Chapter 2

Digital Design Strategies and Techniques

Outline

- Design processing steps
- Analog building blocks for digital primitives
- Using a LUT to implement logic functions
- Discussion of design processing steps
- Synchronous logic rules
- Clocking strategies
- Logic minimization
- What does the synthesizer do ?
- Area/delay optimization

Design Processing Steps

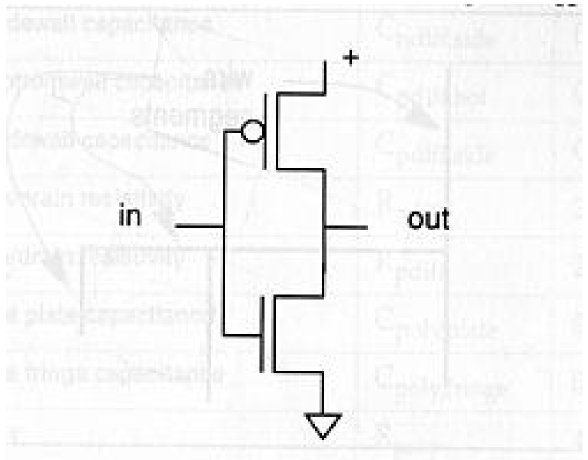
- The processing steps from design to FPGA hardware
  - The design is *parsed for syntax errors*.
  - The design is *minimized and optimized* for the target architecture.
  - *Recognized structure elements* are replaced with selected library modules or cores.
  - *Timing and resource* requirements are estimated.
  - The design is converted to a *netlist*.
  - The design elements and modules are *linked* together and '*black-box*' modules are replaced with library or core module netlists.

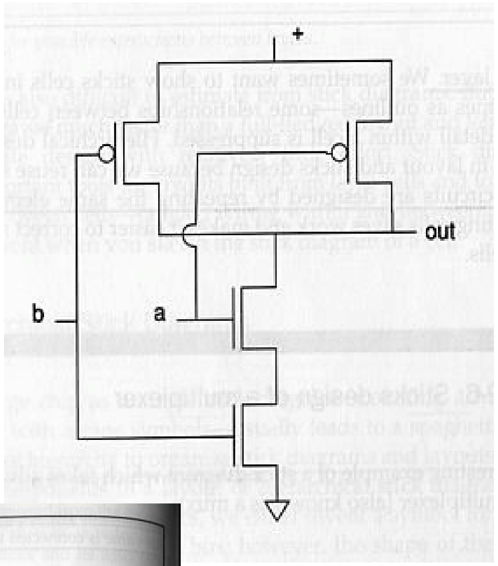
Design Processing Steps

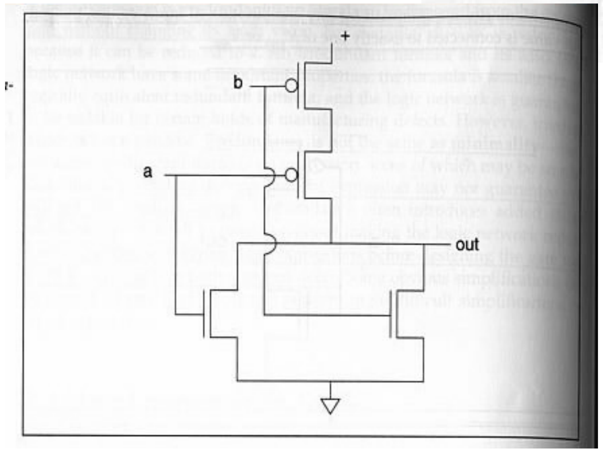
- The processing steps from design to FPGA hardware (cont.)
  - *Floorplanning* and *routing* attempts are made until the *timing* and *resource* constraints are met.
  - Timing and resource reports are extracted from the design. A *timing annotated netlist* is created to support post-routed simulation.
  - The *device configuration* files are created.

Analog Building Blocks for Digital Primitives

- Digital logics are implemented by
  - Transistors
  - Diodes
  - Resistors
- Examples
  - Inverter
    - Fig. 2-1
  - NAND gate
    - Fig. 2-2
  - NOR gate
    - Fig. 2-3







Using a LUT to Implement Logic Functions

- Most FPGAs use a multiplexer (MUX) Look-Up Table (LUT) as a basic logic element.
  - The LUT is versatile
    - Any function of the inputs is possible
    - The LUT is efficiently implemented in silicon
  - Configuration
    - MUX control inputs
      - Logic inputs
    - MUX data inputs
      - Be strapped to logic levels to implement the desired function

Using a LUT to Implement Logic Functions

- Example
  - An inverter implemented using MUX

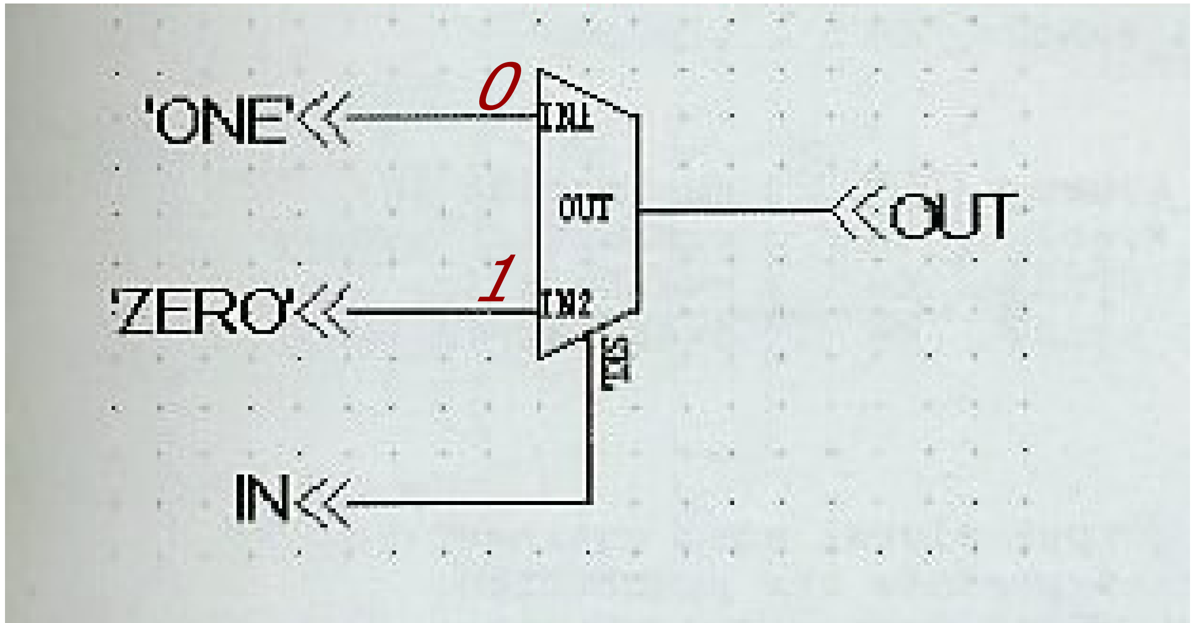


Figure 2-4 MUX Configured as an Inverter

Using a LUT to Implement Logic Functions

- Synthesis example
  - Overheat detector

```
module overheat (clock, reset, overheat_in, pushbutton_in, overheat_out);
input           clock, reset, overheat_in, pushbutton_in;
output          overheat_out;
reg             overheat_out;
reg             pushbutton_sync1, pushbutton_sync2;
reg             overheat_in_sync1, overheat_in_sync2;

// Always synchronize inputs that are not phase related to
//  the system clock.
// Use double-synchronizing flipflops for external signals
//  to minimize metastability problems.
// Even better would be some type of filtering and latching
//  for poorly behaving external signals that will bounce
//  and have slow rise/fall times.

always @ (posedge clock or posedge reset)
begin
                if (reset)
                                begin
                                pushbutton_sync1 <= 1'b0;
                                pushbutton_sync2 <= 1'b0;
                                overheat_in_sync1 <= 1'b0;
                                overheat_in_sync2 <= 1'b0;
                                end
                else            begin
                                pushbutton_sync1 <= pushbutton_in;
                                pushbutton_sync2 <= pushbutton_sync1;
                                overheat_in_sync1 <= overheat_in;
                                overheat_in_sync2 <= overheat_in_sync1;
                                end
end

// Latch the overheat output signal when overheat is
//  asserted and the user presses the pushbutton.
always @ (posedge clock or posedge reset)
                begin
                if (reset)
                                overheat_out <=1'b0;

// Overheat_out is held forever (or until reset).
                else if (overheat_in_sync2 && pushbutton_sync2);
                                overheat_out <=1'b1;
                end
endmodule
```

Using a LUT to Implement Logic Functions

- Graphical version of the netlist

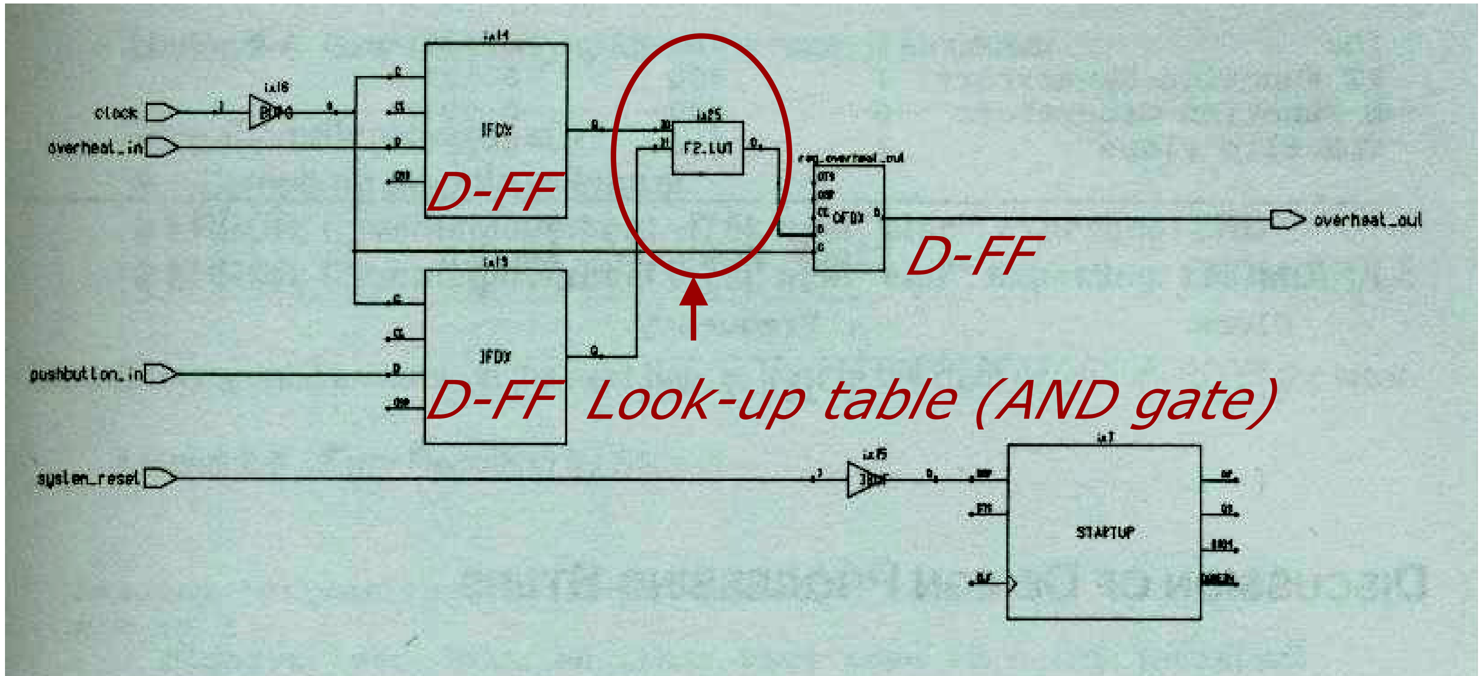


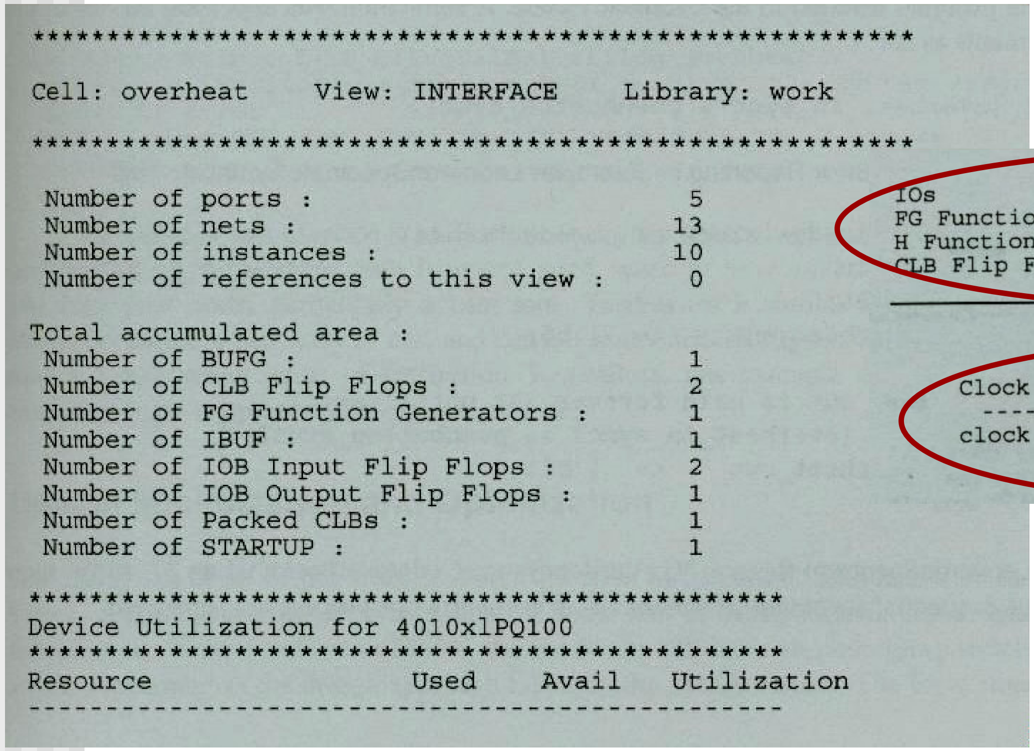
Figure 2-5 Schematic Extracted from the EDIF Netlist

Using a LUT to Implement Logic Functions

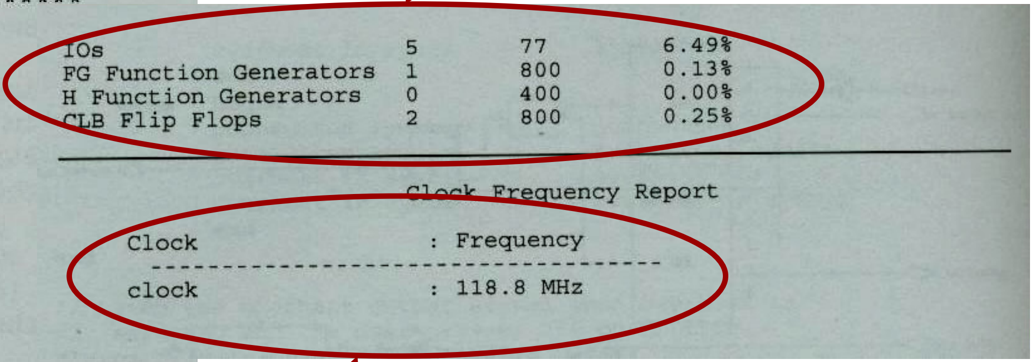
- Graphical version of the netlist
  - Notes
    - The correct use of *global* resources for *clock* and *reset*
    - Using a *LUT for AND gate* that is located between two DFF's
    - Verilog does not support direct assignment of hardware resources
      - It is the designer's job to assure that these inferences are made correctly.

Using a LUT to Implement Logic Functions

- Estimation of the timing and resource requirements



```
*****************************************************************

Cell: overheat     View: INTERFACE     Library: work

*****************************************************************

 Number of ports :                          5
 Number of nets :                          13
 Number of instances :                     10
 Number of references to this view :        0

Total accumulated area :
 Number of BUFG :                           1
 Number of CLB Flip Flops :                 2
 Number of FG Function Generators :         1
 Number of IBUF :                           1
 Number of IOB Input Flip Flops :           2
 Number of IOB Output Flip Flops :          1
 Number of Packed CLBs :                    1
 Number of STARTUP :                        1

***********************************************
Device Utilization for 4010xlPQ100
***********************************************
Resource                   Used    Avail    Utilization
-----------------------------------------------
```



Resource information
Timing information

```
IOs                        5       77        6.49%
FG Function Generators     1       800       0.13%
H Function Generators      0       400       0.00%
CLB Flip Flops             2       800       0.25%
________________________________________________________

                     Clock Frequency Report

    Clock                    : Frequency
   ------------------------------------
    clock                    : 118.8 MHz
```

Discussion of Design Processing Steps

- Syntax checking
  - Identifying the syntax, typing, and other errors
  - Where is the error ?
    - A semicolon is appended on one of the *if* statement *(Listing 2-3, p. 48)*
  - Different tools give different error messages
    - It makes good sense to have several tools available for checking your code.

Discussion of Design Processing Steps

- Design minimization and optimization
  - Hardware configuration
    - FPGA
    - Semi-custom FPGA
    - ASIC
  - The synthesis result is related to the logic structure of FPGA

Discussion of Design Processing Steps

- Xilinx 4K family Configuration Logic Block (CLB)
  - Two 4-input LUTs feeding a pair of flipflops

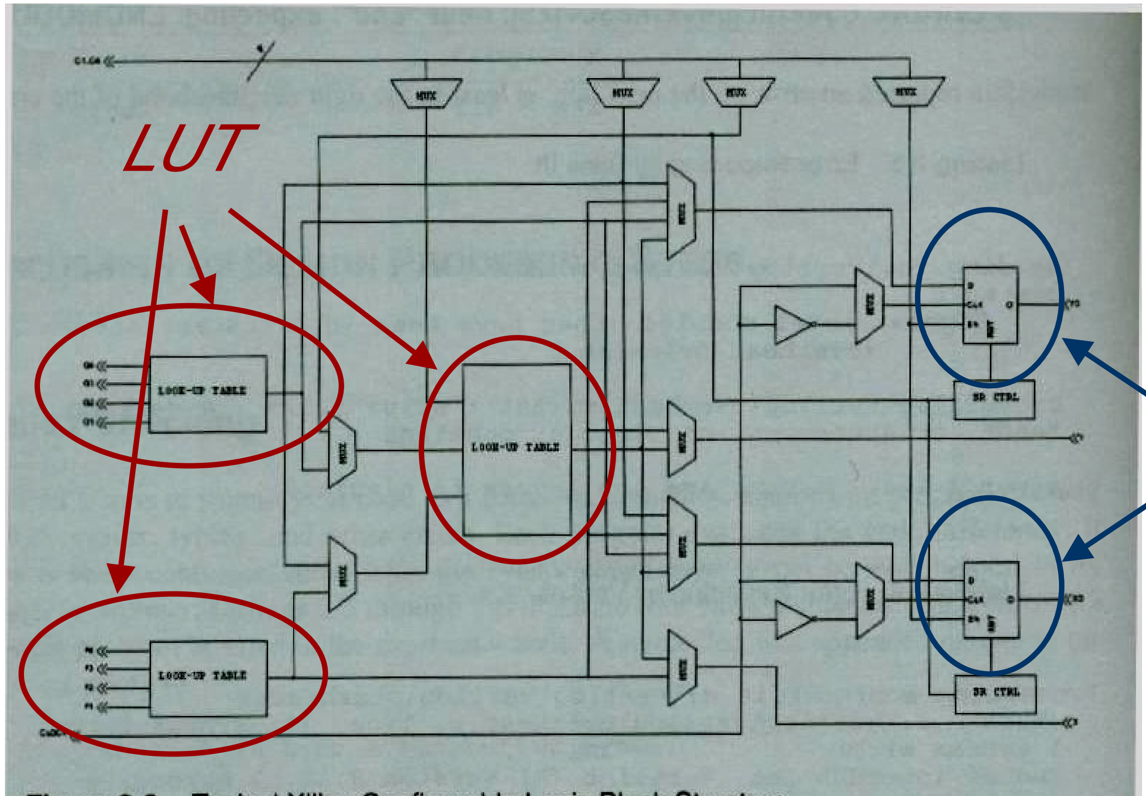


Figure 2-6 Typical Xilinx Configurable Logic Block Structure

Discussion of Design Processing Steps

- Synthesis process steps
  - *Step 1*
    - *Flattening* the design into *large Boolean equations* with *one equation* for each module output, design section output, or register output

Discussion of Design Processing Steps

- Example
  - A simple 2-bit adder code

```
module adder (clock, reset, a, b, c); // Simple adder (no carry input).
          input     clock, reset, a, b;
          output    c;
          reg       [1:0]     c;
          always @ (posedge clock or posedge reset)
          begin     if (reset)
                              c = 2'b0;
          else
                              c = a + b;          // Adder.
                    end
endmodule
```

Discussion of Design Processing Steps

- *Truth table* for simple adder example
  - C[0] could be represented by an XOR gate
  - C[1] could be represented by an AND gate

a	b	C[1]	C[0]
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Discussion of Design Processing Steps

- A flattened version of the simple adder example
  - C[0] <= a^b;
  - C[1] <=a&b;

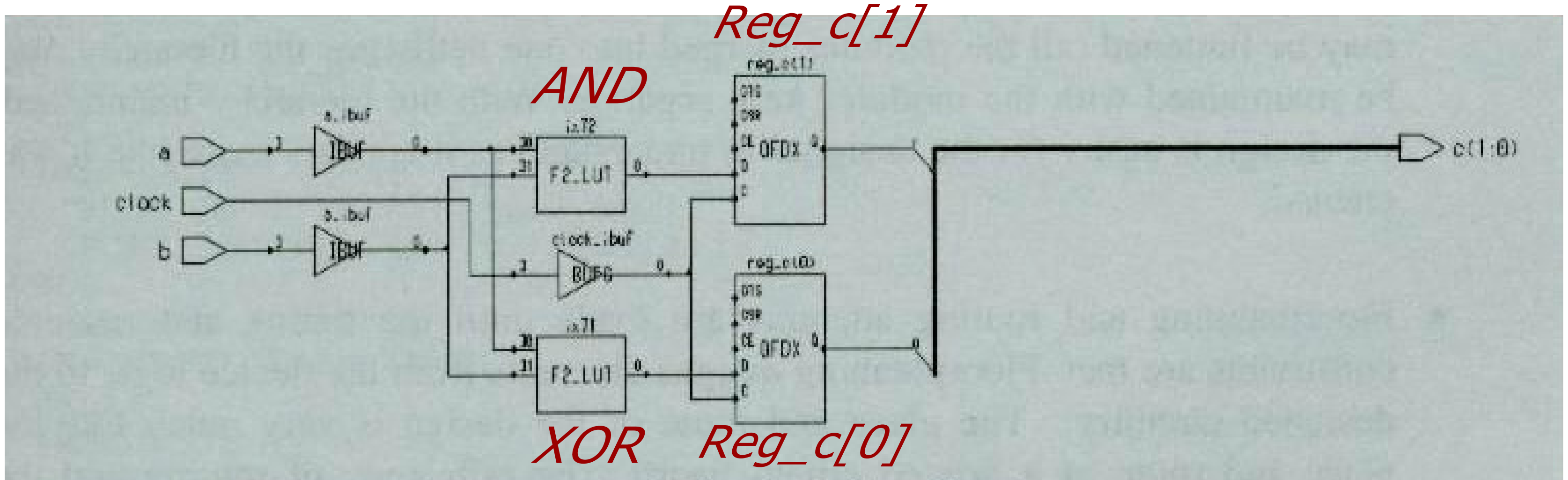


Figure 2-7 Flattened Schematic of Simple Adder Circuit

Discussion of Design Processing Steps

- Synthesis process steps
  - *Step 2*
    - *Minimizing the Boolean equations*
      - By recognizing and removing redundant logic terms
  - *Step 3*
    - Recognized structure elements are replaced with selected modules.
      - E.g. a<=a-1 → Down counter
      - Replacing the logic with a pre-defined circuit optimized for the target architecture for either speed or area

Discussion of Design Processing Steps

- Synthesis process steps
  - *Step 4*
    - *Timing and resource* requirements are *estimated* by the compiler
      - *May not be accurate enough*
      - *Why the estimation is not accurate ?*
        - The manufacturer may have made *changes* to the timing parameters.
        - The *library* and *black-boxes* are *not* yet part of the design netlist.
  - *Step 5*
    - The design is converted to a netlist
      - The most common one is EDIF format

Discussion of Design Processing Steps

- Synthesis process steps
  - *Step 6*
    - The design elements and modules are *linked* together and '*black-box*' modules are replaced with library module netlists.

Discussion of Design Processing Steps

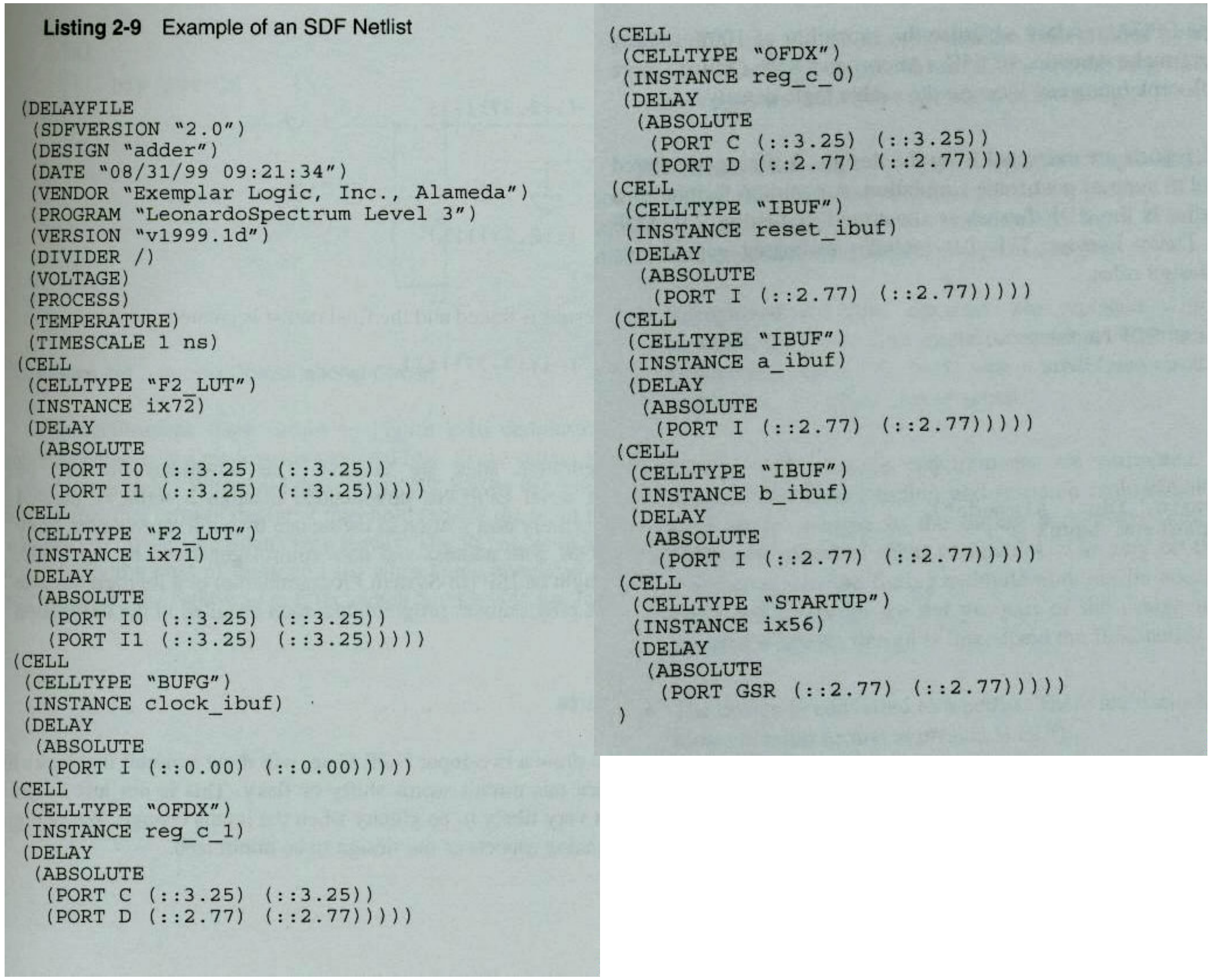
- Synthesis process steps
  - *Step 7*
    - *Floorplanning* and *routing* attempts are made until the *timing* and *resource* constraints are met.
      - Floorplanning assigns elements from device logic to the designed circuitry.
      - The efficiency of routing and the resulting speed of the routed design depend on the *arrangement of the module element*.
    - Typical routing densities of FPGAs
      - Altera → *65%*
      - Xilinx → *85%*
      - *Capability of 100% routing of all logic is only an advertisement.*

Discussion of Design Processing Steps

- Synthesis process steps
  - *Step 8*
    - Timing and resource reports are extracted from the design. A *timing annotated netlist* is created to support post-routed simulation
      - A common format is SDF format
        - SDF→ Standard Delay Format
      - Example
        - Listing 2-9

Discussion of Design Processing Steps

- Synthesis process steps (SDF netlist)



Listing 2-9 Example of an SDF Netlist

```
(DELAYFILE
 (SDFVERSION "2.0")
 (DESIGN "adder")
 (DATE "08/31/99 09:21:34")
 (VENDOR "Exemplar Logic, Inc., Alameda")
 (PROGRAM "LeonardoSpectrum Level 3")
 (VERSION "v1999.1d")
 (DIVIDER /)
 (VOLTAGE)
 (PROCESS)
 (TEMPERATURE)
 (TIMESCALE 1 ns)
(CELL
 (CELLTYPE "F2_LUT")
 (INSTANCE ix72)
 (DELAY
  (ABSOLUTE
   (PORT I0 (::3.25) (::3.25))
   (PORT I1 (::3.25) (::3.25)))))
(CELL
 (CELLTYPE "F2_LUT")
 (INSTANCE ix71)
 (DELAY
  (ABSOLUTE
   (PORT I0 (::3.25) (::3.25))
   (PORT I1 (::3.25) (::3.25)))))
(CELL
 (CELLTYPE "BUFG")
 (INSTANCE clock_ibuf)
 (DELAY
  (ABSOLUTE
   (PORT I (::0.00) (::0.00)))))
(CELL
 (CELLTYPE "OFDX")
 (INSTANCE reg_c_1)
 (DELAY
  (ABSOLUTE
   (PORT C (::3.25) (::3.25))
   (PORT D (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "OFDX")
 (INSTANCE reg_c_0)
 (DELAY
  (ABSOLUTE
   (PORT C (::3.25) (::3.25))
   (PORT D (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "IBUF")
 (INSTANCE reset_ibuf)
 (DELAY
  (ABSOLUTE
   (PORT I (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "IBUF")
 (INSTANCE a_ibuf)
 (DELAY
  (ABSOLUTE
   (PORT I (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "IBUF")
 (INSTANCE b_ibuf)
 (DELAY
  (ABSOLUTE
   (PORT I (::2.77) (::2.77)))))
(CELL
 (CELLTYPE "STARTUP")
 (INSTANCE ix56)
 (DELAY
  (ABSOLUTE
   (PORT GSR (::2.77) (::2.77)))))
)
```

Discussion of Design Processing Steps

- Synthesis process steps
  - *Step 9*
    - The *device configuration* files are created.
    - Way of programming
      - EPROM programming
      - Downloading through serial or parallel cable
      - Stored in memory
    - Programming devices
      - Micro-processors
      - Stand-alone programmer
    - Device types
      - ISP (In-System Programming) type
      - Re-programmable type

Discussion of Design Processing Steps

- Shifty (詭詐的) logic circuits
  - Example 1
    - 2-input NOR gate
      - Seems shifty or flaky (古里古怪的)
      - The output is very likely to be glitchy when the inputs change.
      - Source for glitches→*Switch bounces*

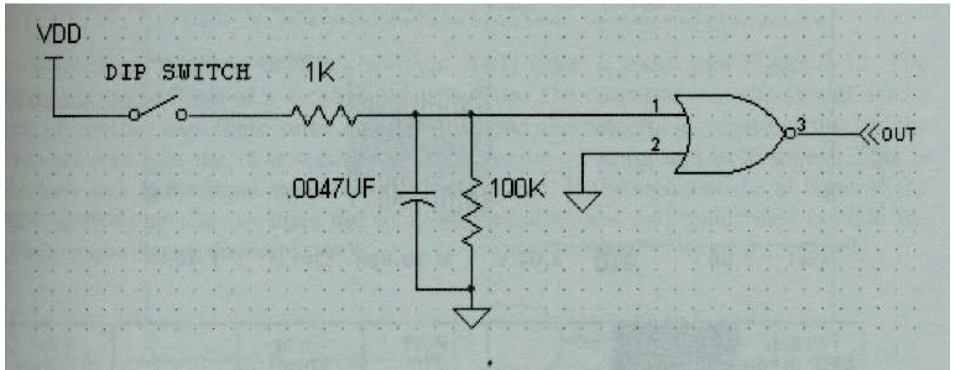


Figure 2-9 Simple Combinational Circuit

Discussion of Design Processing Steps

- Oscilloscope trace of 2-input NOR
  - Switch bounce occurs in DIP switch
    - To use feedback (hysteresis) to filter switch bounce

Node 3

Node 1

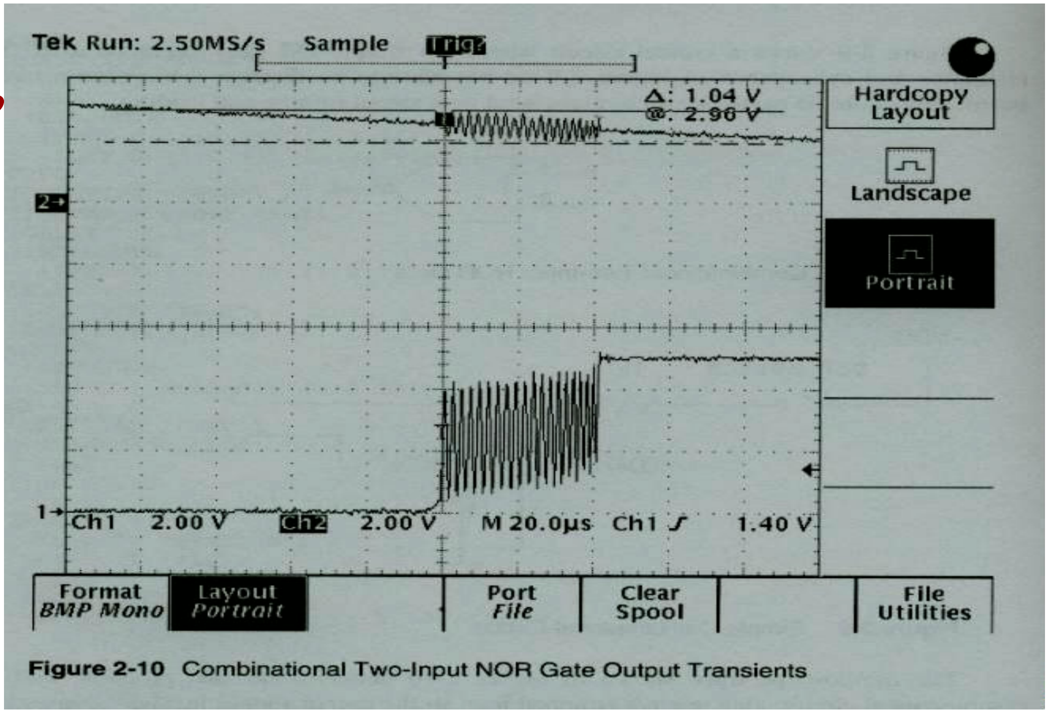


Figure 2-10 Combinational Two-Input NOR Gate Output Transients

Discussion of Design Processing Steps

- Another source for glitches
  - *RC time delay* causing signals traveling with different arrival time
  - R represents
    - The sum of source and routing impedance
  - C represents
    - The net loading
    - Could be controlled by setting fanout constraint in synthesis
  - Example 2
    - 2-input AND gate with RC network

Discussion of Design Processing Steps

- Combinational AND gate

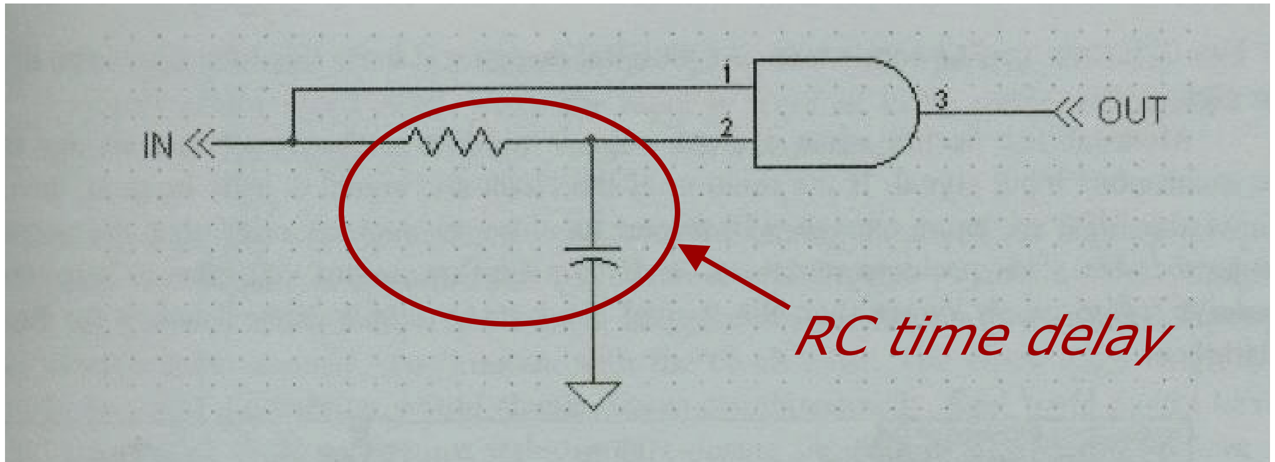


Figure 2-11 Combinational Two-Input AND Gate with RC Network

Discussion of Design Processing Steps

- Solution
  - Synchronous AND gate
  - Adding a D-FF in the output node to remove the glitches by meeting
    - Synchronous logic rules
    - Setup and hold times of the D-FF

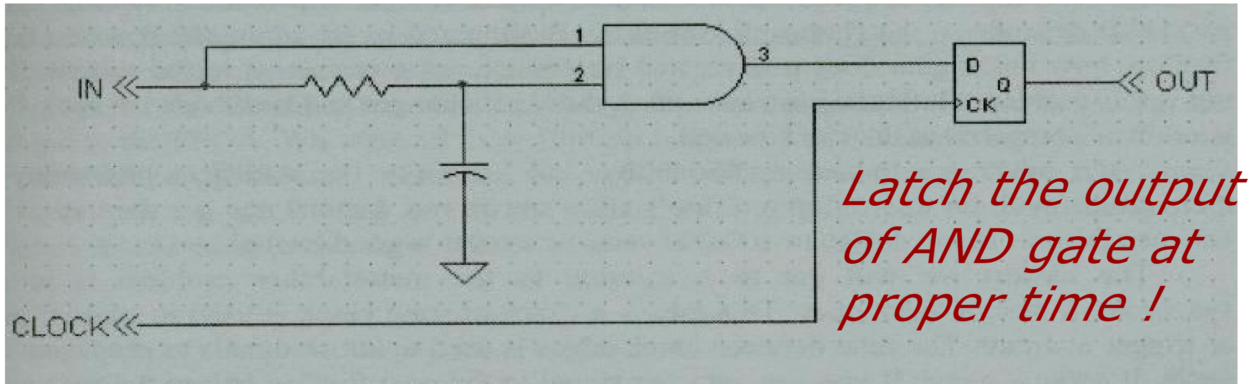


Figure 2-12 Synchronous AND Gate

Synchronous Logic Rules

- Metastability
  - Occurs when a *clock edge* is *random* with respect to a change of an *asynchronous input*
  - The FF output may take a longer time than typical clock-to-Q delay of FF data sheet

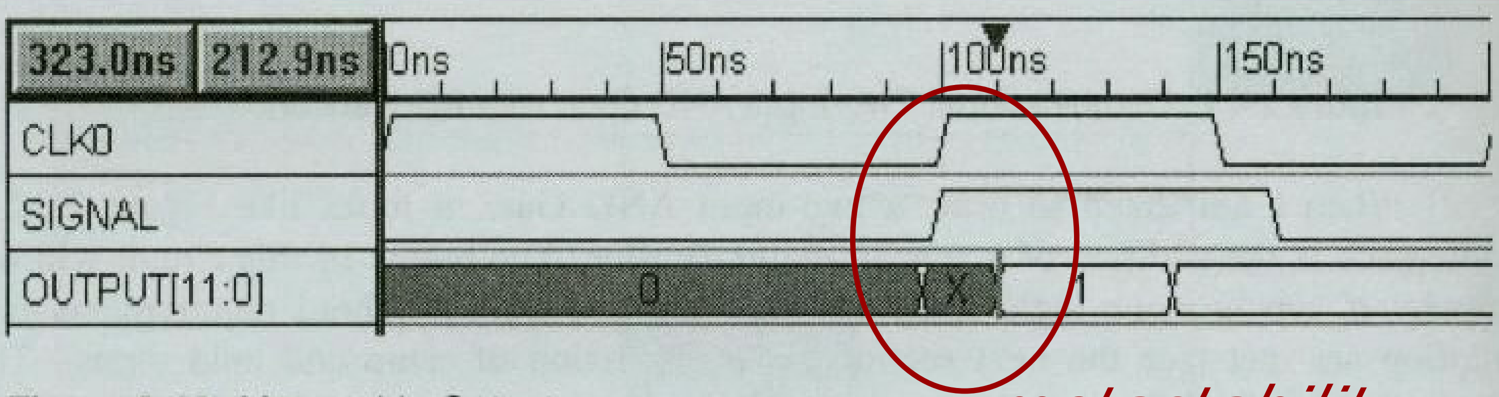


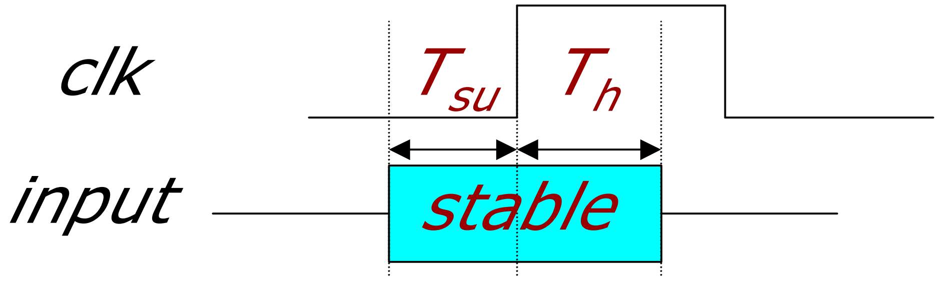
Figure 2-13 Metastable Output

Synchronous Logic Rules

- Solution to the metastability
  - Using *synchronous design technique*
    - Adopting a synchronizing clock to qualify, gate, or trigger a circuit
    - The time between clock edges (clock period) is used to allow signals to propagate and settle
  - *Narrowing the metastability window*
    - Narrowing the setup-and-hold time period by *increasing the speed of FF*

Synchronous Logic Rules

- Setup and hold times
  - The inputs must *meet the setup and hold times* so that the output of FFs is not metastable (can be predictable).
    - The input changes in T_{su} earlier than the synchronizing clock edge
    - The input is required to be stable in T_h later than the synchronizing clock edge

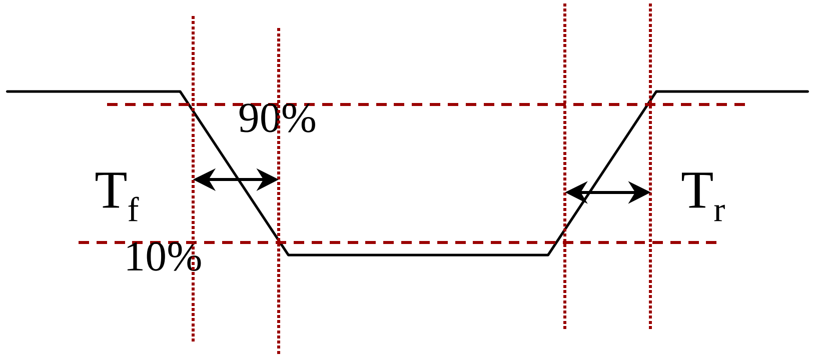


Synchronous Logic Rules

- *Where* do the setup and hold times occur ?
  - *Coming from the analog nature of the flipflop design*
    - The flipflops use *feedback* implemented with cross coupled gates to hold a state.
    - It *takes time* for the FF to achieve their stable state.

Synchronous Logic Rules

- Real World design
  - Real World clock has *rise/fall times*
    - *Do not change abruptly*



  - FF requires *stable inputs during setup/hold time*

Synchronous Logic Rules

- Asynchronous input problem
  - When an *asynchronous input* drives *multiple flipflops*, and the input *changes near the clock edge*, some flipflops outputs *will* change and some *will not*.
  - *RC delay* is caused by the FPGA routing and loads

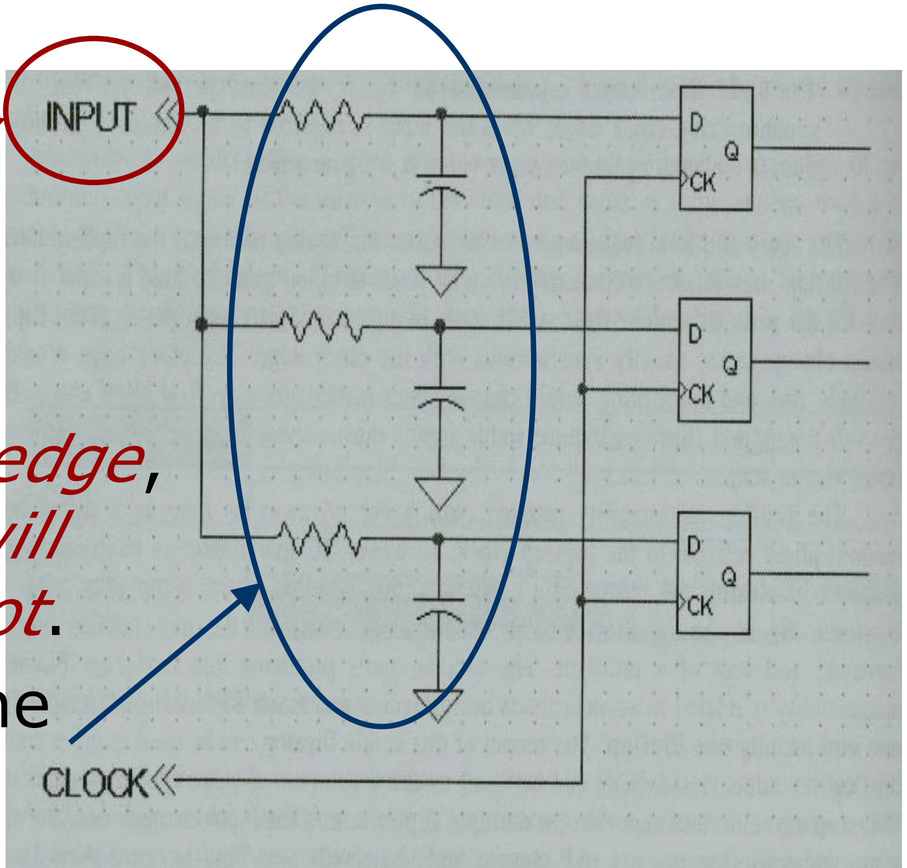


Figure 2-14 Asynchronous Input Problem

Synchronous Logic Rules

- Solution to the asynchronous input problem
  - *Synchronizing* the input with a *single* flipflop
  - *Do not violate* the setup/hold time requirement

Synchronous Logic Rules

- How can I create a nearly trouble-free design ?
  - Always synchronize your inputs !
    - An *asynchronous input* drives *exactly one* FF.
    - The *output of this FF* can be *safely* used to drive the rest of your *synchronous circuitry*.

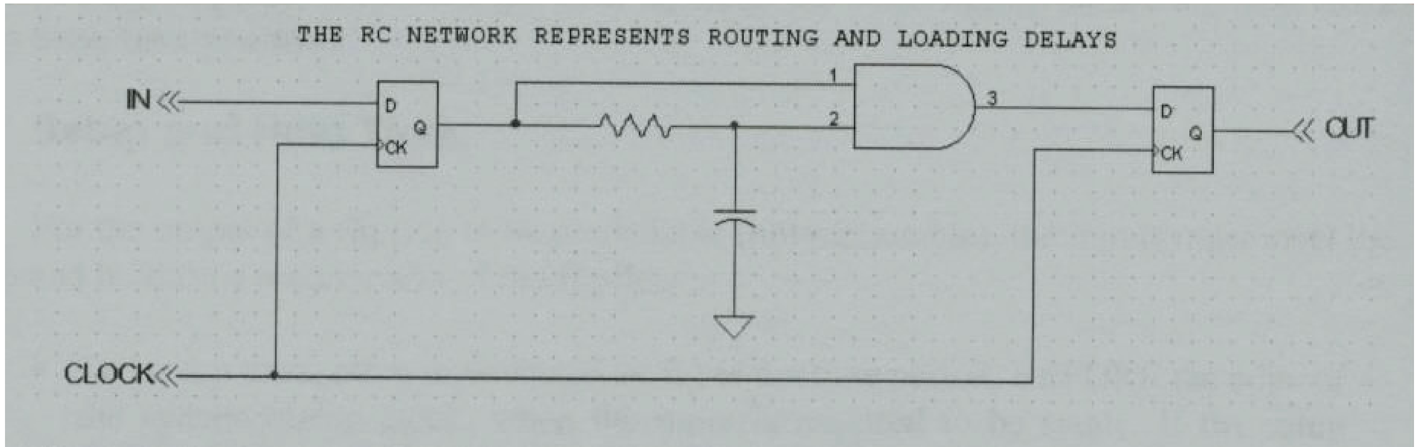


Figure 2-15 Synchronous AND Gate with Synchronous Inputs

Synchronous Logic Rules

- Estimating the max. clock frequency
  - Example: divided-by-two circuit

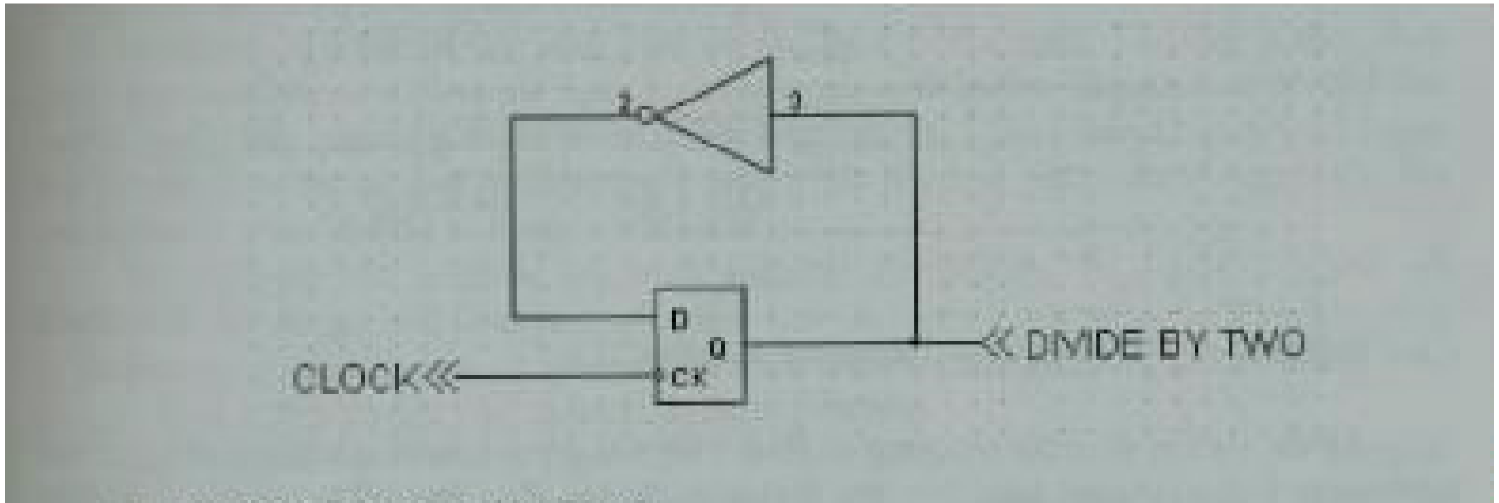


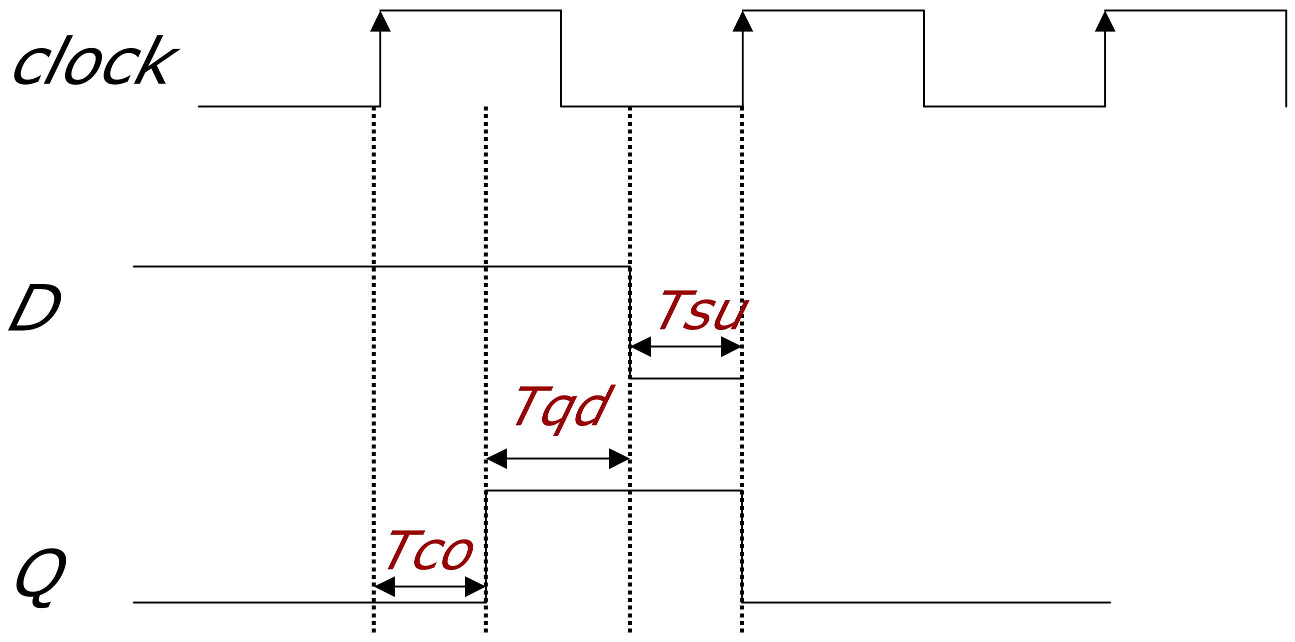
Figure 2-18 Divide-by-Two Circuit

Synchronous Logic Rules

- D-FF specifications
  - FF minimum input set-up time
    - Tsu=1ns
  - FF minimum input hold time
    - Th=1ns
  - Maximum clock-to-output delay
    - Tco=1ns
  - Maximum propagation delay (Q to D)
    - Tqd=1ns

Synchronous Logic Rules

- Timing calculation



Minimum clock period = 3ns→ Max clock freq=333.33MHz

Synchronous Logic Rules

- Cascaded FF v.s. Timing delay
  - How can this circuit work reliably ?(Fig.2-17)
    - What if the min. clock-to-output delay of U1 is less then the hold time requirement of U2 ?
      - *U2 will directly latch the updated value instead of the old value of U1 !!!*
      - *This is not what we want !!!*
  - *FPGA chip designer will create a logic cell that assure the circuit will work !!!*

Synchronous Logic Rules

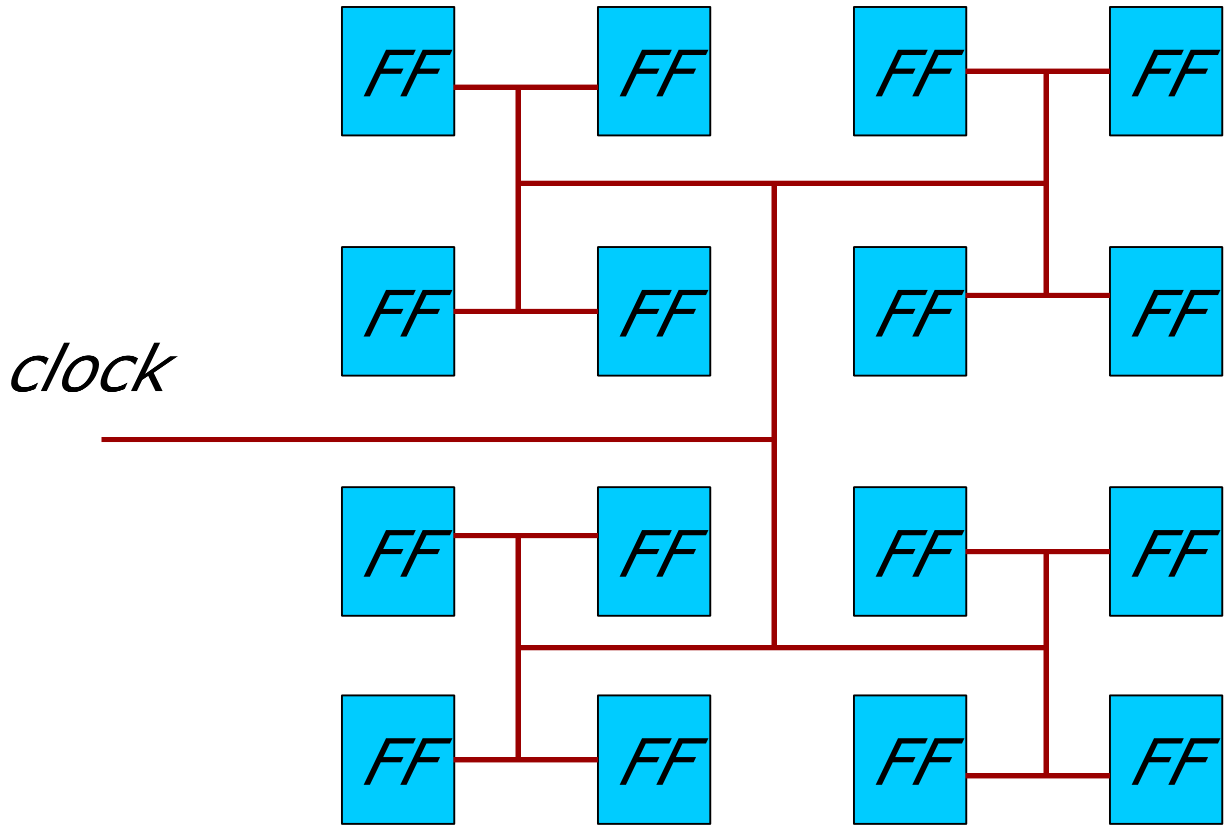
- Clock skew problem
  - Example: two FFs connected in series with clock skew (Fig. 2-19)
  - What is the problem ?
    - *The clock skew will cause the U2 always latches the newest value instead of the old value from U1 !*
    - *Or U2 may latch the unknown value due to the violation of input set-up time caused by the clock skew !*

Synchronous Logic Rules

- What is the solution ?
  - *Minimizing the clock skew situation*
    - FPGA
      - The FPGA chip designer has provided low-skew clock network to assure that the *longest skew of the clock* anywhere across the device *is shorter than* the shortest sum of clock-to-Q and signal routing propagation delay.
    - ASIC
      - H-tree (or balanced) clock distribution network to reduce the clock skew situation

Synchronous Logic Rules

- H-tree clock distribution



Clocking Strategies

- The most important decision the FPGA designer makes is
  - Clocking strategies
    - This must be considered carefully.
    - An error in clocking can doom a design.

Clocking Strategies

- Suggested clocking strategies (1)
  - When designing an ASIC, if power consumption is NOT an issue,
    - Use one master clock on all FFs
    - Replace lower-frequency clocks with clock enables
    - The design has one clock, which results in the simplest timing analysis.

Clocking Strategies

- Suggested clocking strategies (2)
  - When designing an ASIC, if power consumption IS an issue,
    - Running FFs in lower clock frequency in selected parts of the design is OK.
    - Make the design smaller and/or reduce the clock frequency to reduce the power consumption.
    - Minimizing the amount of circuitry running at high speed.
    - Be forced to deal with the problem of synchronizing signals crossing clock domains in exchange for reduced power consumption.

Clocking Strategies

- Suggested clocking strategies (3)
  - When designing an ASIC, but using FPGAs as prototypes,
    - *For fast FPGAs (ASIC-like)*
      - Running the design with one master clock
    - *For slow FPGAs (SRAM-based devices)*
      - Will probably be forced to run modules at the lowest possible speed to get the design work.
      - Drive FFs with multiple clocks,
      - Creating clocks in a central clock generator module,
      - Minimizing the interconnection between clock domains,
      - Making sure signals that cross clock domains are properly synchronized.

Clocking Strategies

- Suggested clocking strategies (4)
  - When doing a *fast* FPGA design,
    - Use up the global clock resources
    - Partitioning the design to minimize signals crossing clock domains
    - Synchronizing the signals properly.
  - When doing a *slow* FPGA design,
    - Either strategy is fine.

Clocking Strategies

- Clock enable
  - Verilog HDL *does not* support dedicated clock enable signals.
  - FPGA synthesis vendors *may provide* this support through compiler directives.
    - The code in Listing 2-10 may be synthesized into *different ways* depending on the targeted FPGA.

Clocking Strategies

- Clock-Enable example (Listing 2-10)

```
module clock_en(out, in, clock, clock_enable1, clock_enable2, reset);

output   out;
input    in, clock, clock_enable1, clock_enable2, reset;
reg      out;

always @ (posedge clock or posedge reset)
         begin
         if (reset)
                  out<=0;
         else if (clock_enable1)
                  out<=out;           // Hold output if not enabled.
         else
                  out<=(in & clock_enable2);
         end
endmodule
```

Clocking Strategies

- Synthesized clock enable
  - Fig. 2-22

Clocking Strategies

- In synthesis, the synthesizer may insert *added logic* into the clock-enable path.
  - Fig. 2-23

Clocking Strategies

- The clock enable implementation in a technology that *does not have a clock-enable feature in its logic block*.
  - Fig. 2-24

Logic Minimization

- A synthesizer can recognize and remove redundant logic.
  - Listing 2-11 v.s. Listing 2-12

Listing 2-11

```
input     test1, test2, test3;
output    sample;

sample = ((test1 & test2 & test3) | (test1 & !test2 & test3)
          | (test1 & test2 & !test3));
```

Using K-map

Listing 2-12

```
input     test1, test2, test3;
output    sample;

sample = ((test1 & (test2 | test3));
```

The compiler can also recognize equivalent logic equations

Logic Minimization

- The synthesizer will not be able to extract the redundant logic if the logic is spread over across register boundaries.
  - Fig. 2-26

Logic Minimization

- How can we take advantages of the synthesis tool's capability to minimize the logic ?
  - *Never create purely combinational modules !!!*
  - *None of the popular FPGA architectures have purely combinational logic elements.*
  - *Mixing the combinational logic with the synchronous logic to allow the synthesis tool to merge the logic into the resources available in the device.*

Logic Minimization

- Combinational logic clouds feeding flipflops
  - Fig. 2-27

What does the synthesizer do ?

- Synthesizer's role
  - Take Verilog HDL and maps it into hardware
  - Procedures
    - *Minimize logic equations* by removing redundant logic terms
    - *Map the resulting Boolean equations onto the available hardware elements in FPGAs*
  - Example
    - 7-segment decoder

What does the synthesizer do ?

- Truth table for 7-segment decoder

BCD input *Segment output*

B3	*B2*	*B1*	*B0*	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

What does the synthesizer do ?

- Let us consider the output of “a” segment

- The reduced equation for “a” segment

What does the synthesizer do ?

- Verilog design for 7-segment display decoder “a” term

```
module seven_seg (clk, reset, bcd_input, a_segment);

input                   clk, reset;
input [3:0]             bcd_input;
output                  a_segment;
reg                     a_segment;

always @ (posedge clk or posedge reset)
            if (reset)
                        a_segment<=0;
            else
            begin case (bcd_input)
{1'b0, 1'b0, 1'b0, 1'b0}: a_segment <= 1'b1;
{1'b0, 1'b0, 1'b0, 1'b1}: a_segment <= 1'b0;
{1'b0, 1'b0, 1'b1, 1'b0}: a_segment <= 1'b1;
{1'b0, 1'b0, 1'b1, 1'b1}: a_segment <= 1'b1;
{1'b0, 1'b1, 1'b0, 1'b0}: a_segment <= 1'b0;
{1'b0, 1'b1, 1'b0, 1'b1}: a_segment <= 1'b1;
{1'b0, 1'b1, 1'b1, 1'b0}: a_segment <= 1'b1;
{1'b0, 1'b1, 1'b1, 1'b1}: a_segment <= 1'b1;
{1'b1, 1'b0, 1'b0, 1'b0}: a_segment <= 1'b1;
{1'b1, 1'b0, 1'b0, 1'b1}: a_segment <= 1'b1;
default: a_segment <= 0;
            endcase
            end
endmodule
```

What does the synthesizer do ?

- Synthesized Logic for 7-segment display decoder “a” term
  - Fig. 2-28

Area/Delay Optimization

- Two fundamental concerns in implementing a design
  - How big is it ?
  - How fast will it operate ?
- The goal of our quest is to achieve *good enough*
  - To meet system requirements like
    - Product cost
    - Development cost
    - Performance
    - Reliability
    - Maintainability
    - Time to market

Area/Delay Optimization

- *The experienced designer always leaves a way out of a problem by insuring that a faster or denser device, if at all possible, is available in the same device footprint.*
  - *Instead of redesigning a circuit board to accommodate a new device at great expenses.*