component, and then shifting out results from each of the boundary scan output ports.

Boundary Scan Description Language (BSDL). A subset of VHDL that defines formats for attributes of VHDL design entities. These attributes contain much of the information required to analyze boundary scan BIT designs.

Built-In Self-Test (BIST). Any test technique that allows a unit to test itself with little or no need for external test equipment or manual test procedures. A unit may be an integrated circuit, board, or system, and the definition implies that the testing process of input stimulation and output response evaluation is integral to the unit being tested.

Built-In Test (BIT). Any test approach using built-in test equipment or self-test hardware or software to test all or part of the unit itself.

Bus. A signal line or a set of lines used by an interface system to connect a number of devices and transfer information. [IEEE]

Bus Signal. In VHDL, a guarded signal that returns to a default value (specified by the resolution function of the signal) whenever all of its drivers are disconnected.

Bus Controller. Sometimes referred to as a bus interface module, this controller is an electronic component that monitors, provides, and controls access to the bus by one or more processors or I/O devices. A bus controller monitors the status of the bus to determine whether to receive an incoming message and vies for control of the bus with other controllers to send out messages.

Bus Functional Model. *See* Bus Interface Model.

Bus Interface Model. A model of the operation of a component with respect to any busses to which it is connected. A bus interface model combines an incomplete model of the processor and the memory portions of a component with an accurate model of the function and timing of the bus or network interface protocol.

Circuit-Level Model. A model that represents a system in terms of transistors or other elements such as switches or gain blocks. It models the electrical behavior of the system at a lower level than the gate level but at a higher level than a full analog model. Circuit-level models consider multiple (but discrete) signal strengths rather than treating signal values as Boolean values or as continuous values.

Combinational Logic. A logic function where a combination of input values always produces the same combination of output values. (The terms “combinative” and “combinatorial” have also been used to mean combinational.)

Compatibility. 1. The degree to which models may be interconnected and used together in the same simulation without modification. 2. The degree to which a VHDL model can be used by or operated on by simulation, analysis, or design tools, such as synthesis tools. [IEEE]

Complete Model. A VHDL model that defines the interface and behavior of a design and includes an electronic data sheet, a test bench, and other supporting information that explicitly describes the characteristics of the design.

Compliant Model. A VHDL model that meets the requirements of the VHDL data item description (DID).

Component. Any logically separable hardware unit. Components can be combined to form a higher level component by being interconnected; thus components are nodes in the design hierarchy. The VHDL DID requires that VHDL model components correspond to physical or logical components.

Concurrent Statement. A VHDL statement that executes asynchronously with no defined order relative to other statements. [VHDL’93 LRM]

Data Flow Model. A model where the architecture is expressed in terms of data transfer, use, and transformation. [IEEE] A data flow model typically uses concurrent signal assignment statements to compute the output signal values directly from the input signals.

Data Set. A named collection of similar and related data records. A WAVES data set is the complete set of files needed to build a WAVES waveform description. A data set consists of a header file, one or more waves files, and zero or more external files. [WAVES]

Declaration. A VHDL construct that defines a declared entity (such as an entity, object, subprogram, configuration, or package) and associates an identifier with it. [VHDL’93 LRM] This association is in effect within a region of text called the scope of the declaration. Within the scope of a declaration there are places in which it is possible to use the identifier to refer to the associated declaration. At such places the identifier is said to denote the associated declaration.

Delay Model. An algorithm used by a VHDL simulator to update the transaction queue for a signal driver.

Delay Time. The time between activation of a VHDL signal assignment statement and the time the updated signal value is scheduled to appear on the signal.

Design Entity. An entity declaration together with an associated architecture body. [VHDL '93 LRM] Different architecture bodies associated with the same entity declaration are different design entities. Different design entities may share the same entity interface but employ different architecture bodies and thus describe different components with the same interface or different views of the same component.

Design File. A text file containing the source code form of one or more VHDL design units.

Design Hierarchy. The complete representation of a design that results from the successive decomposition of a design entity into subcomponents and the binding of those subcomponents to other design entities that may be decomposed in a similar manner. [VHDL '93 LRM]

Design Library. A host-dependent storage facility where intermediate form representations of analyzed VHDL design units are stored. [VHDL'93 LRM] Models contained in a design library may not be available in source form because of licensing and proprietary data restrictions.

Design Unit. Any block of VHDL code that can be independently analyzed and stored in a design library. A design unit is an entity declaration, an architecture body, a configuration declaration, a package declaration, or a package body. [VHDL'93 LRM]

Direction. A component of a WAVES event that indicates whether the event represents a value that is to be driven by the waveform, i.e., is a stimulus, or the event is an expected value coming from the module under test, i.e., is a response. WAVES allows two possible values for the direction of an event: stimulus and response.

Driver. A VHDL mechanism for creating new values for signals. A driver holds the projected output waveform of a signal. [VHDL'93 LRM] A driver consists of a set of time/value pairs that holds the value of each transaction and the time at which the transaction should occur. [Lipsett] The value of a signal is a function of the current values of its drivers. [VHDL'93 LRM]

Edge Detection System. An electronic system that inputs digital images and detects edges in the image, in which an edge is associated with each significant change in intensity between neighboring pixels of the image.

Electrical View. A view of a VHDL model that specifies the voltage and current characteristics for each pin of a component. [EIA]

Electronic Data Sheet. A set of VHDL packages that describe the parameters, data types, physical types, and functions required for the views of a component supported by EIA-567, such as an electrical view, a timing view, and a physical view.

Entity Declaration. A declaration that defines the interface between a given design entity and the environment(s) in which it is used. It may also specify declarations and statements that are part of the interface. A given entity declaration may be shared by many design entities; each of which has a different architecture. Thus an entity declaration can potentially represent a class of design entities, each with the same interface.

Error. A condition that renders a VHDL source description illegal. If the error is detected at the time of analysis of the design unit containing the error, the detection prevents the creation of a library unit for the given source description. A run-time error causes a simulation to terminate.

Event. In VHDL, a change in the current value of a signal, whereas a WAVES event is the occurrence of an event value at some specified time. A WAVES event has three components: an event value, an event time, and an associated signal on which the event occurs.

Event Value. A WAVES data structure that has four components: a state, a strength, a direction, and a relevance. A WAVES event value defines the requirements upon the waveform passing through the unit under test (UUT) at an instant in time.

Fabrication Process. The collection of mechanical and chemical processes used to create an integrated circuit. These fabrication processes have associated rules constraining the performance and the geometry of circuits developed using the processes.

File Slice. A record in a WAVES external file that describes one or more events occurring at the same time.

Fragment of VHDL. A collection of VHDL source statements that do not constitute a complete VHDL design unit that can be separately compiled.

Frame.The set of events in WAVES defined within a slice for a single signal. A frame represents a list of zero or more events.

Functionally Correct. A design is said to be functionally correct if it provides the correct outputs for all possible inputs. It must be assumed that there are no physical faults in the manufactured system and that no errors are caused by timing problems.

Gate-Level Model. A model that describes a system in terms of Boolean logic functions and simple memory devices, such as flip-flops.

Generic. A VHDL interface constant whose value is not fixed until elaboration. A generic is declared in the block header of a block statement, a component declaration, or an entity declaration. Generics provide a mechanism for communicating static information into a block. Unlike constants, the value of the generic can be supplied externally in a component instantiation statement or in a configuration specification. [VHDL'93 LRM]

Guard. *See* Guard Expression.

Guard Expression. A Boolean-valued expression associated with a block statement that controls assignments to guarded signals within the block. A guard expression defines an implicit signal that may be used to control the operation of certain statements within the block. [VHDL'93 LRM]

Guarded Signal. A signal declared as a register of a bus. Such signals have special semantics when their drivers are updated from within a guarded signal assignment statement. [VHDL'93 LRM] There are two forms of guarded signals: bus signals and register signals.

Header File. A WAVES file that specifies how the data set is to be assembled from the WAVES files and the external files and defines the order of analysis for the WAVES files and standard WAVES units. [WAVES] A WAVES header file identifies the WAVES data set, describes the other files in the data set and their intended use (including the target libraries for VHDL packages), identifies VHDL library and packages that already have been analyzed and will be used in the test bench, and defines the order of analysis for the VHDL source code.

Hierarchical Decomposition. A type of modular decomposition in which a system is broken down into a hierarchy of components through a series of top-down refinements. [IEEE]

Hierarchy. A structure in which components are ranked into levels of subordination, each component has zero, one, or more subordinates, and no component has more than one superordinate component. [IEEE]

Implementation Model. A model that reflects the design of a specific physical implementation of a hardware component. Usually, an implementation model is partitioned into several submodels, and each submodel corresponds to a unique physical subcomponent of the component.

Inertial Delay. A VHDL delay model used for switching circuits. A pulse whose duration is shorter than the switching time of the circuit will not be transmitted. Inertial delay is the default delay mode for VHDL signal assignment statements. [VHDL'93 LRM] An inertial delay model removes spikes in the driving value of the target signal driven by an inertially delayed signal assignment.

Infix Operator. A built-in arithmetic, relational, concatenate, or logical function that is represented syntactically by a symbol or reserved word appearing between its two operands in an expression. For example, the addition operator + is an infix operator that appears between its two operands, as in the expression A + B. VHDL built-in operators can be overloaded so that they operate on different types in addition to their native definition.

Initial Value Expression. An expression that specifies the initial value to be assigned to a variable, signal, or constant.

Instruction Set Architecture (ISA). A model of the complete set of instructions recognized by a given processor. An ISA model describes the externally visible state of a programmable processor and the functions the processor performs. An ISA model of a processor executes any machine program for that processor and gives the same results as the physical machine as long as all input stimuli are sent to the ISA model simulation on the same simulated clock cycle as they arrive at the real processor.

Integrated Circuit. A combination of interconnected components constructed on a continuous substrate.

Interchangeable. Two VHDL models of the same module are said to be interchangeable if one model can be substituted for the other as the description of a component in a larger system model without introducing errors into the system.

Interconnection. A mechanism used for electrical communication between two components; it may also be a model of such a mechanism.

Interface. A model that describes a shared boundary or means of transmitting information between units. For example, the interface for a VHDL design entity describes the ports that connect external signals to the internal functions of the component modeled by the entity. The interface for a VHDL package describes the functions and data structures that can be accessed by a user of the package.

Interface Declaration. In VHDL, a declaration that declares the aspects of a VHDL design unit visible to other units using the declared unit.

Interoperable. Two VHDL models of different modules are said to be interoperable if they can be connected together as components of a larger system model without introducing errors into the system model or into simulations of the system model.

Leaf Module. A design entity for which no VHDL structural architecture body is required. As such, it is a leaf node in the hierarchy of components. Examples of possible leaf modules for a structural VHDL model include power supplies, analog circuit blocks, and digital logic gates. In general, a leaf module will be a behavioral model.

Line-Replaceable Unit (LRU). A hardware component of a system that can be replaced in the field if it is found to be faulty.

Logic-Level Model. *See* Gate-Level Model.

Logic-Level Fault Modeling. Models that represent Boolean "stuck-at" values, stuck-at-0 and stuck-at-1. This modeling is used to test how effectively test vectors detect logic-level faults.

Logic Value. For WAVES, a VHDL enumerated type that names all possible signal values that either can be applied to the external inputs of the MUT or can be sensed as external outputs of the MUT.

Match. The WAVES operation that samples the actual response of the MUT (or its model), compares it with the expected response, and produces a flag depending on whether the response was within the tolerances specified by the WAVES waveform generator procedure.

Microelectronic Devices. Monolithic integrated circuits, hybrid integrated circuits, radio frequency (RF) and microwave (hybrid/integrated) circuits, multichip microcircuits, and microcircuit modules. [MIL-HDBK-454]

Model Reference Library. An implementation-dependent storage facility for a set of executable models that can be simulated in a VHDL simulation environment. Models contained in a model reference library may not be provided in source form because of licensing and proprietary data restrictions. Such restricted models are to be avoided in DoD system designs because they may make the described equipment unsupportable in the long term.

Module. A synonym for a component.

Module Under Test (MUT). The component of a test bench that is being tested.

Object. A VHDL object contains a value of a given type. There are four classes of objects in VHDL'93: constants, signals, variables, and files.

Operating Condition. The physical and electronic environment in which physical components are designed to operate, such as temperature range, signal excursions, logic-level definitions, maximum power dissipation, and radiation hardness. An operating condition for a VHDL model is defined in terms of a set of parameters that specify the aspects of the environment and a specific set of values for those parameters.

Operating Point. A specific simulation condition selected from minimum, maximum, and nominal. [EIA] An operating point specifies the values for the operating condition parameters used in a simulation.

Package. A VHDL design unit that contains declarations and definitions. Packages are used to encapsulate definitions of data types, constants, type conversion functions, and utility functions so that these common definitions can be reused throughout a model or across several models.

Partitioning. The process of decomposing a component into its subcomponents.

Performance. A collection of measures of the quality of a design that relate to the timeliness with which the system reacts to stimuli. Measures associated with performance include utilization, throughput, and latency (response time).

Performance Model. A model with incomplete numerical and internal state precision used early in the design cycle to estimate utilization, throughput, and latency.

Period. The time from the beginning of a WAVES slice to the end of the slice.

Physical View. A view that specifies the relationship between the component model and the physical packaging of the component, such as relating port definitions in the component model to the signal and power pins in the physical implementation of the component.

Pin. An electrical connection to a physical component. Pins are classified as signal pins, power pins, or unconnected pins. [EIA]

Port. A VHDL signal that provides a channel for dynamic communication between a module and its environment. A port is a signal declared in the interface list of an entity declaration, in the header of a block statement, or in the interface list of a component declaration. In addition to the characteristics of signals, ports also have an associated mode that constrains the directions of data flow allowed through the port. [VHDL'93 LRM]

Port Interface List. A list of ports that declares the inputs and outputs of a block, component, or design entity. It consists entirely of interface signal declarations.

Power Pin. An electrical connection through which electrical power is supplied for the operation of a physical device. Power pin specification is necessary for the procurement of physical components, but it is not necessary for simulation in VHDL models. [EIA]

Prime Item. A configuration item is a technically complex item such as an aircraft, missile, launcher equipment, fire control equipment, radar set, or training equipment. A prime item requires a B1-level specification for development. The criteria used to consider a configuration item a prime item are described in MIL-STD-490.

Primitive Data Types. A data type that is one of the data types predefined by VHDL. The VHDL primitive data types are `INTEGER`, `REAL`, `TIME`, `CHARACTER`, `BIT`, `BOOLEAN`, and `SEVERITY_LEVEL`.

Primitive Module. A leaf-level module in a design hierarchy.

Printed Circuit Boards. Boards used to mount components. A conductor pattern in, or attached to, the surface of the printed circuit board provides point-to-point electrical connections for the components mounted on the board. [IEEE]

Process. The basic mechanism in VHDL used to describe behavior. All concurrent signal assignment statements can be represented as equivalent processes.

Processor. A hardware component that has the ability to follow a program or list of instructions stored in a memory (RAM or ROM), which allows it to perform some detailed set of tasks. At the simplest level the instructions may be just a set of parameters. At the most complex level, the instructions may be a compiled Ada program.

Processor-Memory-Switch-Level Model. A model that describes a system in terms of processors, memories, and their interconnections, e.g., busses or networks.

Programmable Device. A hardware component whose behavior can be altered after manufacturing of the device. Programmable devices include processors (whose behavior can be changed by changing the program in memory), programmable logic arrays (PLAs), and field programmable gate arrays (FPGAs).

Prototype. An initial or early version of a system in a nondeployable form and usually created to validate certain aspects of the design. It may not have all of the functionality, appearance, or internal complexity of the expected final design. For example, a computer simulation of a hardware component may be considered a prototype in which all physical and timing characteristics are not represented, but the functional characteristics are represented.

Race Condition. Occurs when the behavior of a device depends on the relative arrival order of signal values at a particular component of the device. Such differences may occur because two or more values are ultimately derived from the same signal by computations having potentially different delays.

Register Signal. A guarded VHDL signal that retains its last driven value whenever all of its drivers are disconnected.

Register-Transfer-Level Model. A model that describes a system in terms of registers, combinational circuitry, low-level busses, and control circuits.

Relevance. The component of a WAVES event value that is used to indicate the significance of the event to the simulation. The possible values for a WAVES event relevance are predicted, observed, and required.

Resolution Function. A user-defined function used to compute the value of a signal that has multiple drivers. A resolution function is required whenever a signal has multiple drivers. The resolution function determines the value of the resolved signal as a function of the collection of inputs from the signal's multiple sources. [VHDL'93 LRM] It is invoked whenever the value of any drivers of the signals changes.

Scope. The range of VHDL text to which a declaration applies. For example, the scope of a declaration of an internal variable in a process includes only that process.

Schematic Capture. The process of electronically drawing and storing a schematic diagram. The schematic capture database can be used with simulation to verify design accuracy. [VLSI]

Sequential Logic. A logic relation in which the combination of outputs of the relation is determined not just by the combination of current input values but also by the history of previous inputs to the relation.

Sequential Statement. A VHDL statement that occurs in the body of a process and is executed in the order in which it appears in the program and as controlled by the control statements of the process. Sequential statements are not executed concurrently.

Signal. A VHDL object with a present value, a past history of values, and a possible set of future values. Signals are objects declared by signal declarations or port declarations and are the mechanisms used in VHDL to connect entities. VHDL processes or signal assignment statements create the possible future values of signals. Those connections to a signal that edit the future value of a signal are called drivers. A signal may have multiple drivers, each with a current value and projected future values.

Signal Pin. An electrical connection through which a component exchanges information with other components of the system. Specification of signal pins is necessary for both procurement and simulation. [EIA]

Signal State. The state of a WAVES event value determined by the logic level of the associated signal. A WAVES event can specify one of three logic levels: low, midband, and high. The midband value is used to indicate uncertainty about the value of the signal at the given time. A VHDL signal does not distinguish between state and strength, but data types can be defined for signals that do make this distinction.

Signal State/Strength Value. In VHDL, the encoding of the signal state and strength into a single value. The possible state/strength values for a VHDL signal can be described by an enumerated data type for the signal. IEEE Standard 1164 defines a standard enumerated type in its `std_logic_1164` package. In WAVES, signal state and strength are treated separately.

Signal Strength. The ability of the specific WAVES signal driver to force a logic level in the face of conflicting logic levels from other signal sources. A WAVES event can specify signal strength as disconnected, capacitive, resistive, drive, or supply.

Simulation. The process of applying stimuli to a model over simulated time and producing the corresponding responses from the model at the simulated times at which those responses would occur in an effort to predict how the modeled system will behave.

Simulation Condition. A description of characteristics of the model used for a specific simulation. For example, a simulation condition would specify whether minimum, maximum, or nominal timing was to be used for the simulation and whether assertions on ports would be executed.

Simulation Model. A model that behaves or operates like a given system when provided a set of controlled inputs. [IEEE] A VHDL model that has been prepared for simulation. A VHDL simulation model has been elaborated, the values of all generic constants are set, the configuration has been determined and implemented, and all instances of each specific component have been created.

Slice. A specification of a portion of a WAVES waveform that is created by a single `apply` operation. A slice occurs in a fixed period of time across all signals of the module under test.

Specification. 1. A document that specifies in a complete, precise, verifiable manner the requirements, design, behavior, or other characteristics of a system or component and often the procedures used to determine whether these provisions have been satisfied. [IEEE] 2. A VHDL specification associates additional information with a previously declared named entity. There are three kinds of specifications: attribute specifications, configuration specifications, and disconnection specifications. [VHDL'93 LRM]

Static Analysis. Any kind of analysis of a VHDL model that does not require simulation. Static analysis can be used to check type or that all guarded signals are resolved.

Status. An optional field in a WAVES header file that describes the status of the test set.

Structural Body. A body of a design entity that is composed exclusively of interconnected lower level components.

Structural Model. A model of the physical or logical structure of a system. A structural model may be hierarchical, i.e., a module in a structural model may itself be a structural model. A structural model describes a system purely in terms of its components and the interconnection of these components.

Synthesis. The process of creating a representation of a system at a lower level of design abstraction from a higher level (more abstract) representation. The synthesized representation should have the same function as the higher level representation; it should also meet all constraints specified to the synthesizer.

Tag. The WAVES operation that adds a textual annotation to the waveform at the current time.

Testability. The degree to which the design of a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. [IEEE]

Test Bench. A collection of VHDL modules that apply stimuli to a module under test (MUT), compare the response of the MUT with the expected output, and report any differences between observed and expected responses during simulation.

Test Controller. An electronic circuit dedicated to controlling the testing of its host system and to collecting and storing or reporting the results of the test.

Test Generation. The process of developing a set of test stimuli, expected responses, and a control program. The control program administers the tests to a module under test (MUT) and compares the responses of the MUT to the expected responses.

Test Interface. A collection of input/output (I/O) ports and a protocol for communication through those ports in which the information that is communicated is test data, commands, and results.

Test Pin. In WAVES, an external signal of the module under test to be stimulated or compared with known outputs.

Test Vector. A set of values for all of the external input signals of a module under test. Inputs are driven with the value, whereas outputs are tested against the given value.

Throughput. The total capability of a system or component to process or transmit data during a specified time period. [IEEE] For example, to require that an edge detection system have a throughput rate of 30 images a second means that the system must be able to consume 30 images a second and produce representations of the edges in each of the images consumed, again at a rate of 30 images a second.

Timing Budget. A hierarchical set of throughput limits or response time limits that partition the timing requirements for a system into timing requirements for the components of the system.

Timing View. A view that specifies the signal propagation and timing constraints associated with each signal pin as a function of the operating point. [EIA]

Trace. In WAVES, a sequence of WAVES events that captures the interaction between a WAVES data set and its environment. [WAVES]

Trace File. An output of a simulation that contains one record per event and is sorted by the times of the events. For each event the time the event took place, the signal whose change in values caused the event, the new value of the signal, and sometimes the VHDL process causing the change in value are included. Some VHDL simulation systems provide these transparently and automatically.

Transaction. An element of a driver that holds a single time and value pair. The time represents the simulation time when the value will become the current value of the driver. A transaction is the result of the execution of a signal assignment statement affecting the associated driver and does not necessarily represent a change to the value of a signal. [VHDL'93 LRM]

Transport Delay. A VHDL delay model in which all intermediate driving values of a signal, regardless of their duration, are preserved. A transport delay is used by the driver of a signal driven by a transport-delayed signal assignment. Transport delay is characteristic of hardware devices (such as transmission lines) that have almost infinite frequency response.

Type. An association of a name with a set of values and a set of operations. The set of operations includes the basic operations and predefined operators, as well as the explicitly declared subprograms that have a parameter or a result of the type.

Type Checking. The process of checking that the types of values transmitted between entity ports, subprogram parameters, expression operands, and objects are syntactically consistent. For example, type checking verifies that the types of values assigned to a signal by all of the signal assignment statements are the same.

Use Clause. A clause that implicitly associates the local VHDL name for a named design unit with the library in which that design unit resides.

Validation. The process of evaluating a model during or at the end of a development process to determine whether it satisfies the specified functional, performance, and interface requirements. [IEEE]

Value Dictionary. A WAVES function that translates the logic values created by the module under test into event values.

Variable. A VHDL object that holds data. It has only a single value that is the current value at any time, which may be changed by assignment. A variable is in contrast with signal drivers, which have a present value, a past history of values, and several future values stored in transactions.

Verification. The process of determining whether the products of a given phase of the development cycle fulfill all of the requirements established during the previous phase. [IEEE]

Verification and Validation. The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with the specified requirements. [IEEE]

VHSIC Hardware Description Language (VHDL). An IEEE standard language to describe digital electronic systems.

Very High-Speed Integrated Circuit (VHSIC) Program. A program that developed technology (including the VHDL) for the design and manufacture of high-speed digital integrated circuits with 1.25 (Phase I) and 0.5 (Phase II) micrometer feature sizes for military applications. Many Phase I VHSICs incorporate built-in test capabilities, and Phase II VHSICs comply with VHSIC interoperability standards.

View. A set of logically related data that represents the significant characteristics of a component with respect to the logical scope of the data. For a VHDL model of a component, a view is typically represented by a set of VHDL packages containing declarations of data, which characterize a view.

Wait Statement. A mechanism within the VHDL to synchronize activities in different processes. A wait statement describes a condition on input signals of a process. Only when those conditions are met will sequential execution of the process continue. A wait statement causes a process to suspend until the conditions given in the wait statement are satisfied, at which point the process resumes sequential execution.

Waveform. 1. A VHDL waveform is a series of time-ordered transactions; each of which represents the value of the driver of a signal. [VHDL'93 LRM] 2. A WAVES waveform is a sequence of time-ordered events across a set of signals [WAVES].

Waveform and Vector Exchange Specification (WAVES). The Waveform and Vector Exchange Specification is a standard method used to describe highly structured sets of test vectors, discrete event simulator output, and automatic test equipment input. WAVES is designed to facilitate the exchange of information between design environments and automatic test equipment. WAVES is expressed as a subset of IEEE Std 1076 VHDL.

Waveform Generator Procedure (WGP). The procedure in a WAVES data set that generates a waveform and monitors the response of the module under test to the waveform.

INDEX

SUBJECT TERM (KEY WORD) LISTING

ASIC
Computer
Computer aided design
Design
Hardware simulation
Integrated circuits
Microelectronics
Modeling
Models
Simulation
VLSI
Very high speed integrated circuits
VHSIC
VHSIC hardware description language
Waveform and vector exchange specification
WAVES

Custodians	Preparing activity:
Army—ER	DLA-ES
Navy—EC	(Project 5962-1236)
Air Force—11	
DLA-DH	

Review Activities:
Air Force 17,19

STANDARDIZATION DOCUMENT IMPROVEMENT PROPOSAL

INSTRUCTIONS

1. The preparing activity must complete blocks 1, 2, 3, and 8. In block 1, both the document number and revision letter should be given.
2. The submitter of this form must complete blocks 4, 5, 6, and 7.
3. The preparing activity must provide a reply within 30 days from receipt of the form.

NOTE: This form may not be used to request copies of documents, nor request waivers, or clarification of requirements on current contracts. Comments submitted on this form do not constitute or imply authorization to waive any portion of the referenced document(s) or to amend contractual requirements.

▨	**1. DOCUMENT NUMBER** MIL-HDBK-62	**2. DOCUMENT DATE *(YYMMDD)*** 96/09/13

3. DOCUMENT TITLE

DOCUMENTATION OF DIGITAL ELECTRONIC SYSTEMS WITH VHDL

4. NATURE OF CHANGE ***(Identify paragraph number and include proposed rewrite, if possible. Attach extra sheets as needed.)***

5. REASON FOR RECOMMENDATION

6. SUBMITTER

a. NAME *(Last, First, Middle Initial)*	**b. ORGANIZATION**	
c. ADDRESS *(Include Zip Code)*	**d. TELEPHONE *(Include Area Code)***	**7. DATE SUBMITTED *(YYMMDD)***

8. PREPARING ACTIVITY

a. NAME Defense Supply Center Columbus	**b. TELEPHONE *(Include Area Code)*** **(1) Commercial** 614-692-0536 **(2) DSN** 850-0536
c. ADDRESS *(Include Zip Code)* Director-VAS 3990 East Broad Street Columbus, OH 43216-5000	**IF YOU DO NOT RECEIVE A REPLY WITHIN 45 DAYS, CONTACT:** **Defense Quality and Standardization Office** **5203 Leesburg Pike, Suite 1403, Falls Church, VA 22041-3466** **Telephone (703) 756-2340 AUTOVON 289-2340**