



|| JAI SRI GURUDEV ||
Sri AdichunchanagiriShikshana Trust (R)

SJB INSTITUTE OF TECHNOLOGY

BGS Health & Education City, Kengeri , Bangalore – 60 .

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

Verilog HDL [18EC56]

Introduction to Subject & Preliminary Concepts

By:

Mrs. LATHA S
Assistant Professor,
Dept. of ECE, SJBIT

Outline

- Course Outline
- Recommended Books
- Prerequisites of the subject
- Module 1: Overview of Digital Design with Verilog HDL

Syllabus-18EC56

B. E. (EC / TC) Choice Based Credit System (CBCS) and Outcome Based Education (OBE) SEMESTER – V			
Verilog HDL			
Course Code	18EC56	IA Marks	40
Number of Lecture Hours/Week	03	Exam Marks	60
Total Number of Lecture Hours	40 (08 Hours per Module)	Exam Hours	03
CREDITS– 03			
Course Learning Objectives: <ul style="list-style-type: none"> • Learn different Verilog HDL constructs. • Familiarize the different levels of abstraction in Verilog. • Understand Verilog Tasks, Functions and Directives. • Understand timing and delay Simulation. • Understand the concept of logic synthesis and its impact in verification 			
Module 1			RBT Level
Overview of Digital Design with Verilog HDL: Evolution of CAD, emergence of HDLs, typical HDL-flow, why Verilog HDL?, trends in HDLs. Hierarchical Modeling Concepts: Top-down and bottom-up design methodology, differences between modules and module instances, parts of a simulation, design block, stimulus block.			L1,L2,L3
Module 2			
Basic Concepts: Lexical conventions, data types, system tasks, compiler directives. Modules and Ports: Module definition, port declaration, connecting ports, hierarchical name referencing.			L1,L2,L3

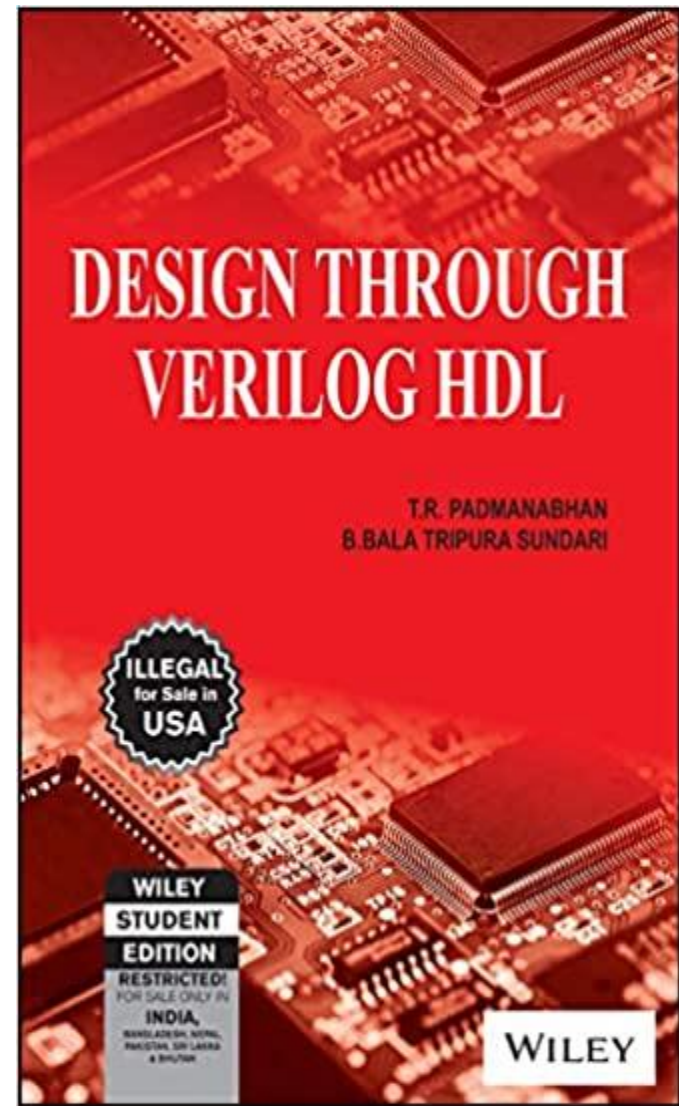
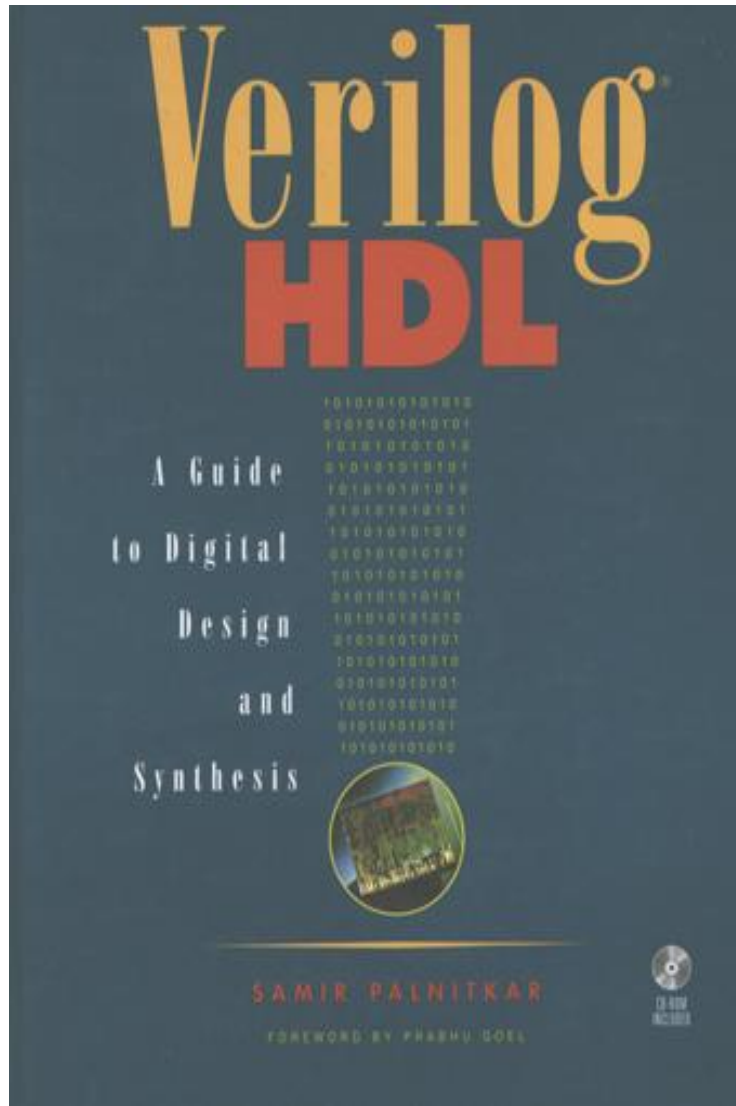
Module 3	
Gate-Level Modeling: Modeling using basic Verilog gate primitives, description of and/or and buf/not type gates, rise, fall and turn-off delays, min, max, and typical delays. Dataflow Modeling: Continuous assignments, delay specification, expressions, operators, operands, operator types.	L1,L2,L3
Module 4	
Behavioral Modeling: Structured procedures, initial and always, blocking and non-blocking statements, delay control, generate statement, event control, conditional statements, Multiway branching, loops, sequential and parallel blocks. Tasks and Functions: Differences between tasks and functions, declaration, invocation, automatic tasks and functions.	L1,L2,L3
Module 5	
Useful Modeling Techniques: Procedural continuous assignments, overriding parameters, conditional compilation and execution, useful system tasks. Logic Synthesis with Verilog: Logic Synthesis, Impact of logic synthesis, Verilog HDL Synthesis, Synthesis design flow, Verification of Gate-Level Netlist. (Chapter 14 till 14.5 of Text).	L1,L2,L3

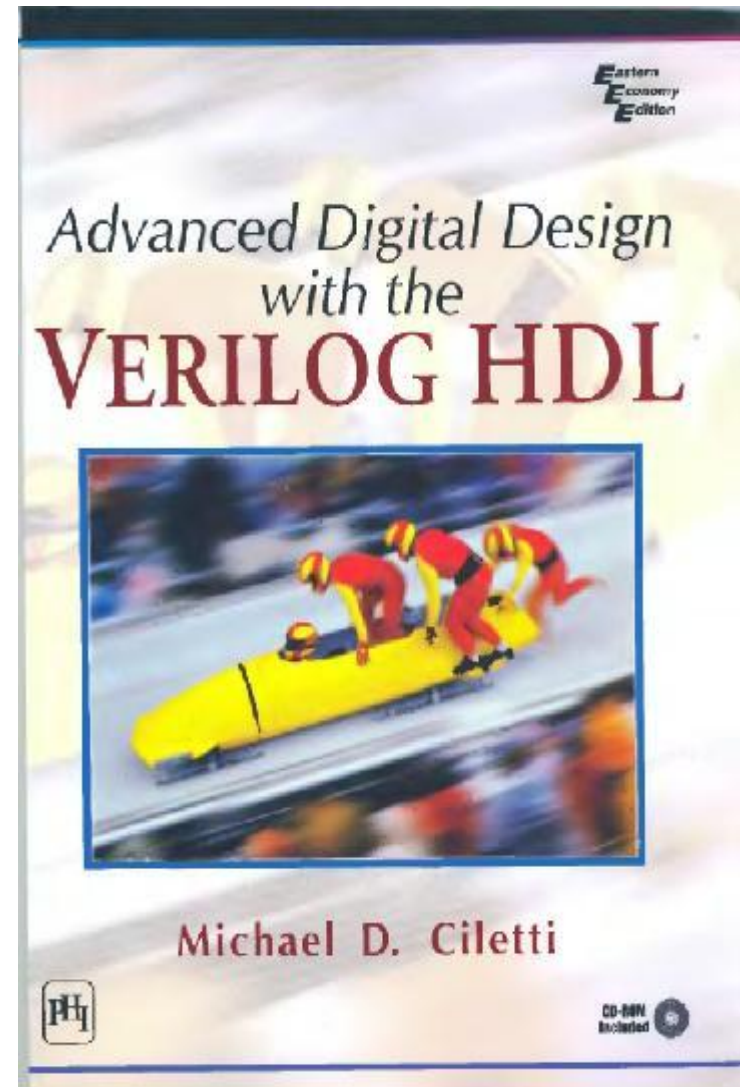
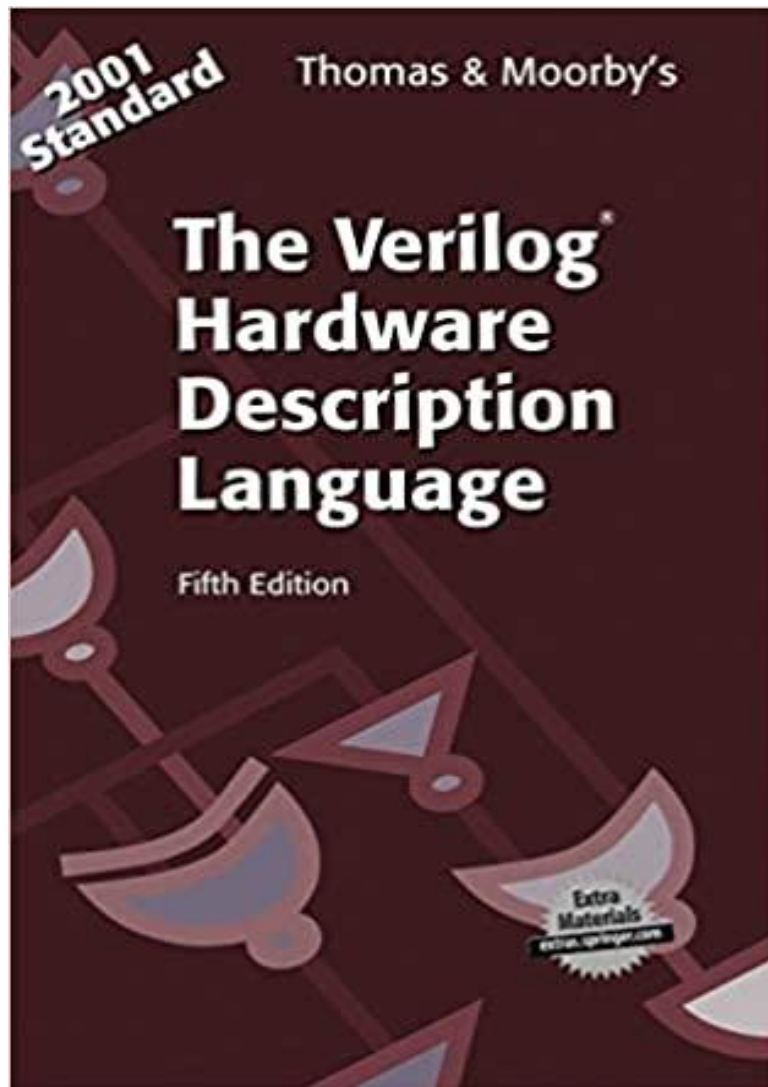
Course Outcomes:

At the end of this course, students should be able to

- Write Verilog programs in gate, dataflow (RTL), behavioral and switch modeling levels of Abstraction.
- Design and verify the functionality of digital circuit/system using test benches.
- Identify the suitable Abstraction level for a particular digital design.
- Write the programs more effectively using Verilog tasks, functions and directives.
- Perform timing and delay Simulation
- Interpret the various constructs in logic synthesis.

-
- **Text Book:**
 - Samir Palnitkar, “**Verilog HDL: A Guide to Digital Design and Synthesis**”, Pearson Education, Second Edition.
 - **Reference Books:**
 - 1. Donald E. Thomas, Philip R. Moorby, “The Verilog Hardware Description Language”, Springer, Fifth edition.
 - 2. Michael D. Ciletti, “Advanced Digital Design with the Verilog HDL” Pearson (Prentice Hall), Second edition.
 - 3. Padmanabhan, Tripura Sundari, “Design through Verilog HDL”, Wiley, 2016 or earlier.
-





Module 1: Overview of Digital Design with Verilog

Mrs. LATHA S
Assistant Professor
Dept. of ECE, SJBIT

Overview of Digital Design with Verilog HDL

- *Evolution of computer aided digital circuit design*
- Emergence of HDLs
- Typical design flow
- Importance of HDLs
- Popularity of Verilog HDL
- Trends in HDLs

Evolution of Computer Aided Digital Design

- Digital circuits were designed with
 - Vacuum tubes
 - Transistors
 - Integrated circuits (ICs)
 - SSI
 - MSI : hundreds of gates
 - LSI : thousands of gates
 - CAD techniques began to evolve
 - circuit and Logic simulation about 100 transistors
 - VLSI : more than 100,000 transistors
 - ULSI : Ultra Large Scale Integration

Overview of Digital Design with Verilog HDL

- Evolution of computer aided digital circuit design
- *Emergence of HDLs*
- Typical design flow
- Importance of HDLs
- Popularity of Verilog HDL
- Trends in HDLs

Emergence of HDLs

- *Hardware Description Language (HDL)*
 - A **hardware description language** is the language that describes the hardware of digital systems in textual form and resembles a programming language, but specifically oriented to describing hardware structures and behavior.
 - Allowed designed to model the *concurrency* of processes found in hardware elements
 - Verilog HDL originated in 1983
 - VHDL was developed under contract from DARPA
 - Could be used to describe digital circuits at a *register transfer level* (RTL)
 - Specify how the data flows between registers and how the design processes the data
 - Logic synthesis tools can be used to produce gate-level netlist from the RTL description automatically

Different Levels of Abstraction

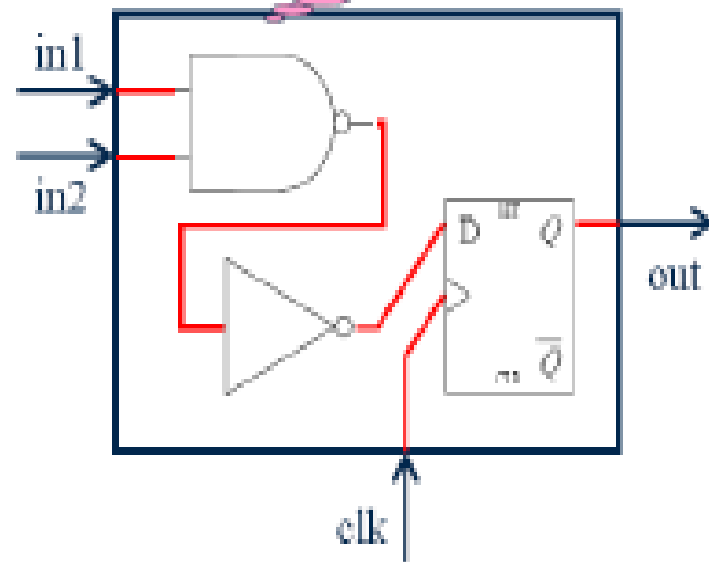
- Architecture / Algorithm Level
 - Describe the functionality (behavior) of a circuit
- Register Transfer Logic (RTL) Level
 - Describe the data flow of a circuit
- Gate Level
 - Describe the connectivity (structure) of a circuit
- Switch Level

An Example of Verilog HDL

```
module top(clk, in1, in2, out);  
  input  clk, in1, in2;  
  output out;  
  
  reg out;  
  
  always @(posedge clk)  
  begin  
    out = in1 & in2;  
  end  
  
endmodule
```

RTL

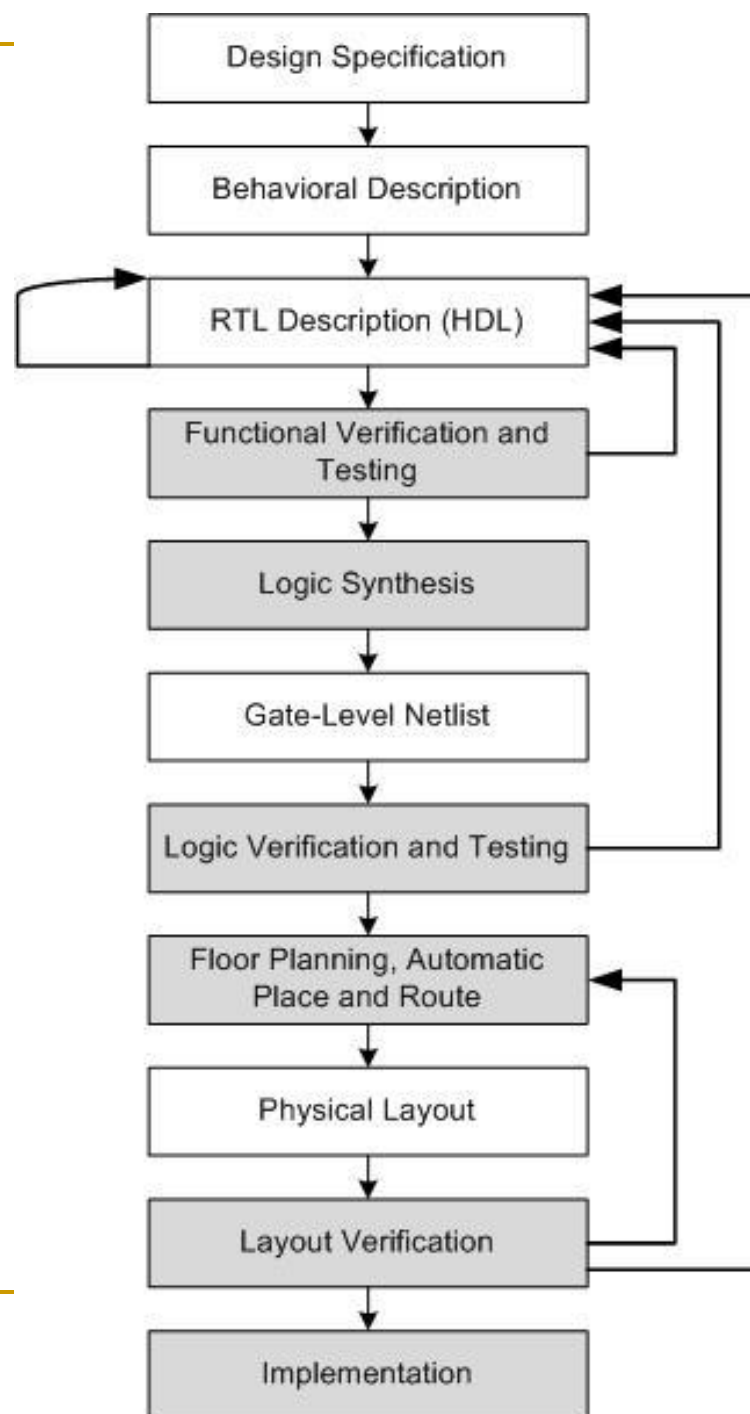
Gate-level



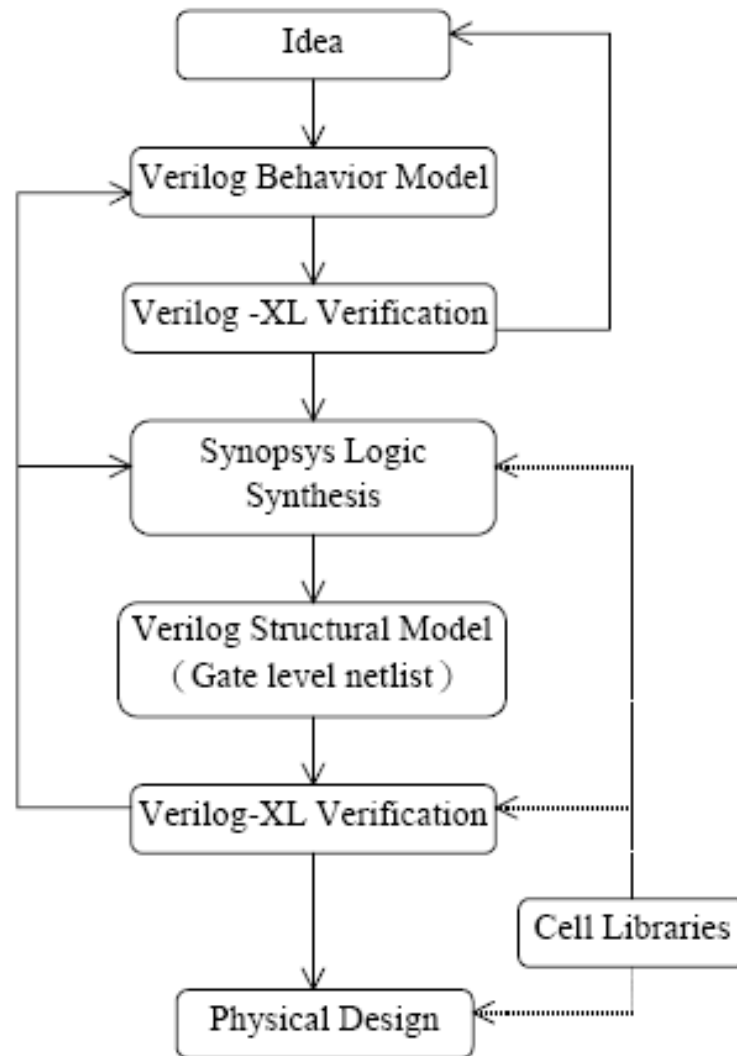
Overview of Digital Design with Verilog HDL

- Evolution of computer aided digital circuit design
- Emergence of HDLs
- *Typical design flow*
- Importance of HDLs
- Popularity of Verilog HDL
- Trends in HDLs

Typical Design Flow for Designing VLSI IC



Design Flow of using HDL



Overview of Digital Design with Verilog HDL

- Evolution of computer aided digital circuit design
- Emergence of HDLs
- Typical design flow
- *Importance of HDLs*
- Popularity of Verilog HDL
- Trends in HDLs

Why use the HDL ?

- Difficult to design directly on hardware
- Mixed-level modeling and simulation
- Easier to explore different design options
- Reduce design time and cost

Advantages of HDLs

- Advantages compared to traditional schematic-based design
 - Design with RTL description + logic synthesis tool
 - Abstract level
 - Independent to fabrication technology
 - Reuse when fabrication technology changing
 - Functional verification can be done early
 - Optimized to meet the desired functionality
 - Analogous to computer programming
 - Textual description with comments

Overview of Digital Design with Verilog HDL

- Evolution of computer aided digital circuit design
- Emergence of HDLs
- Typical design flow
- Importance of HDLs
- *Popularity of Verilog HDL*
- Trends in HDLs

History of the Verilog HDL

- **1984:** Gateway Design Automation introduced the Verilog-XL digital logic simulator
 - The Verilog language was part of the Verilog-XL simulator
 - The language was mostly created by 1 person, Phil Moorby
 - The language was intended to be used with only 1 product
- **1989:** Gateway merged into Cadence Design Systems
- **1990:** Cadence made the Verilog HDL public domain
 - Open Verilog International (OVI) controlled the language

History of the Verilog HDL (Cont'd)

- **1995**: The IEEE standardized the Verilog HDL (IEEE 1364)
- **2001**: The IEEE enhanced the Verilog HDL for modeling scalable designs, deep sub-micron accuracy, etc.

Useful Features of the Verilog HDL

- A general-purpose HDL
 - Easy to learn and use
 - Syntax is similar to C (VHDL is similar to PASCAL)
- Allows different levels of abstraction to be mixed in the same model
 - In terms of switches, gates, RTL, or behavioral code
 - Need to learn only for stimulus and hierarchical design
- Most popular logic synthesis tools support Verilog
- Rich of Verilog HDL libraries
 - Provided by fabrication vendors for postlogic synthesis simulation
 - Allows the widest choice of vendors while designing a chip
- With powerful PLI (Programming Language Interface)
 - Write custom C code to interact with internal data structure

Overview of Digital Design with Verilog HDL

- Evolution of computer aided digital circuit design
- Emergence of HDLs
- Typical design flow
- Importance of HDLs
- Popularity of Verilog HDL
- Trends in HDLs

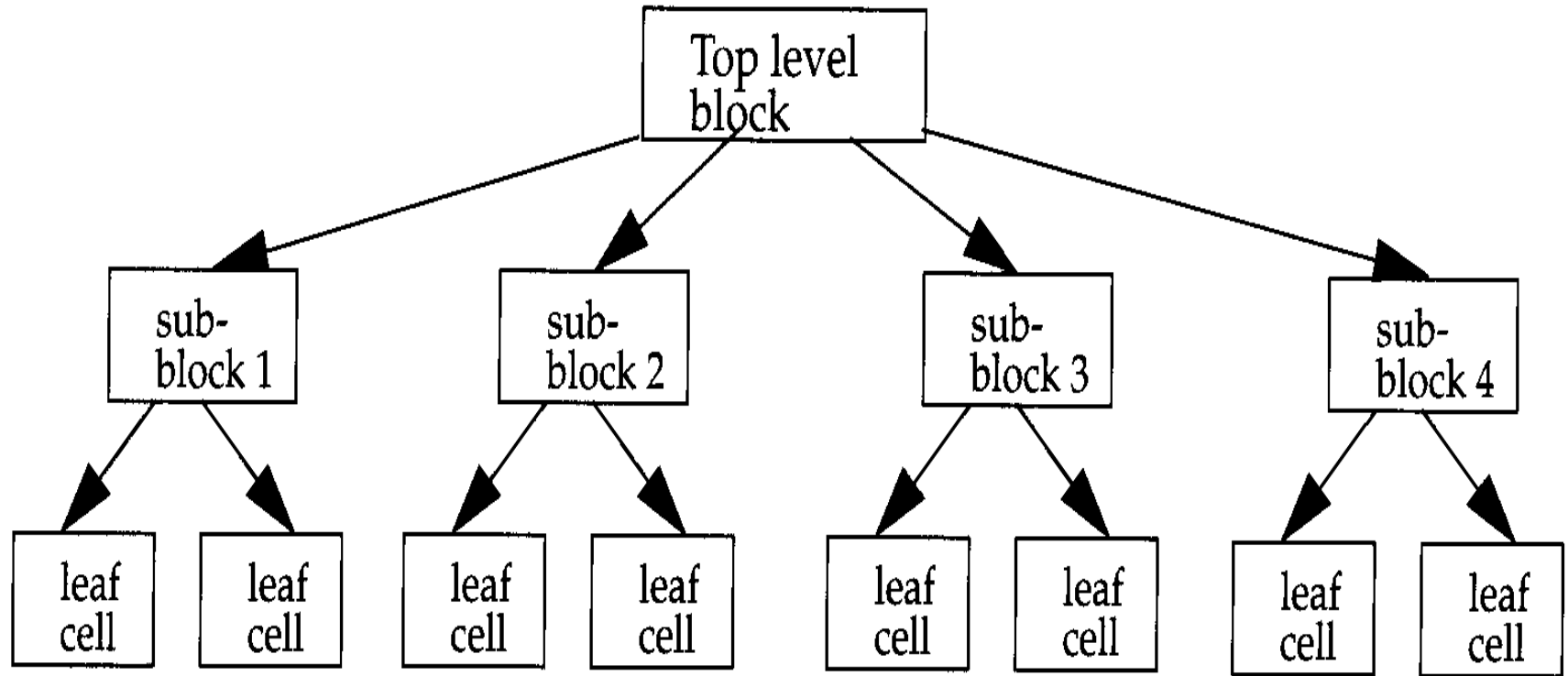
Trends in HDLs

- Higher levels of abstraction
 - Think only in terms of functionality for designers
 - CAD tools take care of the implementation details
- Behavioral modeling
 - Design directly in terms of algorithms and
 - the behavior of the circuit
- Formal verification
- Supports for Mixed-level design
 - Ex: very high speed and timing-critical circuits like μ Ps
 - Mix gate-level description directly into the RTL description
- System-level design in a mixed bottom-up methodology
 - Use either existing Verilog modules, basic building blocks, or IPs
 - Ex: SystemC for SoC designs

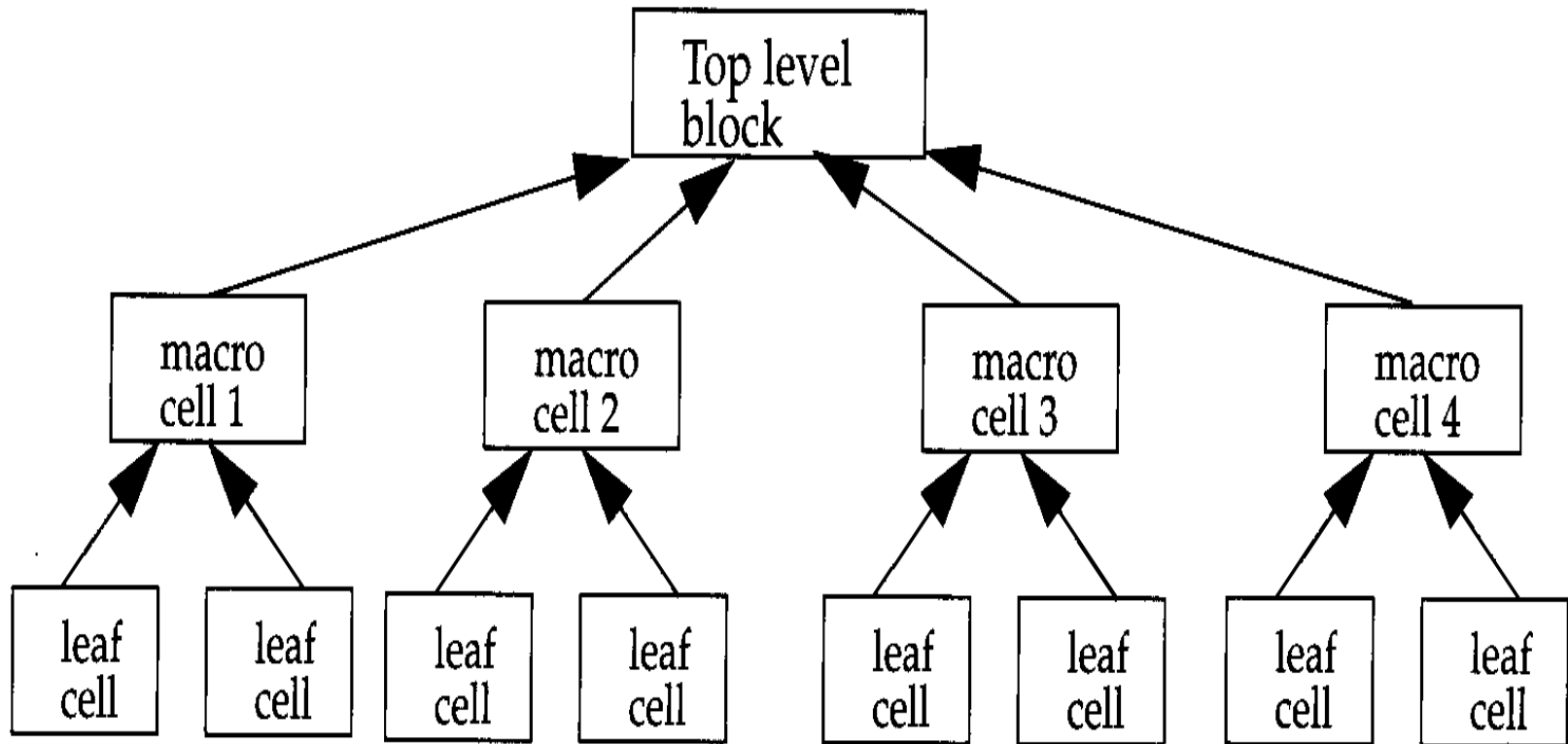
Hierarchical Modelling Concepts

- **Learning Objectives**
- Understand top-down and bottom-up design methodologies for digital design.
- Explain differences between modules and module instances in Verilog.
- Describe four levels of abstraction—behavioral, data flow, gate level, and switch level—to represent the same module.
- Describe components required for the simulation of a digital design. Define a stimulus block and a design block. Explain two methods of applying stimulus.

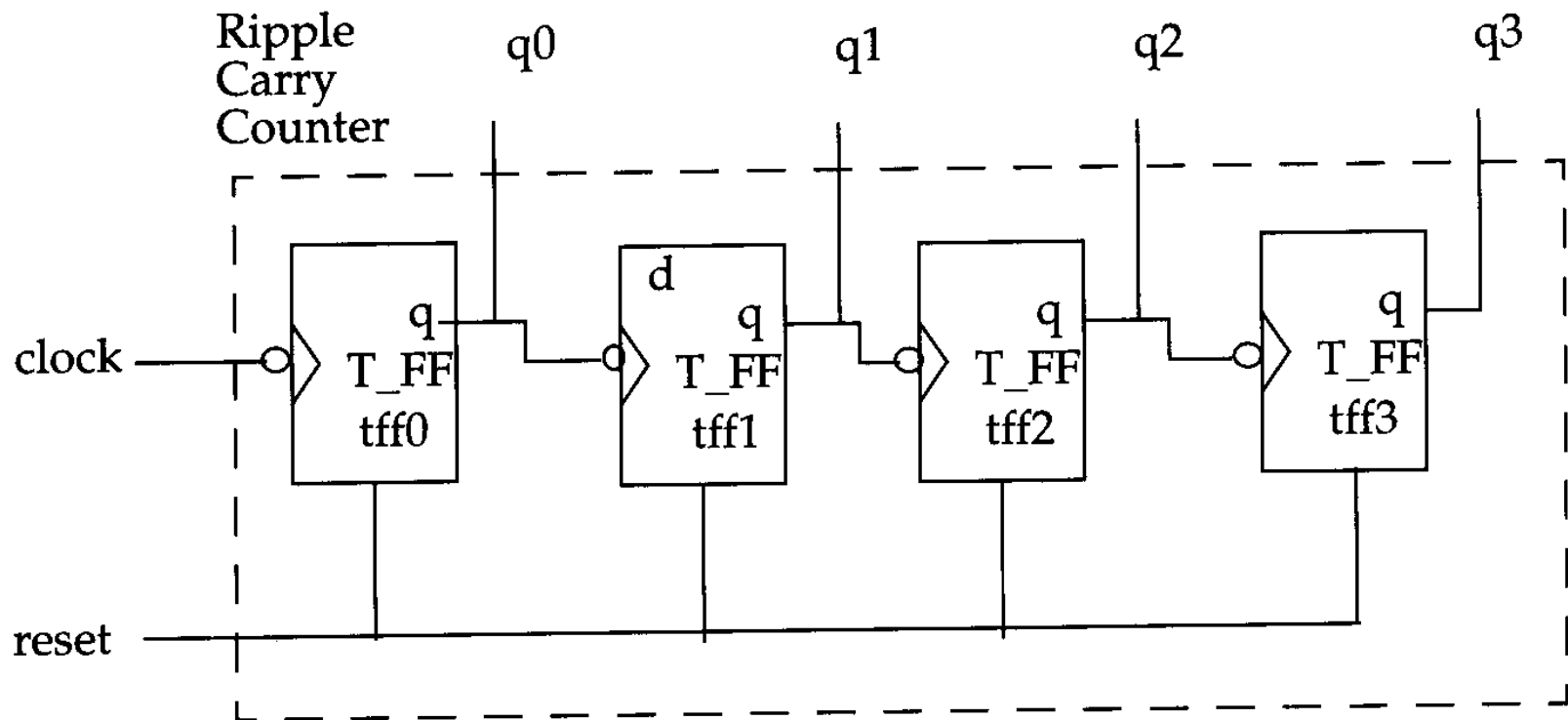
Top Down Design Methodology



Bottom Up Design Methodology



Example : Design of 4 bit Ripple Counter using Top Down approach

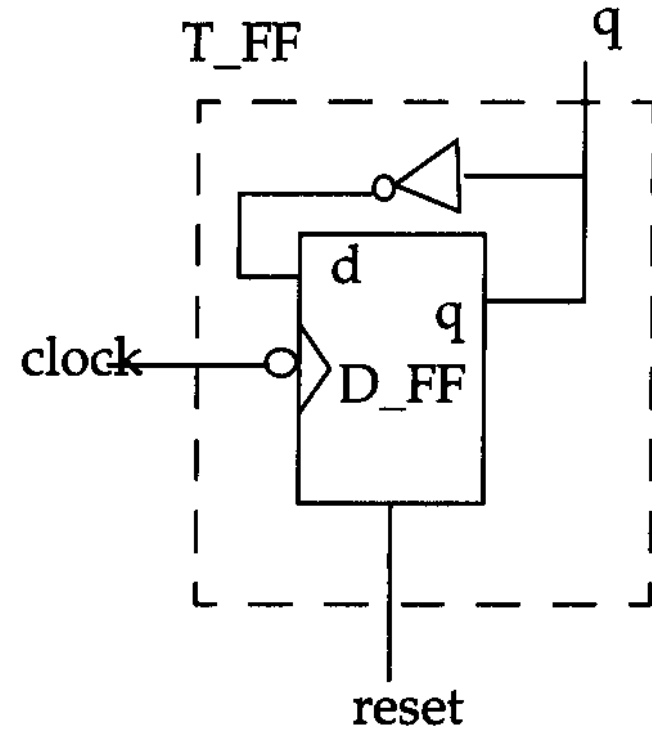


Ripple Counter

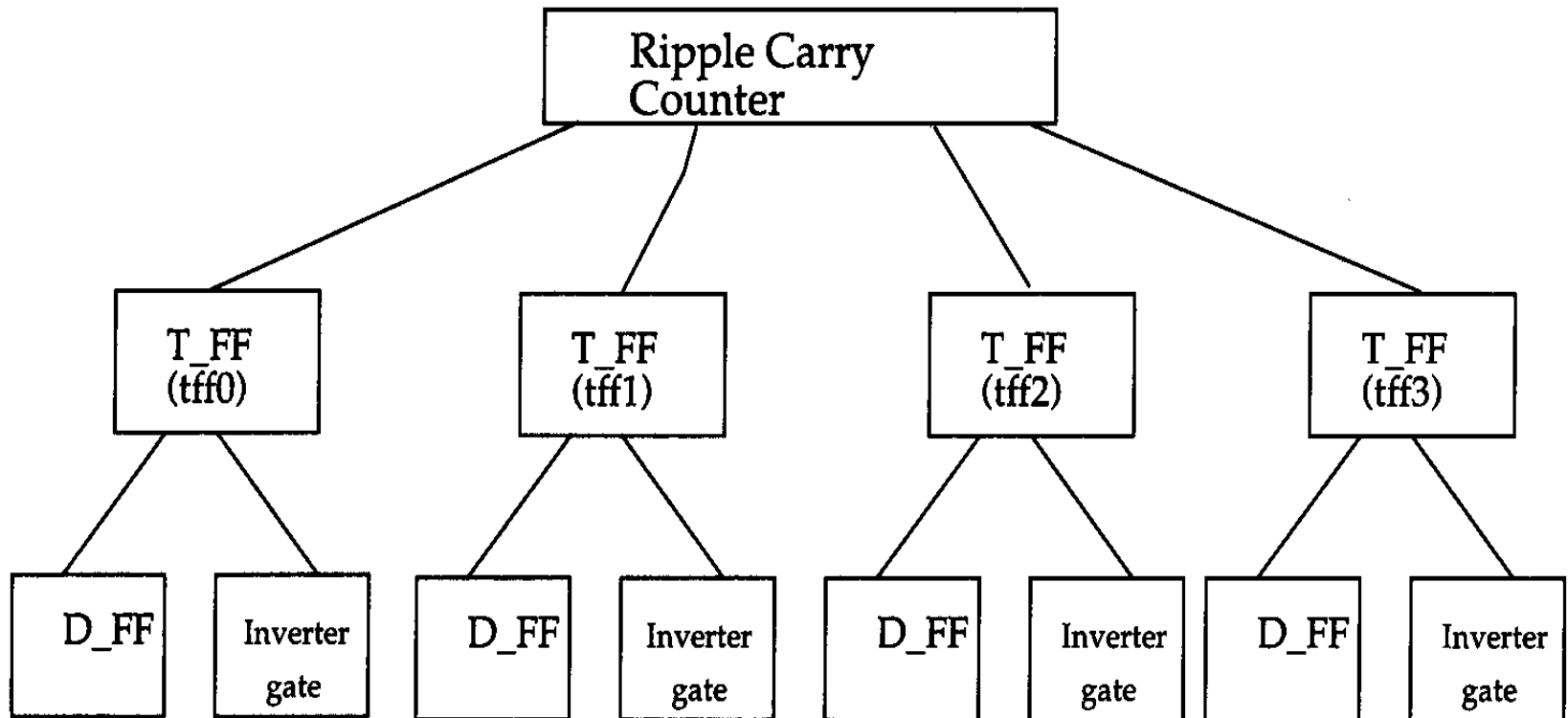
- A n-bit ripple counter can count up to 2^n states. It is also known as MOD n counter. It is known as ripple counter because of the way the clock pulse ripples its way through the flip-flops. Some of the features of ripple counter are:
- It is an asynchronous counter.
- Different flip-flops are used with a different clock pulse.
- All the flip-flops are used in toggle mode.
- Only one flip-flop is applied with an external clock pulse and another flip-flop clock is obtained from the output of the previous flip-flop.
- The flip-flop applied with external clock pulse act as LSB (Least Significant Bit) in the counting sequence.

T-Flip flop

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



Design Hierarchy of 4 Bit ripple Counter



Verilog - Module

- A module is the building block in Verilog.
- It is declared by the keyword *module* and is always terminated by the keyword *endmodule*.
- Each statement is terminated with a semicolon, but there is no semi-colon after

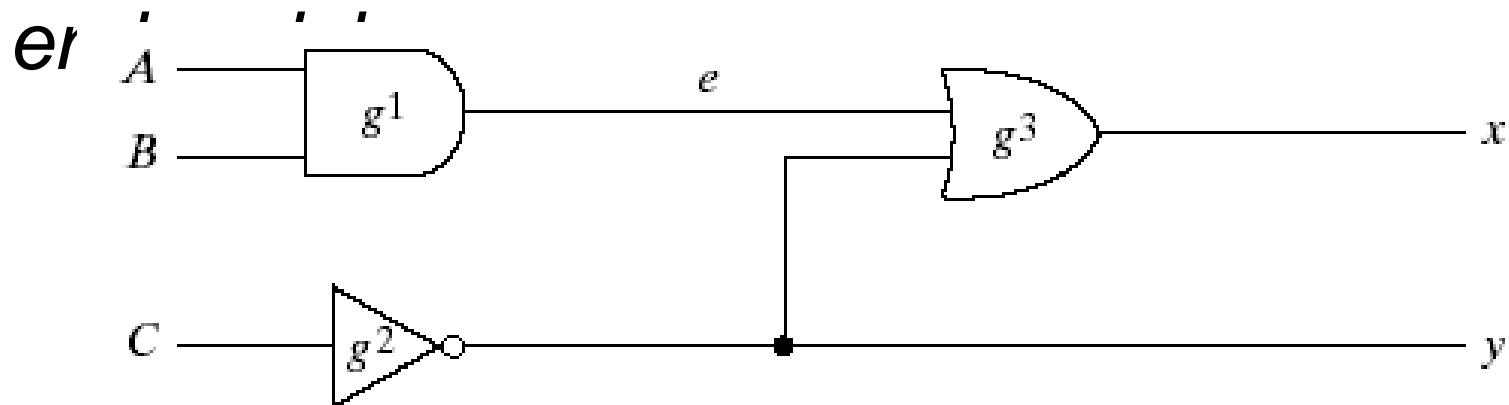


Fig. 3-37 Circuit to Demonstrate HDL

Verilog Modules

- Basic building block in Verilog
 - Hierarchical design (top down vs. bottom-up)
 - Multiple modules in a single file
 - Order of definition not important
 - Modules are:
 - Declared
 - Instantiated
 - Modules declarations cannot be nested

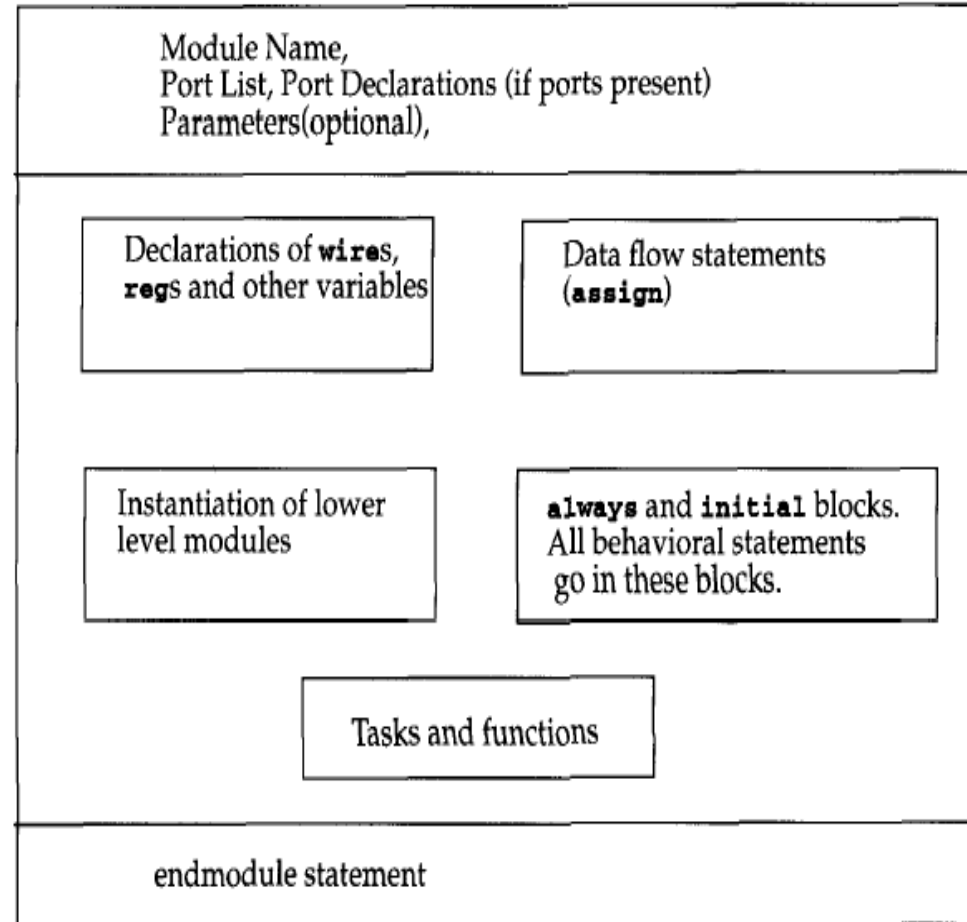


Figure 4-1 Components of a Verilog Module

Module Representation

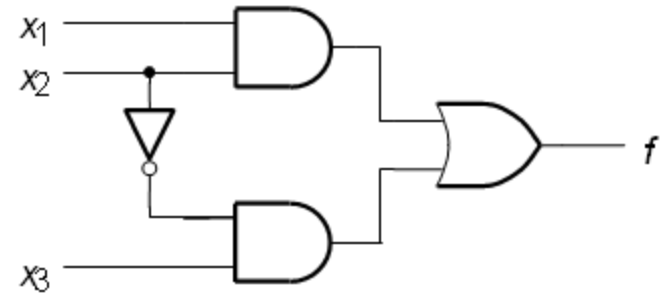
■ Module

- A logic circuit → *module*
- Its ports: inputs and outputs
- Begins with **module**, ends with **endmodule**

■ Module<module name>(module terminal list);

<Module internals>

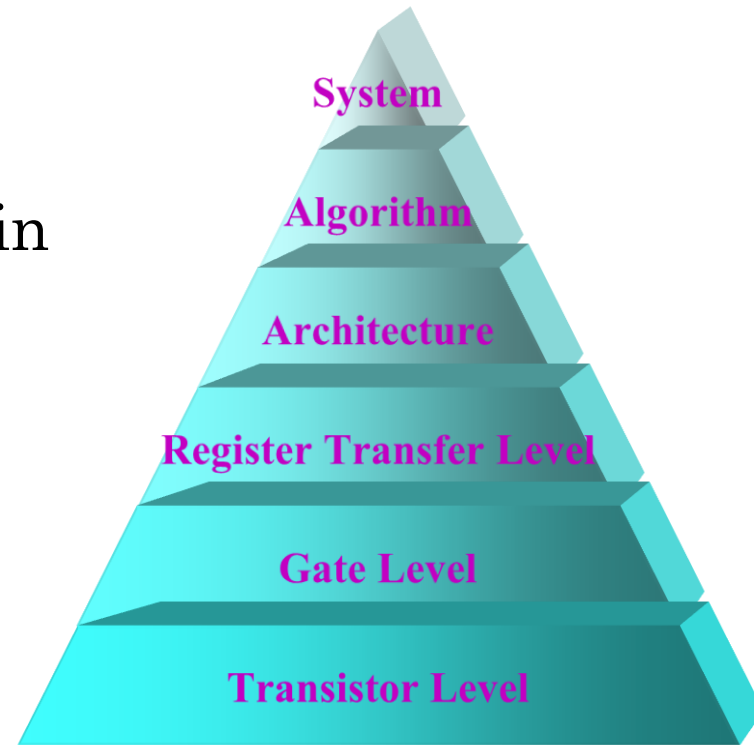
■ End module



```
module example1 (x1, x2, x3, f);  
    input x1, x2, x3;  
    output f;  
  
    and (g, x1, x2);  
    not (k, x2);  
    and (h, k, x3);  
    or (f, g, h);  
  
endmodule
```

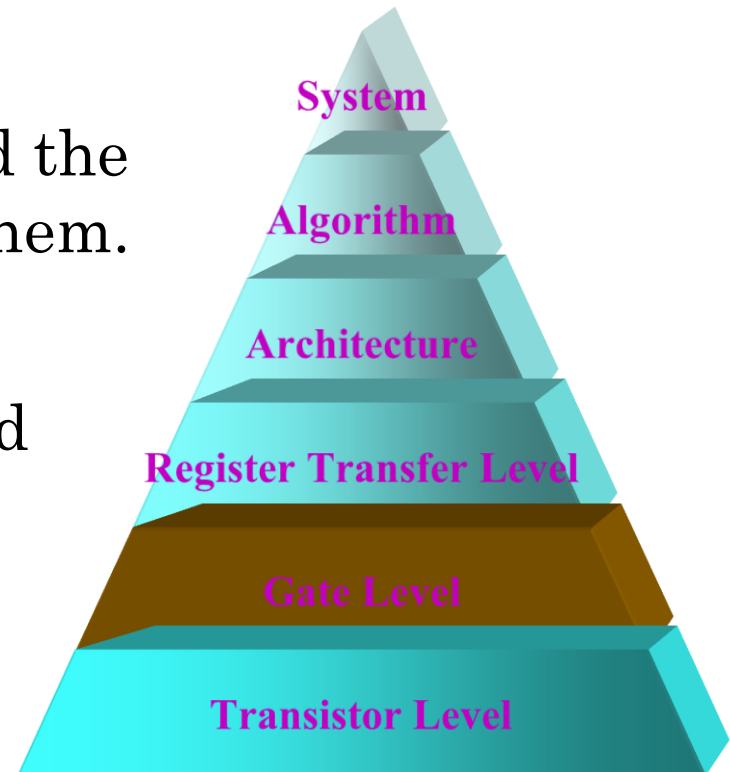
Different Levels of Abstraction

- Behavioural/Architectural / Algorithmic Level
 - Implement a design algorithm in high-level language constructs.
- Data flow level/Register Transfer Level
 - Describes the flow of data between registers and how a design process these data.



Different Levels of Abstraction

- Gate Level
 - Describe the logic gates and the interconnections between them.
- Switch (Transistor) Level
 - Describe the transistors and the interconnections between them.



Verilog Module Instances

- A module provides template from which we can create actual objects.
- When module is invoked, Verilog creates a unique from the template. Each object has its own name, variables, parameters and I/O interface.
- The process of creating objects from template is called *Instantiation*, and the objects are called *Instances*.

Module Instantiation (cont'd)

■ General syntax

```
<module_name> <instance_name> (port connection list);
```

■ Example:

```
// assuming module ripple_carry_counter(q, clk, reset);  
ripple_carry_counter cntrl(wire_vec1, wire2, wire3);
```

Example Full Adder

```
module fastr(x, y, cin, sum, cout);
```

```
    input x,y,cin;
```

```
    output sum,cout;
```

```
    wire t1,t2,t3;
```

```
    ha ha1 ( x,y,t1,t2);
```

```
    ha ha2 ( t1,cin,sum,t3 );
```

```
    or or2 ( cout ,t3,t2);
```

```
endmodule
```

```
module ha(x, y, s, c);
```

```
    input x,y;
```

```
    output s,c;
```

```
    assign s = x^y;
```

```
    assign c = x&y;
```

```
endmodule
```

Module Instantiation

- Recall the Ripple-carry counter and TFF

```
module TFF(q, clk, reset);  
    output q;  
    input clk, reset;  
    ...  
endmodule
```

```
module ripple_carry_counter(q, clk, reset);  
    output [3:0] q;  
    input clk, reset;  
  
    //4 instances of the module TFF are created.  
    TFF tff0(q[0],clk, reset);  
    TFF tff1(q[1],q[0], reset);  
    TFF tff2(q[2],q[1], reset);  
    TFF tff3(q[3],q[2], reset);  
endmodule
```

Module Instances-Ripple Counter

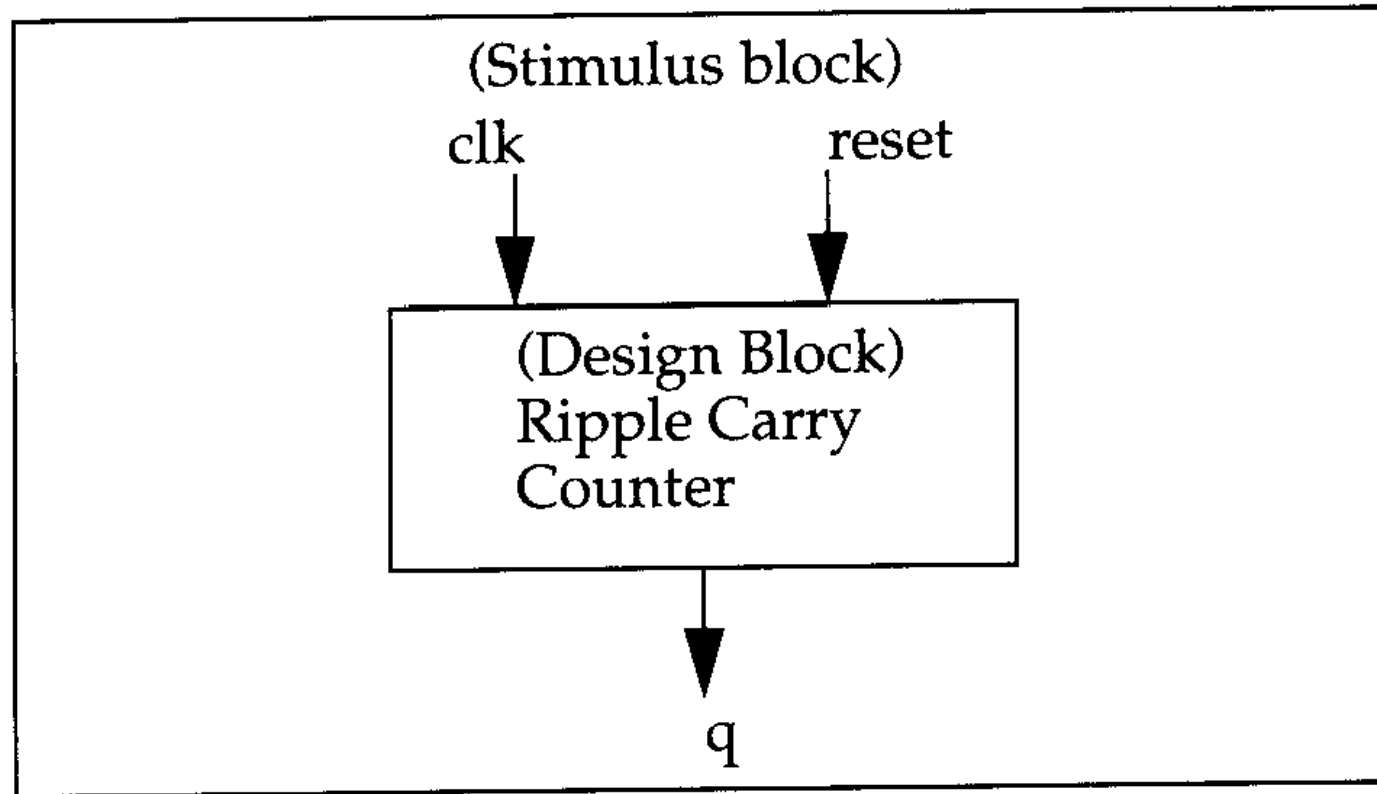
```
module reg4 (q,d,clock);
    output [3:0] q;
    input [3:0] d;
    input clock;
    wire [3:0] q, d;
    wire clock;
    //port order connection,
    //2nd port not connected
    dff u1 (q[0], , d[0], clock);
    //port name connection,
    //qb not connected
    dff u2 (.clk(clock),.q(q[1]),.data(d[1]));
    //explicit parameter redefine
    dff u3 (q[2], ,d[2], clock);
    defparam u3.delay = 3.2;
    //implicit parameter redefine
    dff #(2) u4 (q[3], , d[3], clock);
endmodule
```

```
module dff (q,qb,data,clk);
    output q, qb;
    input data, clk;
    //default delay parameter
    parameter delay = 1;
    dff_udp #(delay) (q,data,clk);
    not (qb, q);
endmodule
```

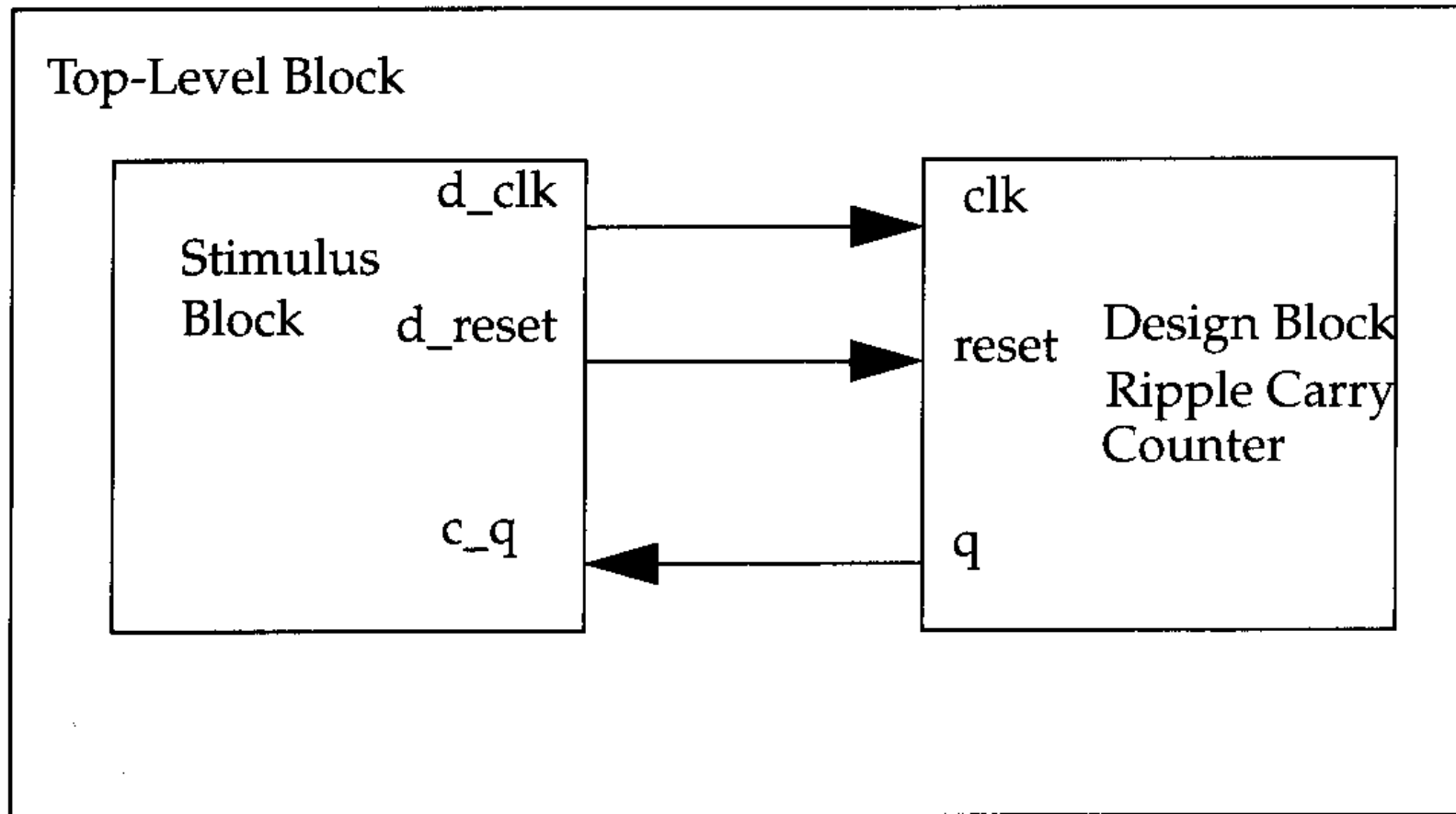
Components of Simulation

- The functionality of the design block can be tested by applying stimulus and checking results such a block is called stimulus block.
- Separate stimulus and design block.
- Stimulus block can be written using verilog, separate language is not required. Stimulus block is also called as *Test bench*.
- Different test benches can be written to test a design block

Stimulus Block Instantiates Design Block(Ripple Counter)



Stimulus and design Blocks Instantiated in a Dummy top level Module Design



Example:- 4 bit Ripple Counter

- To illustrate the concepts discussed in the previous sections, let us build the complete simulation of a ripple carry counter.
- We will define the design block and the stimulus block.
- We will apply stimulus to the design block and monitor the outputs.
- As we develop the Verilog models, you do not need to understand the exact syntax of each construct at this stage.
- At this point, you should simply try to understand the design process. We discuss the syntax in much greater detail in the later modules.

Example: 4 bit Ripple Counter

```
module ripple_carry_counter(q, clk, reset);  
    output [3:0] q;  
    input clk, reset;  
    //4 instances of the module T_FF are created.  
    T_FF tff0(q[0],clk, reset);  
    T_FF tff1(q[1],q[0], reset);  
    T_FF tff2(q[2],q[1], reset);  
    T_FF tff3(q[3],q[2], reset);  
endmodule
```

■ Example 2- Ripple Carry Counter Top Block

```
module T_FF(q, clk, reset);  
output q;  
input clk, reset; wire d;  
D_FF dff0(q, d, clk, reset);  
not n1(d, q); // not is a Verilog-provided primitive. case sensitive  
endmodule
```

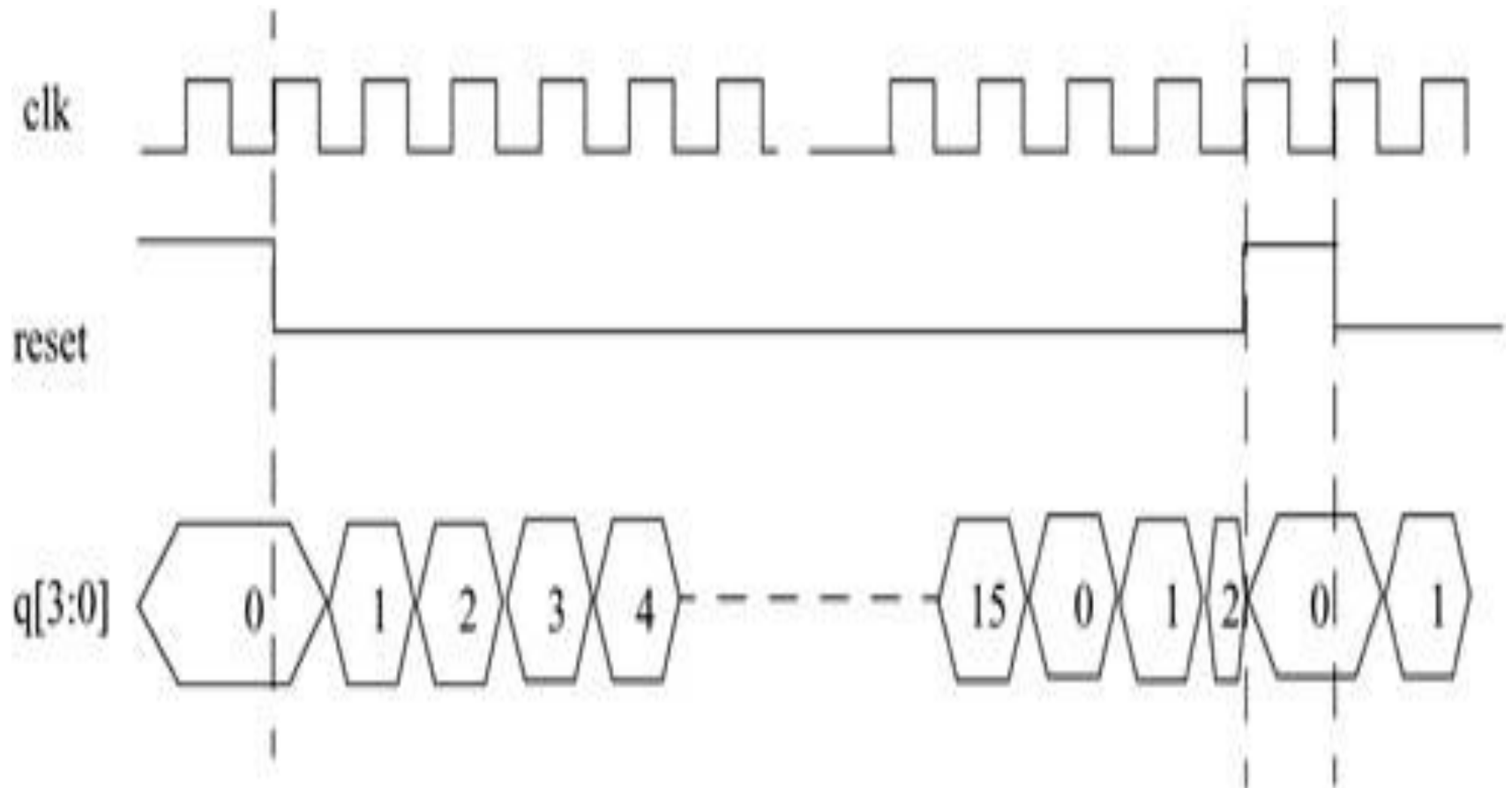
Example 3 . Flipflop D_F

- `// module D_FF with synchronous reset module`
`D_FF(q, d, clk, reset);`
- `output q;`
- `input d, clk, reset;`
- `reg q;`
- `// Lots of new constructs. Ignore the functionality of the // constructs. //`
`Concentrate on how the design block is built in a top-down fashion.`
- `always @(posedge reset or negedge clk)`
- `if (reset) q <= 1'b0;`
- `else q <= d;`
- `endmodule`

Stimulus Block

- We need to write the stimulus block to check if the ripple carry counter design is functioning correctly.
- In this case, we must control the signals clk and reset so that the regular function of the ripple carry counter and the asynchronous reset mechanism are both tested.
- Consider the waveforms shown in Figure 1-9 to test the design.
- Waveforms for clk, reset, and 4-bit output q are shown. The cycle time for clk is 10 units;
- The reset signal stays up from time 0 to 15 and then goes up again from time 195 to 205. Output q counts from 0 to 15.

Stimulus Block



Example 1-6 Stimulus Block

- `module stimulus; reg clk;`
- `reg reset; wire[3:0] q;`
- *// instantiate the design block*
- `ripple_carry_counter r1(q, clk, reset);`
- *// Control the clk signal that drives the design block. Cycle time = 10 initial*
- `clk = 1'b0; //set clk to 0 always`
- `#5 clk = ~clk; //toggle clk every 5 time units`
- *// Control the reset signal that drives the design block*
- *// reset is asserted from 0 to 20 and from 200 to 220. initial*
- `begin`
- `reset = 1'b1;`
- `#15 reset = 1'b0;`
- `#180 reset = 1'b1;`
- `#10 reset = 1'b0;`
- `#20 $finish; //terminate the simulation`
- `end`
- *// Monitor the outputs*
- `initial`
- `$monitor($time, " Output q = %d", q);`
- `endmodule`

Example 2-7. Output of the Simulation

- 0 Output $q = 0$ 20 Output $q = 1$ 30 Output $q = 2$ 40 Output $q = 3$ 50 Output $q = 4$ 60 Output $q = 5$ 70 Output $q = 6$ 80 Output $q = 7$ 90 Output $q = 8$ 100 Output $q = 9$ 110 Output $q = 10$ 120 Output $q = 11$ 130 Output $q = 12$ 140 Output $q = 13$ 150 Output $q = 14$ 160 Output $q = 15$ 170 Output $q = 0$ 180 Output $q = 1$ 190 Output $q = 2$ 195 Output $q = 0$ 210 Output $q = 1$ 220 Output $q = 2$

```
0 Output q = 0
20 Output q = 1
30 Output q = 2
40 Output q = 3
50 Output q = 4
60 Output q = 5
70 Output q = 6
80 Output q = 7
90 Output q = 8
100 Output q = 9
110 Output q = 10
120 Output q = 11
130 Output q = 12
140 Output q = 13
150 Output q = 14
160 Output q = 15
170 Output q = 0
180 Output q = 1
190 Output q = 2
195 Output q = 0
210 Output q = 1
220 Output q = 2
```


Summary

- In this module we discussed the following concepts.
- Two kinds of design methodologies are used for digital design: top-down and bottom-up. A combination of these two methodologies is used in today's digital designs. As designs become very complex, it is important to follow these structured approaches to manage the design process.
- Modules are the basic building blocks in Verilog. Modules are used in a design by instantiation. An instance of a module has a unique identity and is different from other instances of the same module. Each instance has an independent copy of the internals of the module. It is important to understand the difference between modules and instances.

- There are two distinct components in a simulation: a design block and a stimulus block. A stimulus block is used to test the design block. The stimulus block is usually the top-level block. There are two different styles of applying stimulus to a design block.
- The example of the ripple carry counter explains the step-by-step process of building all the blocks required in a simulation.

Outcomes of Module-1

- After completion of the module the students are able to:
- Understand the importance, trends of HDL and design flow and design methodologies for digital design.
- Differentiate the modules and module instances in Verilog with an example.
- Define stimulus block and design block

Recommended questions

- Discuss in brief about the evolution of CAD tools and HDLs used in digital system design.
- Explain the typical VLSI IC design flow with the help of flow chart.
- Discuss the trends in HDLs?
- Why Verilog HDL has evolved as popular HDL in digital circuit design?
- Explain the advantages of using HDLs over traditional schematic based design.
- Describe the digital system design using hierarchical design methodologies with an example.

- Apply the top-down design methodology to demonstrate the design of ripple carry counter.
- Apply the bottom-up design methodology to demonstrate the design of 4-bit ripple carry adder.
- Write Verilog HDL program to describe the 4-bit ripple carry counter.
- Define Module and an Instance. Describe 4 different description styles of Verilog HDL.
- Differentiate simulation and synthesis. What is stimulus?
- Write test bench to test the 4-bit ripple carry counter.
- Write a test bench to test the 4-bit ripple carry adder.

Reference / Text Book Details

Sl.No	Title of Book	Author	Publication	Edition
1	Verilog HDL: A Guide to Digital Design and Synthesis	Samir Palnitkar	Pearson Education	2 nd
2	VHDL for Programmable Logic	Kevin Skahill	PHI/Pearson education	2 nd
3	The Verilog Hardware Description Language	Donald E. Thomas, Philip R. Moorby	Springer Science+Business Media, LLC	5 th
4	Advanced Digital Design with the Verilog HDL	Michael D. Ciletti	Pearson (Prentice Hall)	2 nd
5	Design through Verilog HDL	Padmanabhan, Tripura Sundari	Wiley	Latest

Thank You





|| JAI SRI GURUDEV ||
Sri AdichunchanagiriShikshana Trust (R)
SJB INSTITUTE OF TECHNOLOGY
BGS Health & Education City, Kengeri , Bangalore – 60 .

***DEPARTMENT OF ELECTRONICS & COMMUNICATION
ENGINEERING***

Verilog HDL [18EC56]

Module 2: BASIC CONCEPTS AND MODULES AND PORTS

By:

Mrs. LATHA S
Assistant Professor,
Dept. of ECE, SJBIT

Content

- Basic Concepts: Lexical conventions
- Data types,
- System tasks,
- Compiler directives.
- Modules and Ports:
- Module definition,
- Port declaration,
- Connecting ports,
- Hierarchical name
- Referencing

Learning Objectives

- Understand the lexical conventions and define the logic value set and data type.
 - Identify useful system tasks and basic compiler directives.
 - Identify and understanding of components of a Verilog module definition.
 - Understand the port connection rules and connection to external signals by ordered list and by name
-

Lexical Conventions

- The basic lexical conventions used by Verilog HDL are similar to those in the C programming language.
 - Verilog contains a stream of tokens. Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords.
 - Verilog HDL is a case-sensitive language.
 - ***All keywords are in lowercase.***
-

Whitespace

- Blank spaces (`\b`) , tabs (`\t`) and newlines (`\n`) comprise the whitespace.
- Whitespace is ignored by Verilog except when it separates tokens.
- Whitespace is not ignored in strings.

Comments

- Comments can be inserted in the code for readability and documentation.
 - There are two ways to write comments. A one-line comment starts with "//".
 - Verilog skips from that point to the end of line.
 - A multiple- line comment starts with "/*" and ends with "*/". Multiple-line comments cannot be nested.
 - However, one-line comments can be embedded in multiple-line comments.
-

Comment Syntax

- `a = b && c; // This is a one-line comment`
- `/* This is a multiple line comment`
- `*/`
- `/* This is /* an illegal */ comment */`
- `/* This is //a legal comment */`

Operators

- Operators are of three types: unary, binary, and ternary.
- Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.
- `a = ~ b;` // `~` is a unary operator. `b` is the operand
- `a = b && c;` // `&&` is a binary operator. `b` and `c` are operands
- `a = b ? c : d;` // `?:` is a ternary operator. `b`, `c` and `d` are operands

Number Specification

- There are two types of number specification in Verilog: sized and unsized.
 - SIZED NUMBERS
 - Sized numbers are represented as `<size> '<base format> <number>`.
 - `<size>` is written only in decimal and specifies the number of bits in the number.
-

- Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).
- The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base.
- Uppercase letters are legal for number specification.

Syntax

- 4'b1111 // This is a 4-bit binary number
- 12'habc // This is a 12-bit hexadecimal number
- 16'd255 // This is a 16-bit decimal number

Unsigned Numbers

- Numbers that are specified without a <base format> specification are decimal numbers by default.
- Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine- specific (must be at least 32).
- 23456 // This is a 32-bit decimal number by default
- 'hc3 // This is a 32-bit hexadecimal number
- 'o21 // This is a 32-bit octal number

X or Z values

- Verilog has two symbols for unknown and high impedance values.
- These values are very important for modeling real circuits.
- An unknown value is denoted by an x.
- A high impedance value is denoted by z.
- 12'h13x // This is a 12-bit hex number; 4 least significant bits unknown
- 6'hx // This is a 6-bit hex number
- 32'bz // This is a 32-bit high impedance number

- An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base and one bit for a number in the binary base.
- If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z.
- This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

- Negative numbers can be specified by putting a minus sign before the size for a constant number.
- Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>.
- An optional signed specifier can be added for signed arithmetic.
- -6'd3 // 8-bit negative number stored as 2's complement of 3
- -6'sd3 // Used for performing signed integer math
- 4'd-2 // Illegal specification

Underscore characters and question marks

- An underscore character "_" is allowed anywhere in a number except the first character.
- Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.
- A question mark "?" is the Verilog HDL alternative for z in the context of numbers.
- The ? is used to enhance readability in the casex and casez statements.

Strings

- A string is a sequence of characters that are enclosed by double quotes.
- The restriction on a string is that it must be contained on a single line, that is, without a carriage return.
- It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.
- "Hello Verilog World" // is a string "a / b" // is a string.

Identifiers and Keywords

- Keywords are special identifiers reserved to define the language constructs.
- Keywords are in lowercase.
- Identifiers are names given to objects so that they can be referenced in the design.
- Identifiers are made up of alphanumeric characters, the underscore (_), or the dollar sign (\$).
- Identifiers are case sensitive. Identifiers start with an alphabetic character or an underscore. They cannot start with a digit or a \$ sign (The \$ sign as the first character is reserved for system tasks)
- `reg value; // reg is a keyword; value is an identifier`
- `input clk; // input is a keyword, clk is an identifier`

Escaped Identifiers


- Escaped identifiers begin with the backslash (\) character and end with whitespace (space, tab, or newline).
- All characters between backslash and whitespace are processed literally. Any printable ASCII character can be included in escaped identifiers.
- Neither the backslash nor the terminating whitespace is considered to be a part of the identifier.
- `\a+b-c`
- `**my_name**`

Data Types:- Value Set

- Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table 2-1.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

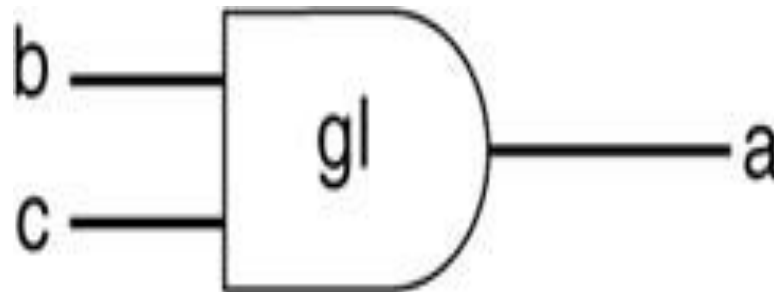
- In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in Table2-2.

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	
pull	riving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	weakest
highz	High Impedance	

-
- If two signals of unequal strengths are driven on a wire, the stronger signal prevails.
 - For example, if two signals of strength `strong1` and `weak0` contend, the result is resolved as a `strong1`.
 - If two signals of equal strengths are driven on a wire, the result is unknown.
 - If two signals of strength `strong1` and `strong0` conflict, the result is an `x`.
-

Nets

- Nets represent connections between hardware elements.
- Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to.
- In Figure 2.1 net a is connected to the output of and gate g1. Net a will continuously assume the value computed at the output of gate g1, which is b & c.



- Nets are declared primarily with the keyword wire.
- Nets are one-bit values by default unless they are declared explicitly as vectors.
- The terms wire and net are often used interchangeably.
- The default value of a net is z (except the trireg net, which defaults to x). Nets get the output value of their drivers.
- If a net has no driver, it gets the value z.
- wire a; // Declare net a for the above circuit
- wire b,c; // Declare two wires b,c for the above circuit
- wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.

Registers

- Registers represent data storage elements.
- Registers retain value until another value is placed onto them.
- In Verilog, the term register merely means a variable that can hold a value.
- Unlike a net, a register does not need a driver.
- Verilog registers do not need a clock as hardware register do.
- Values of registers can be changed anytime in a simulation by assigning a new value to the register.
- Register data types are commonly declared by the keyword `reg`.

- **Example 3-1 Example of Register**

- `reg reset; // declare a variable reset that can hold its value`
- `initial // keyword to specify the initial value of reg.`
- `reset = 1'b1; //initialize reset to 1 to reset the digital circuit.`
- `#100 reset = 1'b0; // after 100 time units reset is deasserted.`
- `End`

- **Example 2-2 Signed Register Declaration**

- `reg signed [63:0] m; // 64 bit signed value`
 - `integer i; // 32 bit signed value`
-

Vectors

- Nets or reg data types can be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit).
- `wire a; // scalar net variable,`
- `wire [7:0] bus; // 8-bit bus`
- `wire [31:0] busA, busB, busC; // 3 buses of 32-bit width.` `reg clock; // scalar register, default`
- `reg [0:40] virtual_addr; // Vector register, virtual address 41 bits wide.`
- Vectors can be declared at `[high# : low#]` or `[low# : high#]`,

Vector Part Select

- For the vector declarations shown above, it is possible to address bits or parts of vectors.
- `busA[7]` // bit # 7 of vector `busA`
- `bus[2:0]` // Three least significant bits of vector `bus`,
- *// using `bus[0:2]` is illegal because the significant bit should always be on the left of a range specification.*
- `virtual_addr[0:1]` // Two most significant bits of vector `virtual_addr`.

Vector Part Select

- For the vector declarations shown above, it is possible to address bits or parts of vectors.
`busA[7]` // bit # 7 of vector `busA`
- `bus[2:0]` // Three least significant bits of vector `bus`,
- *// using `bus[0:2]` is illegal because the significant bit should always be on the left of a range specification*
- `virtual_addr[0:1]` // Two most significant bits of vector `virtual_addr`

Variable Vector Part Select

- Another ability provided in Verilog HDL is to have variable part selects of a vector.
- This allows part selects to be put in for loops to select various parts of the vector.
- There are two special part-select operators:
[<starting_bit>+:width] - part-select increments from starting bit.
- [<starting_bit>-:width] - part-select decrements from starting bit.
- The starting bit of the part select can be varied, but the width has to be constant.

- The following example shows the use of variable vector part select:
- `reg [255:0] data1; //Little endian notation reg`
- `[0:255] data2; //Big endian notation reg [7:0] byte;`
- `//Using a variable part select, one can choose parts`
- `byte = data1[31 -:8]; //starting bit = 31, width =8 => data[31:24]`
- `byte = data1[24 +:8]; //starting bit = 24, width =8 => data[31:24]`
- `byte = data2[31 -:8]; //starting bit = 31, width =8 => data[24:31]`
- `byte = data2[24 +:8]; //starting bit = 24, width =8 => data[24:31]`

- `//The starting bit can also be a variable. The width has to be constant.`
- `//Therefore, one can use the variable part select`
- `//in a loop to select all bytes of the vector. for (j=0; j<=31; j=j+1)`
- `byte = data1[(j*8)+:8]; //Sequence is [7:0], [15:8]... [255:248]`
- `//Can initialize a part of the vector`
- `data1[(byteNum*8)+:8] = 8'b0; //If byteNum = 1, clear 8 bits [15:8]`

Data Types- Integers

- An integer is a general purpose register data type used for manipulating quantities.
- Integers are declared by the keyword **integer**.
- The default width for an integer is the host-machine word size, which is implementation-specific but is at least 32 bits.
- `integer counter; // general purpose variable used as a counter. initial`
- `counter = -1; // A negative one is stored in the counter`

Data Types- Real

- Real number constants and real register data types are declared with the keyword `real`.
- They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3×10^6).
- Real numbers cannot have a range declaration, and their default value is 0.
- When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

Example

- `real delta; // Define a real variable called delta`
`initial begin`
- `delta = 4e10; // delta is assigned in scientific notation`
- `delta = 2.13; // delta is assigned a value 2.13`
`end integer i; // Define an integer i`
- `initial`
- `i = delta; // i gets the value 2 (rounded value of 2.13)`

Data Types- Time

- Verilog simulation is done with respect to simulation time.
- A special time register data type is used in Verilog to store simulation time.
- A time variable is declared with the keyword **time**. The width for time register data types is implementation-specific but is at least 64 bits.
- The system function **\$time** is invoked to get the current simulation time.
- **time save_sim_time; // Define a time variable save_sim_time initial.**
- **save_sim_time = \$time; // Save the current simulation time**

Data Types- Arrays

- Arrays are allowed in Verilog for reg, integer, time, real, real time and vector register data types.
- Multi- dimensional arrays can also be declared with any number of dimensions.
- Arrays of nets can also be used to connect ports of generated instances.
- Each element of the array can be used in the same fashion as a scalar or vector net. Arrays are accessed by <array_name>[<subscript>].
- For multi- dimensional arrays, indexes need to be provided for each dimension.

Examples

- `integer count[0:7];` // An array of 8 count variables
- `count[5] = 0;` // Reset 5th element of array of count variables
- `chk_point[1:100];` // Array of 100 time checkpoint variables.
- `chk_point[100] = 0;` // Reset 100th time check point value

Memories

- In digital simulation, one often needs to model register files, RAMs, and ROMs.
- Memories are modeled in Verilog simply as a one-dimensional array of registers.
- Each element of the array is known as an element or word and is addressed by a single array index.
- Each word can be one or more bits. It is important to differentiate between n 1-bit registers and one n -bit register.
- A particular word in memory is obtained by using the address as a memory array subscript.

Example- Memory Declaration

- `reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words`
- `reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words(bytes)`
`membyte[511] // Fetches 1 byte word whose address is 511.`

Parameters

- Verilog allows constants to be defined in a module by the keyword parameter. Parameter
- `parameter port_id = 5; // Defines a constant port_id`
- `parameter cache_line_width = 256; // Constant defines width of cache line`
`parameter signed [15:0] WIDTH; // Fixed sign and range for parameter WIDTH`
- Parameters cannot be used as variables.

Strings

- Strings can be stored in reg. The width of the register variables must be large enough to hold the string.
- Each character in the string takes up 8 bits (1 byte). If the width of the register is greater than the size of the string, Verilog fills bits to the left of the string with zeros.
- If the register width is smaller than the string width, Verilog truncates the leftmost bits of the string.
- It is always safe to declare a string that is slightly wider than necessary. `reg [8*18:1] string_value; // Declare a variable that is 18 bytes wide initial`
- `string_value = "Hello Verilog World"; // String can be stored in variable`

Special characters

Escaped Characters	Character Displayed
<code>\n</code>	newline
<code>\t</code>	tab
<code>%%</code>	%
<code>\\</code>	\
<code>\"</code>	"
<code>\ooo</code>	Character written in 1?3 octal digits

System Tasks and Compiler Directives

- Verilog provides standard system tasks for certain routine operations. All system tasks appear in the form \$<keyword>.
 - Operations such as displaying on the screen, monitoring values of nets, stopping, and finishing are done by system tasks.
-

Displaying information

- `$display` is the main system task for displaying values of variables or strings or expressions. This is one of the most useful tasks in Verilog.
- Usage: `$display(p1, p2, p3, . . . , pn);`
- `p1, p2, p3, . . . , pn` can be quoted strings or variables or expressions. The format of `$display` is very similar to `printf` in C.
- A `$display` inserts a newline at the end of the string by default.
- A `$display` without any arguments produces a newline.

Monitoring information

- Verilog provides a mechanism to monitor a signal when its value changes. This facility is provided by the
- \$monitor task.
- Usage: \$monitor(p1,p2,p3,.. ,pn);
- The parameters p1, p2, ... , pn can be variables, signal names, or quoted strings. A format similar to the
- \$display task is used in the \$monitor task.

- \$monitor continuously monitors the values of the variables or signals specified in the parameter list and displays all parameters in the list whenever the value of any one variable or signal changes.
 - Unlike \$display, \$monitor needs to be invoked only once. Only one monitoring list can be active at a time.
 - If there is more than one \$monitor statement in your simulation, the last \$monitor statement will be the active statement. The earlier \$monitor statements will be overridden.
 - Two tasks are used to switch monitoring on and off.
-

Usage:

- \$monitoron;
 - \$monitoroff;
 - The \$monitoron task enables monitoring, and the \$monitoroff task disables monitoring during a simulation.
-

Example of Monitor Statement

//Monitor time and value of the signals clock and reset

//Clock toggles every 5 time units and reset goes down
at 10 time units initial

begin

\$monitor (\$time," Value of signals clock = %b reset =
%b", clock,reset); end

Partial output of the monitor statement:

-- 0 Value of signals clock = 0 reset = 1

-- 5 Value of signals clock = 1 reset = 1

-- 10 Value of signals clock = 0 reset = 0

Stopping and finishing in a simulation

- The task \$stop is provided to stop during a simulation. Usage: \$stop;
 - The \$stop task puts the simulation in an interactive mode. The designer can then debug the design from the interactive mode. The \$stop task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design.
 - The \$finish task terminates the simulation.
-

Example of Stop and Finish Tasks

- // Stop at time 100 in the simulation and examine the results
- // Finish the simulation at time 1000. initial
- begin clock = 0;
- reset = 1;
- #100 \$stop; // This will suspend the simulation at time = 100 #900 \$finish; // This will terminate the simulation at time = 1000 end

Compiler Directives

- Compiler directives are provided in Verilog. All compiler directives are defined by using the
- ``<keyword>` construct. The two most useful compiler directives are
- **`define** - The ``define` directive is used to define text macros in Verilog. The Verilog compiler substitutes the text of the macro wherever it encounters a ``<macro_name>`.
- This is similar to the `#define` construct in C. The defined constants or text macros are used in the Verilog code by preceding them with a ``` (back tick).

Example for `define Directive

- //define a text macro that defines default word size
- //Used as 'WORD_SIZE in the code 'define WORD_SIZE 32
- //define an alias. A \$stop will be substituted wherever 'S appears 'define S \$stop;
- //define a frequently used text string 'define WORD_REG reg [31:0]

- **`include**

- The `include directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation.
 - This works similarly to the #include in the C programming language.
-

Example for ``include` Directive

- `// Include the file header.v, which contains declarations in the main verilog file design.v.`
``include header.v`
- ...
- ...
- `<Verilog code in file design.v>`

Modules

- Module is a basic building block in Verilog.
 - A module definition always begins with the keyword module.
 - The module name, port list, port declarations, and optional parameters must come first in a module definition.
 - Port list and port declarations are present only if the module has any ports to interact with the external environment.
-

- The five components within a module are: variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions.
- These components can be in any order and at an
- The endmodule statement must always come last in a module definition.
- All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs.
- Verilog allows multiple modules to be defined in a single file. The modules can be defined in any order in the file. y place in the module definition.

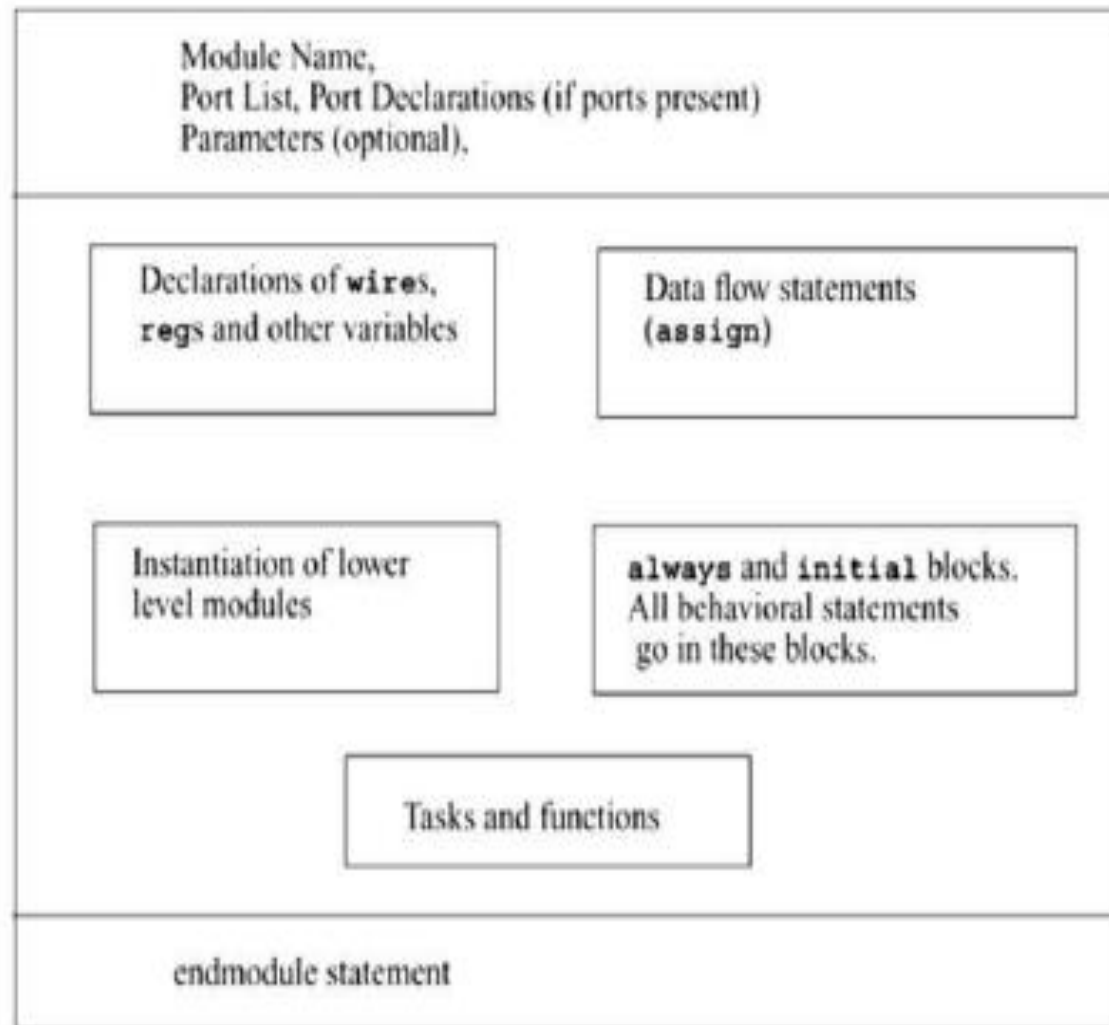


Figure 2.2.:Components of a Verilog Module

Example of Components of SR Latch

- The SR latch has S and R as the input ports and Q and Qbar as the output ports. The SR latch and its stimulus can be modeled as shown in Exam
- // This example illustrates the different components of a module
- // Module name and port list
- // SR_latch module
- module SR_latch(Q, Qbar, Sbar, Rbar);

-
- //Port declarations output Q, Qbar; input Sbar, Rbar
 - // Instantiate lower-level modules
 - // In this case, instantiate Verilog primitive nand gates
 - // Note how the wires are connected in a cross-coupled fashion.
 - nand n1(Q, Sbar, Qbar);
 - nand n2(Qbar, Rbar, Q);
 - // endmodule statement
 - endmodule
-

- `// Module name and port list`
- `// Stimulus module`
- `module Top;`
- `// Declarations of wire, reg, and other variables`
- `reg set, reset;`
- `// Instantiate lower-level modules`
- `// In this case, instantiate SR_latch Feed inverted set and reset signals to the SR latch`
- `SR_latch m1(q, qbar, ~set, ~reset);`
- `// Behavioral block, initial initial`
- `begin`
- `$monitor($time, " set = %b, reset= %b, q= %b\n",set,reset,q); set = 0; reset = 0;`
- `#5 reset = 1;`
- `#5 reset = 0;`
- `#5 set = 1; end`
- `// endmodule statement`
- `endmodule`

- From the above example following characteristics are noticed:
- In the SR latch definition above ,all components described in Figure 2-2 need not be present in a module.
- We do not find variable declarations, dataflow (assign) statements, or behavioral blocks (always or initial).
- However, the stimulus block for the SR latch contains module name, wire, reg, and variable declarations, instantiation of lower level modules, behavioral block (initial), and endmodule statement but does not contain port list, port declarations, and data flow (assign) statements.
- Thus, all parts except module, module name, and endmodule are optional and can be mixed and matched as per design needs.

Ports

- Ports provide the interface by which a module can communicate with its environment.
 - For example, the input/output pins of an IC chip are its ports.
 - The environment can interact with the module only through its ports.
 - The internals of the module are not visible to the environment.
 - This provides a very powerful flexibility to the designer. The internals of the module can be changed without affecting the environment as long as the interface is not modified.
 - Ports are also referred to as terminals.
-

List of Ports

- A module definition contains an optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list. Consider a 4-bit full adder that is instantiated inside a top-level module *Top*. The diagram for the input/output ports is shown in Figure 2-4.

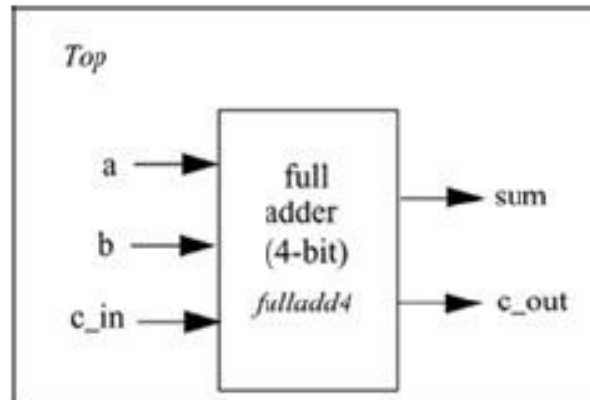


Figure 2-4. I/O Ports for *Top* and Full Adder

- From the above figure, the module Top is a top-level module. The module fulladd4 is instantiated below Top.
- The module fulladd4 takes input on ports a, b, and c_in and produces an output on ports sum and c_out. Thus, module fulladd4 performs an addition for its environment.
- The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment.
- Thus, it does not have a list of ports. The module names and port lists for both module declarations in Verilog are as shown in below example.
- **Example of List of Ports**
- `module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports`
- `module Top; // No list of ports, top-level module in simulation`

Port Declaration

- All ports in the list of ports must be declared in the module. Ports can be declared as follows: input -Input port
- output- Output port inout- Bidirectional port
- Each port in the port list is defined as input, output, or inout, based on the direction of the port signal.
- Thus, for the example of the the port declarations will be as shown in example below.

Example for Port Declarations

- module fulladd4(sum, c_out, a, b, c_in);

- //Begin port declarations section

- output[3:0] sum;

- output c_cout;

- input [3:0] a, b; input c_in;

- //End port declarations section

- ...

- <module internals>

- ... endmodule

- All port declarations are implicitly declared as wire in Verilog.
- Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout.
- Input or inout ports are normally declared as wires.
- However, if output ports hold their value, they must be declared as reg.
- Ports of the type input and inout cannot be declared as reg because reg variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.

- Alternate syntax for port declaration is shown in below example. This syntax avoids the duplication of naming the ports in both the module definition statement and the module port list definitions.
- If a port is declared but no data type is specified, then, under specific circumstances, the signal will default to a wire data type.
- `module fulladd4(output reg [3:0] sum, output reg c_out,`
- `input [3:0] a, b, //wire by default input c_in);`
- `//wire by default`
- `...`
- `<module internals>`
- `...`
- `endmodule`

Port Connection Rules

- A port as consisting of two units, one unit that is internal to the module and another that is external to the module.
 - The internal and external units are connected. There are rules governing port connections when modules are instantiated within other modules.
 - The Verilog simulator complains if any port connection rules are violated. These rules are summarized in Figure2.5
-

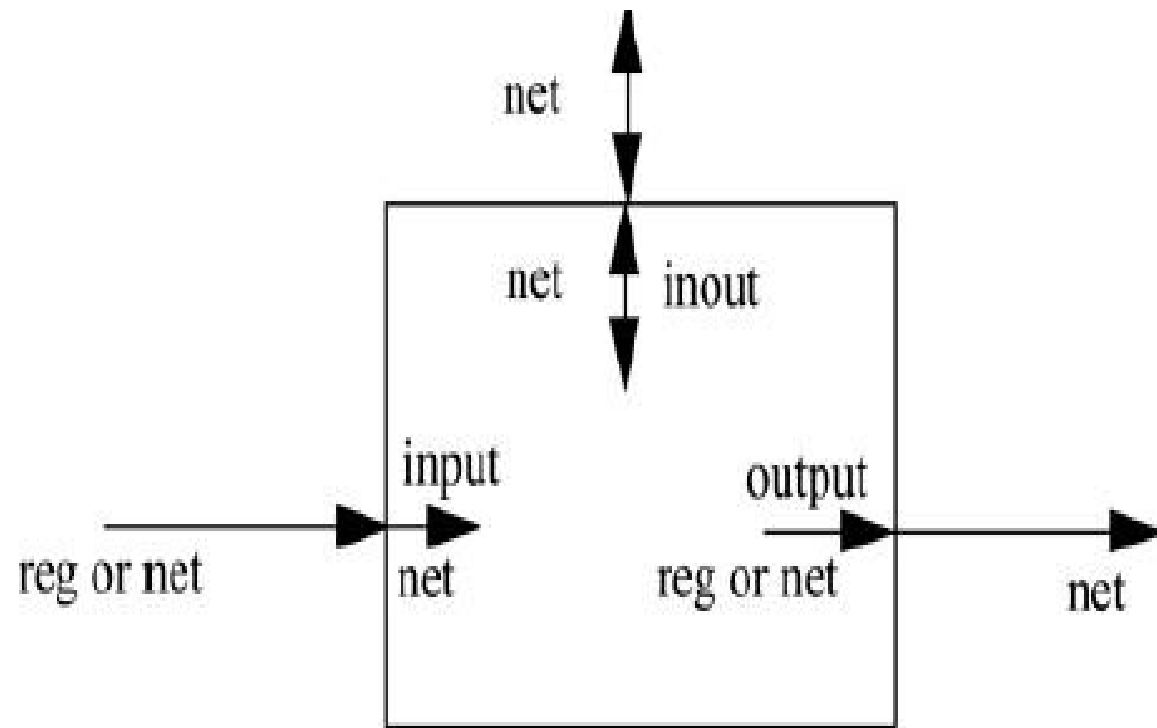


Figure 2-5. Port Connection Rules

■ **Inputs**

- Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.

■ **Outputs**

- Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.

■ **Inouts**

- Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.

■ **Width matching**

- It is legal to connect internal and external items of different sizes when making intermodule port connections. However, a warning is typically issued that the widths do not match.

- **Unconnected ports**

- Verilog allows ports to remain unconnected. For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals. You can let a port remain unconnected by instantiating a module as shown below

- `fulladd4 fa0 (SUM, A, B, C_IN); // Output port c_out is unconnected`

- **Example of illegal port connection**

- To illustrate port connection rules, assume that the module `fulladd4 Example` is instantiated in the stimulus block `Top`. Below example shows an illegal port connection
-

Example 2-14 Illegal Port Connection

- module Top;
- //Declare connection variables reg [3:0]A,B;
- reg C_IN;
- reg [3:0] SUM;
- wire C_OUT;
- //Instantiate fulladd4, call it fa0
- fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
- //Illegal connection because output port sum in module fulladd4
- //is connected to a register variable SUM in module Top.
- <stimulus>
-
- endmodule
- This problem is rectified if the variable SUM is declared as a net (wire).

Connecting Ports to External Signals

- There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition.
- These two methods cannot be mixed. These methods are
 - **Connecting by ordered list**
 - **Connecting ports by name**

Connecting by ordered list

- The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition.
- Consider the module fulladd4. To connect signals in module Top by ordered list, the Verilog code is shown in below example.
- Notice that the external signals SUM, C_OUT, A, B, and C_IN appear in exactly the same order as the ports sum, c_out, a, b, and c_in in module definition of fulladd4.

Example 2-15 Connection by Ordered List

- module Top;
- //Declare connection variables reg [3:0]A,B;
- reg C_IN;
- wire [3:0] SUM; wire C_OUT;
- //Instantiate fulladd4, call it fa_ordered.
- //Signals are connected to ports in order (by position) fulladd4 fa_ordered (SUM, C_OUT, A, B, C_IN);
- ...
- <stimulus>
- ... endmodule

-
- `module fulladd4(sum, c_out, a, b, c_in);`
 - `output[3:0] sum; output c_cout; input [3:0] a,`
`b; input c_in;`
 - `...`
 - `<module internals>`
 - `... endmodule`
-

Connecting ports by name

- For large designs where modules have, say, 50 ports, remembering the order of the ports in the module definition is impractical and error-prone.
- Verilog provides the capability to connect external signals to ports by the port names, rather than by position.
- Another advantage of connecting ports by name is that as long as the port name is not changed, the order of ports in the port list of a module can be rearranged without changing the port connections in module instantiations.
- We could connect the ports by name in above example by instantiating the module fulladd4, as follows.



- // Instantiate module fa_byname and connect signals to ports by name
- `fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A),);`
- Note that only those ports that are to be connected to external signals must be specified in port connection by name.
- Unconnected ports can be dropped. For example, if the port c_out were to be kept unconnected, the instantiation of fulladd4 would look as follows. The port c_out is simply dropped from the port list.
- // Instantiate module fa_byname and connect signals to ports by name
- `fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);`

Reference / Text Book Details

Sl.No	Title of Book	Author	Publication	Edition
1	Verilog HDL: A Guide to Digital Design and Synthesis	Samir Palnitkar	Pearson Education	2 nd
2	VHDL for Programmable Logic	Kevin Skahill	PHI/Pearson education	2 nd
3	The Verilog Hardware Description Language	Donald E. Thomas, Philip R. Moorby	Springer Science+Business Media, LLC	5 th
4	Advanced Digital Design with the Verilog HDL	Michael D. Ciletti	Pearson (Prentice Hall)	2 nd
5	Design through Verilog HDL	Padmanabhan, Tripura Sundari	Wiley	Latest

Thank You





|| JAI SRI GURUDEV ||
Sri AdichunchanagiriShikshana Trust (R)
SJB INSTITUTE OF TECHNOLOGY
BGS Health & Education City, Kengeri , Bangalore – 60 .

***DEPARTMENT OF ELECTRONICS & COMMUNICATION
ENGINEERING***

Verilog HDL [18EC56]

Module 3: Gate-Level and Data Flow Modelling

By:

Mrs. LATHA S
Assistant Professor,
Dept. of ECE, SJBIT

Content

1. Basic Verilog gate primitives
2. Description of and/or and buf/not type gates
3. Rise, fall and turn-off delays
4. Min, max, and typical delays
5. Continuous assignments
6. Delay specification
7. Expressions, Operators, operands
8. Operator types

Learning Objectives

- Identify logic gate primitives provided in Verilog.
 - Understand instantiation of gates, gate symbols, and truth tables for and/or and buf/not type gates.
 - Understand how to construct a Verilog description from the logic diagram of the circuit.
 - Describe rise, fall, and turn-off delays in the gate-level design and Explain min, max, and type delays in the gate-level design
-

Learning Objectives

- Describe the continuous assignment (assign) statement, restrictions on the assign statement, and the implicit continuous assignment statement.
- Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statements and Define expressions, operators, and operands.
- Use dataflow constructs to model practical digital circuits in Verilog

Gate Types

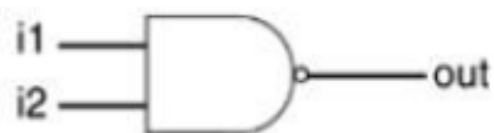
- A logic circuit can be designed by use of logic gates.
- Verilog supports basic logic gates as predefined primitives.
- These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition.
- All logic circuits can be designed by using basic gates. There are two classes of basic gates: **and/or gates** and **buf/not gates**.

And/Or Gates

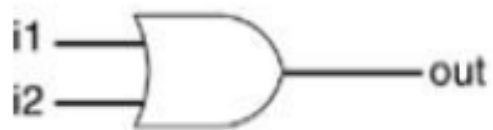
- And/or gates have one scalar output and multiple scalar inputs.
- The first terminal in the list of gate terminals is an output and the other terminals are inputs.
- The output of a gate is evaluated as soon as one of the inputs changes.
- The and/or gates available in Verilog are: **and**, **or**, **xor**, **nand**, **nor**, **xnor**.
- The corresponding logic symbols for these gates are shown in Figure 3-1. Consider the gates with two inputs.
- The output terminal is denoted by out. Input terminals are denoted by i1 and i2.



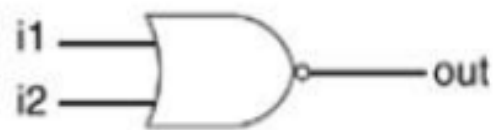
and



nand



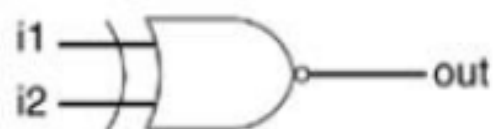
or



nor



xor



xnor

Figure 3-1. Basic Gates

Gate Instantiation of And / Or Gates

- wire OUT, IN1, IN2;
- **// basic gate instantiations.**
- and a1(OUT, IN1, IN2);
- nand na1(OUT, IN1, IN2);
- or or1(OUT, IN1, IN2);
- nor nor1(OUT, IN1, IN2);
- xor x1(OUT, IN1, IN2);
- xnor nx1(OUT, IN1, IN2);
- // More than two inputs; 3 input nand gate
- nand na1_3inp(OUT, IN1, IN2, IN3);
- **// gate instantiation without instance name**
- and (OUT, IN1, IN2); **// legal gate instantiation**

Truth Table

	and	i1			
		0	1	x	z
i2	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

	nand	i1			
		0	1	x	z
i2	0	1	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x

		i1			
		0	1	x	z
i2	or				
	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

		i1			
		0	1	x	z
i2	nor				
	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x

	xor	i1			
		0	1	x	z
i2	0	0	1	x	x
	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

	xnor	i1			
		0	1	x	z
i2	0	1	0	x	x
	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

Buf/Not Gates

- Buf/not gates have one scalar input and one or more scalar outputs. The last terminal in the port list is connected
- to the input. Other terminals are connected to the outputs. We will discuss gates that have one input and one
- output. Two basic buf/not gate primitives are provided in Verilog.
- The symbols for these logic gates are shown in Figure 3-2.

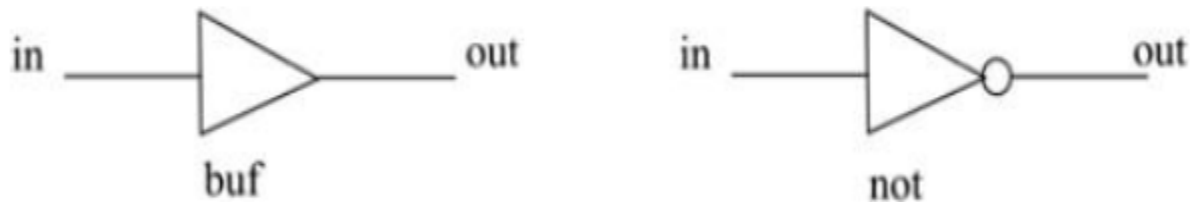


Figure 3-2. Buf/not Gates

Example 3-2 Gate Instantiations of Buf/Not Gates

- These gates are instantiated in Verilog as shown Example 3-2. Notice that these gates can have multiple outputs but exactly one input, which is the last terminal in the port list.
- `// basic gate instantiations.`
- `buf b1(OUT1, IN);`
- `not n1(OUT1, IN);`
- `// More than two outputs`
- `buf b1_2out(OUT1, OUT2, IN);`
- `// gate instantiation without instance name`
- `not (OUT1, IN); // legal gate instantiation`
- Truth tables for gates with one input and one output are shown in Table 3-2.

Table 3-2. Truth Tables for Buf/Not Gates

buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x

Bufif/notif

- Gates with an additional control signal on buf and not gates are also available.
- These gates propagate only if their control signal is asserted. They propagate z if their control signal is deasserted. Symbols for bufif/notif are shown in Figure

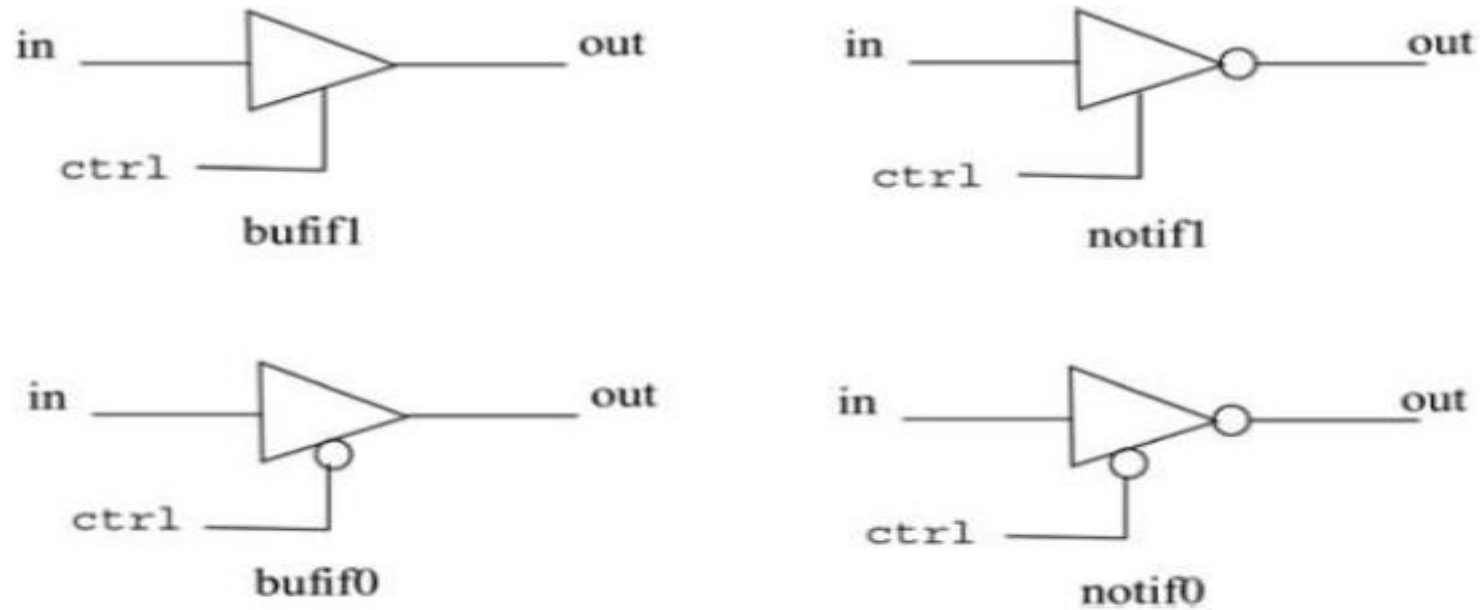


Figure 3-3. Bufif/notif Gates

Table 3-3. Truth Tables for Bufif/Notif Gates

		ctrl			
bufif1		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
bufif0		0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

		ctrl			
notif1		0	1	x	z
in	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

		ctrl			
notif0		0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

Example 3-3 Gate Instantiations of Bufif/Notif Gates

- //Instantiation of bufif gates.
 - bufif1 b1 (out, in, ctrl);
 - bufif0 b0 (out, in, ctrl);
 - //Instantiation of notif gates
 - notif1 n1 (out, in, ctrl);
 - notif0 n0 (out, in, ctrl);
-

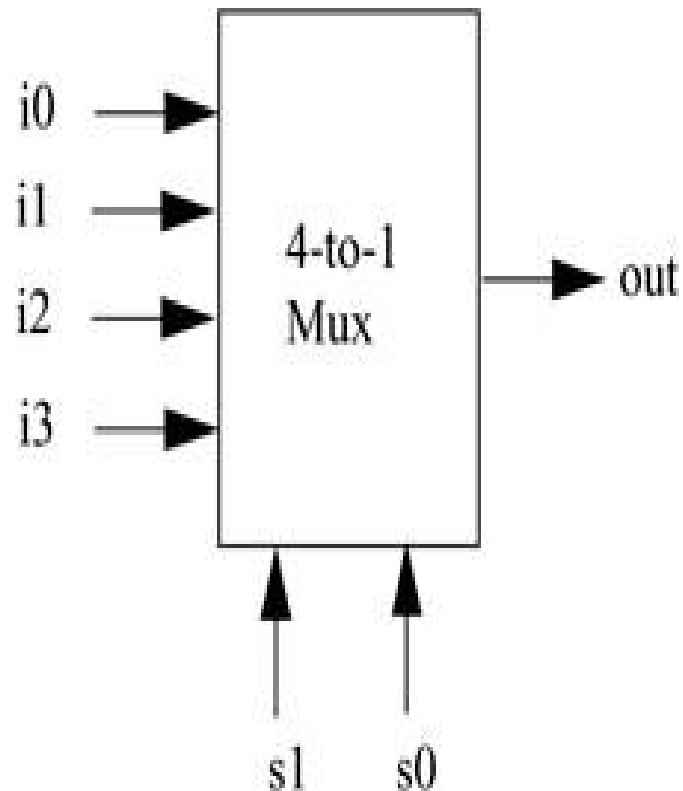
Array of Instances

- There are many situations when repetitive instances are required.
 - These instances differ from each other only by the index of the vector to which they are connected.
 - To simplify specification of such instances, Verilog HDL allows an array of primitive instances to be defined.
-

Simple Array of Primitive Instances

- `wire [7:0] OUT, IN1, IN2;`
- `// basic gate instantiations.`
- `nand n_gate[7:0](OUT, IN1, IN2);`
- `// This is equivalent to the following 8 instantiations`
- `nand n_gate0(OUT[0], IN1[0], IN2[0]);`
- `nand n_gate1(OUT[1], IN1[1], IN2[1]);`
- `nand n_gate2(OUT[2], IN1[2], IN2[2]);`
- `nand n_gate3(OUT[3], IN1[3], IN2[3]);`
- `nand n_gate4(OUT[4], IN1[4], IN2[4]);`
- `nand n_gate5(OUT[5], IN1[5], IN2[5]);`
- `nand n_gate6(OUT[6], IN1[6], IN2[6]);`
- `nand n_gate7(OUT[7], IN1[7], IN2[7]);`

Gate-level multiplexer



s_1	s_0	out
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_3

Logic Diagram for Multiplexer

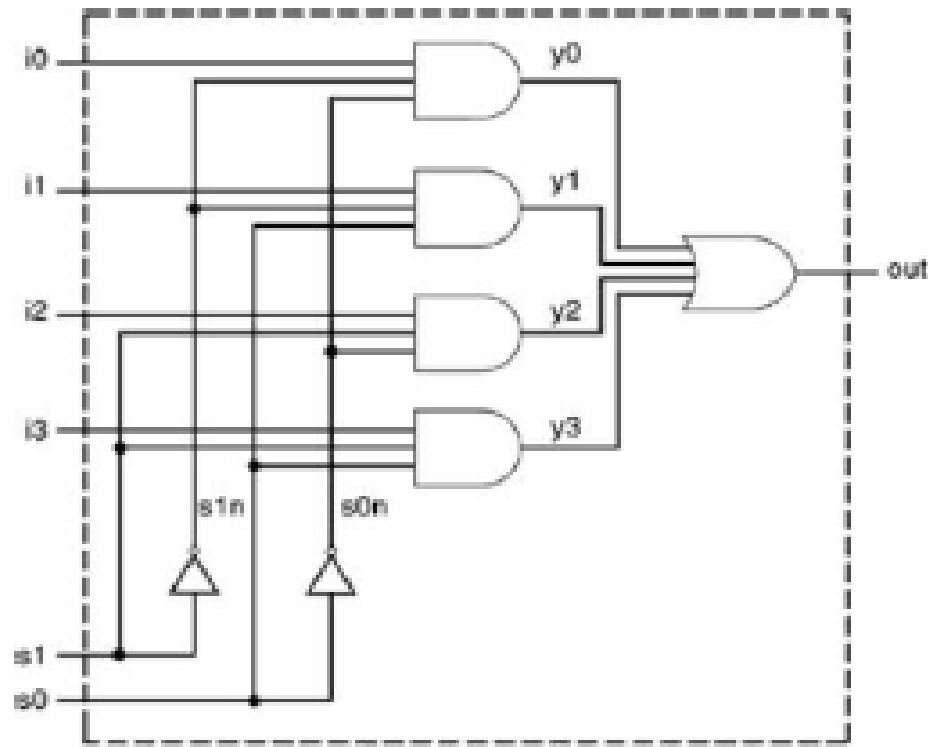


Figure 3-5. Logic Diagram for Multiplexer

Example 3-5 Verilog Description of Multiplexer

- // Module 4-to-1 multiplexer. Port list is taken exactly from// the I/O diagram.
- module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
- // Port declarations from the I/O diagram
- output out;
- input i0, i1, i2, i3;
- input s1, s0;
- // Internal wire declarations
- wire s1n, s0n;
- wire y0, y1, y2, y3;

-
- // Gate instantiations
 - // Create s1n and s0n signals.
 - not (s1n, s1);
 - not (s0n, s0);
 - // 3-input and gates instantiated
 - and (y0, i0, s1n, s0n);
 - and (y1, i1, s1n, s0);
 - and (y2, i2, s1, s0n);
 - and (y3, i3, s1, s0);
 - // 4-input or gate instantiated
 - or (out, y0, y1, y2, y3);
-

Stimulus for Multiplexer

- `// Define the stimulus module (no ports)`
- `module stimulus;`
- `// Declare variables to be connected// to inputs`
- `reg IN0, IN1, IN2, IN3;`
- `reg S1, S0;`
- `// Declare output wire`
- `wire OUTPUT;`
- `// Instantiate the multiplexer`
- `mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);`

-
- `// Stimulate the inputs`
 - `// Define the stimulus module (no ports)`
 - `initial`
 - `begin`
 - `// set input lines`
 - `IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;`
 - `#1 $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n",IN0,IN1,IN2,IN3);`
 - `// choose IN0`
 - `S1 = 0; S0 = 0;`
 - `#1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);`
 - `// choose IN1`
-

-
- S1 = 0; S0 = 1;
 - #1 \$display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
 - // choose IN2
 - S1 = 1; S0 = 0;
 - #1 \$display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
 - // choose IN3
 - S1 = 1; S0 = 1;
 - #1 \$display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
 - end
 - endmodule
-

- The output of the simulation is shown below. Each combination of the select signals is tested.
- $IN0 = 1, IN1 = 0, IN2 = 1, IN3 = 0$
- $S1 = 0, S0 = 0, OUTPUT = 1$
- $S1 = 0, S0 = 1, OUTPUT = 0$
- $S1 = 1, S0 = 0, OUTPUT = 1$
- $S1 = 1, S0 = 1, OUTPUT = 0$

1-bit Full Adder

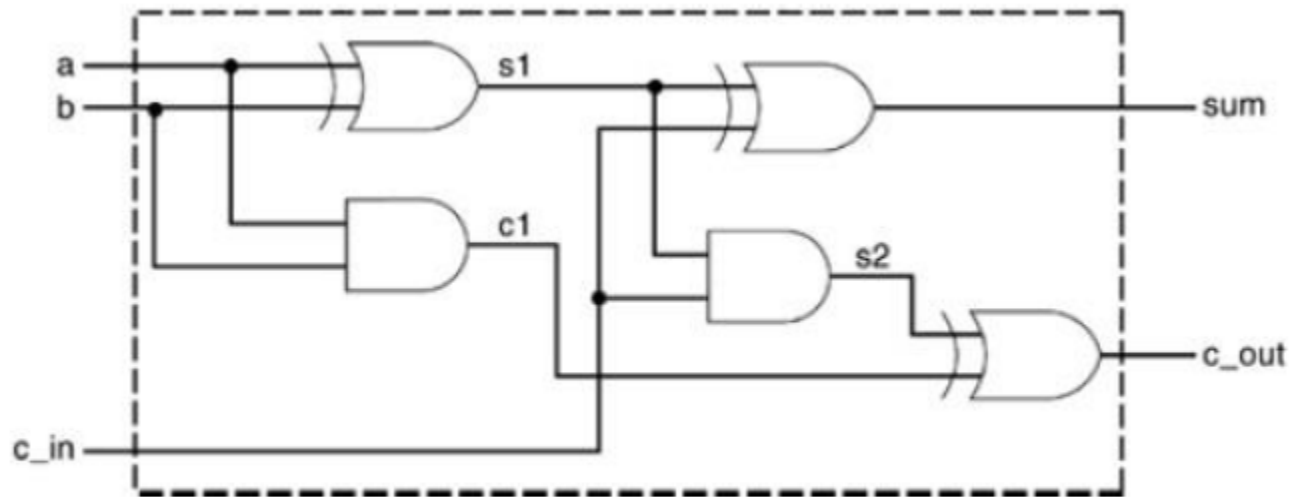


Figure 3-6. 1-bit Full Adder

■ Example 3-7 Verilog Description for 1-bit Full Adder

- *// Define a 1-bit full adder*
- module fulladd(sum, c_out, a, b, c_in);
- *// I/O port declarations*
- output sum, c_out;
- input a, b, c_in;
- *// Internal nets*
- wire s1, c1, c2;
- *// Instantiate logic gate primitives*
- xor (s1, a, b);
- and (c1, a, b);
- xor (sum, s1, c_in);
- and (c2, s1, c_in);
- xor (c_out, c2, c1);
- endmodule

4-bit Ripple Carry Full Adder

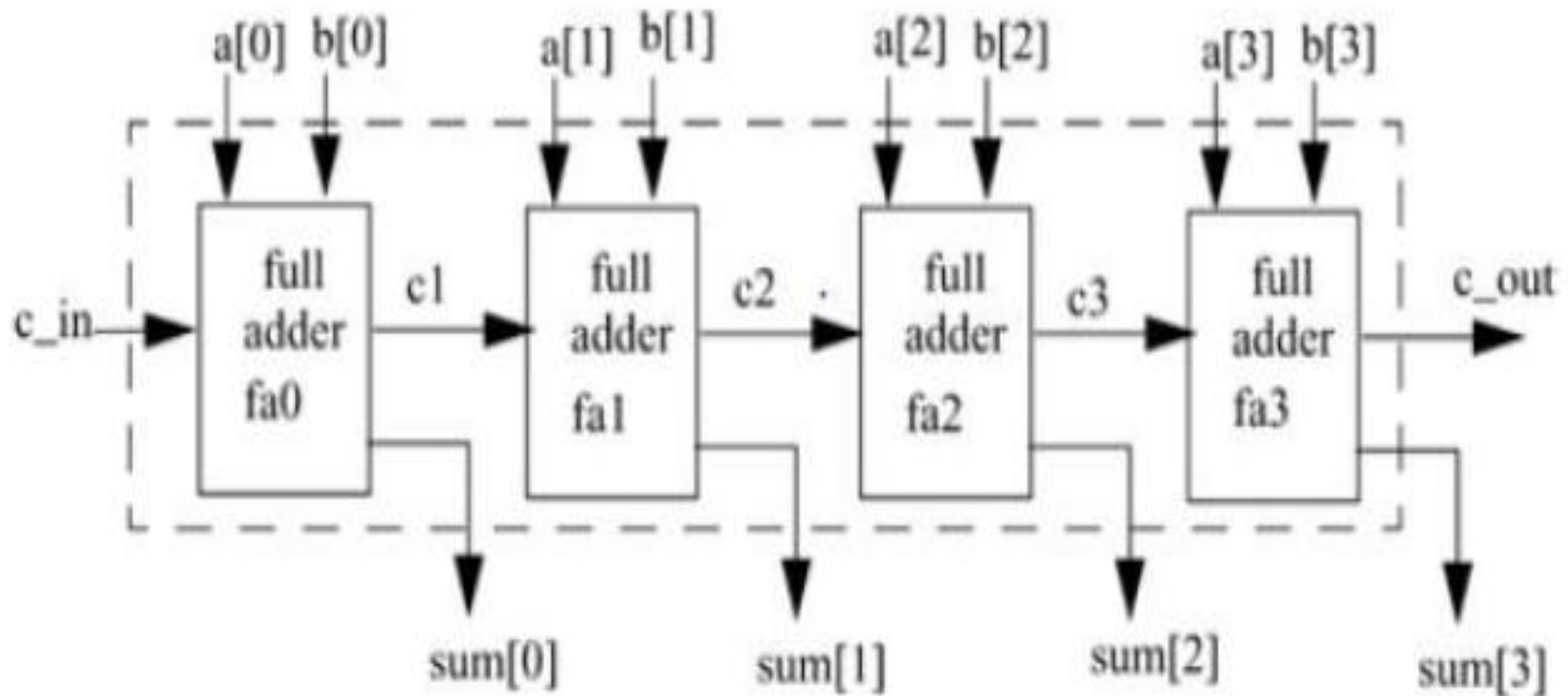


Figure 3-7. 4-bit Ripple Carry Full Adder

- **Example 3-8 Verilog Description for 4-bit Ripple Carry Full Adder**
- **// Define a 4-bit full adder**
- `module fulladd4(sum, c_out, a, b, c_in);`
- **// I/O port declarations**
- `output [3:0] sum;`
- `output c_out;`
- `input[3:0] a, b;`
- `input c_in;`
- **// Internal nets**
- `wire c1, c2, c3;`

-
- // Instantiate four 1-bit full adders.
 - fulladd fa0(sum[0], c1, a[0], b[0], c_in);
 - fulladd fa1(sum[1], c2, a[1], b[1], c1);
 - fulladd fa2(sum[2], c3, a[2], b[2], c2);
 - fulladd fa3(sum[3], c_out, a[3], b[3], c3);
 - endmodule
-

-
- **Example 3-9 Stimulus for 4-bit Ripple Carry Full Adder**
 - `// Define the stimulus (top level module)`
 - `module stimulus;`
 - `// Set up variables`
 - `reg [3:0] A, B;`
 - `reg C_IN;`
 - `wire [3:0] SUM;`
 - `wire C_OUT;`
 - `// Instantiate the 4-bit full adder. call it FA1_4`
 - `fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);`
-

- // Set up the monitoring for the signal values
- initial
- begin
- \$monitor(\$time," A= %b, B=%b, C_IN= %b, --
- C_OUT= %b, SUM= %b\n",
- A, B, C_IN, C_OUT, SUM);
- End
- // Stimulate inputs
- initial
- begin
- A = 4'd0; B = 4'd0; C_IN = 1'b0;
- #5 A = 4'd3; B = 4'd4;

-
- #5 A = 4'd2; B = 4'd5;
 - #5 A = 4'd9; B = 4'd9;
 - #5 A = 4'd10; B = 4'd15;
 - #5 A = 4'd10; B = 4'd5; C_IN = 1'b1;
 - end
 - endmodule
-

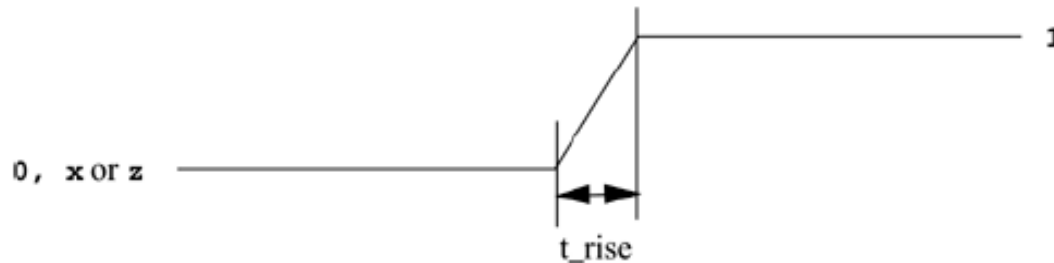
-
- The output of the simulation is shown below.
 - 0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000
 - 5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111
 - 10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111
 - 15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010
 - 20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001
 - 25 A= 1010, B=0101, C_IN= 1, --- C_OUT= 1, SUM= 0000
-

Gate Delays

- Until now, circuits are described without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them.
- Gate delays allow the Verilog user to specify delays through the logic circuits.
- Pin-to-pin delays can also be specified in Verilog.
- There are three types of delays from the inputs to the output of a primitive gate : **Rise, Fall, and Turn-off Delays**

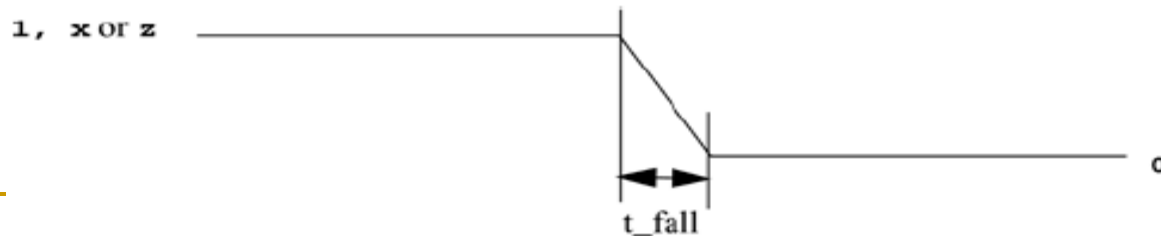
■ Rise delay

- The rise delay is associated with a gate output transition to a 1 from another value.



■ Fall delay

- The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

- The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.
- If the value changes to x, the minimum of the three delays is considered.
- Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions.
- If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays.
- If all three delays are specified, they refer to rise, fall, and turn-off delay values.
- If no delays are specified, the default value is zero. Examples of delay specification are shown in Example 3-10.

Example 3-10 Types of Delay Specification

- `// Delay of delay_time for all transitions`
- `and #(delay_time) a1(out, i1, i2);`
- `// Rise and Fall Delay Specification.`
- `and #(rise_val, fall_val) a2(out, i1, i2);`
- `// Rise, Fall, and Turn-off Delay Specification`
- `bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);`

- Examples of delay specification are shown below.
- `and #(5) a1(out, i1, i2); //Delay of 5 for all transitions`
- `and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6`
- `bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off= 5`

Min/Typ/Max Values

- Verilog provides an additional level of control for each type of delay mentioned above.
- For each type of delay? rise, fall, and turn-off? three values, min, typ, and max, can be specified.
- Any one value can be chosen at the start of the simulation.
- Min/typ/max values are used to model devices whose delays vary within a minimum and maximum range be
- **Min value:-** The min value is the minimum delay value that the designer expects the gate to have.
- **Typ val:-** The typ value is the typical delay value that the designer expects the gate to have.

■ **Max value**

- The max value is the maximum delay value that the designer expects the gate to have.
- Min, typ, or max values can be chosen at Verilog run time.
- Method of choosing a min/typ/max value may vary for different simulators or operating systems. (For Verilog-XL , the values are chosen by specifying options +maxdelays, +typdelays, and +mindelays at run time.
- If no option is specified, the typical delay value is the default).
- This allows the designers the flexibility of building three delay values for each transition into their design.
- The
- ~~■ designer can experiment with delay values without modifying the design.~~

■ Example 3-11 Min, Max, and Typical Delay Values

■ // One delay

- // if +mindelays, delay= 4
- // if +typdelays, delay= 5
- // if +maxdelays, delay= 6
- and #(4:5:6) a1(out, i1, i2);

■ // Two delays

- // if +mindelays, rise= 3, fall= 5, turn-off = min(3,5)
 - // if +typdelays, rise= 4, fall= 6, turn-off = min(4,6)
 - // if +maxdelays, rise= 5, fall= 7, turn-off = min(5,7)
 - and #(3:4:5, 5:6:7) a2(out, i1, i2);
-

- **// Three delays**
- // if +mindelays, rise= 2 fall= 3 turn-off = 4
- // if +typdelays, rise= 3 fall= 4 turn-off = 5
- // if +maxdelays, rise= 4 fall= 5 turn-off = 6
- and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
- Examples of invoking the Verilog-XL simulator with the command-line options are shown below. Assume that the module with delays is declared in the file test.v.
- **//invoke simulation with maximum delay**
- > verilog test.v +maxdelays
- **//invoke simulation with minimum delay**
- > verilog test.v +mindelays
- **//invoke simulation with typical delay**
- > verilog test.v +typdelays

Delay Example

- Let us consider a simple example to illustrate the use of gate delays to model timing in the logic circuits.
- A simple module called D implements the following logic equations:
- $out = (a \cdot b) + c$
- The gate-level implementation is shown in Module D (Figure 3-8). The module contains two gates with delays of 5 and 4 time units.

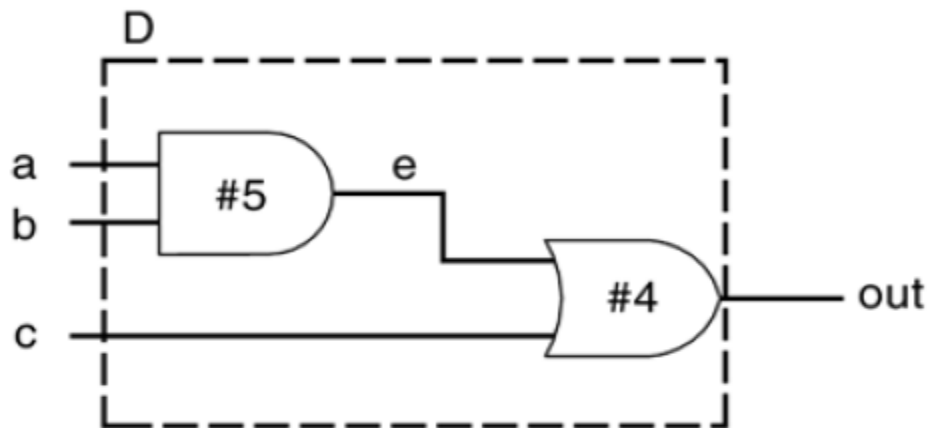


Figure 3-8. Module D

-
- **Example 3-12 Verilog Definition for Module D with Delay**
 - **// Define a simple combination module called D**
 - `module D (out, a, b, c);`
 - **// I/O port declarations**
 - `output out;`
 - `input a,b,c;`
 - **// Internal nets**
 - `wire e;`
 - **// Instantiate primitive gates to build the circuit**
 - `and #(5) a1(e, a, b); //Delay of 5 on gate a1`
 - `or #(4) o1(out, e,c); //Delay of 4 on gate o1`
 - `endmodule`
-

■ **Example 3-13 Stimulus for Module D with Delay**

- `// Stimulus (top-level module)`
- `module stimulus;`
- `// Declare variables`
- `reg A, B, C;`
- `wire OUT;`
- `// Instantiate the module D`
- `D d1(OUT, A, B, C);`
- `// Stimulate the inputs. Finish the simulation at 40 time units.`
- `initial`
- `begin`
- `A= 1'b0; B= 1'b0; C= 1'b0;`
- `#10 A= 1'b1; B= 1'b1; C= 1'b1;`
- `#10 A= 1'b1; B= 1'b0; C= 1'b0;`
- `#20 $finish;`
- `end`
- `endmodule`

- The waveforms from the simulation are shown in Figure 3-9 to illustrate the effect of specifying delays on
- gates.
- The waveforms are not drawn to scale. However, simulation time at each transition is specified below the
- transition.
- The outputs E and OUT are initially unknown.
- At time 10, after A, B, and C all transition to 1, OUT transitions to 1 after a delay of 4 time units and E changes value to 1 after 5 time units.
- At time 20, B and C transition to 0. E changes value to 0 after 5 time units, and OUT transitions to 0, 4 time units after E changes.

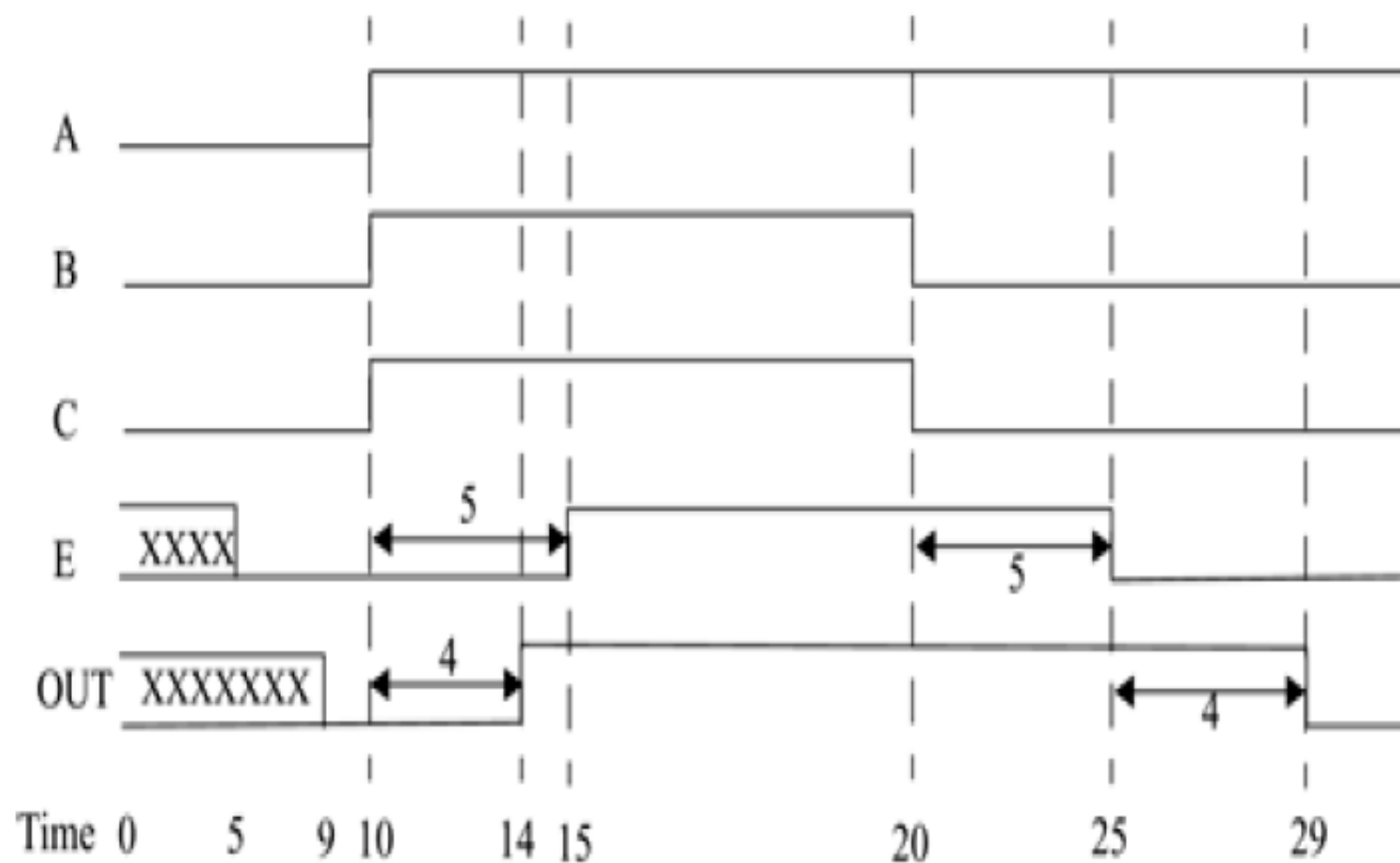


Figure 3-9. Waveforms for Delay Simulation of module D

Data Flow Modeling

- For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connects every gate individually.
- Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large.
- Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level.
- Dataflow modeling provides a powerful way to implement a design.
- Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

Continuous Assignments

- A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This
- assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction.
- The assignment statement starts with the keyword assign. The syntax of an assign statement is as follows.
- `continuous_assign ::= assign [drive_strength] [delay3]
list_of_net_assignments ;`
- `list_of_net_assignments ::= net_assignment { ,
net_assignment }`
- `net_assignment ::= net_lvalue = expression`

- The default value for drive strength is strong1 and strong0. The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates. Continuous assignments have the following characteristics:
- The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
- Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right hand- side operands changes and the value is assigned to the left-hand-side net.
- The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
- Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value.
- This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

- **Example 3-14 Examples of Continuous Assignment**
- *// Continuous assign. out is a net. i1 and i2 are nets.*
- `assign out = i1 & i2;`
- *// Continuous assign for vector nets. addr is a 16-bit vector net // addr1 and addr2 are 16-bit vector registers.*
- `assign addr[15:0] =addr1_bits[15:0]^addr2_bits[15:0];`
- *// Concatenation. Left-hand side is a concatenation of a scalar// net and a vector net.*
- `assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;`

Implicit Continuous Assignment

- Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration
- assignment per net because a net is declared only once.
- In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.
- **//Regular continuous assignment**
- wire out;
- assign out = in1 & in2;
- **//Same effect is achieved by an implicit continuous assignment**
- wire out = in1 & in2;

Implicit Net Declaration

- If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name.
- If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.
- `wire i1, i2;`
- `assign out = i1 & i2;` //Note that out was not declared as a wire
- `//but an implicit wire declaration for out`
- `//is done by the simulator`

Delays

- Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side.
 - Three ways of specifying delays in continuous assignment statements are **regular assignment delay**, **implicit continuous assignment delay**, and **net declaration delay**.
-

Regular Assignment Delay

- The first method is to assign a delay value in a continuous assignment statement.
- The delay value is specified after the keyword assign.
- Any change in values of in1 or in2 will result in a delay of 10 time units before re-computation of the expression in1 & in2, and the result will be assigned to out.
- If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of re-computation are considered.
- This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

- `assign #10 out = in1 & in2; // Delay in a continuous assign`
- 1. When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
- 2. When in1 goes low at 60, out changes to low at 70.
- 3. However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.
- 4. Hence, at the time of re-computation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is no propagated to the output.

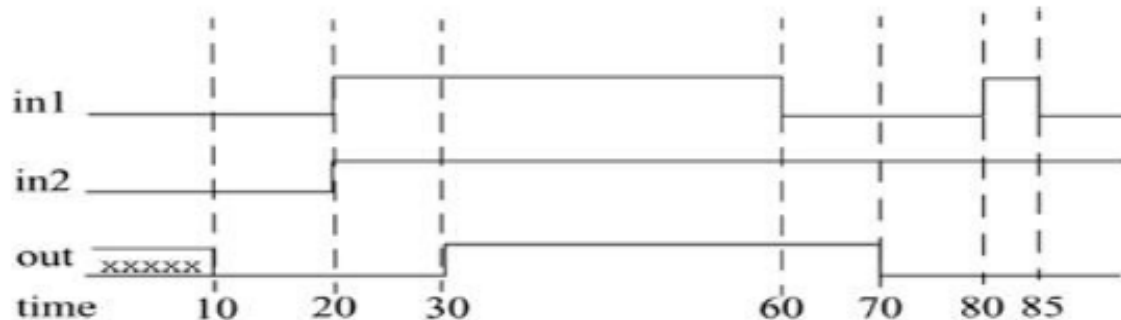


Figure 3-10. Waveforms for Delay Simulation

Implicit Continuous Assignment Delay

- An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.
- `//implicit continuous assignment delay`
- `wire #10 out = in1 & in2;`
- `//same as`
- `wire out;`
- `assign #10 out = in1 & in2;`
- The declaration above has the same effect as defining a wire out and declaring a continuous assignment on out.

Net Declaration Delay

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly.
- Net declaration delays can also be used in gate-level modeling.
- **//Net Delays**
- wire # 10 out;
- assign out = in1 & in2;
- **//The above statement has the same effect as the following.**
- wire out;
- assign #10 out = in1 & in2;

Expressions, Operators, and Operands

- Dataflow modeling describes the design in terms of expressions instead of primitive gates.
- Expressions, operators, and operands form the basis of dataflow modeling.
- Expressions are constructs that combine operators and operands to produce
- // Examples of expressions. Combines operands and operators
- $a \wedge b$
- `addr1[20:17] + addr2[20:17]`
- `in1 | in2`

Operands

- Operands can be any one of the data types defined, Data Types.
- Some constructs will take only certain types of operands.
- Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls
- **Examples:** integer count, final_count;
- `final_count = count + 1;`//count is an integer operand
- `real a, b, c;`

- `c = a - b;` *//a and b are real operands*
- `reg [15:0] reg1, reg2;`
- `reg [3:0] reg_out;`
- `reg_out = reg1[3:0] ^ reg2[3:0];`*//reg1[3:0] and reg2[3:0] are //part-select register operands*
- `reg ret_value;`
- `ret_value=calculate_parity(A,B);`*//calculate_parity is a//function type operand*

Operators

- Operators act on the operands to produce desired results.
- Verilog provides various types of operators.
Operator
- Types `d1 && d2` // `&&` is an operator on operands `d1` and `d2`.
- `!a[0]` // `!` is an operator on operand `a[0]`
- `B >> 1` // `>>` is an operator on operands `B` and `1`

Operator Types

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two

■ Arithmetic

- There are five arithmetic operators in Verilog.

```
■ module Arithmetic (A, B, Y1, Y2, Y3, Y4, Y5);  
■     input [2:0] A, B;  
■     output [3:0] Y1;  
■     output [4:0] Y3;  
■     output [2:0] Y2, Y4, Y5;  
■     reg [3:0] Y1;  
■     reg [4:0] Y3;  
■     reg [2:0] Y2, Y4, Y5;  
■     always @(A or B)  
■     begin  
■         Y1=A+B;//addition  
■         Y2=A-B;//subtraction  
■         Y3=A*B;//multiplication  
■         Y4=A/B;//division  
■         Y5=A%B;//modulus of A divided by B  
■     end  
■ endmodule
```

Logical and Relational Operators

Logical	!	logical negation	one
	&&	logical and	two
		logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two

Equality and Bitwise Operators

Equality	<code>==</code>	equality	two
	<code>!=</code>	inequality	two
	<code>===</code>	case equality	two
	<code>!==</code>	case inequality	two

Bitwise	<code>~</code>	bitwise negation	one
	<code>&</code>	bitwise and	two
	<code> </code>	bitwise or	two
	<code>^</code>	bitwise xor	two
	<code>^~</code> or <code>~^</code>	bitwise xnor	two

■ Equality and inequality

- Equality and inequality operators are used in exactly the same way as relational operators and return a true or false indication depending on whether any two operands are equivalent or not.

```
■ module Equality (A, B, Y1, Y2, Y3);  
■     input [2:0] A, B;  
■     output Y1, Y2;  
■     output [2:0] Y3;  
■     reg Y1, Y2;  
■     reg [2:0] Y3;  
■     always @(A or B)  
■     begin  
■         Y1=A==B;//Y1=1 if A equivalent to B  
■         Y2=A!=B;//Y2=1 if A not equivalent to B  
■         if (A==B)//parenthesis needed  
■             Y3=A;  
■         else  
■             Y3=B;  
■     end  
■ endmodule
```

■ Bit-wise

- Logical bit-wise operators take two single or multiple operands on either side of the operator and return a single bit result. The only exception is the **NOT** operator, which negates the single operand that follows. Verilog does not have the equivalent of **NAND** or **NOR** operator, their function is implemented by negating the **AND** and **OR** operators.

```
■ module Bitwise (A, B, Y);  
■     input [6:0] A;  
■     input [5:0] B;  
■     output [6:0] Y;  
■     reg [6:0] Y;  
■     always @(A or B)  
■     begin  
■         Y(0)=A(0)&B(0); //binary AND  
■         Y(1)=A(1)|B(1); //binary OR  
■         Y(2)=!(A(2)&B(2)); //negated AND  
■         Y(3)=!(A(3)|B(3)); //negated OR  
■         Y(4)=A(4)^B(4); //binary XOR  
■         Y(5)=A(5)~^B(5); //binary XNOR  
■         Y(6)=!A(6); //unary negation  
■     end  
■ endmodule
```


Reduction	&	reduction and	one
	~&	reduction nand	one
		reduction or	one
	~	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

■ Reduction

- Verilog has six reduction operators, these operators accept a single vectored (multiple bit) operand, performs the appropriate bit-wise reduction on all bits of the operand, and returns a single bit result. For example, the four bits of A are **AND**ed together to produce Y1.

- **module** Reduction (A, Y1, Y2, Y3, Y4, Y5, Y6);

- **input** [3:0] A;

- **output** Y1, Y2, Y3, Y4, Y5, Y6;

- **reg** Y1, Y2, Y3, Y4, Y5, Y6;

- **always** @(A)

- **begin**

- Y1=&A; //reduction AND

- Y2=|A; //reduction OR

- Y3=~&A; //reduction NAND

- Y4=~|A; //reduction NOR

- Y5 ^=A; //reduction XOR

- Y6=~^A; //reduction XNOR

- **end**

- **endmodule**

■ **Shift**

- Shift operators require two operands. The operand before the operator contains data to be shifted and the operand after the operator contains the number of single bit shift operations to be performed. **0** is being used to fill the blank positions.

- **module** Shift (A, Y1, Y2);

-

- **input** [7:0] A;

- **output** [7:0] Y1, Y2;

- **parameter** B=3; reg [7:0] Y1, Y2;

-

- **always** @(A)

- **begin**

- Y1=A<<B; //logical shift left

- Y2=A>>B; //logical shift right

- **end**

- **endmodule**

■ Concatenation and Replication

- The concatenation operator "{ , }" combines (concatenates) the bits of two or more data objects. The objects may be scalar (single bit) or vectored (multiple bit). Multiple concatenations may be performed with a constant prefix and is known as replication.

- **module** Concatenation (A, B, Y);
 - **input** [2:0] A, B;
 - **output** [14:0] Y;
 - **parameter** C=3'b011;
 - **reg** [14:0] Y;
 - **always** @(A or B)
 - **begin**
 - Y={A, B, (2{C}), 3'b110};
 - **end**
- **endmodule**
-

■ **Conditional**

- An expression using conditional operator evaluates the logical expression before the "?".
 - If the expression is true then the expression before the colon (:) is evaluated and assigned to the output.
 - If the logical expression is false then the expression after the colon is evaluated and assigned to the output.
-

4-to-1 Multiplexer

- Gate-level modeling of a 4-to-1 multiplexer, Example. The logic diagram for the multiplexer is given in Figure 3.4 and the gate-level Verilog description is shown in Example.
- We describe the multiplexer, using dataflow statements.
- We show two methods to model the multiplexer by using dataflow statements.
- **Method 1: logic equation**
- We can use assignment statements instead of gates to model the logic equations of the multiplexer.
- Notice that everything is same as the gate-level Verilog description except that computation of out is done by specifying one logic equation by using operators instead of individual gate instantiations.

- I/O ports remain the same.
- This important so that the interface with the environment does not change. Only the internals of the module change.
- **Example 4-to-1 Multiplexer, Using Logic Equations**
- **// Module 4-to-1 multiplexer using data flow. logic equation**
- `module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);`
- `output out;`
- `input i0, i1, i2, i3;`
- `input s1, s0;`
- **//Logic equation for out**
- `assign out = (~s1 & ~s0 & i0)|(~s1 & s0 & i1) |(s1 & ~s0 & i2) |(s1 & s0 & i3) ;`
- `endmodule`

■ **Method 2: Conditional Operator**

- There is a more concise way to specify the 4-to-1 multiplexers.
- Example of 4-to-1 Multiplexer, Using Conditional Operators
- `// Module 4-to-1 multiplexer using data flow. Conditional operator.`
- `module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);`
- `// Port declarations from the I/O diagram`
- `output out;`
- `input i0, i1, i2, i3`
- `input s1, s0;`
- `assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0) ;`
- `endmodule`

4 bit Full Adder

- **Method 1: dataflow operators**
- **Example 4-bit Full Adder, Using Dataflow Operators**
- *// Define a 4-bit full adder by using dataflow statements.*
- `module fulladd4(sum, c_out, a, b, c_in);`
- *// I/O port declarations*
- `output [3:0] sum;`
- `output c_out;`
- `input[3:0] a, b;`
- `input c_in;`
- *// Specify the function of a full adder*
- `assign {c_out, sum} = a + b + c_in;`
- `endmodule`

Example 4-bit Full Adder with Carry Lookahead

- module fulladd4(sum, c_out, a, b, c_in);
- // Inputs and outputs
- output [3:0] sum;
- output c_out;
- input [3:0] a,b;
- input c_in;
- // Internal wires
- wire p0,g0, p1,g1, p2,g2, p3,g3;
- wire
- compute the p for each stage
- assign p0 = a[0] ^ b[0],
- p1 = a[1] ^ b[1],
- p2 = a[2] ^ b[2],
- p3 = a[3] ^ b[3]; c4, c3, c2, c1;

- compute the g for each stage
- assign $g_0 = a[0] \& b[0]$,
- $g_1 = a[1] \& b[1]$,
- $g_2 = a[2] \& b[2]$,
- $g_3 = a[3] \& b[3]$;
- // compute the carry for each stage
- // Note that c_in
- carry lookahead computation
- assign $c_1 = g_0 \mid (p_0 \& c_{in})$,
- $c_2 = g_1 \mid (p_1 \& g_0) \mid (p_1 \& p_0 \& c_{in})$,
- $c_3 = g_2 \mid (p_2 \& g_1) \mid (p_2 \& p_1 \& g_0) \mid (p_2 \& p_1 \& p_0 \& c_{in})$,
- $c_4 = g_3 \mid (p_3 \& g_2) \mid (p_3 \& p_2 \& g_1) \mid (p_3 \& p_2 \& p_1 \& g_0) \mid$
 $(p_3 \& p_2 \& p_1 \& p_0 \& c_{in})$;

- `// Compute Sum`
- `assign sum[0] = p0 ^ c_in,`
- `sum[1] = p1 ^ c1,`
- `sum[2] = p2 ^ c2,`
- `sum[3] = p3 ^ c3;`
- `// Assign carry output`
- `assign c_out = c4;`
- `Endmodule`

Ripple Counter

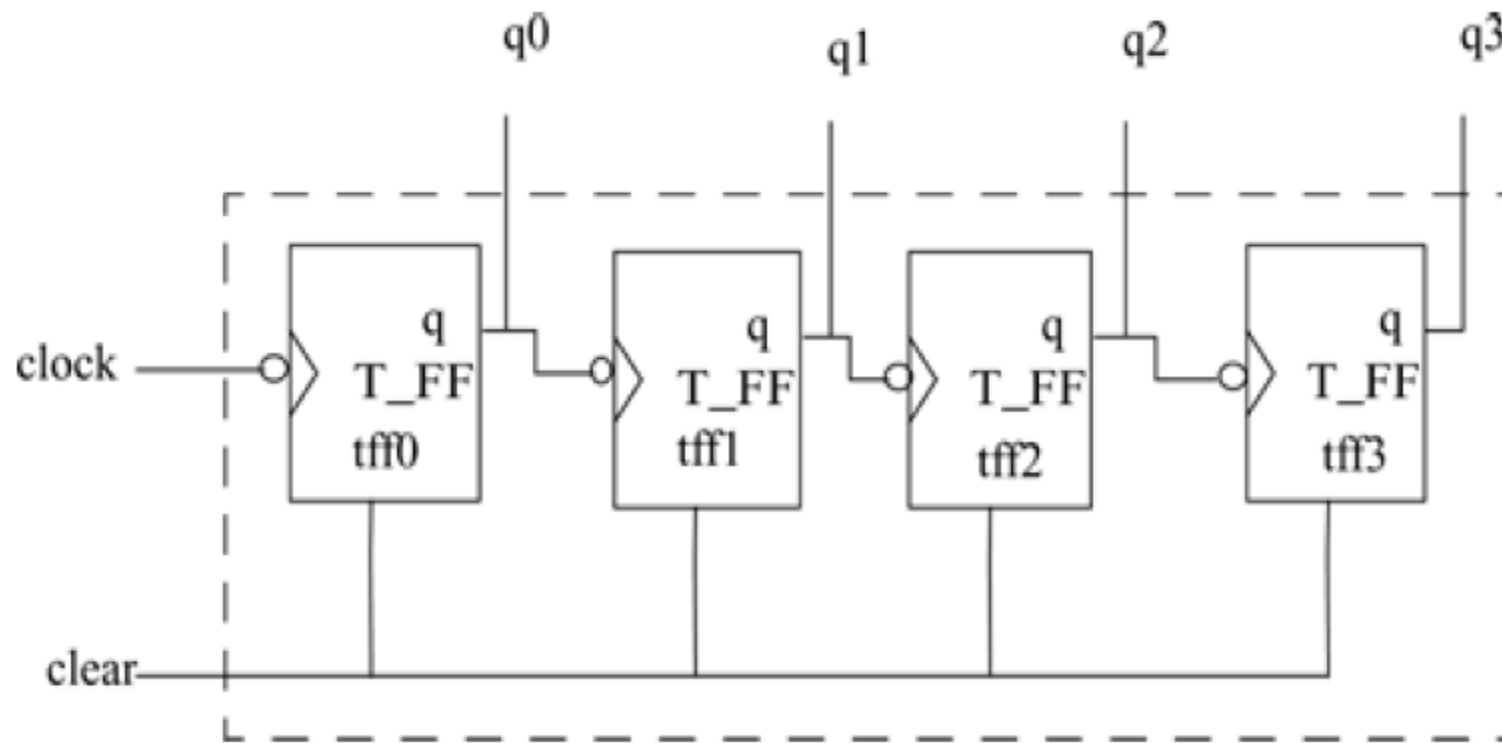


Figure 3.11 4 bit ripple counter

Example: Verilog Code for Ripple Counter

- `module counter(Q , clock, clear);`
- `// I/O ports`
- `output [3:0] Q;`
- `input clock, clear;`
- `// Instantiate the T flipflops`
- `T_FF tff0(Q[0], clock, clear);`
- `T_FF tff1(Q[1], Q[0], clear);`
- `T_FF tff2(Q[2], Q[1], clear);`
- `T_FF tff3(Q[3], Q[2], clear);`
- `endmodule`

Stimulus Module for Ripple Counter

- // Top level stimulus module
- module stimulus;
- // Declare variables for stimulating input
- reg CLOCK, CLEAR;
- wire [3:0] Q;
- initial
- \$monitor(\$time, " Count Q = %b Clear= %b", Q[3:0],CLEAR);
- // Instantiate the design block counter
- counter c1(Q, CLOCK, CLEAR)

-
- `// Stimulate the Clear Signal`
 - `initial`
 - `begin`
 - `CLEAR = 1'b1;`
 - `#34 CLEAR = 1'b0;`
 - `#200 CLEAR = 1'b1;`
 - `#50 CLEAR = 1'b0;`
 - `End`
 - `// Set up the clock to toggle every 10 time units`
 - `initial`
 - `begin`
 - `CLOCK = 1'b0;`
 - `forever #10 CLOCK = ~CLOCK;`
 - `end`
-

-
- // Finish the simulation at time 400
 - initial
 - begin
 - #400 \$finish;
 - end
 - endmodule
-

The output of the simulation is shown below. Note that the clear signal resets the count to zero.

0 Count Q = 0000 Clear= 1

34 Count Q = 0000 Clear= 0

40 Count Q = 0001 Clear= 0

60 Count Q = 0010 Clear= 0

80 Count Q = 0011 Clear= 0

100 Count Q = 0100 Clear= 0

120 Count Q = 0101 Clear= 0

140 Count Q = 0110 Clear= 0

160 Count Q = 0111 Clear= 0

180 Count Q = 1000 Clear= 0

200 Count Q = 1001 Clear= 0

220 Count Q = 1010 Clear= 0

234 Count Q = 0000 Clear= 1

284 Count Q = 0000 Clear= 0

300 Count Q = 0001 Clear= 0

320 Count Q = 0010 Clear= 0

340 Count Q = 0011 Clear= 0

360 Count Q = 0100 Clear= 0

380 Count Q = 0101 Clear= 0

T-Flipflop

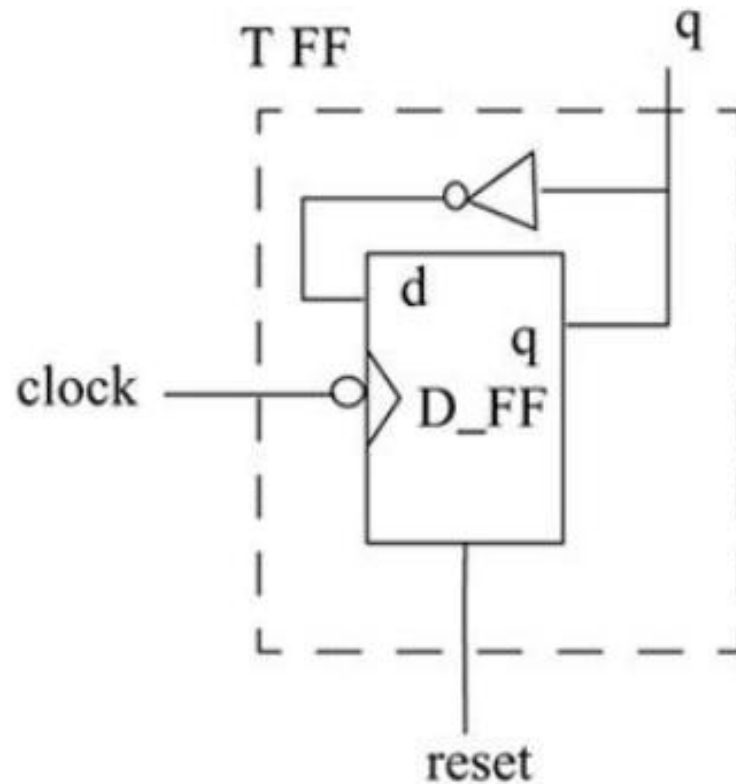


Figure 3.12 T-flipflop is built with one D-flipflop and an inverter gate

Example : Verilog Code for T-flipflop

- // Edge-triggered T-flipflop. Toggles every clock
- // cycle.
- module T_FF(q, clk, clear);
- // I/O ports
- output q;
- input clk, clear;
- // Instantiate the edge-triggered DFF
- // Complement of output q is fed back.
- // Notice qbar not needed. Unconnected port.
- edge_dff ff1(q, ,~q, clk, clear);
- endmodule

D-flipflop

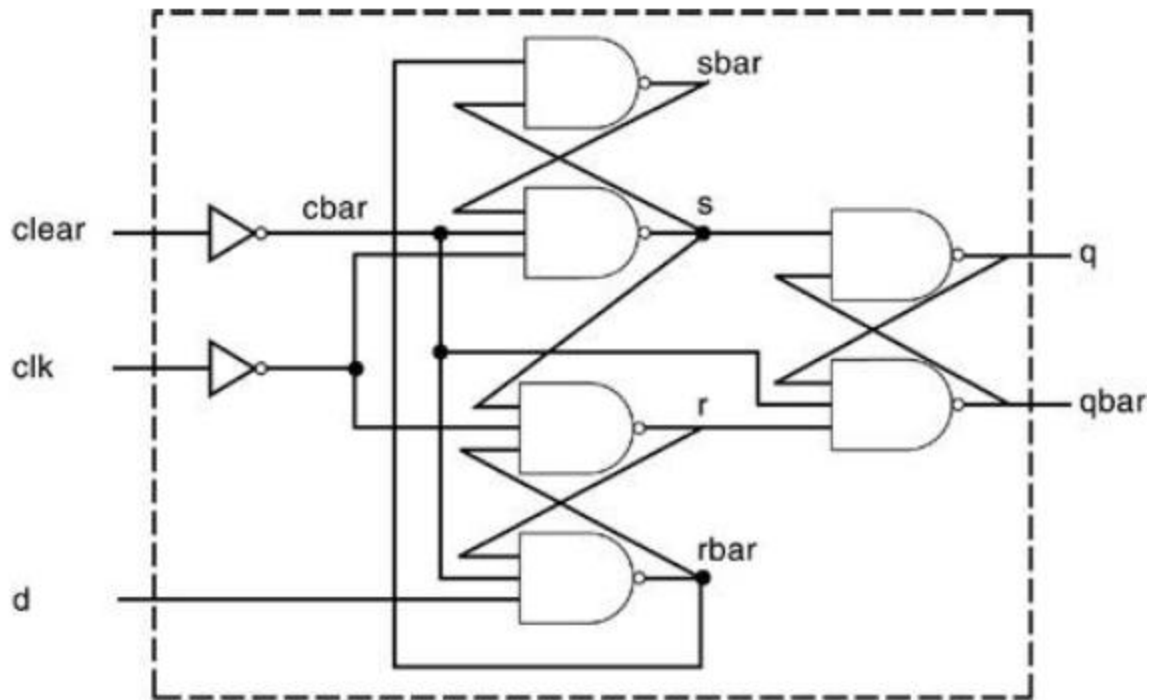


Figure 3.13 Negative Edge-Triggered D-flipflop with Clear

Verilog Code for Edge-Triggered D-flipflop

- // Edge-triggered D flipflop
- module edge_dff(q, qbar, d, clk, clear);
- // Inputs and outputs
- output q,qbar;
- input d, clk, clear;
- // Internal variables
- wire s, sbar, r, rbar,cbar;

-
- `// dataflow statements`
 - `// Create a complement of signal clear`
 - `assign cbar = ~clear;`
 - `// Input latches; A latch is level sensitive. An edge-sensitive`
 - `// flip-flop is implemented by using 3 SR latches.`
 - `assign sbar = ~(rbar & s),`
 - `s = ~(sbar & cbar & ~clk),`
 - `r = ~(rbar & ~clk & s),`
 - `rbar = ~(r & cbar & d);`
 - `// Output latch`
 - `assign q = ~(s & qbar),`
 - `qbar = ~(q & r & cbar);`
 - `endmodule`
-

Module Outcomes

- After completion of the module the students are able to:
- Identify logic gate primitives provided in Verilog and Understand instantiation of gates, gate symbols, and truth tables for and/or and buf/not type gates.
- Understand how to construct a Verilog description from the logic diagram of the circuit.
- Describe rise, fall, and turn-off delays in the gate-level design and Explain min, max, and type delays in the gate-level design
- Describe the continuous assignment (assign) statement, restrictions on the assign statement, and the implicit continuous assignment statement.
- Explain assignment delay, implicit assignment delay, and net declaration delay for continuous assignment statements and Define expressions, operators, and operands.
- Use dataflow constructs to model practical digital circuits in Verilog

Recommended questions

- 1. Write the truth table of all the basic gates. Input values consisting of '0', '1', 'x', 'z'.
- 2. What are the primitive gates supported by Verilog HDL? Write the Verilog HDL statements to instantiate all the primitive gates.
- 3. Use gate level description of Verilog HDL to design 4 to 1 multiplexer. Write truth table, top-level block, logic expression and logic diagram. Also write the stimulus block for the same.
- 4. Explain the different types of buffers and not gates with the help of truth table, logic symbol, logic expression
- 5. Use gate level description of Verilog HDL to describe the 4-bit ripple carry counter. Also write a stimulus block for 4-bit ripple carry adder.

- 6. How to model the delays of a logic gate using Verilog HDL? Give examples. Also explain the different delays associated with digital circuits.
- 7. Write gate level description to implement function $y = a.b + c$, with 5 and 4 time units of gate delay for AND and OR gate respectively. Also write the stimulus block and simulation waveform.
- 8. With syntax describe the continuous assignment statement.
- 9. Show how different delays associated with logic circuit are modelled using dataflow description.
- 10. Explain different operators supported by Verilog HDL.
- 11. What is an expression associated with dataflow description? What are the different types of operands in an expression?

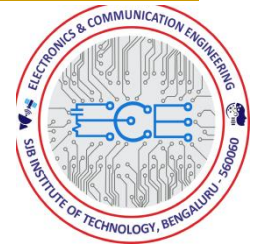
-
- 12. Discuss the precedence of operators.
 - 13. Use dataflow description style of Verilog HDL to design 4:1 multiplexer with and without using conditional operator.
 - 14. Use dataflow description style of Verilog HDL to design 4-bit adder using
 - i. Ripple carry logic.
 - ii. Carry look ahead logic.
 - 15. Use dataflow description style, gate level description of Verilog HDL to design 4-bit ripple carry counter. Also write the stimulus block to verify the same.
-

Reference / Text Book Details

Sl.No	Title of Book	Author	Publication	Edition
1	Verilog HDL: A Guide to Digital Design and Synthesis	Samir Palnitkar	Pearson Education	2 nd
2	VHDL for Programmable Logic	Kevin Skahill	PHI/Pearson education	2 nd
3	The Verilog Hardware Description Language	Donald E. Thomas, Philip R. Moorby	Springer Science+Business Media, LLC	5 th
4	Advanced Digital Design with the Verilog HDL	Michael D. Ciletti	Pearson (Prentice Hall)	2 nd
5	Design through Verilog HDL	Padmanabhan, Tripura Sundari	Wiley	Latest

Thank You





|| JAI SRI GURUDEV ||

Sri AdichunchanagiriShikshana Trust (R)

SJB INSTITUTE OF TECHNOLOGY

BGS Health & Education City, Kengeri , Bangalore – 60 .

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

Verilog HDL [18EC56]

Module 4: Behavioural Modelling and Tasks Functions

By:

Mrs. LATHA S
Assistant Professor,
Dept. of ECE, SJBIT

Content

- Structured procedures
 - Initial and always,
 - Blocking and non-blocking statements
 - Delay control, generate statement
 - Event control, conditional statements
 - Multiway branching, loops
 - sequential and parallel blocks
 - Tasks and Functions: Differences between tasks and functions, declaration, invocation, automatic tasks and functions.
-

Learning Objectives

- To Explain the significance of structured procedures always and initial in behavioral modeling.
- To Define blocking and nonblocking procedural assignments.
- To Understand delay-based timing control mechanism in behavioral modeling. Use regular delays, intra-assignment delays, and zero delays.
- To Describe event-based timing control mechanism in behavioral modeling. Use regular event control, named event control, and event OR control.

Learning Objectives

- To Use level-sensitive timing control mechanism in behavioral modeling.
- To Explain conditional statements using if and else.
- To Describe multiway branching, using case, casex, and casez statements.
- To Understand looping statements such as while, for, repeat, and forever.
- To Define sequential and parallel blocks

Structured Procedures

- There are two structured procedure statements in Verilog: **always** and **initial**.
 - These statements are the two most basic statements in behavioral modeling.
 - All other behavioral statements can appear only inside these structured procedure state
 - The statements **always** and **initial** cannot be nested. ments.
-

Initial Statement

- All statements inside an initial statement constitute an initial block.
- An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again. If there are multiple initial blocks, each block starts to execute concurrently at time 0.
- Each block finishes execution independently of other blocks.
- Multiple behavioral statements must be grouped, typically using the keywords begin and end.
- If there is only one behavioral statement, grouping is not necessary. This is similar to the begin-end blocks in Pascal programming language or the { } grouping in the C programming language

■ **Example 4.1:Initial Statement**

- module stimulus; reg x,y, a,b, m;
- initial
- m = 1'b0; //single statement; does not need to be grouped
- initial
- begin
- #5 a = 1'b1; //multiple statements; need to be grouped
- #25 b = 1'b0;
- end
- Initial
- begin
- #10 x = 1'b0;
- #25 y = 1'b1;
- end
- initial
- #50 \$finish;
- endmodule

- In the above example, the three initial statements start to execute in parallel at time 0.
- If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time. Thus, the execution sequence of the statements inside the initial blocks will be as follows.
- time statement executed
- 0 m = 1'b0;
- 5 a = 1'b1;
- 10 x = 1'b0;
- 30 b = 1'b0;
- 35 y = 1'b1;
- 50 \$finish;
- The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.

Combined Variable Declaration and Initialization

- Variables can be initialized when they are declared. Example 4-2 shows such a declaration.
- **Example 4-2 Initial Value Assignment**
- //The clock variable is defined first **reg clock;**
- //The value of clock is set to 0 **initial clock = 0;**
- //Instead of the above method, clock variable
- //can be initialized at the time of declaration
- //This is allowed only for variables declared
- //at module level. **reg clock = 0;**

Combined Port/Data Declaration and Initialization

- The combined port/data declaration can also be combined with an initialization. Example 4-3 shows such a declaration.
- **Example 4-3 Combined Port/Data Declaration and Variable Initialization**
- `module adder (sum, co, a, b, ci);`
- `output reg [7:0] sum = 0; //Initialize 8 bit output sum`
`output reg co = 0; //Initialize 1 bit output co`
- `input [7:0] a, b; input ci;`
- `-----`
- `endmodule`

Combined ANSI C Style Port Declaration and Initialization

- Verilog-2001 introduced an abbreviated module **port declaration** enhancement, often referred to as “**ANSI-C**” style **port declarations**, where each module **port** could be declared just once and include the **port** position, **port** direction and **port** data type all in a single **declaration**
- ANSI C style port declaration can also be combined with an initialization. Example 4-4 shows such a declaration.
- **Example 4-4 Combined ANSI C Port Declaration and Variable Initialization**
- module adder (output reg [7:0] sum = 0, //Initialize 8 bit output output reg co = 0, //Initialize 1 bit output co
- input [7:0] a, b, input ci
-);
- --
- endmodule

Always Statement

- All behavioral statements inside an always statement constitute an always block.
 - The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion.
 - This statement is used to model a block of activity that is repeated continuously in a digital circuit.
 - An example is a clock generator module that toggles the clock signal every half cycle.
-

- In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on. Example 4-5 illustrates one method to model a clock generator in Verilog.

- **Example 4-5 always Statement**

- `module clock_gen (output reg clock);`
- `//Initialize clock at time zero initial`
- `clock = 1'b0;`
- `//Toggle clock every half-cycle (time period = 10)`
- `Always`
- `#10 clock = ~clock;`
- `initial`
- `#1000 $finish;`
- `endmodule`

- In Example 4-5, the always statement starts at time 0 and executes the statement $\text{clock} = \sim\text{clock}$ every 10 time units.
- Notice that the initialization of clock has to be done inside a separate initial statement.
- If we put the initialization of clock inside the always block, clock will be initialized every time the always is entered.
- Also, the simulation must be halted inside an initial statement. If there is no \$stop or \$finish statement to halt the simulation, the clock generator will run forever.

Procedural Assignments

- Procedural assignments update values of reg, integer, real, or time variables.
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.
- These are unlike continuous assignments, Dataflow Modeling, where one assignment statement can cause the value of the right-hand-side expression to be continuously expression sly placed onto the left-hand-side net.
- The syntax for the simplest form of procedural assignment is shown below.
- `assignment ::= variable_lvalue = [delay_or_event_control] expression`

- The left-hand side of a procedural assignment <lvalue> can be one of the following:
- A reg, integer, real, or time register variable or a memory element
- A bit select of these variables (e.g., addr[0])
- A part select of these variables (e.g., addr[31:16])
- A concatenation of any of the above
- The right-hand side can be any expression that evaluates to a value. In behavioral modeling, all operators can be used in behavioral expressions.
- There are two types of procedural assignment statements: **blocking and nonblocking.**

Blocking Assignments

- Blocking assignment statements are executed in the order they are specified in a sequential block.
- A blocking assignment will not block execution of statements that follow in a parallel block.
- The = operator is used to specify blocking assignments.
- A **blocking assignment** gets its name because a **blocking assignment** must evaluate the RHS arguments and complete the **assignment** without interruption from any other **Verilog statement**.
- The **assignment** is said to "**block**" other **assignments** until the current **assignment** has completed.

■ **Example 4-6 Blocking Statements**

- `reg x, y, z;`
- `reg [15:0] reg_a, reg_b; integer count;`
- `//All behavioral statements must be inside an initial or always block initial`
- `begin`
- `x = 0; y = 1; z = 1; //Scalar assignments count = 0;`
`//Assignment to integer variables`
- `reg_a = 16'b0; reg_b = reg_a; //initialize vectors`
- `#15 reg_a[2] = 1'b1; //Bit select assignment with delay`
- `#10 reg_b[15:13] = {x, y, z} //Assign result of concatenation to part select of a vector`
- `count = count + 1; //Assignment to an integer (increment)`
- `end`

- In Example 4-6, the statement $y = 1$ is executed only after $x = 0$ is executed. The behavior in a particular block is sequential in a begin-end block if blocking statements are used, because the statements can execute only in sequence.
- The statement $\text{count} = \text{count} + 1$ is executed last. The simulation times at which the statements are executed are as follows:
- All statements $x = 0$ through $\text{reg_b} = \text{reg_a}$ are executed at time 0
- Statement $\text{reg_a}[2] = 0$ at time = 15
- Statement $\text{reg_b}[15:13] = \{x, y, z\}$ at time = 25
- Statement $\text{count} = \text{count} + 1$ at time = 25
- Since there is a delay of 15 and 10 in the preceding statements, ~~$\text{count} = \text{count} + 1$ will be executed at time =~~ 25 units

Nonblocking Assignments

- Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block.
- A `<=` operator is used to specify nonblocking assignments. Note that this operator has the same symbol as a relational operator, `less_than_equal_to`.
- The operator `<=` is interpreted as a relational operator in an expression and as an assignment operator in the context of a nonblocking assignment.
- To illustrate the behavior of nonblocking statements and its difference from blocking statements, let us consider Example 4-7, where we convert some blocking assignments to nonblocking assignments, and observe the behavior.

■ Example 4-7 Nonblocking Assignments

- reg x, y, z;
- reg [15:0] reg_a, reg_b; integer count; **//All behavioral statements must be inside an initial or always block**
- initial
- begin
- x = 0; y = 1; z = 1; //Scalar assignments
- count = 0; //Assignment to integer variables
- reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
- reg_a[2] <= #15 1'b1; //Bit select assignment with delay
- reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation //to part select of a vector
- count <= count + 1; //Assignment to an integer (increment)
- end

- In this example, the statements $x = 0$ through $\text{reg_b} = \text{reg_a}$ are executed sequentially at time 0.
- Then the three nonblocking assignments are processed at the same simulation time.
- $\text{reg_a}[2] = 0$ is scheduled to execute after 15 units (i.e., time = 15)
- $\text{reg_b}[15:13] = \{x, y, z\}$ is scheduled to execute after 10 time units (i.e., time = 10)
- $\text{count} = \text{count} + 1$ is scheduled to be executed without any delay (i.e., time = 0)
- Thus, the simulator schedules a non blocking assignment statement to execute and continues to the next statement in the block without waiting for the non blocking statement to complete execution.

- Typically, nonblocking assignment statements are executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed.
 - In the example above, we mixed blocking and non blocking assignments to illustrate their behaviour.
 - However, it is recommended that blocking and non blocking assignments not be mixed in the same always block.
-

- module block_nonblock();
- reg a, b, c, d , e, f ;
- // Blocking assignments
- initial begin
- a = #10 1'b1;// The simulator assigns 1 to a at time 10
- b = #20 1'b0;// The simulator assigns 0 to b at time 30
- c = #40 1'b1;// The simulator assigns 1 to c at time 70
- end
- // Nonblocking assignments
- initial
- begin 13
- d <= #10 1'b1;// The simulator assigns 1 to d at time 10
- e <= #20 1'b0;// The simulator assigns 0 to e at time 20
- f <= #40 1'b1;// The simulator assigns 1 to f at time 40
- end
- endmodule

Application of non blocking assignments

- They are used as a method to model several concurrent data transfers that take place after a common event.
- Consider the following example where three concurrent data transfers take place at the positive edge of clock.
- `always @(posedge clock) begin`
- `reg1 <= #1 in1;`
- `reg2 <= @(negedge clock) in2 ^ in3;`
- `reg3 <= #1 reg1; //The old value of reg1`
- `end`

- At each positive edge of clock, the following sequence takes place for the non blocking assignments.
- A read operation is performed on each right-hand-side variable, in1, in2, in3, and reg1, at the positive edge of clock.
- The right-hand-side expressions are evaluated, and the results are stored internally in the simulator.
- The write operations to the left-hand-side variables are scheduled to be executed at the time specified by the intra-assignment delay in each assignment, that is, schedule "write" to reg1 after 1 time unit, to reg2 at the next negative edge of clock, and to reg3 after 1 time unit.

- The write operations are executed at the scheduled time steps.
 - The order in which the write operations are executed is not important because the internally stored right-hand-side expression values are used to assign to the left-hand-side values.
 - For example, note that reg3 is assigned the old value of reg1 that was stored after the read operation, even if the write operation wrote a new value to reg1 before the write operation to reg3 was executed.
 - Thus, the final values of reg1, reg2, and reg3 are not dependent on the order in which the assignments are processed.
-

Nonblocking Statements to Eliminate Race Conditions

- **//Illustration 1: Two concurrent always blocks with blocking statements//**
- `always @(posedge clock) a = b;`
- `always @(posedge clock) b = a;`
- **//Illustration 2: Two concurrent always blocks with nonblocking statements//**
- `always @(posedge clock) a <= b;`
- `always @(posedge clock) b <= a;`
- In Example 4-8, in Illustration 1, there is a race condition when blocking statements are used. Either `a = b` would be executed before `b = a`, or vice versa, depending on the simulator implementation..

- Thus, values of registers a and b will not be swapped. Instead, both registers will get the same value (previous value of a or b), based on the Verilog simulator implementation
- However, nonblocking statements used in Illustration 2 eliminate the race condition. At the positive edge of clock, the values of all right-hand-side variables are "read," and the right-hand-side expressions are evaluated and stored in temporary variables.
- During the write operation, the values stored in the temporary variables are assigned to the left-hand-side variables. Separating the read and write operations ensures that the values of registers a and b are swapped correctly, regardless of the order in which the write operations are performed.

Example 4-9 Implementing Nonblocking Assignments using Blocking Assignments

- Example 4-9 shows how non blocking assignments shown in Illustration 2 could be emulated using blocking assignments.
- //Emulate the behavior of nonblocking assignments by
- //using temporary variables and blocking assignments always @(posedge clock)
- begin
- //Read operation
- //store values of right-hand-side expressions in temporary variables
- temp_a = a;
- temp_b = b;

-
- `//Write operation`
 - `//Assign values of temporary variables to left-hand-side variables`
 - `a = temp_b;`
 - `b = temp_a;`
 - `End`
 - For digital design, use of nonblocking assignments in place of blocking assignments is highly recommended in places where concurrent data transfers take place after a common event.
-

- In such cases, blocking assignments can potentially cause race conditions because the final result depends on the order in which the assignments are evaluated.
- Nonblocking assignments can be used effectively to model concurrent data transfers because the final result is not dependent on the order in which the assignments are evaluated.
- Typical applications of nonblocking assignments include pipeline modeling and modeling of several mutually exclusive data transfers.
- On the downside, nonblocking assignments can potentially cause degradation in the simulator performance and increase in memory usage.

Timing Controls

- Various behavioral timing control constructs are available in Verilog.
- In Verilog, if there are no timing control statements, the simulation time does not advance.
- Timing controls provide a way to specify the simulation time at which procedural statements will execute.
- There are three methods of timing control: **delay-based timing control, event-based timing control, and level-sensitive timing control.**

Delay-Based Timing Control

- Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed.
- Delays are specified by the symbol #.
- Syntax for the delay-based timing control statement is shown below.
- `delay3 ::= # delay_value | # (delay_value [, delay_value [, delay_value]])`
- `delay2 ::= # delay_value | # (delay_value [, delay_value])`
`delay_value ::= unsigned_number |`
`parameter_identifier | specparam_identifier |`
`mintypmax_expression`

-
- Delay-based timing control can be specified by a number, identifier, or a mintypmax_expression.
 - There are three types of delay control for procedural assignments: regular delay control, intra-assignment delay control, and zero delay control.
 - **Regular delay control**
 - Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment. Usage of regular delay control is shown in Example 4-10.
-

Example 4-10 Regular Delay Control

- //define parameters
- parameter latency = 20;
- parameter delta = 2;
- //define register variables
- reg x, y, z, p, q;
- initial
- begin
- x = 0; // no delay control // delay control with a number.
Delay execution of // y = 1 by 10units
- #10 y = 1;
- ~~#latency z = 0; // Delay control with identifier. Delay of 20 units~~

- `#(latency + delta) p = 1; // Delay control with expression`
- `#y x = x + 1; // Delay control with identifier. Take value of y.`
- `#(4:5:6) q = 0; // Minimum, typical and maximum delay values.`
- `end`
- In Example 4-10, the execution of a procedural assignment is delayed by the number specified by the delay control.
- For begin-end groups, delay is always relative to time when the statement is encountered.
- Thus, `y = 1` is executed 10 units after it is encountered in the activity flow.

Intra-assignment delay control

- Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator. Such delay specification alters the flow of activity in a different manner.
- Example 4-11 shows the contrast between intra-assignment delays and regular delays
- `//define register variables`
- `reg x, y, z;`
- `//intra assignment delays`
- `initial`
- `begin`
- `x = 0; z = 0;`
- `y = #5 x + z; //Take value of x and z at the time=0, evaluate //x + z and then wait 5 time units to assign value to y.`
- `end`

- //Equivalent method with temporary variables and regular delay control initial
- begin
- $x = 0; z = 0;$
- $\text{temp_xz} = x + z;$
- #5 $y = \text{temp_xz};$ //Take value of $x + z$ at the current time and
- //store it in a temporary variable. Even though x and z might change between 0 and 5, //the value assigned to y at time 5 is unaffected.
- end
- Regular delays defer the execution of the entire assignment.
- Intra-assignment delays compute the righthand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable. Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.

Zero delay control

- Procedural statements in different always-initial blocks may be evaluated at the same simulation time.
- The order of execution of these statements in different always-initial blocks is nondeterministic.
- Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed.
- This is used to eliminate race conditions.
- However, if there are multiple zero delay statements, the order between them is nondeterministic.
- Example 4-12 illustrates zero delay control.

Example 4-12:- Zero Delay Control

- initial
- begin
- $x = 0;$
- $y = 0;$
- end
- initial
- begin
- $\#0 \ x = 1;$ *//zero delay control*
- $\#0 \ y = 1;$
- end

- In Example 4-12, four statements? $x = 0$, $y = 0$, $x = 1$, $y = 1$ are to be executed at simulation time 0.
- However, since $x = 1$ and $y = 1$ have #0, they will be executed last.
- Thus, at the end of time 0, x will have value 1 and y will have value 1.
- The order in which $x = 1$ and $y = 1$ are executed is not deterministic.
- The above example was used as an illustration. However, using #0 is not a recommended practice.

Event-Based Timing Control

- An event is the change in the value on a register or a net. Events can be utilized to trigger execution of a statement or a block of statements. There are four types of event-based timing control: regular event control, named event control, event OR control, and level sensitive timing control.
- **Regular event control**
- The @ symbol is used to specify an event control. Statements can be executed on changes in signal value or at a positive or negative transition of the signal value. The keyword posedge is used for a positive transition, as shown in Example 4-13.

■ **Example 4-13 Regular Event Control**

- `@(clock) q = d; //q = d` is executed whenever signal clock changes value
- `@(posedge clock) q = d; //q = d` is executed whenever signal clock does
- `//a positive transition (0 to 1,x or z, x to 1, z to 1)`
- `@(negedge clock) q = d; //q = d` is executed whenever signal clock does
- `//a negative transition (1 to 0,x or z, x to 0, z to 0)`
- `q = @(posedge clock) d; //d` is evaluated immediately and assigned to q at the positive edge of clock

Named event control

- Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event (see Example 4-14).
 - The event does not hold any data. A named event is declared by the keyword `event`. An event is triggered by the symbol `->`.
 - The triggering of the event is recognized by the symbol `@`.
-

■ **Example 4-14 Named Event Control**

- `//This is an example of a data buffer storing data after the last packet of data has arrived.`
 - `event received_data; //Define an event called received_data`
 - `always @(posedge clock) //check at each positive clock edge`
 - `begin`
 - `if(last_data_packet) //If this is the last data packet`
 - `->received_data; //trigger the event received_data end`
 - `always @(received_data) //Await triggering of event received_data`
 - `//When event is triggered, store all four`
 - `//packets of received data in data buffer`
 - `//use concatenation operator { }`
 - `data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};`
-

Event OR Control

- Sometimes a transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements.
- This is expressed as an OR of events or signals. The list of events or signals expressed as an OR is also known as a sensitivity list.
- The keyword or is used to specify multiple triggers, as shown in Example 4-15.

- **Example 4-15 Event OR Control (Sensitivity List)**
- `//A level-sensitive latch with asynchronous reset`
- `always @(reset or clock or d)`
- `//Wait for reset or clock or d to change`
- `begin`
- `if (reset) //if reset signal is high, set q to 0.`
- `q = 1'b0;`
- `else if(clock) //if clock is high, latch input`
- `q = d;`
- `end`
- Sensitivity lists can also be specified using the `","` (comma) operator instead of the or operator.

-
- **Example 4-16 Sensitivity List with Comma Operator**
 - `//A level-sensitive latch with asynchronous reset`
 - `always @(reset, clock, d)`
 - `//Wait for reset or clock or d to change`
 - `begin`
 - `if (reset) //if reset signal is high, set q to 0.`
 - `q = 1'b0;`
 - `else if(clock) //if clock is high, latch input`
 - `q = d;`
 - `End`
-

- When the number of input variables to a combination logic block are very large, sensitivity lists can become very cumbersome to write.
- Moreover, if an input variable is missed from the sensitivity list, the block will not behave like a combinational logic block.
- To solve this problem, Verilog HDL contains two special symbols: `@*` and `@(*)`.
- Both symbols exhibit identical behavior. These special symbols are sensitive to a change on any signal that may be read by the statement group that follows this symbol

- **Example 4-17 Use of @* Operator**

- //Combination logic block using the or operator
 - //Cumbersome to write and it is easy to miss one input to the block
 - always @(a or b or c or d or e or f or g or h or p or m)
 - begin
 - out1 = a ? b+c : d+e;
 - out2 = f ? g+h : p+m;
 - end
-

- //Instead of the above method, use @(*) symbol
- //Alternately, the @* symbol can be used
- //All input variables are automatically included in the
- //sensitivity list.
- always @(*)
- begin
- out1 = a ? b+c : d+e;
- out2 = f ? g+h : p+m;
- end

Level-Sensitive Timing Control

- Event control discussed earlier waited for the change of a signal value or the triggering of an event.
- The symbol @ provided edge-sensitive control.
- Verilog also allows level sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed.
- The keyword wait is used for level sensitive constructs.
- **always**
- **wait (count_enable) #20 count = count + 1;**
- In the above example, the value of count_enable is monitored continuously. If count_enable is 0, the statement is not entered. If it is logical 1, the statement count = count + 1 is executed after 20 time units. If count_enable stays at 1, count will be incremented every 20 time units.

Conditional Statements

- Conditional statements are used for making decisions based upon certain conditions.
- These conditions are used to decide whether or not a statement should be executed.
- Keywords `if` and `else` are used for conditional statements.
- There are three types of conditional statements. Usage of conditional statements is shown below.
- **//Type 1 conditional statement. No else statement.**
- `//Statement executes or does not execute.`
- `if (<expression>) true_statement ;`

- //Type 2 conditional statement. One else statement
- //Either true_statement or false_statement is evaluated
- `if (<expression>) true_statement ; else false_statement ;`
- //Type 3 conditional statement. Nested if-else-if.
- //Choice of multiple statements. Only one is executed.
- `if (<expression1>) true_statement1 ;`
- `else if (<expression2>) true_statement2 ;`
- `else if (<expression3>) true_statement3 ;`
- `else default_statement ;`
- The <expression> is evaluated. If it is true (1 or a non-zero value), the true_statement is executed. However, if it is false (zero) or ambiguous (x), the false_statement is executed. The <expression> can contain any operators. Each true_statement or false_statement can be a single statement or a block of multiple statements. A block must be grouped, typically by using keywords begin and end. A single statement need not be grouped.

■ **Example 4-18 Conditional Statement Examples**

■ **//Type 1 statements**

■ `if(!lock) buffer = data; if(enable) out = in;`

■ **//Type 2 statements**

■ `if (number_queued < MAX_Q_DEPTH) begin`

■ `data_queue = data;`

■ `number_queued = number_queued + 1; end`

■ `else`

■ `$display("Queue Full. Try again");`

- **//Type 3 statements**
- **//Execute statements based on ALU control signal. if (alu_control == 0)**
- **y = x + z;**
- **else if(alu_control == 1) y = x - z;**
- **else if(alu_control == 2) y = x * z;**
- **else**
- **\$display("Invalid ALU control signal");**

Multiway Branching

- Conditional Statements, there were many alternatives, from which one was chosen. The nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the case statement.
- **Case Statement**
- The keywords case, endcase, and default are used in the case statement.. case (expression)
- alternative1: statement1; alternative2: statement2;
- alternative3: statement3;
- ...
- default: default_statement;
- endcase

- Each of statement1, statement2 , default_statement can be a single statement or a block of multiple statements.
 - A block of multiple statements must be grouped by keywords begin and end.
 - The expression is compared to the alternatives in the order they are written.
 - For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the default_statement is executed. The default_statement is optional.
 - Placing of multiple default statements in one case statement is not allowed. The case statements can be nested. The following Verilog code implements the type 3 conditional statement in Example 4-18.
-

-
- `//Execute statements based on the ALU control signal`
 - `reg [1:0] alu_control;`
 - `...`
 - `...`
 - `case (alu_control)`
 - `2'd0 : y = x + z;`
 - `2'd1 : y = x - z;`
 - `2'd2 : y = x * z;`
 - `default : $display("Invalid ALU control signal");`
 - `endcase`
-

■ **Example 4-19 4-to-1 Multiplexer with Case Statement**

- `module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);`
- `// Port declarations from the I/O diagram output out;`
- `input i0, i1, i2, i3;`
- `input s1, s0;`
- `reg out;`
- `always @(s1 or s0 or i0 or i1 or i2 or i3)`
- `case (s1, s0) //Switch based on concatenation of control signals`
- `2'd0 : out = i0;`
- `2'd1 : out = i1;`
- `2'd2 : out = i2;`
- `2'd3 : out = i3;`
- `default: $display("Invalid control signals"); endcase`
- `endmodule`

Casex, Casez Keywords

- There are two variations of the case statement. They are denoted by keywords, casex and casez.
- casez treats all z values in the case alternatives or the case expression as don't cares. All bit positions with z can also be represented by ? in that position.
- casex treats all x and z values in the case item or the case expression as don't cares.
- The use of casex and casez allows comparison of only non-x or -z positions in the case expression and the case alternatives. Example 4-21 illustrates the decoding of state bits in a finite state machine using a casex statement. The use of casez is similar. Only one bit is considered to determine the next state and the other bits are ignored.

- **Example 4-21 casex Use reg [3:0] encoding; integer state;**
- casex (encoding) //logic value x represents a don't care bit.
- 4'b1xxx : next_state = 3;
- 4'bx1xx : next_state = 2;
- 4'bxx1x : next_state = 1;
- 4'bxxx1 : next_state = 0;
- default : next_state = 0; endcase
- Thus, an input encoding = 4'b10xz would cause next_state = 3 to be executed.

Loops

- There are four types of looping statements in Verilog: while, for, repeat, and forever. The syntax of these loops is very similar to the syntax of loops in the C programming language. All looping statements can appear only inside an initial or always block. Loops may contain delay expressions.
- **While Loop**
- The keyword while is used to specify this loop. The while loop executes until the while expression is not true. If the loop is entered when the while-expression is not true, the loop is not executed at all. Each expression can contain the operators. Any logical expression can be specified with these operators. If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end. Example 4-22 illustrates the use of the while loop.

- `//Illustration 1: Increment count from 0 to 127. Exit at count 128.`
- `//Display the count variable.`
- `integer count;`
- `initial`
- `begin`
- `count = 0;`
- `while (count < 128) //Execute loop till count is 127.`
- `//exit at count 128`
- `begin`
- `$display("Count = %d", count);`
- `count = count + 1;`
- `end`
- `end`

- **//Illustration 2: Find the first bit with a value 1 in flag (vector variable)**
- 'define TRUE 1'b1';
- 'define FALSE 1'b0; reg [15:0] flag;
- integer i; //integer to keep count
- reg continue;
- initial
- begin
- flag = 16'b 0010_0000_0000_0000; i = 0;
- continue = 'TRUE; 148
- while((i < 16) && continue) **//Multiple conditions using operators.**
- begin
- if (flag[i]) begin
- \$display("Encountered a TRUE bit at element number %d", i); continue = 'FALSE;
- end
- i = i + 1;
- end
- end

For Loop

- The keyword `for` is used to specify this loop. The `for` loop contains three parts:
- An initial condition
- A check to see if the terminating condition is true
- A procedural assignment to change value of the control variable
- The counter described in Example 4-22 can be coded as a `for` loop (Example 4-23). The initialization condition and the incrementing procedural assignment are included in the `for` loop and do not need to be specified separately. Thus, the `for` loop provides a more compact loop structure than the `while` loop. Note, however, that the `while` loop is more general-purpose than the `for` loop. The `for` loop cannot be used in place of the `while` loop in all situations.

■ **Example 4-23 For Loop**

- integer count;
- initial
- for (count=0; count < 128; count = count + 1)
- \$display("Count = %d", count);
- for loops can also be used to initialize an array or memory, as shown below. //Initialize array elements 'define MAX_STATES 32
- integer state [0: 'MAX_STATES-1]; //Integer array state with elements 0:31
- integer i;
- initial begin
- for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0 state[i] = 0;
- for(i = 1; i < 32; i = i + 2) //initialize all odd locations with 1
- state[i] = 1;
- end

Repeat Loop

- The keyword **repeat** is used for this loop.
- The repeat construct executes the loop a fixed number of times. A repeat construct cannot be used to loop on a general logical expression.
- A while loop is used for that purpose.
- A repeat construct must contain a number, which can be a constant, a variable or a signal value.
- However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.
- The counter in Example 4-22 can be expressed with the repeat loop, as shown in Illustration 1.

■ **Example 4-24 Repeat Loop**

- //Illustration 1 : increment and display count from 0 to 127
 - integer count;
 - initial
 - begin count = 0;
 - repeat(128)
 - begin
 - \$display("Count = %d", count); count = count + 1;
 - end
 - end
-

- `//Illustration 2 : Data buffer module example //After it receives a data_start signal. //Reads data for next 8 cycles.`
- `module data_buffer(data_start, data, clock);`
- `parameter cycles = 8;`
- `input data_start;`
- `input [15:0] data; input clock;`
- `reg [15:0] buffer [0:7]; integer i;`
- `always @(posedge clock) begin`
- `if(data_start) //data start signal is true begin`
- `i = 0;`
- `repeat(cycles) //Store data at the posedge of next 8 clock //cycles`
- `begin`
- `@(posedge clock) buffer[i] = data; //waits till next // posedge to latch`
- `data i = i + 1;`
- `end`
- `end`
- `end`
- `endmodule`

Forever loop

- The keyword `forever` is used to express this loop.
- The loop does not contain any expression and executes forever until the `$finish` task is encountered.
- The loop is equivalent to a while loop with an expression that always evaluates to true, e.g., `while (1)`. A forever loop can be exited by use of the `disable` statement.
- A forever loop is typically used in conjunction with timing control constructs.
- If timing control constructs are not used, the Verilog simulator would execute this statement infinitely without advancing simulation time and the rest of the design would never be executed. Example 4-25 explains the use of the forever statement

■ **Example 4-25 Forever Loop**

- **//Example 1: Clock generation**
- **//Use forever loop instead of always block reg clock;**
- **initial**
- **begin**
- **clock = 1'b0;**
- **forever #10 clock = ~clock; //Clock with period of 20 units**
- **end**
- **//Example 2: Synchronize two register values at every positive edge of //clock**
- **reg clock; reg x, y;**
- **initial**
- **forever @(posedge clock) x = y;**

Sequential and Parallel Blocks

- Block statements are used to group multiple statements to act together as one.
- In previous examples, we used keywords begin and end to group multiple statements.
- Thus, we used sequential blocks where the statements in the block execute one after another.
- In this section we discuss the block types: sequential blocks and parallel blocks.
- We also discuss three special features of blocks: named blocks, disabling named blocks, and nested blocks.
- There are two types of blocks: sequential blocks and parallel blocks

Sequential blocks

- The keywords **begin** and **end** are used to group statements into sequential blocks.
- Sequential blocks have the following characteristics:
- The statements in a sequential block are processed in the order they are specified.
- A statement is executed only after its preceding statement completes execution (except for nonblocking assignments with intra- assignment timing control).
- If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.

■ Example 4-26 Sequential Blocks

- In Illustration 1, the final values are $x = 0$, $y = 1$, $z = 1$, $w = 2$ at simulation time 0.
- //Illustration 1: Sequential block without delay
- reg x, y;
- reg [1:0] z, w;
- initial
- begin
- x = 1'b0;
- y = 1'b1;
- z = {x, y};
- w = {y, x};
- end

- In Illustration 2, the final values are the same except that the simulation time is 35 at the end of the block.
 - //Illustration 2: Sequential blocks with delay.
 - reg x, y;
 - reg [1:0] z, w; initial
 - begin
 - x = 1'b0; //completes at simulation time 0
 - #5 y = 1'b1; //completes at simulation time 5
 - #10 z = {x, y}; //completes at simulation time 15
 - #20 w = {y, x}; //completes at simulation time 35
 - end
-

Parallel blocks

- Parallel blocks, specified by keywords **fork** and **join**, provide interesting simulation features.
- Parallel blocks have the following characteristics:
- Statements in a parallel block are executed concurrently.
- Ordering of statements is controlled by the delay or event control assigned to each statement.
- If delay or event control is specified, it is relative to the time the block was entered.
- Notice the fundamental difference between sequential and parallel blocks.
- **All statements in a parallel block start at the time when the block was entered. Thus, the order in which the statements are written in the block is not important.**

- Let us consider the sequential block with delay in Example 4-26 and convert it to a parallel block. The converted Verilog code is shown in Example 4-27. The result of simulation remains the same except that all statements start in parallel at time 0. Hence, the block finishes at time 20 instead of time 35.
- `//Example 1: Parallel blocks with delay. reg x, y;`
- `reg [1:0] z, w; initial`
- `fork`
- `x = 1'b0; //completes at simulation time 0`
- `#5 y = 1'b1; //completes at simulation time 5`
- `#10 z = {x, y}; //completes at simulation time 10`
- `#20 w = {y, x}; //completes at simulation time 20`
- `join`

- **//Parallel blocks with deliberate race condition**

- reg x, y;

- reg [1:0] z, w;

- initial

- fork

- x = 1'b0;

- y = 1'b1;

- z = {x, y};

- w = {y, x};

- join

Special Features of Blocks

- We discuss three special features available with block statements: **nested blocks, named blocks and disabling of named blocks.**
- **Nested blocks:- Blocks can be nested.**
- **Sequential and parallel blocks can be mixed, as shown in Example 4-28. Example 4-28 Nested Blocks**
- `//Nested blocks initial`
- `begin`
- `x = 1'b0;`
- `fork`
- `#5 y = 1'b1;`
- `#10 z = {x, y};`
- `join`
- `#20 w = {y, x};`
- `end`

■ **Named blocks**

- Blocks can be given names.
 - Local variables can be declared for the named block.
 - Named blocks are a part of the design hierarchy.
Variables in a named block can be accessed by using hierarchical name referencing.
 - Named blocks can be disabled, i.e., their execution can be stopped.
 - Example 4-29 shows naming of blocks and hierarchical naming of blocks.
-

■ Example 4-29 Named Blocks

- `//Named blocks module top; initial`
- `begin: block1 //sequential block named block1 integer i;`
`//integer i is static and local to block1`
- `// can be accessed by hierarchical name, top.block1.i`
- `...`
- `...`
- `end initial`
- `fork: block2 //parallel block named block2`
- `reg i; // register i is static and local to block2`
- `// can be accessed by hierarchical name, top.block2.i`
- `...`
- `...`
- `join`

Disabling named blocks

- The keyword `disable` provides a way to terminate the execution of a named block.
- `Disable` can be used to get out of loops, handle error conditions, or control execution of pieces of code, based on a control signal.
- Disabling a block causes the execution control to be passed to the statement immediately succeeding the block.
- For C programmers, this is very similar to the `break` statement used to exit a loop.

Task and Functions

- **Learning Objectives**
 - Describe the differences between tasks and functions.
 - Identify the conditions required for tasks to be defined. Understand task declaration and invocation.
 - Explain the conditions necessary for functions to be defined. Understand function declaration and invocation.
-

Task and Functions

- A designer is frequently required to implement the same functionality at many places in a behavioral design.
 - This means that the commonly used parts should be abstracted into routines and the routines must be invoked instead of repeating the code.
 - Most programming languages provide procedures or subroutines to accomplish this.
 - Verilog provides tasks and functions to break up large behavioral designs into smaller pieces.
 - Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design.
-

- Tasks have input, output, and inout arguments; functions have input arguments.
 - Thus, values can be passed into and out from tasks and functions.
 - Considering the analogy of FORTRAN, tasks are similar to SUBROUTINE and functions are similar to FUNCTION.
 - Tasks and functions are included in the design hierarchy.
 - Like named blocks, tasks or functions can be addressed by means of hierarchical names.
-

Differences between Tasks and Functions

Table 8-1. Tasks and Functions

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one input argument. They can have more than one input .	Tasks may have zero or more arguments of type input , output , or inout .
Functions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout arguments.

- Both task and functions must be defined in a module and are local to the module.
- Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments.
- Functions are used when common Verilog code is purely combinational, executes in zero simulation time and provides exactly one output
- Functions are typically used for conversions and commonly used calculations.
- Task can have input, output and in-out ports
- Functions can have input ports. In addition they can have local variables, integers, real or events.
- Tasks and functions cannot have wires, they contain behavioral statements only.
- Tasks and functions do not contain always and initial statements but are called from always block, initial block and other task and functions.

Task

- Tasks are declared with the keywords **task** and **endtask**. **Tasks must be used if any one of the following conditions is true for the procedure:**
- There are delay, timing, or event control constructs in the procedure.
- The procedure has zero or more than one output arguments.
- The procedure has no input arguments.
- I/O declaration use keywords input, output or inout, based on the type of argument declared.
- Input and output arguments are passed into the task.
- Input arguments are processed in the task statements.
- Output and inout argument values are passed back to the variables in the task invocation statement when the task is completed.
- Task can invoke other tasks or functions.
- Ports are used to connect external signals to the module.
- I/O arguments in a task are used to pass values to and from the task.
Arguments are processed in the task statements.

Task Declaration and Invocation

- `task_declaration ::=`
- `task [automatic] task_identifier ;`
- `{ task_item_declaration }`
- `statement`
- `endtask`
- **`| task [automatic] task_identifier (task_port_list) ;`**
- `{ block_item_declaration }`
- `statement`
- `endtask`

-
- `task_item_declaration ::=`
 - `block_item_declaration`
 - `| { attribute_instance } tf_input_declaration ;`
 - `| { attribute_instance } tf_output_declaration ;`
 - `| { attribute_instance } tf_inout_declaration ;`
 - `task_port_list ::= task_port_item { , task_port_item }`
 - `task_port_item ::=`
 - `{ attribute_instance } tf_input_declaration`
 - `| { attribute_instance } tf_output_declaration`
 - `| { attribute_instance } tf_inout_declaration`
-

- **tf_input_declaration ::=**
 - **input [reg] [signed] [range]**
list_of_port_identifiers
 - | input [task_port_type] list_of_port_identifiers
- **tf_output_declaration ::=**
 - **output [reg] [signed] [range]**
list_of_port_identifiers
 - | output [task_port_type] list_of_port_identifiers
- **tf_inout_declaration ::=**
 - **inout [reg] [signed] [range]**
list_of_port_identifiers
 - | inout [task_port_type] list_of_port_identifiers
- **task_port_type ::=**
 - ~~time | real | realtime | integer~~

- I/O declarations use keywords input, output, or inout, based on the type of argument declared.
- Input and inout arguments are passed into the task.
- Input arguments are processed in the task statements. Output and inout argument values are passed back to the variables in the task invocation statement when the task is completed.
- Tasks can invoke other tasks or functions.
- Although the keywords input, inout, and output used for I/O arguments in a task are the same as the keywords used to declare ports in modules, there is a difference.
- Ports are used to connect external signals to the module. I/O arguments in a task are used to pass values to and from the task.

Task Examples

- **Use of input and output arguments**
 - `//Define a module called operation that contains the task bitwise_oper`
 - `module operation;`
 - `...`
 - `parameter delay = 10;`
 - `reg [15:0] A, B;`
 - `reg [15:0] AB_AND, AB_OR, AB_XOR;`
 - `always @(A or B) //whenever A or B changes in value`
 - `begin`
 - `//invoke the task bitwise_oper. provide 2 input arguments A, B`
 - `//Expect 3 output arguments AB_AND, AB_OR, AB_XOR`
-

- //The arguments must be specified in the same order as they //appear in the task declaration.
 - bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
 - end
 - ...
 - //define task bitwise_oper
 - task bitwise_oper;
 - output [15:0] ab_and, ab_or, ab_xor; //outputs from the task
 - input [15:0] a, b; //inputs to the task
 - begin
 - #delay ab_and = a & b;
 - ab_or = a | b;
 - ab_xor = a ^ b;
 - end
 - endtask
 - ...
-
- endmodule

- In the above task, the input values passed to the task are A and B.
- Hence, when the task is entered, $a = A$ and $b = B$. The three output values are computed after a delay.
- This delay is specified by the parameter delay, which is 10 units for this example.
- When the task is completed, the output values are passed back to the calling output arguments.
- Therefore, $AB_AND = ab_and$, $AB_OR = ab_or$, and $AB_XOR = ab_xor$ when the task is completed.

■ **Example 9-3. Task Definition using ANSI C Style Argument Declaration**

- `//define task bitwise_oper`
 - `task bitwise_oper (output [15:0] ab_and, ab_or, ab_xor,`
 - `input [15:0] a, b);`
 - `begin`
 - `#delay ab_and = a & b;`
 - `ab_or = a | b;`
 - `ab_xor = a ^ b;`
 - `end`
 - `endtask`
-

Asymmetric Sequence Generator

- Tasks can directly operate on reg variables defined in the module.
- Example 8-4 directly operates on the reg variable clock to continuously produce an asymmetric sequence.
- The clock is initialized with an initialization sequence.
- Example 9-4. Direct Operation on reg Variables
- `//Define a module that contains the task asymmetric_sequence`
- `module sequence;`
- `reg clock;`
- `...`
- `initial`
- `init_sequence; //Invoke the task init_sequence`
- `...`
- `always`
- `begin`
- `asymmetric_sequence; //Invoke the task asymmetric_sequence`
- `End`

- //Initialization sequence
- task init_sequence;
- begin
- clock = 1'b0;
- end
- endtask
- //define task to generate asymmetric sequence
- //operate directly on the clock defined in the module.
- task asymmetric_sequence;
- begin
- #12 clock = 1'b0;
- #5 clock = 1'b1;
- #3 clock = 1'b0;
- #10 clock = 1'b1;
- end
- endtask
- Endmodule

Functions

- Functions are declared with the keywords **function** and **endfunction**.
- Functions are used if all of the following conditions are true for the procedure:
 - 1. There are no delay, timing, or event control constructs in the procedure.
 - 2. The procedure returns a single value.
 - 3. There is at least one input argument.
 - 4. There are no output or inout arguments.
 - 5. There are no nonblocking assignments.

Function Declaration and Invocation

■ Example 9-6. Syntax for Functions

- `function_declaration ::=`
- **`function [automatic] [signed] [range_or_type]`**
- `function_identifier ;`
- `function_item_declaration {function_item_declaration }`
- `function_statement`
- `endfunction`
- **`| function [automatic] [signed] [range_or_type]`**
- **`function_identifier (function_port_list) ;`**
- `block_item_declaration { block_item_declaration }`
- `function_statement`
- `endfunction`

-
- `function_item_declaration ::=`
 - `block_item_declaration`
 - `| tf_input_declaration ;`
 - `function_port_list ::= { attribute_instance }`
`tf_input_declaration {,`
 - `{ attribute_instance } tf_input_declaration }`
 - `range_or_type ::= range | integer | real | realtime`
`| time`
-

- When a function is declared, a register with name `function_idenfifer` is declared implicitly inside Verilog.
- The output of a function is passed back by setting the value of the register `function_idenfifer` appropriately.
- The function is invoked by specifying function name and input arguments.
- At the end of function execution, the return value is placed where the function was invoked.
- The optional `range_or_type` specifies the width of the internal register.
- If no range or type is specified, the default bit width is 1. Functions are very similar to `FUNCTION` in FORTRAN.
- Notice that at least one input argument must be defined for a function. There are no output arguments for functions because the implicit register *function_idenfifer* contains the output value. Also, functions cannot invoke other tasks. They can invoke only other functions.

Function Examples

- **Parity calculation**
- `//Define a module that contains the function calc_parity`
- `module parity;`
- `...`
- `reg [31:0] addr;`
- `reg parity;`
- `//Compute new parity whenever address value changes`
- `always @(addr)`
- `begin`
- `parity = calc_parity(addr); //First invocation of calc_parity`
- `$display("Parity calculated = %b", calc_parity(addr));`
- `//Second invocation of calc_parity`
- `end`

-
- **//define the parity calculation function**
 - function calc_parity;
 - input [31:0] address;
 - begin
 - //set the output value appropriately. Use the implicit
 - //internal register calc_parity.
 - calc_parity = ^address; //Return the xor of all address bits.
 - end
 - endfunction
 - ...
 - ...
 - endmodule
-

-
- **Example 9-8. Function Definition using ANSI C Style Argument Declaration**
 - `//define the parity calculation function using ANSI C Style arguments`
 - `function calc_parity (input [31:0] address);`
 - `begin`
 - `//set the output value appropriately. Use the implicit`
 - `//internal register calc_parity.`
 - `calc_parity = ^address; //Return the xor of all address bits.`
 - `end`
 - `endfunction`
-

Automatic (Recursive) Functions

- Functions are normally used non-recursively . If a function is called concurrently from two locations, the results are non-deterministic because both calls operate on the same variable space.
- However, the keyword `automatic` can be used to declare a recursive (automatic) function where all function declarations are allocated dynamically for each recursive calls.
- Each call to an automatic function operates in an independent variable space. Automatic function items cannot be accessed by hierarchical references. Automatic functions can be invoked through the use of their hierarchical name.

■ **Example 9-10. Recursive (Automatic) Functions**

- //Define a factorial with a recursive function
- module top;
- ...
- // Define the function
- function automatic integer factorial;
- input [31:0] oper;
- integer i;
- begin
- if (operand >= 2)
- factorial = factorial (oper -1) * oper; //recursive call
- else
- factorial = 1 ;
- end
- endfunction

-
- `// Call the function`
 - `integer result;`
 - `initial`
 - `begin`
 - `result = factorial(4); // Call the factorial of 7`
 - `$display("Factorial of 4 is %0d", result); //Displays 24`
 - `end`
 - `...`
 - `...`
 - `endmodule`
-

Constant Functions

- A constant function[1] is a regular Verilog HDL function, but with certain restrictions.
- These functions can be used to reference complex values and can be used instead of constants
- **Example:-Constant Functions**-shows how a constant function can be used to compute the width of the address bus in a module.
- **//Define a RAM model**
- module ram (...);
- parameter RAM_DEPTH = 256;
- input [clogb2(RAM_DEPTH)-1:0] addr_bus; //width of bus computed
- //by calling constant//function defined below//Result of clogb2 = 8
- //input [7:0] addr_bus;
- --
- --

-
- `//Constant function`
 - `function integer clogb2(input integer depth);`
 - `begin`
 - `for(clogb2=0; depth >0; clogb2=clogb2+1)`
 - `depth = depth >> 1;`
 - `end`
 - `endfunction`
 - `--`
 - `--`
 - `endmodule`
-

Signed Functions

- Signed functions allow signed operations to be performed on the function return values.
- Example 8-12 shows an example of a signed function.
- **Example 9-12. Signed Functions**
- `module top;`
- `//Signed function declaration`
- `//Returns a 64 bit signed value`
- `function signed [63:0] compute_signed(input [63:0]`
`vector);`
- `--`
- `--`
- `endfunction`

-
- `//Call to the signed function from the higher module`
 - `if(compute_signed(vector) < -3)`
 - `begin`
 - `--`
 - `end`
 - `endmodule`
-

■ **Recommended Questions**

- 1. Describe the following statements with an example: initial and always
 - 2. What are blocking and non-blocking assignment statements? Explain with examples.
 - 3. With syntax explain conditional, branching and loop statements available in Verilog HDL behavioural description.
 - 4. Describe sequential and parallel blocks of Verilog HDL.
 - 5. Write Verilog HDL program of 4:1 mux using CASE statement.
 - 6. Write Verilog HDL program of 4:1 mux using If-else statement.
 - 7. Write Verilog HDL program of 4-bit synchronous up counter.
 - 8. Write Verilog HDL program of 4-bit asynchronous down counter.
 - 9. Write Verilog HDL program to simulate traffic signal controller
-

Reference / Text Book Details

Sl.No	Title of Book	Author	Publication	Edition
1	Verilog HDL: A Guide to Digital Design and Synthesis	Samir Palnitkar	Pearson Education	2 nd
2	VHDL for Programmable Logic	Kevin Skahill	PHI/Pearson education	2 nd
3	The Verilog Hardware Description Language	Donald E. Thomas, Philip R. Moorby	Springer Science+Business Media, LLC	5 th
4	Advanced Digital Design with the Verilog HDL	Michael D. Ciletti	Pearson (Prentice Hall)	2 nd
5	Design through Verilog HDL	Padmanabhan, Tripura Sundari	Wiley	Latest

Thank You





|| JAI SRI GURUDEV ||
Sri AdichunchanagiriShikshana Trust (R)
SJB INSTITUTE OF TECHNOLOGY
BGS Health & Education City, Kengeri , Bangalore – 60 .

***DEPARTMENT OF ELECTRONICS & COMMUNICATION
ENGINEERING***

Verilog HDL [18EC56]

Module 5: Useful Modelling Techniques

By:

Mrs. LATHA S
Assistant Professor,
Dept. of ECE, SJBIT

Content

Procedural continuous assignments

overriding parameters,

conditional compilation and execution

useful system tasks

Logic Synthesis

Impact of logic synthesis

Verilog HDL Synthesis

Synthesis design flow, Verification of Gate-Level netlist

Learning Objectives

- Describe procedural continuous assignment statements **assign, deassign, force, and release. Explain their significance in modeling and debugging.**
- Understand how to override parameters by using the **defparam statement at the time of module instantiation.**
- Explain conditional compilation and execution of parts of the Verilog description.
- Identify system tasks for file output, displaying hierarchy, strobing, random number generation, memory initialization, and value change dump

Procedural Continuous Assignments

- Procedural assignments assign a value to a register. The value stays in the register until another procedural assignment puts another value in that register.
- Procedural continuous assignments behave differently. They are procedural statements which allow values of expressions to be driven continuously onto registers or nets for limited periods of time.
- Procedural continuous assignments override existing assignments to a register or net. They provide a useful extension to the regular procedural assignment statement.

Assign and Deassign

- The keywords **assign** and **deassign** are used to express the first type of procedural continuous assignment.
 - The left-hand side of procedural continuous assignments can be only be a register or a concatenation of registers.
 - It cannot be a part or bit select of a net or an array of registers.
 - Procedural continuous assignments override the effect of regular procedural assignments.
 - Procedural continuous assignments are normally used for controlled periods of time.
-

D-Flipflop with Procedural Continuous Assignments

- // Negative edge-triggered D-flipflop with asynchronous reset
- module edge_dff(q, qbar, d, clk, reset);
- // Inputs and outputs
- output q,qbar;
- input d, clk, reset;
- reg q, qbar; //declare q and qbar are registers
- always @(negedge clk) //assign value of q & qbar at active edge of clock.
- begin
- q = d;
- qbar = ~d;
- end

- always @(reset) //Override the regular assignments to q and qbar
- //whenever reset goes high. Use of procedural continuous //assignments.
- if(reset)
- begin //if reset is high, override regular assignments to q with
- //the new values, using procedural continuous assignment.
- assign q = 1'b0;
- assign qbar = 1'b1;
- end
- else
- begin //If reset goes low, remove the overriding values by //deassigning the registers. After this the regular
- //assignments q = d and qbar = ~d will be able to change
- //the registers on the next negative edge of clock.
- deassign q;
- deassign qbar;
- end
- endmodule

Force and Release

- Keywords force and release are used to express the second form of the procedural continuous assignments.
 - They can be used to override assignments on both registers and nets.
 - Force and release statements are typically used in the interactive debugging process, where certain registers or nets are forced to a value and the effect on other registers and nets is noted.
 - It is recommended that force and release statements not be used inside design blocks. They should appear only in stimulus or as debug statements.
-

Force and Release on registers

- A force on a register overrides any procedural assignments or procedural continuous assignments on the register until the register is released.
- The register variables will continue to store the forced value after being released, but can then be changed by a future procedural assignment.
- To override the values of q and $qbar$ in Example 5-1 for a limited period of time, we could do the following:

- module stimulus;
- ...
- //instantiate the d-flipflop
- edge_dff dff(Q, Qbar, D, CLK, RESET);
- ...
- initial
- begin
- //these statements force value of 1 on dff.q between time 50 and
- //100, regardless of the actual output of the edge_dff.
- #50 force dff.q = 1'b1; //force value of q to 1 at time 50.
- #50 release dff.q; //release the value of q at time 100.
- end
- ...
- endmodule

Force and Release on nets

- Force on nets overrides any continuous assignments until the net is released. The net will immediately return to its normal driven value when it is released. A net can be forced to an expression or a value.
- module top;
- ...
- assign out = a & b & c; //continuous assignment on net *out*
- ...
- initial
- #50 force out = a | b & c;
- #50 release out;
- end
- ...
- endmodule

-
- In the example above, a new expression is forced on the net from time 50 to time 100.
 - From time 50 to time 100, when the force statement is active, the expression `a | b & c` will be re-evaluated and assigned to `out` whenever values of signals `a` or `b` or `c` change.
 - Thus, the force statement behaves like a continuous assignment except that it is active for only a limited period of time.
-

Overriding Parameters

- Parameters can be defined in a module definition, as was discussed earlier in Section 3.2.8, Parameters.
- However, during compilation of Verilog modules, parameter values can be altered separately for each module instance.
- This allows us to pass a distinct set of parameter values to each module during compilation regardless of predefined parameter values.
- There are two ways to override parameter values: through the **defparam** statement or through module instance parameter value assignment.

Defparam Statement

- **Example 5-2. Defparam Statement**
- `//Define a module hello_world`
- `module hello_world;`
- `parameter id_num = 0; //define a module
identification number = 0`
- `initial //display the module identification
number`
- `$display("Displaying hello_world id number =
%d", id_num);`
- `endmodule`

-
- `//define top-level module`
 - `module top;`
 - `//change parameter values in the instantiated modules`
 - `//Use defparam statement`
 - `defparam w1.id_num = 1, w2.id_num = 2;`
 - `//instantiate two hello_world modules`
 - `hello_world w1();`
 - `hello_world w2();`
 - `endmodule`
-

- In Example 5-2, the module `hello_world` was defined with a default `id_num = 0`. However, when the module instances `w1` and `w2` of the type `hello_world` are created, their `id_num` values are modified with the `defparam` statement. If we simulate the above design, we would get the following output
- **Displaying hello_world id number = 1**
- **Displaying hello_world id number = 2**
- Multiple `defparam` statements can appear in a module. Any parameter can be overridden with the `defparam` statement. The `defparam` construct is now considered to be a bad coding style and it is recommended that alternative styles be used in Verilog HDL code.
- Note that the module `hello_world` can also be defined using an ANSI C style parameter declaration. Figure 5-3 shows the ANSI C style parameter declaration for the module `hello_world`.

Conditional Compilation and Execution

- A portion of Verilog might be suitable for one environment but not for another.
- The designer does not wish to create two versions of Verilog design for the two environments.
- Instead, the designer can specify that the particular portion of the code be compiled only if a certain flag is set. This is called *conditional compilation*.
- A designer might also want to execute certain parts of the Verilog design only when a flag is set at run time. This is called *conditional execution*.

Conditional Compilation

- Conditional compilation can be accomplished by using compiler directives
- **`ifdef, `ifndef, `else, `elsif, and `endif. Example 5-5 contains Verilog source code to be compiled conditionally.**
- **Example 5-5. Conditional Compilation**
- `//Conditional Compilation`
- `//Example 1`
- `'ifdef TEST //compile module test only if text macro TEST is defined`
- `module test;`
- `...`
- `endmodule`

-
- 'else //compile the module *stimulus as default*
 - module stimulus;
 - ...
 - ...
 - endmodule
 - 'endif //completion of 'ifdef directive
-

- //Example 2
- module top;
- bus_master b1(); //instantiate module unconditionally
- 'ifdef ADD_B2
- bus_master b2(); //b2 is instantiated conditionally if text macro
- //ADD_B2 is defined
- 'elsif ADD_B3
- bus_master b3(); //b3 is instantiated conditionally if text macro
- //ADD_B3 is defined
- 'else
- bus_master b4(); //b4 is instantiate by default
- 'endif
- 'ifndef IGNORE_B5
- bus_master b5(); //b5 is instantiated conditionally if text macro
- //IGNORE_B5 is not defined
- 'endif
- endmodule

- The ``ifdef` and ``ifndef` directives can appear anywhere in the design.
- A designer can conditionally compile statements, modules, blocks, declarations, and other compiler directives.
- The ``else` directive is optional.
- A maximum of one ``else` directive can accompany an ``ifdef` or ``ifndef`.
- Any number of ``elsif` directives can accompany an ``ifdef` or ``ifndef`. An ``ifdef` or ``ifndef` is always closed by a corresponding ``endif`.

- *The conditional compile flag can be set by using the `define statement inside the Verilog file.*
- *In the example above, we could define the flags by defining text macros TEST and ADD_B2 at compile time by using the `define statement.*
- *The Verilog compiler simply skips the portion if the conditional compile flag is not set. A Boolean expression, such as TEST && ADD_B2, is not allowed with the `ifdef statement.*

Conditional Execution

- Conditional execution flags allow the designer to control statement execution flow at run time.
- All statements are compiled but executed conditionally.
- Conditional execution flags can be used only for behavioral statements.
- The system task keyword **\$test\$plusargs** is used for conditional execution.

-
- **Example 5-6. Conditional Execution with \$test\$plusargs**
 - `//Conditional execution`
 - `module test;`
 - `...`
 - `...`
 - `initial`
 - `begin`
 - `if($test$plusargs("DISPLAY_VAR"))`
 - `$display("Display = %b ", {a,b,c}); //display only if flag is set`
 - `else`
 - `//Conditional execution`
 - `$display("No Display"); //otherwise no display`
 - `end`
 - `endmodule`
-

- Conditional execution can be further controlled by using the system task keyword \$value\$plusargs.
- This system task allows testing for arguments to an invocation option.
- \$value\$plusargs returns a 0 if a matching invocation was not found and non-zero if a matching option was found.
- Example 5-7 shows an example of \$value\$plusargs.
- **Example 5-7. Conditional Execution with \$value\$plusargs**
- //Conditional execution with \$value\$plusargs
- module test;
- reg [8*128-1:0] test_string;
- integer clk_period;
- ...
- ...

- initial
- begin
- if(\$value\$plusargs("testname=%s", test_string))
- \$readmemh(test_string, vectors); //Read test vectors
- else
- //otherwise display error message
- \$display("Test name option not specified");
- if(\$value\$plusargs("clk_t=%d", clk_period))
- forever #(clk_period/2) clk = ~clk; //Set up clock
- else
- //otherwise display error message
- \$display("Clock period option name not specified");
- end
- //For example, to invoke the above options invoke simulator with
- //+testname=test1.vec +clk_t=10
- //Test name = "test1.vec" and clk_period = 10
- endmodule

Time Scales

- Often, in a single simulation, delay values in one module need to be defined by using certain time unit, e.g., 1 μ s, and delay values in another module need to be defined by using a different time unit, e.g. 100 ns.
- Verilog HDL allows the reference time unit for modules to be specified with **the `timescale compiler directive.**

- **Usage: ``timescale <reference_time_unit> / <time_precision>`**
- The `<reference_time_unit>` specifies the unit of measurement for times and delays.
- The `<time_precision>` specifies the precision to which the delays are rounded off during simulation.
- Only 1, 10, and 100 are valid integers for specifying time unit and time precision. Consider the two modules, *dummy1* and *dummy2*, in Example 5-8.

-
- `//Define a time scale for the module dummy1`
 - `//Reference time unit is 100 nanoseconds and precision is 1 ns`
 - ``timescale 100 ns / 1 ns`
 - `module dummy1;`
 - `reg toggle;`
 - `//initialize toggle`
 - `initial`
 - `toggle = 1'b0;`
 - `//Flip the toggle register every 5 time units`
-

-
- //In this module 5 time units = 500 ns = .5 μ s
 - always #5
 - begin
 - toggle = ~toggle;
 - \$display("%d , In %m toggle = %b ", \$time, toggle);
 - end
 - endmodule
-

- //Define a time scale for the module dummy2
- //Reference time unit is 1 microsecond and precision is 10 ns
- `timescale 1 us / 10 ns
- module dummy2;
- reg toggle;
- //initialize toggle
- initial
- toggle = 1'b0;
- //Flip the toggle register every 5 time units
- //In this module 5 time units = 5 μ s = 5000 ns
- always #5
- begin
- toggle = ~toggle;
- \$display("%d , In %m toggle = %b ", \$time, toggle);
- end
- endmodule

- The two modules `dummy1` and `dummy2` are identical in all respects, except that the time unit for `dummy1` is 100 ns and the time unit for `dummy2` is 1 μ s.
- Thus the `$display` statement in `dummy1` will be executed 10 times for each `$display` executed in `dummy2`.
- The `$time` task reports the simulation time in terms of the reference time unit for the module in which it is invoked.
- The first few `$display` statements are shown in the simulation output below to illustrate the effect of the ``timescale` directive.

- 5 , In dummy1 toggle = 1
- 10 , In dummy1 toggle = 0
- 15 , In dummy1 toggle = 1
- 20 , In dummy1 toggle = 0
- 25 , In dummy1 toggle = 1
- 30 , In dummy1 toggle = 0
- 35 , In dummy1 toggle = 1
- 40 , In dummy1 toggle = 0
- 45 , In dummy1 toggle = 1
- --> 5 , In dummy2 toggle = 1
- 50 , In dummy1 toggle = 0
- 55 , In dummy1 toggle = 1
- Notice that the **\$display** statement in *dummy2* ***executes once for every ten \$display statements in dummy1.***

Useful System Tasks

- We discuss system tasks [1] for file output, displaying hierarchy, strobing, random number generation, memory initialization, and value change dump.
- **File Output**
- Output from Verilog normally goes to the standard output and the file verilog.log. It is possible to redirect the output of Verilog to a chosen file.
- **Opening a file**
- A file can be opened with the system task **\$fopen**.
- Usage: **\$fopen("<name_of_file>"); [2]**
- Usage: **<file_handle> = \$fopen("<name_of_file>");**

- The task `$fopen` returns a 32-bit value called a multichannel descriptor.[3]
- Only one bit is set in a multichannel descriptor.
- The standard output has a multichannel descriptor with the least significant bit (bit 0) set.
- Standard output is also called channel 0. The standard output is always open.
- Each successive call to `$fopen` opens a new channel and returns a 32-bit descriptor with bit 1 set, bit 2 set, and so on, up to bit 30 set.
- Bit 31 is reserved. The channel number corresponds to the individual bit set in the multichannel descriptor. Example 9-9 illustrates the use of file descriptors.

File Descriptors

- //Multichannel descriptor
- integer handle1, handle2, handle3; //integers are 32-bit values
- //standard output is open; descriptor = 32'h0000_0001 (bit 0 set)
- initial
- begin
- handle1 = \$fopen("file1.out"); //handle1 = 32'h0000_0002 (bit 1 set)
- handle2 = \$fopen("file2.out"); //handle2 = 32'h0000_0004 (bit 2 set)
- handle3 = \$fopen("file3.out"); //handle3 = 32'h0000_0008 (bit 3 set)
- end

Writing to files

- The system tasks `$fdisplay`, `$fmonitor`, `$fwrite`, and `$fstrobe` are used to write to files.
- Note that these tasks are similar in syntax to regular system tasks `$display`, `$monitor`, etc., but they provide the additional capability of writing to file. We will consider only `$fdisplay` and `$fmonitor` tasks.
- *Usage:* `$fdisplay(<file_descriptor>, p1, p2 ..., pn);`
 `$fmonitor(<file_descriptor>, p1, p2,..., pn);`
- `p1, p2, ..., pn` can be variables, signal names, or quoted strings.
- A `file_descriptor` is a multichannel descriptor that can be a file handle or a bitwise combination of file handles. Verilog will write the output to all files that have a 1 associated with them in the file descriptor.

-
- `//All handles defined in Example 9-9`
 - `//Writing to files`
 - `integer desc1, desc2, desc3; //three file descriptors`
 - `initial`
 - `begin`
 - `desc1 = handle1 | 1; //bitwise or; desc1 = 32'h0000_0003`
 - `$fdisplay(desc1, "Display 1");//write to files file1.out & stdout`
 - `desc2 = handle2 | handle1; //desc2 = 32'h0000_0006`
 - `$fdisplay(desc2, "Display 2");//write to files file1.out & file2.out`
 - `desc3 = handle3 ; //desc3 = 32'h0000_0008`
 - `$fdisplay(desc3, "Display 3");//write to file file3.out only`
 - `end`
-

- **Closing files**

- Files can be closed with the system task **\$fclose**.

- Usage: ***\$fclose(<file_handle>);***

- //Closing Files

- **\$fclose(handle1);**

- A file cannot be written to once it is closed. The corresponding bit in the multichannel descriptor is set to 0. The next **\$fopen call can reuse the bit.**
-

Displaying Hierarchy

- Hierarchy at any level can be displayed by means of the %m option in any of the display tasks, \$display, \$write task, \$monitor, or \$strobe task, as discussed briefly in Section 4.3, Hierarchical Names.
 - This is a very useful option. For example, when multiple instances of a module execute the same Verilog code, the %m option will distinguish from which module instance the output is coming.
 - No argument is needed for the %m option in the display tasks. See Example 9-10.
-

Displaying Hierarchy

- //Displaying hierarchy information
- module M;
- ...
- initial
- \$display("Displaying in %m");
- endmodule
- //instantiate module M
- module top;
- ...
- M m1();
- M m2();
- //Displaying hierarchy information
- M m3();
- endmodule

-
- The output from the simulation will look like the following:
 - Displaying in top.m1
 - Displaying in top.m2
 - Displaying in top.m3
 - This feature can display full hierarchical names, including module instances, tasks, functions, and named blocks.
-

Strobing

- **Strobing** is done with the system task keyword **\$strobe**. This task is very similar to the **\$display** task except for a slight difference.
- If many other statements are executed in the same time unit as the **\$display** task, the order in which the statements and the **\$display** task are executed is nondeterministic.
- If **\$strobe** is used, it is always executed after all other assignment statements in the same time unit have executed.
- Thus, **\$strobe** provides a synchronization mechanism to ensure that data is displayed only after all other assignment statements, which change the data in that time step, have executed.

■ **Example 5-11. Strobing**

- `//Strobing`
- `always @(posedge clock)`
- `begin`
- `a = b;`
- `c = d;`
- `end`
- `always @(posedge clock)`
- `$strobe("Displaying a = %b, c = %b", a, c); // display values at posedge`
- In Example 9-11, the values at positive edge of clock will be displayed only after statements `a = b` and `c = d` execute.
- If `$display` was used, `$display` might execute before statements `a = b` and `c = d`, thus displaying different values.

Random Number Generation

- Random number generation capabilities are required for generating a random set of test vectors.
- Random testing is important because it often catches hidden bugs in the design.
- Random vector generation is also used in performance analysis of chip architectures.
- The system task **\$random is used for generating a random number.**
- **Usage: \$random;**
- **\$random(<seed>;**
- The value of <seed> is optional and is used to ensure the same random number sequence each time the test is run. The <seed> parameter can either be a **reg, integer, or time variable.** **The task \$random returns a 32-bit signed integer.**

Random Number Generation

- //Generate random numbers and apply them to a simple ROM
 - module test;
 - integer r_seed;
 - reg [31:0] addr;//input to ROM
 - wire [31:0] data;//output from ROM
 - ...
 - ROM rom1(data, addr);
 - initial
 - r_seed = 2; //arbitrarily define the seed as 2.
 - always @(posedge clock)
 - addr = \$random(r_seed); //generates random numbers
 - ...
 - <check output of ROM against expected results>
-
- ...
 - endmodule

-
- Generation of Positive and Negative Numbers by \$random Task
 - reg [23:0] rand1, rand2;
 - rand1 = \$random % 60; //Generates a random number between -59 and 59
 - rand2 = {\$random} % 60; //Addition of concatenation operator to
 - //\$random generates a positive value between //0 and 59.
-

■ **Initializing Memory from File**

- We discussed how to declare memories in Section 3.2.7, Memories.
- Verilog provides a very useful system task to initialize memories from a data file.
- Two tasks are provided to read numbers in binary or hexadecimal format.
- Keywords \$readmemb and \$readmemh are used to initialize memories.
- Usage: \$readmemb("<file_name>", <memory_name>);
 \$readmemb("<file_name>", <memory_name>, <start_addr>);
- \$readmemb("<file_name>", <memory_name>, <start_addr>, <finish_addr>);
- Identical syntax for \$readmemh.
- The <file_name> and <memory_name> are mandatory; <start_addr> and <finish_addr> are optional.
- Defaults are start index of memory array for <start_addr> and end of the data file or memory for <finish_addr>.

■

Initializing Memory

- module test;
- reg [7:0] memory[0:7]; //declare an 8-byte memory
- integer i;
- initial
- begin
- //read memory file init.dat. address locations given in memory
- \$readmemb("init.dat", memory);
- module test;
- //display contents of initialized memory
- for(i=0; i < 8; i = i + 1)
- \$display("Memory [%0d] = %b", i, memory[i]);
- end
- endmodule

-
- The file *init.dat* contains the initialization data. Addresses are specified in the data file with @<address>. Addresses are specified as hexadecimal numbers. Data is separated by whitespaces. Data can contain **x** or **z**. **Uninitialized locations default to x. A sample file, *init.dat*, is shown below.**

- @002
 - 11111111 01010101
 - 00000000 10101010
 - @006
 - 1111zzzz 00001111
-

- When the test module is simulated, we will get the following output:
- Memory [0] = xxxxxxxx
- Memory [1] = xxxxxxxx
- Memory [2] = 11111111
- Memory [3] = 01010101
- Memory [4] = 00000000
- Memory [5] = 10101010
- Memory [6] = 1111zzzz
- Memory [7] = 00001111

Value Change Dump File (VCD)

- A value change dump (VCD) is an ASCII file that contains information about simulation time, scope and signal definitions, and signal value changes in the simulation run.
- All signals or a selected set of signals in a design can be written to a VCD file during simulation.
- Postprocessing tools can take the VCD file as input and visually display hierarchical information, signal values, and signal waveforms.
- Many postprocessing tools as well as tools integrated into the simulator are now commercially available.
- For simulation of large designs, designers dump selected signals to a VCD file and use a postprocessing tool to debug, analyze, and verify the simulation output.

Debugging and Analysis of Simulation with VCD File

- System tasks are provided for selecting module instances or module instance signals to dump (\$dumpvars), name of VCD file (\$dumpfile), starting and stopping the dump process (\$dumpon, \$dumpoff), and generating checkpoints (\$dumpall).
- The uses of each task are shown in Example 9-15.
- //specify name of VCD file. Otherwise, default name is
- //assigned by the simulator.

-
- initial
 - `$dumpfile("myfile.dmp");` //Simulation info dumped to myfile.dmp
 - `//Dump signals in a module`
 - initial
 - `$dumpvars;` //no arguments, dump all signals in the design
 - initial
 - `$dumpvars(1, top);` //dump variables in module instance top.
-

- //Number 1 indicates levels of hierarchy.
Dump one
- //hierarchy level below top, i.e., dump variables in top,
- //but not signals in modules instantiated by top.
- initial
- \$dumpvars(2, top.m1); //dump up to 2 levels of hierarchy below top.m1
- initial
- \$dumpvars(0, top.m1); //Number 0 means dump the entire hierarchy

- // below top.m1
- //Start and stop dump process
- initial
- begin
- \$dumpon; //start the dump process.
- #100000 \$dumpoff; //stop the dump process after 100,000 time units
- end
- //Create a checkpoint. Dump current value of all VCD variables
- initial
- \$dumpall;
- The \$dumpfile and \$dumpvars tasks are normally specified at the beginning of the simulation. The \$dumpon, \$dumpoff, and \$dumpall control the dump process during the simulation.[5]

Questions

- 1) Using assign and deassign statements, design a positive edge-triggered D-flipflop with asynchronous *clear* ($q=0$) and *preset* ($q=1$).
- 2) Using primitive gates, design a 1-bit full adder FA. Instantiate the full adder inside a stimulus module. Force the *sum output to a & b & c_in for the time between 15 and 35 units*.

Questions

- What will be the output of the **\$display** statement shown below?
- module top;
- A a1();
- endmodule
- module A;
- B b1();
- endmodule
- module B;
- initial
- \$display("I am inside instance %m");
- endmodule

Questions

- Identify the files to which the following display statements will write:
- //File output with multi-channel descriptor
- module test;
- integer handle1,handle2,handle3; //file handles
- //open files
- initial
- begin
- handle1 = \$fopen("f1.out");
- handle2 = \$fopen("f2.out");
- handle3 = \$fopen("f3.out");
- end

Questions

- //Display statements to files
- initial
- begin
- //File output with multi-channel descriptor
- #5;
- \$fdisplay(4, "Display Statement # 1");
- \$fdisplay(15, "Display Statement # 2");
- \$fdisplay(6, "Display Statement # 3");
- \$fdisplay(10, "Display Statement # 4");
- \$fdisplay(0, "Display Statement # 5");
- end
- endmodule

Logic Synthesis with Verilog HDL

- Define logic synthesis and explain the benefits of logic synthesis.
- Identify Verilog HDL constructs and operators accepted in logic synthesis. Understand how the logic synthesis tool interprets these constructs.
- Explain a typical design flow, using logic synthesis. Describe the components in the logic synthesis-based design flow.
- Describe verification of the gate-level netlist produced by logic synthesis.
- Understand techniques for writing efficient RTL descriptions.
- Describe partitioning techniques to help logic synthesis provide the optimal gate-level netlist.
- Design combinational and sequential circuits, using logic synthesis.

What Is Logic Synthesis?

- Simply speaking, logic synthesis is the process of converting a high-level description of the design into an optimized gate-level representation, given a standard cell library and certain design constraints.
- A standard cell library can have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adders, muxes, and special flip-flops.
- A standard cell library is also known as the technology library.
- Logic synthesis always existed even in the days of schematic gate-level design, but it was always done inside the designer's mind.

- The designer would first understand the architectural description. Then he would consider design constraints such as *timing, area, testability, and power*.
 - The designer would partition the design into high-level blocks, draw them on a piece of paper or a computer terminal, and describe the functionality of the circuit. This was the *high-level description*.
 - Finally, each block would be implemented on a hand-drawn schematic, using the cells available in the *standard cell library*.
 - The last step was the most complex process in the design flow and required several time-consuming design iterations before an optimized gate-level representation that met all design constraints was obtained.
-

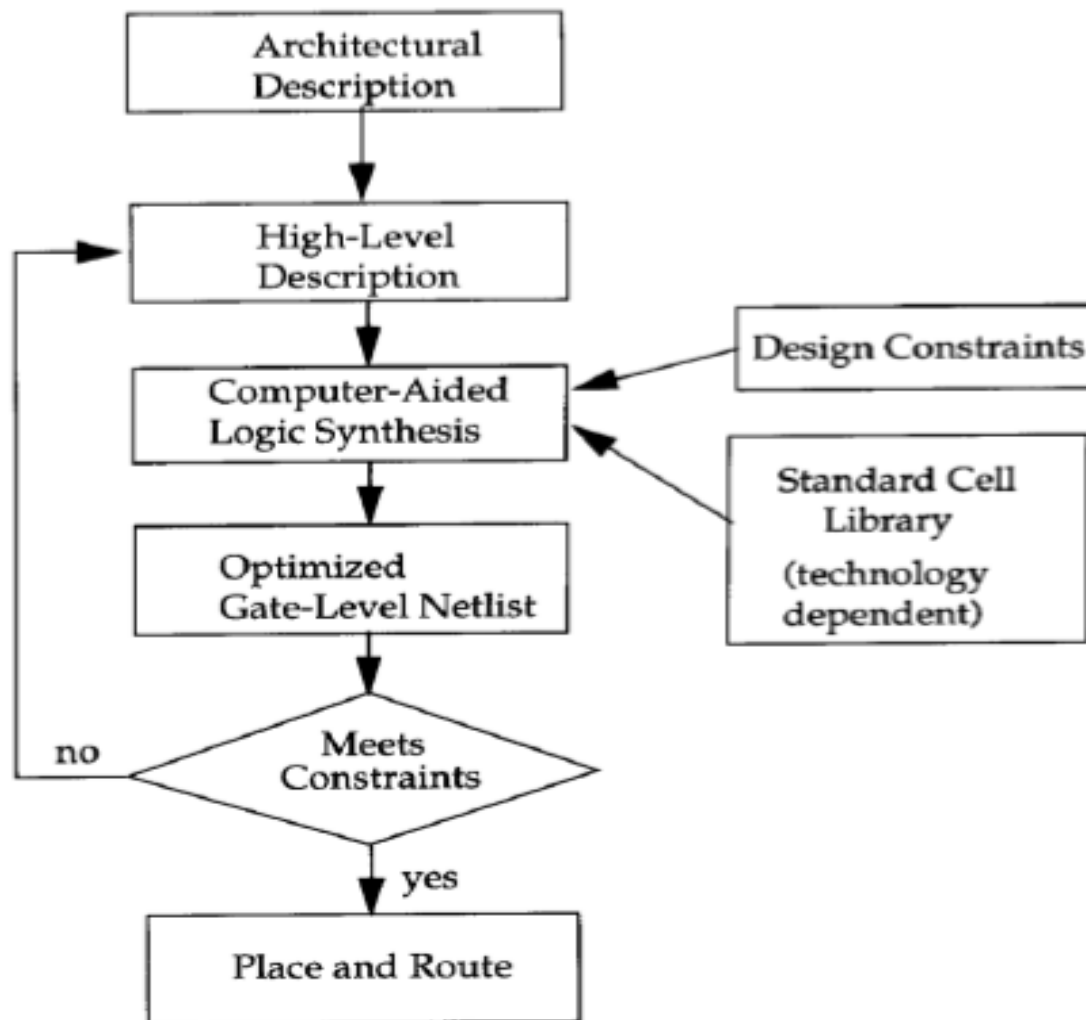


Figure 14-2. Basic Computer-Aided Logic Synthesis Process

Verilog HDL Synthesis

- The term RTL(**Register transfer level**) is used for an HDL description style that utilizes a combination of data flow and behavioral constructs.
- Logic synthesis tools take the register transfer-level HDL description and convert it to an optimized gate-level netlist.
- Verilog and VHDL are the two most popular HDLs used to describe the functionality at the RTL level. In this chapter, we discuss RTL-based logic synthesis with Verilog HDL.
- Behavioral synthesis tools that convert a behavioral description into an RTL description are slowly evolving, but RTL-based synthesis is currently the most popular design method. Thus, we will address only RTL-based synthesis in this chapter

Verilog Constructs

Table 14-1. Verilog HDL Constructs for Logic Synthesis

Construct Type	Keyword or Description	Notes
ports	input, inout, output	
parameters	parameter	
module definition	module	
signals and variables	wire, reg, tri	Vectors are allowed
instantiation	module instances, primitive gate instances	E.g., mymux m1(out, i0, i1, s); E.g., nand (out, a, b);
functions and tasks	function, task	Timing constructs ignored
procedural	always, if, then, else, case, casex, casez	initial is not supported
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
loops	for, while, forever,	while and forever loops must contain @(posedge clk) or @(negedge clk)

Verilog Operators

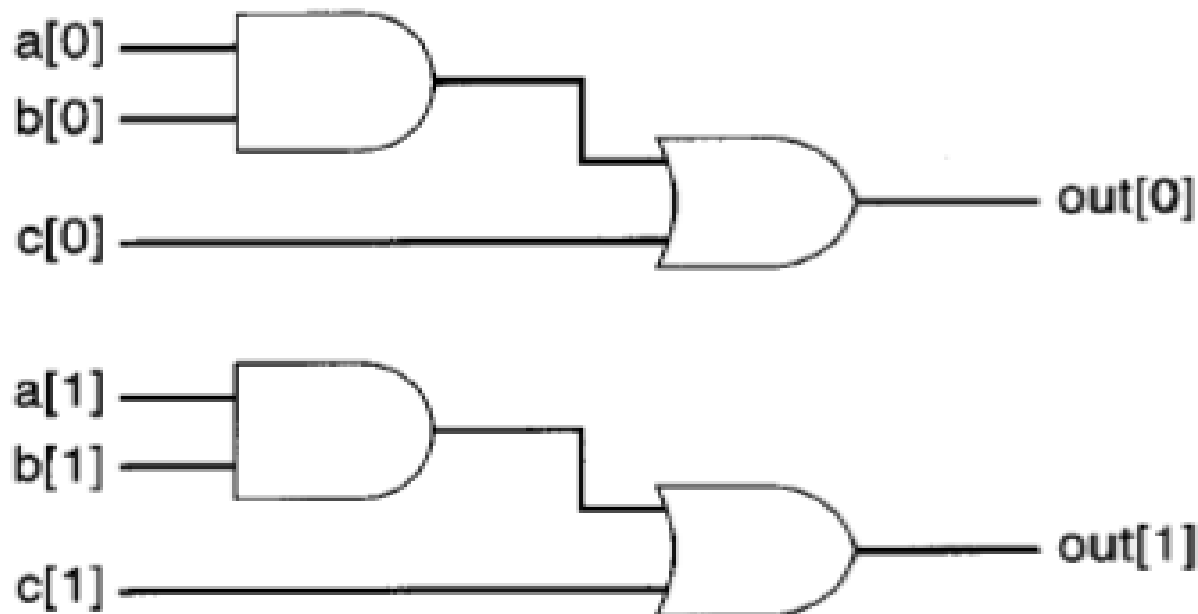
- Almost all operators in Verilog are allowed for logic synthesis.
- Table 14-2 is a list of the operators allowed. Only operators such as `===` and `!==` that are related to x and z are not allowed, because equality with x and z does not have much meaning in logic synthesis.
- While writing expressions, it is recommended that you use parentheses to group logic the way you want it to appear.
- If you rely on operator precedence, logic synthesis tools might produce an undesirable logic structure.

Interpretation of a Few Verilog Constructs

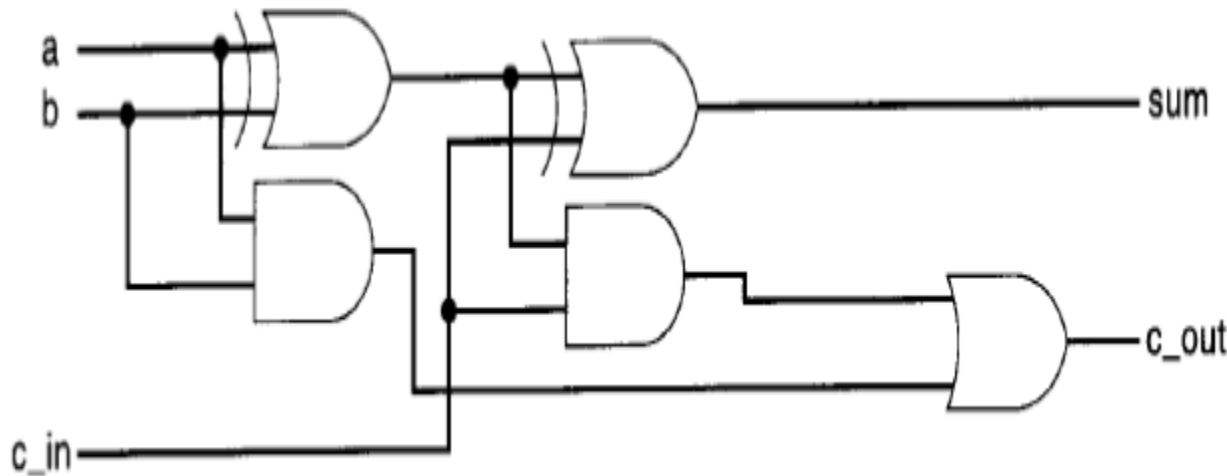
- Having described the basic Verilog constructs, let us try to understand how logic synthesis tools frequently interpret these constructs and translate them to logic gates.
- **Assign statement**
- The assign construct is the most fundamental construct used to describe combinational logic at an RTL level. Given below is a logic expression that uses the assign statement.
- `assign out = (a & b) | c;`
- This will frequently translate to the following gate-level representation:



- If a , b , c , and out are 2-bit vectors $[1:0]$, then the above **assign statement** will frequently **translate to two identical circuits for each bit**.



- If arithmetic operators are used, each arithmetic operator is implemented in terms of arithmetic hardware blocks available to the logic synthesis tool. A 1-bit full adder is implemented below.
- `assign {c_out, sum} = a + b + c_in;`
- Assuming that the 1-bit full adder is available internally in the logic synthesis tool, the above assign statement is often interpreted by logic synthesis tools as follows:



- If a multiple-bit adder is synthesized, the synthesis tool will perform optimization and the designer might get a result that looks different from the above figure.
- If a conditional operator **? is used, a *multiplexer circuit is inferred.***
- assign out = (s) ? i1 : i0;
- It frequently translates to the gate-level representation shown in Figure 14-3.

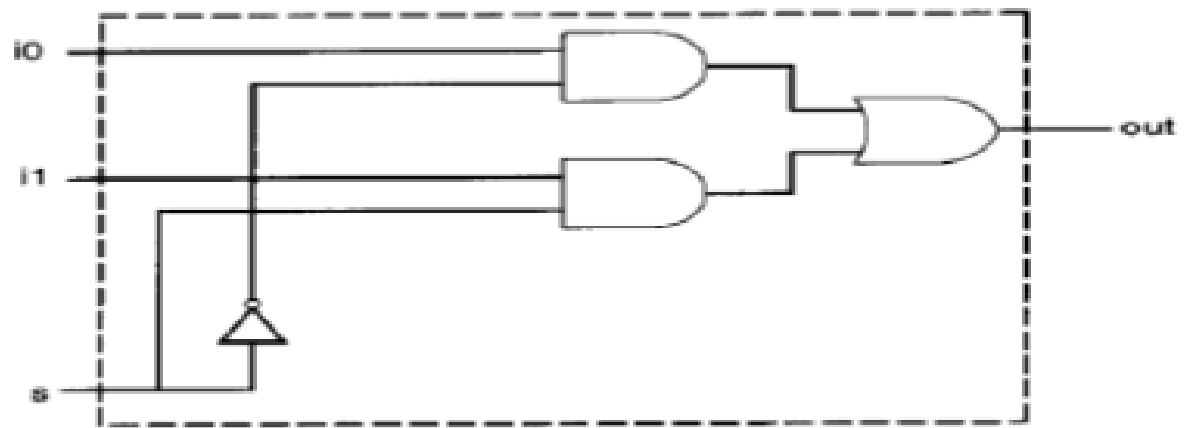


Figure 14-3 Multiplexer Description

- The if-else statement
- Single *if-else statements translate to multiplexers where the control signal is the signal or variable in the if clause.*
- if(s)
- out = i1;
- else
- out = i0;
- The above statement will frequently translate to the gate-level description shown in Figure 14-3. In general,
- multiple *if-else-if statements do not synthesize to large multiplexers*

- The case statement
- The case statement also can be used to infer multiplexers. The above multiplexer would have been inferred from the
- following description that uses case statements:
- case (s)
- 1'b0 : out = i0;
- 1'b1 : out = i1;
- endcase
- Large case statements may be used to infer large multiplexers.

- **for loops**

- The for loops can be used to build cascaded combinational logic. For example, the following for loop builds an
 - 8-bit full adder:
 - `c = c_in;`
 - `for(i=0; i <=7; i = i + 1)`
 - `{c, sum[i]} = a[i] + b[i] + c; // builds an 8-bit ripple adder`
 - `c_out = c;`
-

■ **The always statement**

- The always statement can be used to infer sequential and combinational logic.
- For sequential logic, the always statement must be controlled by the change in the value of a clock signal *clk*.
- `always @(posedge clk)`
- `q <= d;`
- This is inferred as a positive edge-triggered D-flipflop with *d* as input, *q* as output, and *clk* as the clocking signal.
- Similarly, the following Verilog description creates a level-sensitive latch:
- `always @(clk or d)`
- `if (clk)`
- `q <= d;`

Synthesis Design Flow-RTL to Gates

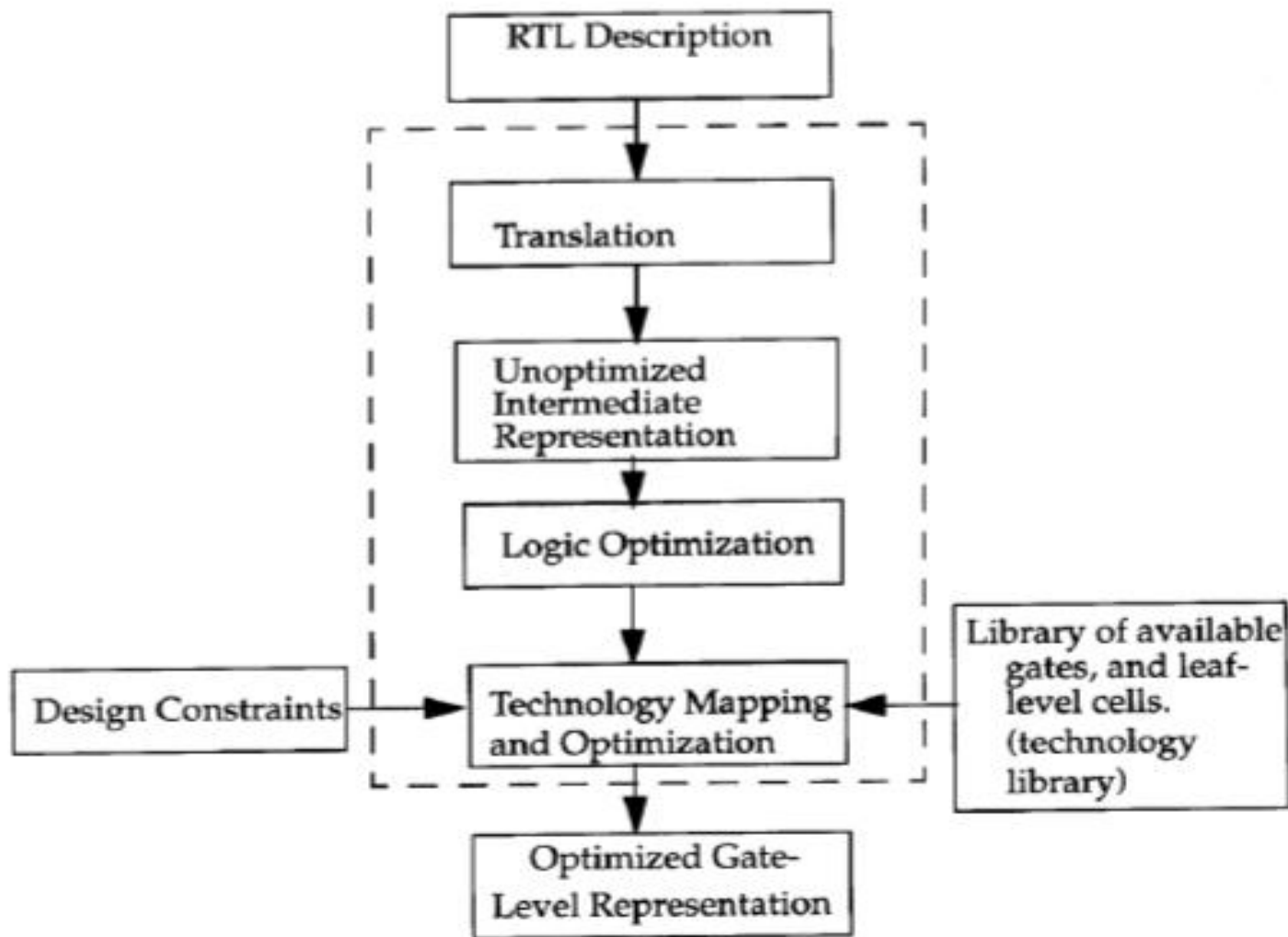


Figure 14-4. Logic Synthesis Flow from RTL to Gates

An Example of RTL-to-Gates

- **Design specification**
- A magnitude comparator checks if one number is greater than, equal to, or less than another number. Design a 4-bit magnitude comparator IC chip that has the following specifications:
- The name of the design is `magnitude_comparator`
- Inputs A and B are 4-bit inputs. No x or z values will appear on A and B inputs
- Output `A_gt_B` is true if A is greater than B
- Output `A_lt_B` is true if A is less than B
- Output `A_eq_B` is true if A is equal to B
- The magnitude comparator circuit must be as fast as possible. Area can be compromised for speed.

■ RTL description

- The RTL description that describes the magnitude comparator is shown in Example 14-1. This is a technology independent description. The designer does not have to worry about the target technology at this point.

■ Example 14-1. RTL for Magnitude Comparator

- `//Module magnitude comparator`
- `module magnitude_comparator(A_gt_B, A_lt_B, A_eq_B, A,`
`B);//Comparison output`
- `output A_gt_B, A_lt_B, A_eq_B;`
- `//4-bits numbers input`
- `input [3:0] A, B;`
- `assign A_gt_B = (A > B); //A greater than B`
- `assign A_lt_B = (A < B); //A less than B`
- `assign A_eq_B = (A == B); //A equal to B`
- `endmodule`

Reference / Text Book Details

Sl.No	Title of Book	Author	Publication	Edition
1	Verilog HDL: A Guide to Digital Design and Synthesis	Samir Palnitkar	Pearson Education	2 nd
2	VHDL for Programmable Logic	Kevin Skahill	PHI/Pearson education	2 nd
3	The Verilog Hardware Description Language	Donald E. Thomas, Philip R. Moorby	Springer Science+Business Media, LLC	5 th
4	Advanced Digital Design with the Verilog HDL	Michael D. Ciletti	Pearson (Prentice Hall)	2 nd
5	Design through Verilog HDL	Padmanabhan, Tripura Sundari	Wiley	Latest

Thank You

