MODULE -1

OVERVIEW OF DIGITAL DESIGN WITH VERILOG HDL

1.1 : Objectives

- Understand the importance and trends of HDL.
- Understand the design flow and design methodologies for digital design.
- Explain the difference between modules and module instances in Verilog.
- Describe four levels of abstraction and define stimulus block and design block.

1.2 Evolution of Computer-Aided Digital Design

In early days digital circuits were designed with vacuum tubes and transistor. Then integrated circuits chips were invented which consists of logic gates embed on them. As technology advances from SSI (Small Scale Integration), MSI (Medium Scale Integration), LSI (Large Scale Integration), designers could implement thousands of gates on a single chip. So the testing of circuits and designing became complicated hence Electronic Design Automation (EDA) techniques to verify functionality of building blocks were one.

The advances in semiconductor technology continue to increase the power and complexity of digital systems with the invent of VLSI (very Large Scale Integration) with more than 10000 transistors. Because of the complexity of circuit, breadboard design became impossible and gave rise to computer aided techniques to design and verify VLSI digital circuits. These computer aided programs and tools allow us to design, do automatic placement and routing and Abe to develop hierarchical based development and hence prototype development by downloading of programmable chips (like - ASIC, FPGA, CPLD) before fabrication.

1.3 Emergence of HDLs

In the field of digital design, the complexity in designing a circuit gave birth to standard languages to describe digital circuits (ie. Hardware Description Languages - HDL). HDL is a Computer Aided design (CAD) tool for the modern design and synthesis of digital systems. HDLs were been used to model hardware elements very concurrently. Verilog HDL and VHDL are most popular HDLs.

In initial days of HDL, designing and verification were done using tool but synthesis (ie translation of RTL to schematic circuit) used to be done manually which become tediously as technology advances. Later

tool is automated to generate the schematic of RTL developed.

Digital circuits are described at Registers Transfer Level (RTL) by using HDL. Then logic synthesis tool will generate details of gates and interconnection to implement circuits. This synthesized result can be used for fabrication by having placement and routing details. Verify functionality using simulation. HDLs are used for system-level design - simulation of system boards, interconnect buses, FPGAs and PALs. Verilog HDL is IEEE standard - IEEE 1364-2001.

Note: RTL - designer has to specify how the data flows between registers and how the design processes the data.

1.4 Typical Design Flow

A typical design flow (HDL flow) for designing VLSI IC circuits is as shown in figure 1.1

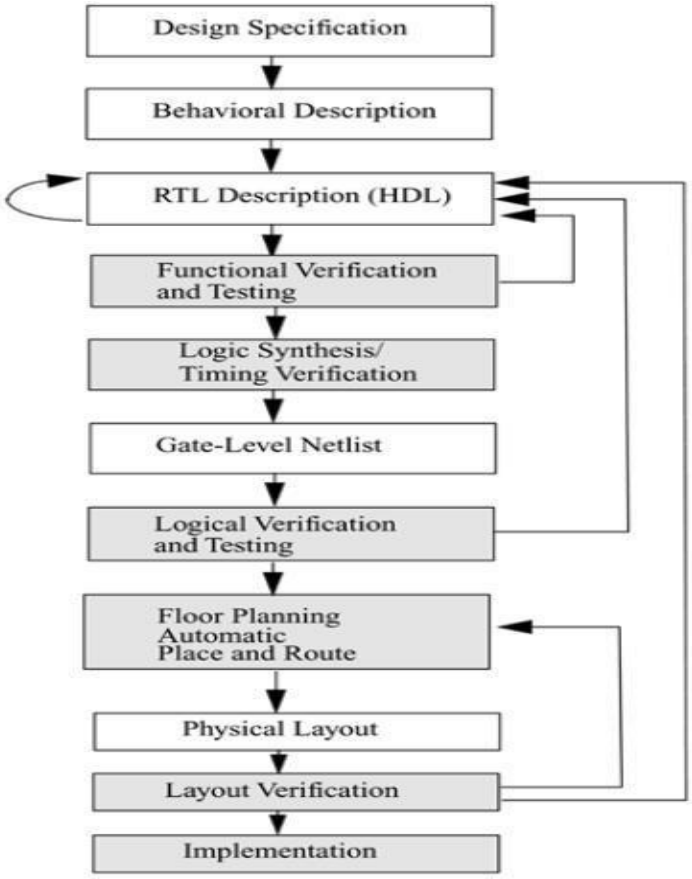


Figure: 1.1: Typical design flow

The design flow in any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. At this point, the architects

do not need to think about how they will implement this circuit. A behavioral description is then created to analyze the design in terms of functionality, performance, and compliance to standards, and other high-level issues. Behavioral descriptions are often written with HDLs.

New EDA tools have emerged to simulate behavioral descriptions of circuits. These tools combine the powerful concepts from HDLs and object oriented languages such as C++. These tools can be used instead of writing behavioral descriptions in Verilog HDL. The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. From this point onward, the design process is done with the assistance of EDA tools.

Logic synthesis tools convert the RTL description to a gate-level net list. Logic synthesis tools ensure that the gate-level net list meets timing, area, and power specifications.

A gate-level net list is a description of the circuit in terms of gates and connections between them. The gate-level net list is input to an Automatic Place and Route tool, which creates a layout.

The layout is verified and then fabricated on a chip.

Thus, most digital design activity is concentrated on manually optimizing the RTL description of the circuit. After the RTL description is frozen, EDA tools are available to assist the designer in further processes. Designing at the RTL level has shrunk the design cycle times from years to a few months. It is also possible to do many design iterations in a short period of time.

Behavioral synthesis tools have begun to emerge recently. These tools can create RTL descriptions from a behavioral or algorithmic description of the circuit. As these tools mature, digital circuit design will become similar to high-level computer programming. Designers will simply implement the algorithm in an HDL at a very abstract level. EDA tools will help the designer convert the behavioral description to a final IC chip.

1.5 Importance of HDLs

HDLs have many advantages that help in developing large digital circuits reaching the optimized circuit design.

- Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology. The logic synthesis tool will optimize the

circuit in area and timing for the new technology.

- By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gate-level netlist or physical layout is minimized.
- Designing with HDLs is similar to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.
- Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design. Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- All fabrication vendors provide Verilog HDL libraries for post-logic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.
- The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

1.6 Trends in HDLs

Increase in speed and complexity go digital circuits will complicate the designer job, but EDA tools make the job easy for designer. Designer has to do high level abstraction designing and need to take care of functionality of the design and EDA tools take care of implementation, and can achieve a almost optimum design.

Digital circuits are designed in HDL at an RTL level, so that logic synthesis tools can create gate level net lists. Behavioral synthesis allowed designers to directly design in terms of algorithms and the behavior of the circuit EDA tool is then used to translate and optimize at each phase of design. Verilog HDL is also used widely for verification. Formal verification uses mathematical techniques to verify the correctness of Verilog HDL

descriptions and to establish equivalency between RTL and gate level net lists. Assertion checking is done to check the transition and important parts of a design.

1.7 Design Methodologies

There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology.

1.7.1 Top-down design methodology:

This designing approach allows early testing, easy change of different technologies, a well structures system design and offers many other advantages.

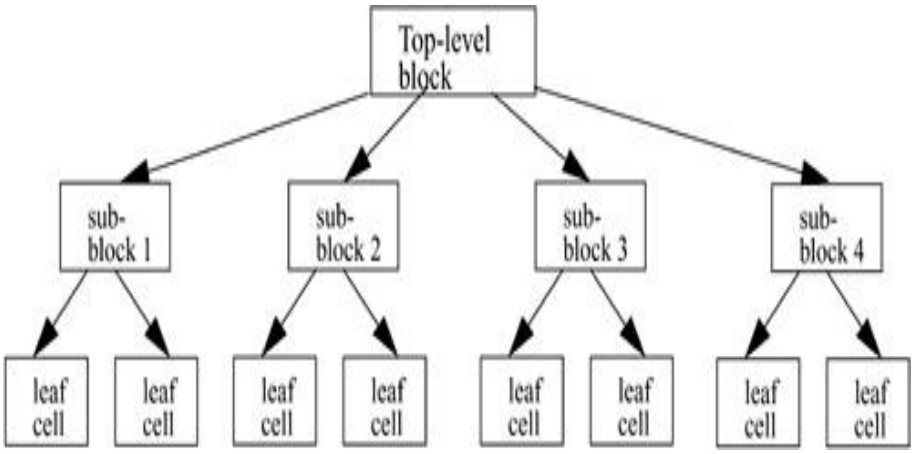


Figure: 1.2: Top-down Design Methodology

In this method, top-level block is defined and sub-blocks necessary to build the top-level block are identified. We further subdivide, sub-blocks until cells cannot be further divided, we call these cells as leaf cells is as shown in figure 1.2.

1.7.2 Bottom-up design methodology:

We first identify the available building blocks and try to build bigger cells out of these, and continue process until we reach the top-level block of the design is as shown in figure 1.3

Most of the time, the combination of these two design methodologies are used to design. Logic designers decide the structure of design and break up the functionality into blocks and sub blocks. And designer will design a optimized circuit for leaf cell and using these will design top level design.

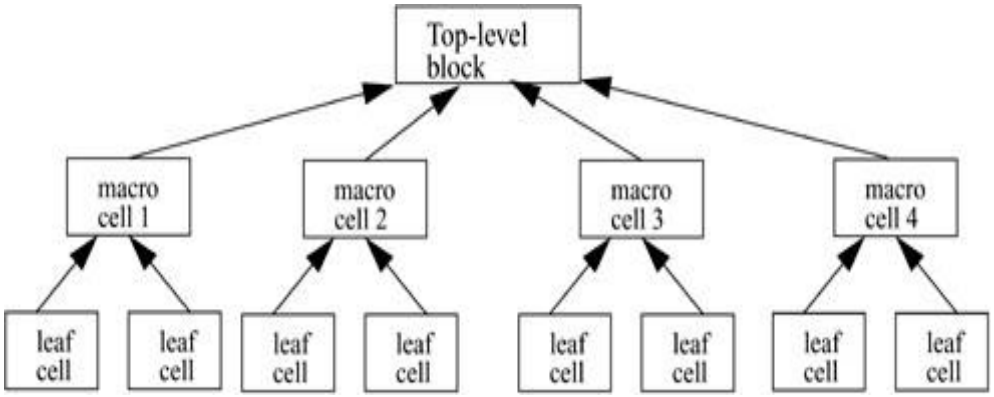


Figure 1-3. Bottom-up Design Methodology

A hierarchical modeling concept is illustrated with an example of 4-bit Ripple Carry Counter.

The ripple carry counter shown in Figure 1.4 is made up of negative edge-triggered toggle flip-flops (T_FF). Each of the T_FFs can be made up from negative edge-triggered D-flip-flops (D_FF) and inverters (assuming q_bar output is not available on the D_FF), as shown in Figure 1.5.

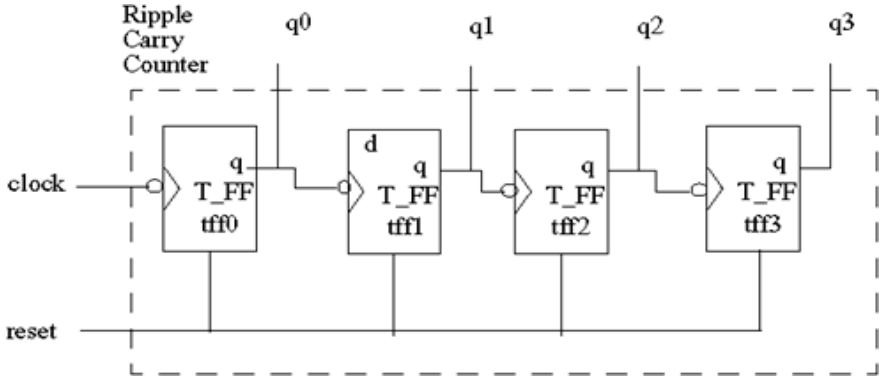


Figure 1.4: Ripple Carry Counter

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

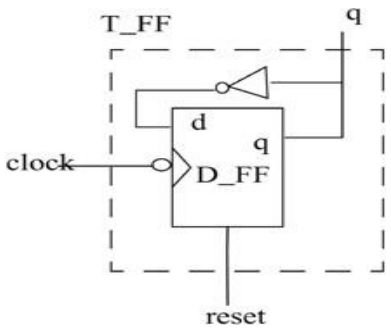


Figure 1-5: T-flip-flop

Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks. The diagram for the

design hierarchy is shown in Figure 1.6.

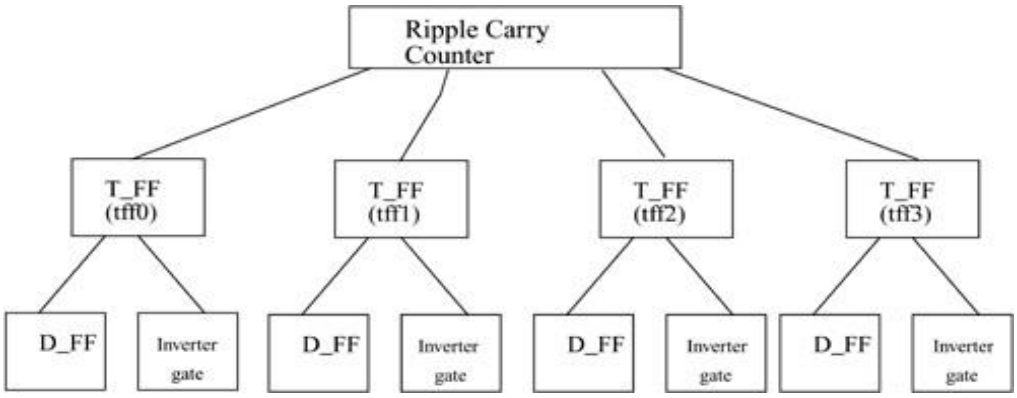


Figure 1.6. Design Hierarchy

In a top-down design methodology, we first have to specify the functionality of the ripple carry counter, which is the top-level block. Then, we implement the counter with T_FFs. We build the T_FFs from the D_FF and an additional inverter gate. Thus, we break bigger blocks into smaller building sub-blocks until we decide that we cannot break up the blocks any further.

A bottom-up methodology flows in the opposite direction. We combine small building blocks and build bigger blocks; e.g., we could build D_FF from and/ or gates, or we could build a custom D_FF from transistors. Thus, the bottom-up flow meets the top-down flow at the level of the D_FF.

1.8 Modules

Verilog provides the concept of a module. A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design. In Verilog, a module is declared by the keyword module. A corresponding keyword endmodule must appear at the end of the module definition.

module <module_name> (<module_terminal_list>);
...
<module internals>
...
... endmodule

Specifically, the T-flipflop could be defined as a module as follows:
module T_FF (q, clock, reset);

```
.
.
<functionality of T-flipflop>
.
.
endmodule
```

Verilog is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The levels are defined below.

• **Behavioral or algorithmic level:** This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

• **Dataflow level:** At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

• **Gate level:** The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

• **Switch level:** This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.

Verilog allows the designer to mix and match all four levels of abstractions in a design.

1.9 Module Instances:

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

In Example of 4 bit ripple carry counter, the top-level block creates four instances from the T-flipflop (T_FF) template. Each T_FF instantiates a D_FF and an inverter gate. Each instance must be given a unique name. Note that // is used to denote single-line comments.

Example of Module Instantiation

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops. Interconnections areshown in figure 1.4 :4-bit Ripple Carry Counter.
module
ripple_carry_counter(q, clk, reset);
```

```
output [3:0] q; //I/O signals and vector declarations
input clk, reset; //I/O signals will be explained later.

//Four instances of the module T_FF are created. Each has a unique name.
//Each instance is passed a set of signals. Notice, that each instance is a copy of the module T_FF.
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

// Define the module T_FF. It instantiates a D-flipflop.
//We assumed that module D-flipflop is defined elsewhere in the design.
//Refer to Figure 1-5 for interconnections.

module T_FF(q, clk, reset);
output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.
not n1(d, q); // not gate is a Verilog primitive.
endmodule
```

In Verilog, it is illegal to nest modules. One module definition cannot contain another module definition within the module and endmodule statements.

Example below shows an illegal module nesting where the module T_FF is defined inside the module definition of the ripple carry counter.

Example for Illegal Module Nesting

```
// Define the top-level module called ripple carry counter.
// It is illegal to define the module T_FF inside this module.

module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;
```

module T_FF(q, clock, reset);// *ILLEGAL MODULE NESTING*

...

<module T_FF internals>

...

endmodule // *END OF ILLEGAL MODULE NESTING*

endmodule

1.20 Components of a Simulation

Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. We call such a block the stimulus block. It is good practice to keep the stimulus and design blocks separate. The stimulus block can be written in Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench. Different test benches can be used to thoroughly test the design block.

Two styles of stimulus application are possible. In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block. In Figure 1-7, the stimulus block becomes the top-level block. It manipulates signals clk and reset, and it checks and displays output signal q.

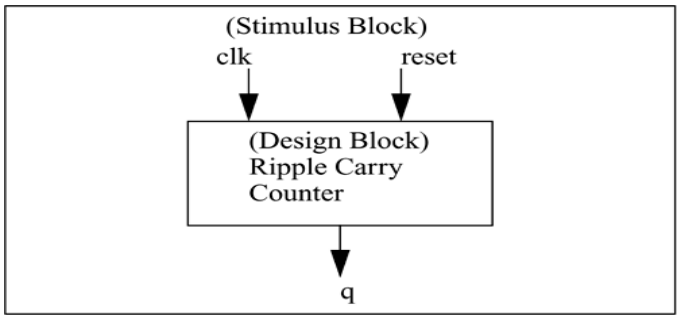


Figure 1.7. Stimulus Block Instantiates Design Block

The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top- level dummy module. The stimulus block interacts with the design block only through the interface. This style of applying stimulus is shown in Figure 1-8. The stimulus module drives the signals d_clk and d_reset, which are connected to the signals clk and reset in the design block.It also checks and displays signal c_q, which is connected to the signal q in the design block. The function of top-level block is simply to instantiate the design and stimulus blocks. Either stimulus style can be used effectively.

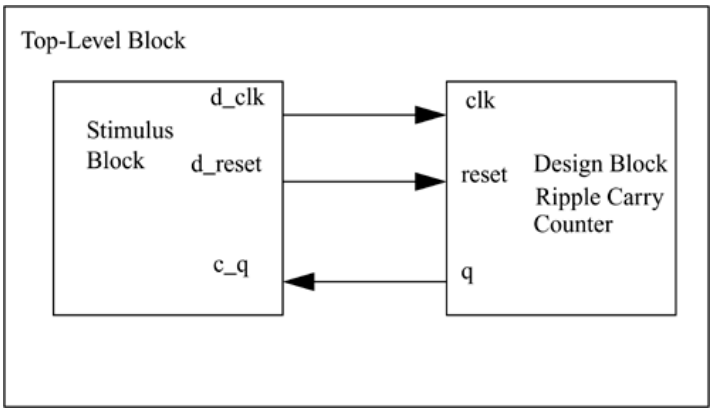


Figure 1.8. Stimulus Block and Design Block Instantiated in a dummy toplevel module

1.21 Example

Consider the example of 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.

1.21.1 Design Block

Consider a top-down design methodology. First, we write the Verilog description of the top-level design block which is the ripple carry counter.

Example of Ripple Carry Counter Top Block

```
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
```

In the above module, four instances of the module T_FF (T-flipflop) are used. Therefore, we must now define the internals of the module T_FF.

Example for Flipflop T_FF

```
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
```

Since T_FF instantiates D_FF, we must now define (Example 1-5) the internals of module D_FF. We assume asynchronous reset for the D_FFF.

Example for 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
```

All modules have been defined down to the lowest-level leaf cells in the design methodology. The design block is now complete.

1.21.2 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.

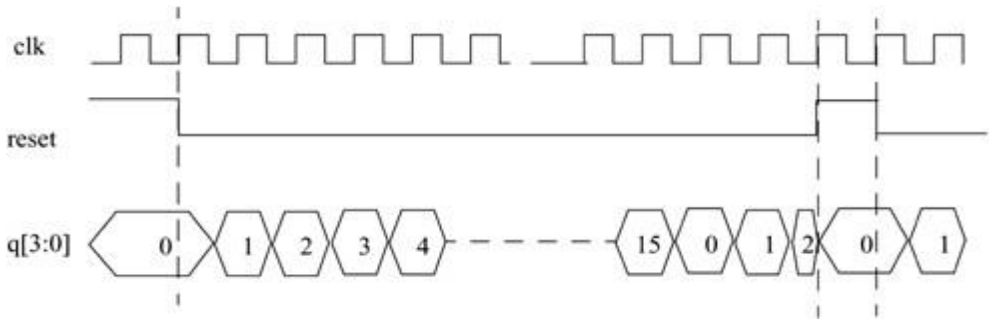


Figure 1.9: Stimulus and Output Waveforms

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

Once the stimulus block is completed, we are ready to run the simulation and verify the functional correctness of the design block.

The output obtained when stimulus and design blocks are simulated is shown in Example 1-7.

Example for an 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

1.22: Outcomes

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

1.23: Recommended questions

1. Discuss in brief about the evolution of CAD tools and HDLs used in digital system design.
2. Explain the typical VLSI IC design flow with the help of flow chart.
3. Discuss the trends in HDLs?
4. Why Verilog HDL has evolved as popular HDL in digital circuit design?
5. Explain the advantages of using HDLs over traditional schematic based design.
6. Describe the digital system design using hierarchical design methodologies with an example.
7. Apply the top-down design methodology to demonstrate the design of ripple carry counter.
8. Apply the bottom-up design methodology to demonstrate the design of 4-bit ripple carry adder.
9. Write Verilog HDL program to describe the 4-bit ripple carry counter.
10. Define Module and an Instance. Describe 4 different description styles of Verilog HDL.
11. Differentiate simulation and synthesis. What is stimulus?
12. Write test bench to test the 4-bit ripple carry counter.
13. Write a test bench to test the 4-bit ripple carry adder.

MODULE-2

BASIC CONCEPTS AND MODULES AND PORTS

2.1 : 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.

2.2 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.**

2.2.1 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.

2.2.2 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.

```
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 */
```

2.2.3 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

2.2.4 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.

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

Unsized 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.

2.2.5 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

2.2.6 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

2.2.7 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

2.3 Data Types

This section discusses the data types used in Verilog.

2.3.1 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.

Table 2-1. Value Levels

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.

Table 2-2. Strength Levels

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

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.

2.3.2 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.

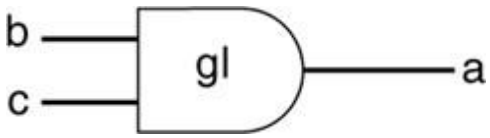


Figure 2.1. Example of Nets

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.

2.3.3 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 registers 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
```

2.3.4 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, default
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#], but the left number in the squared brackets is always the most significant bit of the vector. In the example shown above, bit 0 is the most significant bit 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 shouldalways 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]
```

2.3.5 Integer , Real, and Time Register Data Types

Integer, real, and time register data types are supported in Verilog.

Integer

An integer is a general purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. Although it is possible to use reg as a general-purpose variable, it is more convenient to declare an integer variable for purposes such as counting. The default width for an integer is the host-machine word size, which is implementation-specific but is at least 32 bits. Registers declared as data type reg store values as unsigned quantities, whereas integers store values as signed quantities.

```
integer counter; // general purpose variable used as a counter.
initial
counter = -1; // A negative one is stored in the counter
```

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.

```
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)
```

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

Arrays

Arrays are allowed in Verilog for reg, integer, time, real, realtime 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.

integer count[0:7]; // An array of 8 count variables

reg bool[31:0]; // Array of 32 one-bit boolean register variables time

chk_point[1:100]; // Array of 100 time checkpoint variables reg [4:0]

port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits wide

integer matrix[4:0][0:255]; // Two dimensional array of integers

reg [63:0] array_4d [15:0][7:0][7:0][255:0]; //Four dimensional array

wire [7:0] w_array2 [5:0]; // Declare an array of 8 bit vector wire

wire w_array1[7:0][5:0]; // Declare an array of single bit wires.

It is important not to confuse arrays with net or register vectors. A vector is a single element that is n-bits wide. On the other hand, arrays are multiple elements that are 1-bit or n-bits wide.

Examples of assignments to elements of arrays discussed above are shown below:

count[5] = 0; // Reset 5th element of array of count variables

chk_point[100] = 0; // Reset 100th time check point value

port_id[3] = 0; // Reset 3rd element (a 5-bit value) of port_id array.

matrix[1][0] = 33559; // Set value of element indexed by [1][0] to 33559

port_id = 0; // Illegal syntax - Attempt to write the entire array

matrix [1] = 0; // Illegal syntax - Attempt to write [1][0]..[1][255]

2.3.6 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.

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.

2.3.7 Parameters

Verilog allows constants to be defined in a module by the keyword parameter. Parameters cannot be used as variables. Parameter values for each module instance can be overridden individually at compile time. This allows the module instances to be customized. This aspect is discussed later. Parameter types and sizes can also be defined.

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

2.3.8 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 serve a special purpose in displaying strings, such as newline, tabs, and displaying argument values. Special characters can be displayed in strings only when they are preceded by escape characters, as shown in Table 2-3

Table 2-3. Special Characters

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

2.4 System Tasks and Compiler Directives

In this section, we introduce two special concepts used in Verilog: system tasks and compiler directives.

2.4.1 System Tasks

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 tasks 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.

Usage: $finish;

Examples of $stop and $finish are given below

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
```

2.4.2 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 themain verilog file design.v.
'include header.v
...
...
<Verilog code in file design.v>
...
...
```

2.5 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 any place in the module definition.

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.

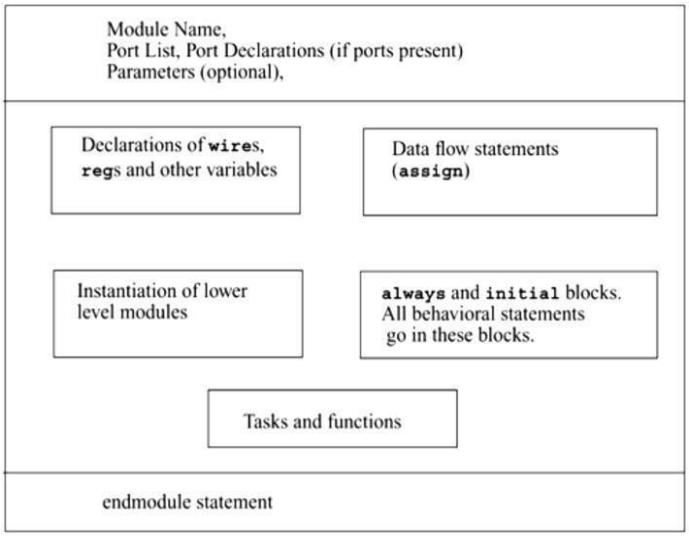


Figure 2.2.:Components of a Verilog Module

Consider a simple example of an SR latch, as shown in Figure 2.3

Figure 2-3. 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 Example.

Example of Components of SR Latch

```
// 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.

2.6 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.

2.6.1 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.

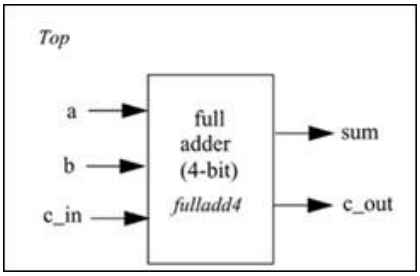


Figure 2-4. I/O Ports for Top and Full Adder

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
```

2.6.2 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.

Example for ANSI C Style Port Declaration Syntax

```
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
```

2.6.3 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

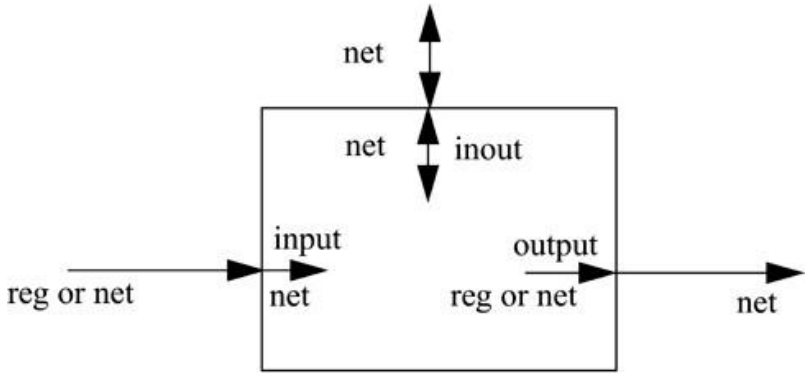


Figure 2-5. Port Connection Rules

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).

2.7 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

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. We could connect the ports by name in above example by instantiating the module fulladd4, as follows. Note that you can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.

```
// 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),);
```

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.

2.8 Hierarchical Names

Every module instance, signal, or variable is defined with an identifier. A particular identifier has a unique place in the design hierarchy. Hierarchical name referencing allows us to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots (".") for each level of hierarchy. Thus, any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier. The top-level module is called the root module because it is not instantiated anywhere. It is the starting point.

To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier.

Consider the simulation of SR latch Example. The design hierarchy is shown in Figure 2.6.

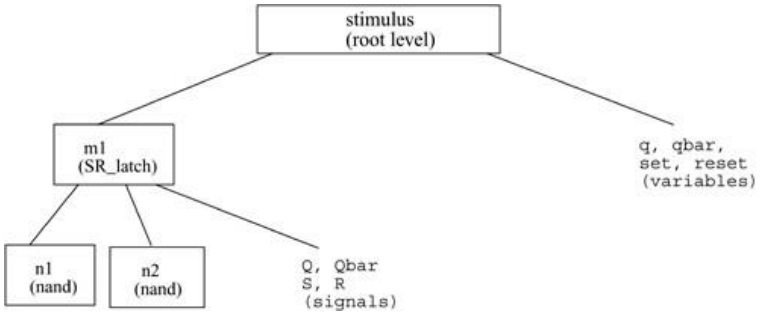


Figure 2-6. Design Hierarchy for SR Latch Simulation

For this simulation, stimulus is the top-level module. Since the top-level module is not instantiated anywhere, it is called the root module. The identifiers defined in this module are q, qbar, set, and reset. The root module instantiates m1, which is a module of type SR_latch. The module m1 instantiates nand gates n1 and n2. Q, Qbar, S, and R are port signals in instance m1. Hierarchical name referencing assigns a unique name to each identifier. To assign hierarchical names, use the module name for root module and instance names for all module instances below the root module.

Example

stimulus
stimulus.q
stimulus.qbar
timulus.set
stimulus.reset
stimulus.m1
stimulus.m1.Q
stimulus.m1.Qbar
stimulus m1.S
stimulus.m1.R
stimulus.n1
stimulus.n2

Each identifier in the design is uniquely specified by its hierarchical path name. To display the level of hierarchy, use the special character %m in the $display task.

2.9 : Outcomes

After completion of the module the students are able to:

- Understand the lexical conventions and different data types of verilog.
- Identify useful system tasks such as $display and $monitor and basic compiler directives.
- Understand different components of a Verilog module definition
- Understand the port connection rules and connection to external signals by ordered list and by name

2.10 : Recommended questions

1. Describe the lexical conventions used in Verilog HDL with examples.
2. Explain different data types of Verilog HDL with examples
3. What are system tasks and compiler directives?
4. What are the uses of $monitor, $display and $finish system tasks? Explain with examples.
5. Explain `define and `include compiler directives.
6. Explain the components of Verilog HDL module.
7. What are the components of SR latch? Write Verilog HDL module of SR latch.
8. Explain the different types of ports supported by Verilog HDL with examples.
9. Explain the port connection rules of Verilog HDL with examples.
10. How hierarchical names helps in addressing any identifier used in the design from any other level of hierarchy? Explain with examples.
11. What are the basic components of a module? Which components are mandatory?

MODULE -3

GATE LEVEL MODELING AND DATA FLOW MODELING

3.1 : 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 typ 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

3.2 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.**

3.2.1 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.

These gates are instantiated to build logic circuits in Verilog. Examples of gate instantiations are shown below. In Example 3-1, for all instances, OUT is connected to the output out, and IN1 and IN2 are connected to the two inputs i1 and i2 of the gate primitives. Note that the instance name does not need to be specified for primitives. This lets the designer instantiate hundreds of gates without giving them a name. More than two inputs can be specified in a gate instantiation. Gates with more than two inputs are

instantiated by simply adding more input ports in the gate instantiation. Verilog automatically instantiates the appropriate gate.

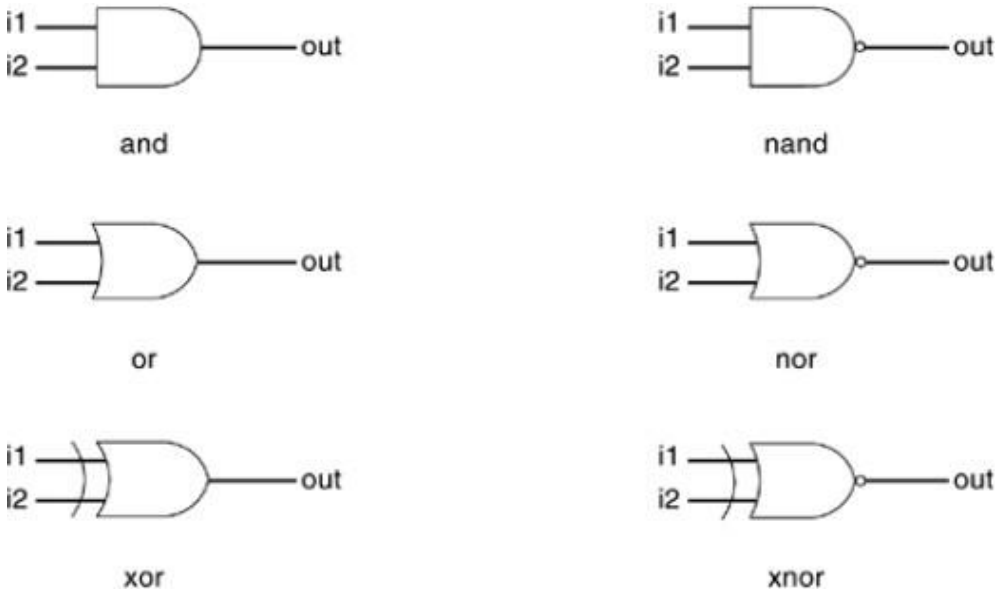


Figure 3-1. Basic Gates

Example 3-1 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
```

The truth tables for these gates define how outputs for the gates are computed from the inputs. Truth tables are defined assuming two inputs. The truth tables for these gates are shown in Table 3-1. Outputs of gates with more than two inputs are computed by applying the truth table iteratively.

Table 3-1. Truth Tables for And/Or

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

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

nor	i1: 0	1	x	z
i2: 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

3.2.2 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.

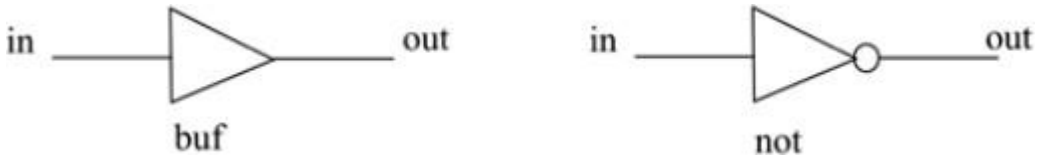


Figure 3-2. Buf/not Gates

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.

Example 3-2 Gate Instantiations of Buf/Not Gates

```
// 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.

bufif1 notif1

bufif0 notif0

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 3-3.

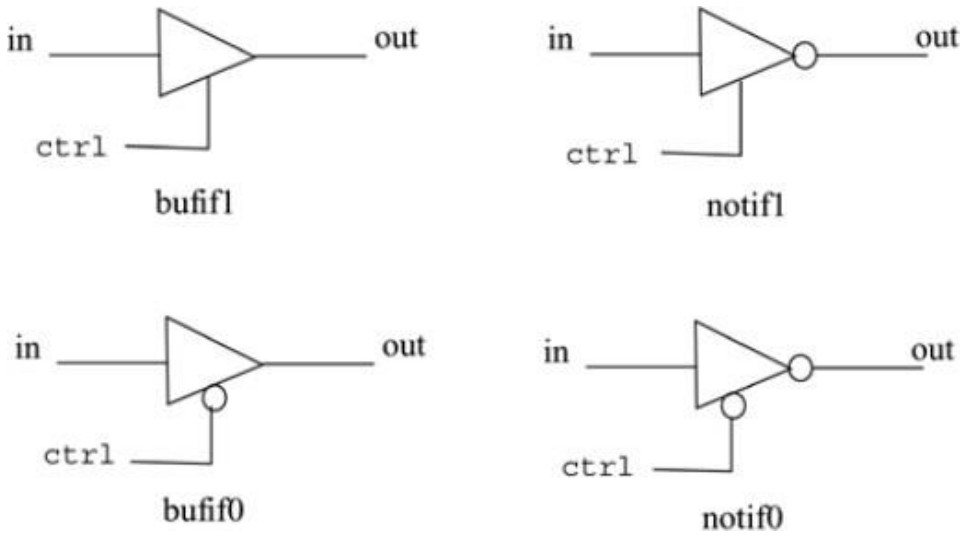


Figure 3-3. Bufif/notif Gates

The truth tables for these gates are shown in Table 3-3

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

These gates are used when a signal is to be driven only when the control signal is asserted. Such a situation is applicable when multiple drivers drive the signal. These drivers are designed to drive the signal on mutually exclusive control signals. Example 3-3 shows examples of instantiation of bufif and notif gates.

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);
```

3.2.3 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. Example3-4 shows an example of an array of instances.

Example 3-4 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]);
```

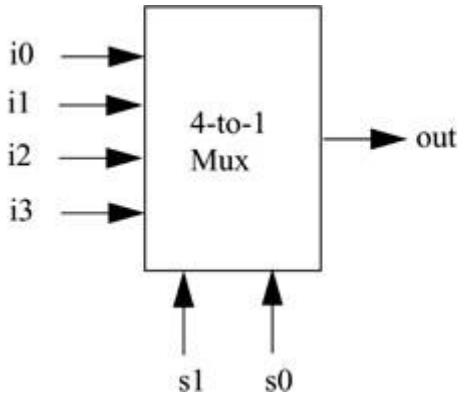
```
nand n_gate7(OUT[7], IN1[7], IN2[7]);
```

3.1.4 Examples

Having understood the various types of gates available in Verilog, consider the real examples that illustrates design of gate-level digital circuits.

Gate-level multiplexer

Consider the design of 4-to-1 multiplexer with 2 select signals. Multiplexers serve a useful purpose in logic design. They can connect two or more sources to a single destination. They can also be used to implement Boolean functions. We will assume for this example that signals s1 and s0 do not get the value x or z. The I/O diagram and the truth table for the multiplexer are shown in Figure 3-4. The I/O diagram will be useful in setting up the port list for the multiplexer.



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Figure 3-4. 4-to-1 Multiplexer

Implement the logic for the multiplexer using basic logic gates. The logic diagram for the multiplexer is shown in Figure 3-5.

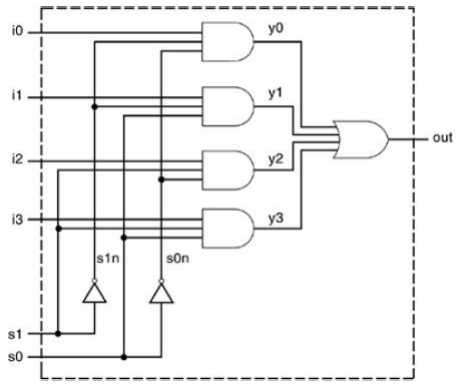


Figure 3-5. Logic Diagram for Multiplexer

The logic diagram has a one-to-one correspondence with the Verilog description. The Verilog description for the multiplexer is shown in Example 3-5. Two intermediate nets, s0n and s1n, are created; they are complements of input signals s1 and s0. Internal nets y0, y1, y2, y3 are also required. Note that instance names are not specified for primitive gates, not, and, and or. Instance names are optional for Verilog primitives but are mandatory for instances of user-defined modules.

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);
```

```
endmodule
```

This multiplexer can be tested with the stimulus shown in Example 3-6. The stimulus checks that each combination of select signals connects the appropriate input to the output. The signal OUTPUT is displayed one time unit after it changes. System task $monitor could also be used to display the signals when they change values.

Example 3-6 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
```

4-bit Ripple Carry Full Adder

Consider the design of a 4-bit full adder whose port list was defined in, List of Ports. We use primitive logic gates, and we apply stimulus to the 4-bit full adder to check functionality. For the sake of simplicity, we will implement a ripple carry adder. The basic building block is a 1-bit full adder. The mathematical equations for a 1-bit full adder are shown below.

sum = (a b cin)

cout = (a b) + cin (a b)

The logic diagram for a 1-bit full adder is shown in Figure 3-6.

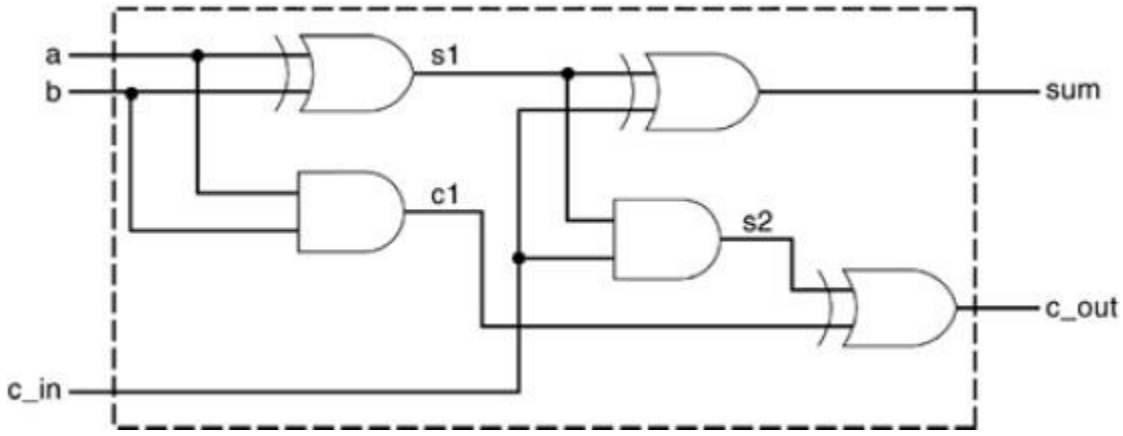


Figure 3-6. 1-bit Full Adder

This logic diagram for the 1-bit full adder is converted to a Verilog description, shown in Example 3-7.

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
```

A 4-bit ripple carry full adder can be constructed from four 1-bit full adders, as shown in Figure 3-7. Notice that fa0, fa1, fa2, and fa3 are instances of the module fulladd (1-bit full adder).

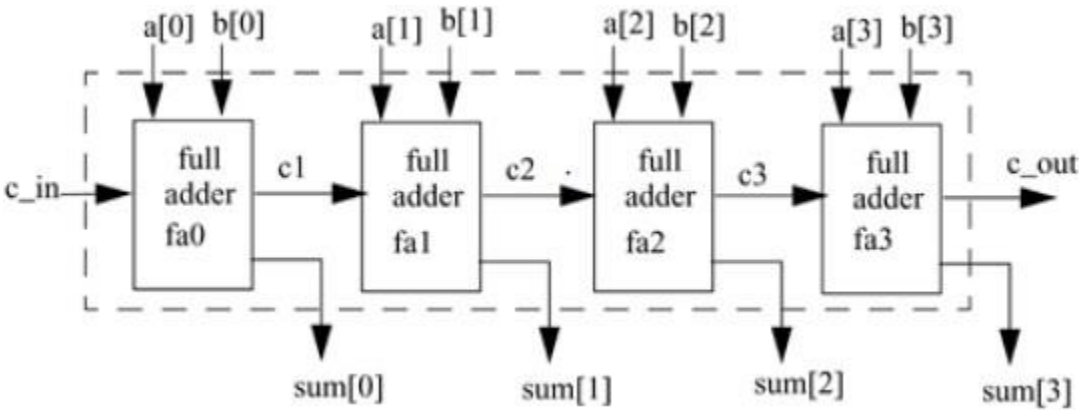


Figure 3-7. 4-bit Ripple Carry Full Adder

This structure can be translated to Verilog as shown in Example 3-8. Note that the port names used in a 1-bit full adder and a 4-bit full adder are the same but they represent different elements. The element sum in a 1-bit adder is a scalar quantity and the element sum in the 4-bit full adder is a 4-bit vector quantity. Verilog keeps names local to a module.

Names are not visible outside the module unless hierarchical name referencing is used. Also note that instance names must be specified when defined modules are instantiated, but when instantiating Verilog primitives, the instance names are optional.

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
```

Finally, the design must be checked by applying stimulus, as shown in Example 3-9. The module stimulus stimulates the 4-bit full adder by applying a few input combinations and monitors the results.

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
```

3.3 Gate Delays

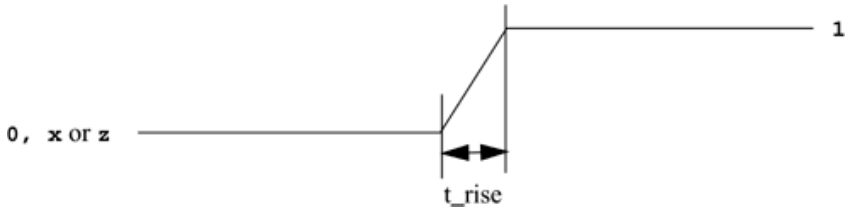
Until now, circuits are described without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

3.3.1 Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

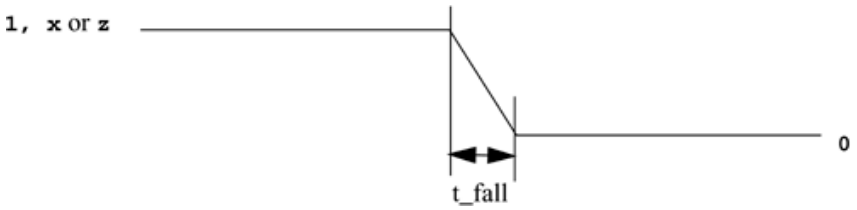
Rise delay

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
```

3.3.2 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 because of the IC fabrication process variations.

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.

Examples of min, typ, and max value specification for Verilog-XL are shown in Example3-11.

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
```

3.3.3 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 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.

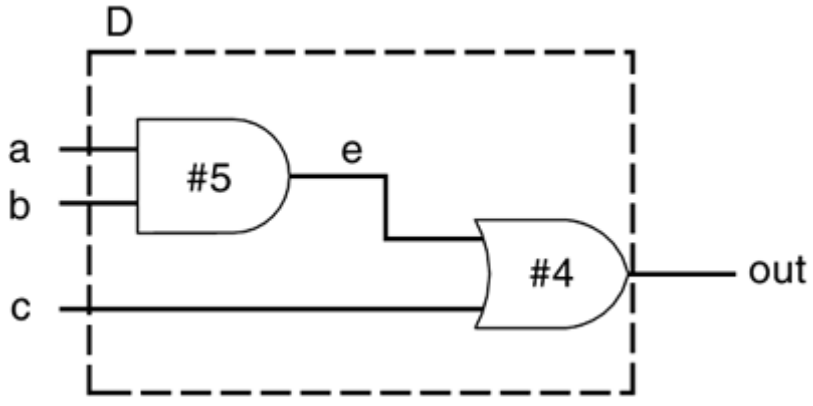


Figure 3-8. Module D

The module D is defined in Verilog as shown in Example 3-12.

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
```

This module is tested by the stimulus file shown in Example 3-13.

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.

1. The outputs E and OUT are initially unknown.

2. 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.

3. 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.

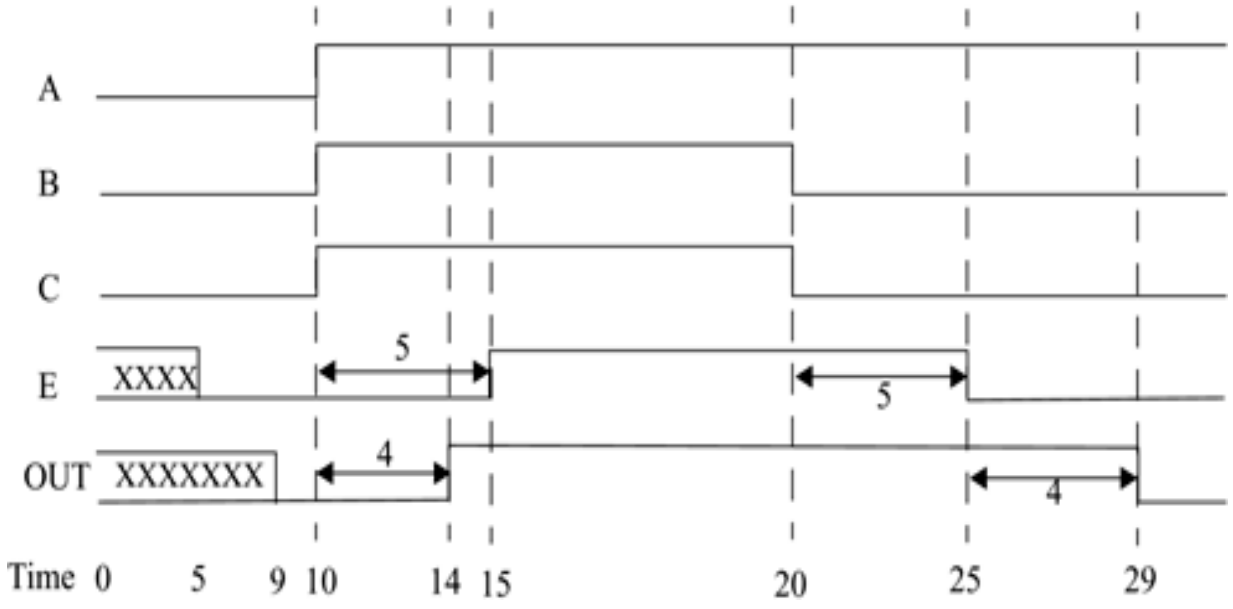


Figure 3-9. Waveforms for Delay Simulation of module D

It is a useful exercise to understand how the timing for each transition in the above waveform corresponds to the gate delays shown in Module D.

3.4 Dataflow 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.

3.4.1 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:

1. 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.

2. 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.

3. The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.

4. 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.

Examples of continuous assignments are shown below. Operators such as &, ^, |, {, } and + used in the examples, At this point, concentrate on how the assign statements are specified.

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;
```

3.4.2 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.

```
// Continuous assign. out is a net.

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
```

3.5 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.

3.5.1 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.

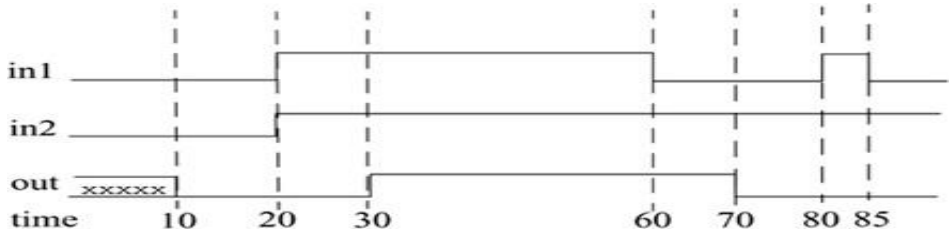


Figure 3-10. Waveforms for Delay Simulation

Inertial delays also apply to gate delays,

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;
```

3.5 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 a result.

```
// Examples of expressions. Combines operands and operators

a ^ b
```

```
addr1[20:17] + addr2[20:17]

in1 | in2
```

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

```
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

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. Table shows the complete listing of operator symbols classified by category.

.

Table 3-4 Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
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	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
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

Examples

A design can be represented in terms of gates, data flow, or a behavioral description. Consider the 4-to-1 multiplexer and 4-bit full adder described earlier. Previously, these designs were directly translated from the logic diagram into a gate-level Verilog description. Here, we describe the same designs in terms of data flow. We also discuss two additional examples: a 4-bit full adder using carry look ahead and a 4-bit counter using negative edge-triggered D-flip-flops.

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. Compare it with the gate-level description. 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

// Compare to gate-level model

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;

//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.

// Compare to gate-level model

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;
```

```
// Use nested conditional operator

assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;

endmodule
```

In the simulation of the multiplexer, the gate-level module can be substituted with the dataflow multiplexer modules described above. The stimulus module will not change. The simulation results will be identical. By encapsulating functionality inside a module, we can replace the gate-level module with a dataflow module without affecting the other modules in the simulation. This is a very powerful feature of Verilog.

4 bit Full Adder

The 4-bit full adder in, Examples, was designed by using gates; the logic diagram is shown in Figure 3.7. In this section, we write the dataflow description for the 4-bit adder. In gates, we had to first describe a 1-bit full adder. Then we built a 4-bit full ripple carry adder. We again illustrate two methods to describe a 4-bit full adder by means of dataflow statements.

Method 1: dataflow operators

A concise description of the adder is defined with the + and { } 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
```

If we substitute the gate-level 4-bit full adder with the dataflow 4-bit full adder, the rest of the modules will not change. The simulation results will be identical.

Method 2: full adder with carry lookahead

In ripple carry adders, the carry must propagate through the gate levels before the sum is available at the output terminals. An n-bit ripple carry adder will have 2n gate levels. The propagation time can be a limiting factor on the speed of the circuit. One of the most popular methods to reduce delay is to use a carry lookahead mechanism. Logic equations for implementing the carry lookahead mechanism can be found in any logic design book. The propagation delay is reduced to four gate levels, irrespective of the number of bits in the adder. The Verilog description for a carry lookahead adder. This module can be substituted in place of the full adder modules described before without changing any other component of the simulation. The simulation results will be unchanged.

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 c4, c3, c2, c1;

// 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];

// compute the g for each stage

assign g0 = a[0] & b[0],

g1 = a[1] & b[1],

g2 = a[2] & b[2],

g3 = a[3] & b[3];

// compute the carry for each stage

// Note that c_in is equivalent c0 in the arithmetic equation for
```

```
// carry lookahead computation
assign c1 = g0 | (p0 & c_in),
c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
(p3 & p2 & p1 & p0 & 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

Consider the design of a 4-bit ripple counter by using negative edge-triggered flipflops. This example was discussed at a very abstract level, Hierarchical Modeling Concepts. We design it using Verilog dataflow statements and test it with a stimulus module. The diagrams for the 4-bit ripple carry counter modules are show the counter being built with four T-flipflops.

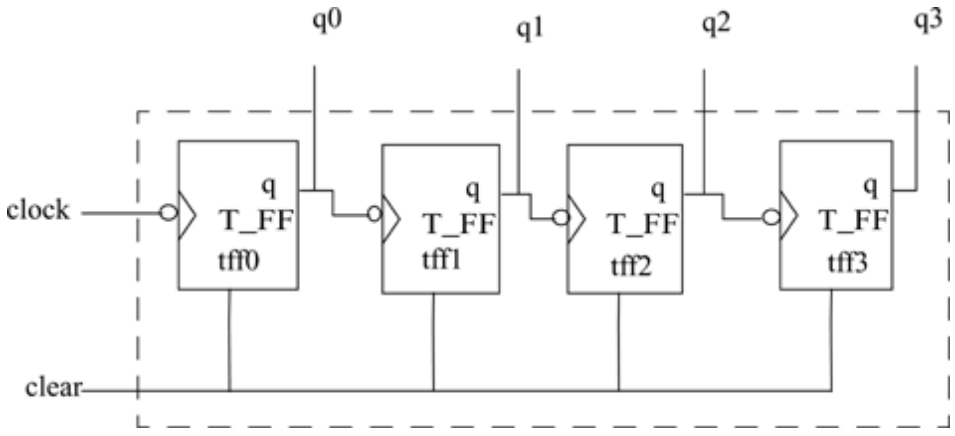


Figure 3.11 4 bit ripple counter

.

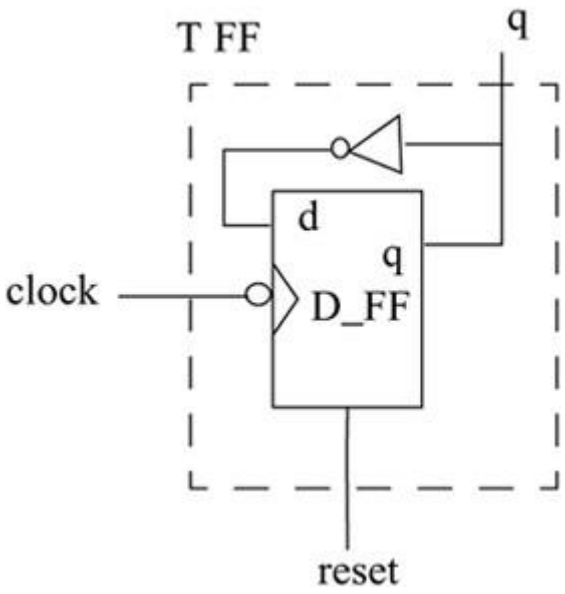


Figure 3.12 T-flipflop is built with one D-flipflop and an inverter gate

Figure 3.13 shows the D-flipflop constructed from basic logic gates.

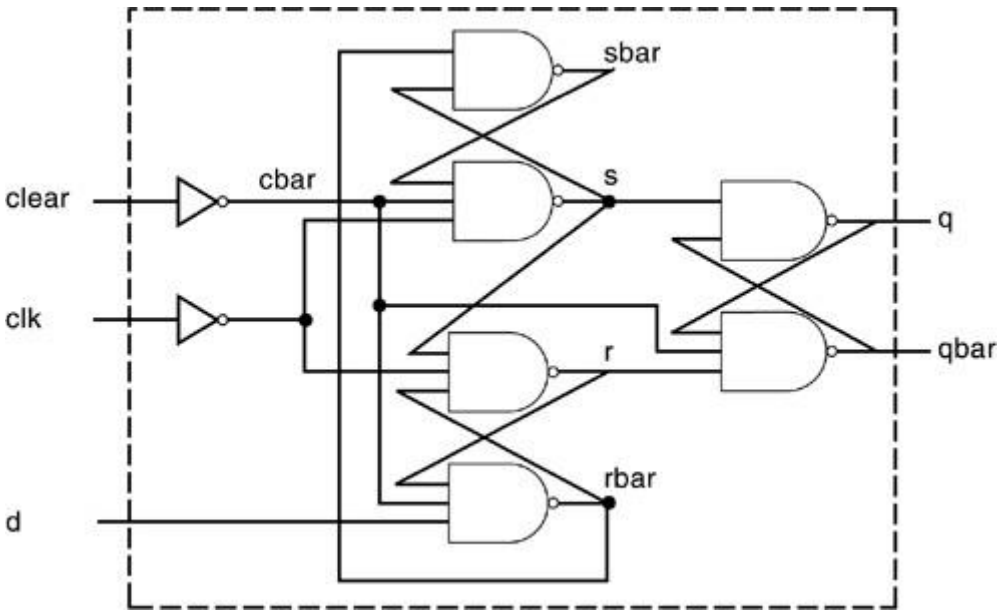


Figure 3.13 Negative Edge-Triggered D-flipflop with Clear

Given the above diagrams, we write the corresponding Verilog, using dataflow statements in a top-down fashion. First we design the module counter. The code is shown in. The code contains instantiation of four T_FF modules.

Example: Verilog Code for Ripple Counter

```
// 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
```

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
```

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
```

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
```

3.6: 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 typ 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

3.7: 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-bitadder

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.

MODULE -4

BEHAVIORAL MODELING

4.1 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.
- 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.

4.2 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 statements. Verilog is a concurrent programming language unlike the C programming language, which is sequential in nature.

Activity flows in Verilog run in parallel rather than in sequence. Each always and initial statement represents a separate activity flow in Verilog. Each activity flow starts at simulation time 0. The statements always and initial cannot be nested. The fundamental difference between the two statements is explained in the following sections

4.2.1 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 illustrates the use of the initial statement.

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

128

#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. The following subsections discussion how to initialize values using alternate shorthand syntax. The use of such shorthand syntax has the same effect as an initial block combined with a variable declaration.

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

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
```

4.2.2 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 = 20)
always
```

```
#10 clock = ~clock;

initial

#1000 $finish;

endmodule
```

In Example 4-5, the always statement starts at time 0 and executes the statement clock = ~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. C programmers might draw an analogy between the always block and an infinite loop.

But hardware designers tend to view it as a continuously repeated activity in a digital circuit starting from power on. The activity is stopped only by power off ($finish) or by an interrupt ($stop).

4.3 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 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.

4.3.1 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.

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 count = count + 1 is executed last. The simulation times at which the statements are executed are as follows:

• All statements x = 0 through reg_b = reg_a are executed at time 0

• Statement reg_a[2] = 0 at time = 15

• Statement reg_b[15:13] = {x, y, z} at time = 25

• Statement count = count + 1 at time = 25

• Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units

Note that for procedural assignments to registers, if the right-hand side has more bits than the register variable, the right-hand side is truncated to match the width of the register variable. The least significant bits are selected and the most significant bits are discarded. If the right-hand side has fewer bits, zeros are filled in the most significant bits of the register variable.

4.3.2 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 reg_b = reg_a are executed sequentially at time 0. Then the three nonblocking assignments are processed at the same simulation time.

1. reg_a[2] = 0 is scheduled to execute after 15 units (i.e., time = 15)

2. reg_b[15:13] = {x, y, z} is scheduled to execute after 10 time units (i.e., time = 10)

3. count = 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 behavior. However, it is recommended that blocking and non blocking assignments not be mixed in the same always block.

Application of non blocking assignments

Having described the behavior of non blocking assignments, it is important to understand why they are used in digital design. 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.

1. 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.

2. 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.

3. 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.

To understand the read and write operations further, consider Example 4-8, which is intended to swap the values of registers a and b at each positive edge of clock, using two concurrent always blocks.

Example 4-8 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;
135
//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-handside 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 shows how nonblocking assignments shown in Illustration 2 could be emulated using blocking assignments.

Example 4-9 Implementing Nonblocking Assignments 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.

4.4 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.

4.4.1 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. We used delay-based timing control statements when writing few modules in the preceding chapters but did not explain them in detail. In this section, we will discuss delay-based timing control statements. 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
#10 y = 1; // delay control with a number. Delay execution of
// y = 1 by 10 units
#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.
//Discussed in gate-level modeling chapter.
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.

Example 4-11 Intra-assignment 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;
temp_xz = x + z;
#5 y = 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
```

Note the difference between intra-assignment delays and regular delays. 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.

4.4.2 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 shows how the above example can be rewritten using the comma operator. Comma operators can also be applied to sensitivity lists that have edge-sensitive triggers.

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

//A positive edge triggered D flipflop with asynchronous falling

//reset can be modeled as shown below

always @(posedge clk, negedge reset) //Note use of comma operator

if(!reset)

q <=0;

else
q <=d;
```

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 shows an example of this special symbol for combinational logic sensitivity lists.

IEEE Standard Verilog Hardware Description Language document for details and restrictions on the @* and @(*) symbols.

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
```

4.4.3 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.

4.5 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");
```

4.6 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.

4.6.1 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
```

The case statement can also act like a many-to-one multiplexer. To understand this, let us model the 4-to-1 multiplexer, using case statements. The I/O ports are unchanged. Notice that an 8-to-1 or 16-to-1 multiplexer can also be easily implemented by case statements.

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
```

The case statement compares 0, 1, x, and z values in the expression and the alternative bit for bit. If the expression and the alternative are of unequal bit width, they are zero filled to match the bit width of the widest of the expression and the alternative. In Example 4- 20, we will define a 1-to-4 demultiplexer for which outputs are completely specified, that is, definitive results are provided even for x and z values on the select signal.

Example 4-20 Case Statement with x and z

```
module demultiplexer1_to_4 (out0, out1, out2, out3, in, s1, s0);

// Port declarations from the I/O diagram

output out0, out1, out2, out3;

reg out0, out1, out2, out3;

input in;

input s1, s0;

always @(s1 or s0 or in)

case ({s1, s0}) //Switch based on control signals

2'b00 : begin out0 = in; out1 = 1'bz; out2 = 1'bz; out3 =

1'bz; end

2'b01 : begin out0 = 1'bz; out1 = in; out2 = 1'bz; out3 =

1'bz; end

2'b10 : begin out0 = 1'bz; out1 = 1'bz; out2 = in; out3 =

1'bz; end

2'b11 : begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 =

in; end

//Account for unknown signals on select. If any select signal is x

//then outputs are x. If any select signal is z, outputs are z.

//If one is x and the other is z, x gets higher priority.

2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :

begin

out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx;

end

2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z :

begin
```

```
out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz;

end

default: $display("Unspecified control signals");

endcase

endmodule
```

In the demultiplexer shown above, multiple input signal combinations such as 2'bz0, 2'bz1, 2,bzz, 2'b0z, and 2'b1z that cause the same block to be executed are put together with a comma (,) symbol.

4.6.2 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 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.

4.7 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.

4.7.1 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.

Example 4-22 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
```

4.7.2 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
```

for loops are generally used when there is a fixed beginning and end to the loop. If the loop is simply looping on a certain condition, it is better to use the while loop.

4.7.3 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 in Example 4-24. Illustration 2 shows how to model a data buffer that latches data at the positive edge of clock for the next eight cycles after it receives a data start signal.

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;

150

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
```

4.7.4 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;
```

4.8 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.

4.8.1 Block Types

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.

We have used numerous examples of sequential blocks in this book. Two more examples of sequential blocks are given in Example 4-26. Statements in the sequential block execute in order. In Illustration 1, the final values are x = 0, y= 1, z = 1, w = 2 at simulation time 0. In Illustration 2, the final values are the same except that the simulation time is 35 at the end of the block.

Example 4-26 Sequential Blocks

```
//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
//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 4-27 Parallel Blocks

```
//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 provide a mechanism to execute statements in parallel. However, it is important to be careful with parallel blocks because of implicit race conditions that might arise if two statements that affect the same variable complete at the same time. Shown below is the parallel version of Illustration 1 from Example 4-26. Race conditions have been deliberately introduced in this example. All statements start at simulation time 0.

The order in which the statements will execute is not known. Variables z and w will get values 1 and 2 if x = 1'b0 and y = 1'b1 execute first. Variables z and w will get values 2'bxx and 2'bxx if x = 1'b0 and y = 1'b1 execute last. Thus, the result of z and w is nondeterministic and dependent on the simulator implementation. In simulation time, all statements in the fork-join block are executed at once. However, in reality, CPUs running simulations can execute only one statement at a time. Different simulators execute statements in different order. Thus, the race condition is a limitation of today's simulators, not of the fork-join block.

```
//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
```

The keyword fork can be viewed as splitting a single flow into independent flows. The keyword join can be seen as joining the independent flows back into a single flow. Independent flows operate concurrently.

4.8.2 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;

154

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.

4.9: 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 r outines 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.

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.

4.9.1 Differences between Tasks and Functions

Tasks and functions serve different purposes in Verilog. We discuss tasks and functions in greater detail in the following sections. However, first it is important to understand differences between tasks and functions, as outlined in Table 8-1.

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	Tasks may contain delay, event, or timing control

Functions	Tasks
timing control statements.	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 form always block, initial block and other task and functions.

4.9.2 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:

1. There are delay, timing, or event control constructs in the procedure.
2. The procedure has zero or more than one output arguments.
3. The procedure has no input arguments.

- I/O declaration use keywords input, output or input, 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.

4.9.3 Task Declaration and Invocation

Task declaration and task invocation syntax are as follows.

Example 9-1. Syntax for Tasks

```
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

We discuss two examples of tasks. The first example illustrates the use of input and output arguments in tasks. The second example models an asymmetric sequence generator that generates an asymmetric sequence on the clock signal.

Use of input and output arguments

Example 9-2 illustrates the use of **input** and **output** arguments in tasks. Consider a task called *bitwise_oper*, which computes the *bitwise and*, *bitwise or*, and *bitwise ex-or* of two 16-bit numbers. The two 16-bit numbers *a* and *b* are inputs and the three outputs are 16-bit numbers *ab_and*, *ab_or*, *ab_xor*. A parameter *delay* is also used in the task.

Example 9-2. Input and Output Arguments in Tasks

```
//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.

Another method of declaring arguments for tasks is the ANSI C style. Example 8-3 shows the *bitwise_oper* task defined with an ANSI C style argument declaration.

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
```

4.10 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.

4.11 Function Declaration and Invocation

The syntax for functions is follows:

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
```

There are some peculiarities of functions. When a function is declared, a register with name *function_identifer* is declared implicitly inside Verilog. The output of a function is passed back by setting the value of the register *function_identifer* 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_identifer* contains the output value. Also, functions *cannot* invoke other tasks. They can invoke only other functions.

4.12 Function Examples

We will discuss two examples. The first example models a parity calculator that returns a 1-bit value. The second example models a 32-bit left/right shift register that returns a 32-bit shifted value.

Parity calculation

Let us discuss a function that calculates the parity of a 32-bit *address* and returns the value. We assume even parity. Example 8-7 shows the definition and invocation of the function *calc_parity*.

Example 9-7. 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
```

Note that in the first invocation of *calc_parity*, the returned value was used to set the **`reg`** *parity*. In the second invocation, the value returned was directly used inside the **`$display`** task. Thus, the returned value is placed wherever the function was invoked.

Another method of declaring arguments for functions is the ANSI C style. Example 8-8 shows the *calc_parity* function defined with an ANSI C style argument declaration.

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
```

Left/right shifter

To illustrate how a range for the output value of a function can be specified, let us consider a function that shifts a 32-bit value to the left or right by one bit, based on a *control* signal. Example 8-9 shows the implementation of the left/right shifter.

Example 9-9. Left/Right Shifter

```
//Define a module that contains the function shift
module shifter;
```

```
...
//Left/right shifter
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr, right_addr;
reg control;

//Compute the right- and left-shifted values whenever
//a new address value appears
always @(addr)
begin
        //call the function defined below to do left and right shift.
         left_addr = shift(addr, `LEFT_SHIFT);
         right_addr = shift(addr, `RIGHT_SHIFT);
end
...
...
//define shift function. The output is a 32-bit value.
function [31:0] shift;
input [31:0] address;
input control;
begin
        //set the output value appropriately based on a control signal.
        shift = (control == `LEFT_SHIFT) ?(address << 1) : (address >> 1);

end
endfunction
...
...
endmodule
```

4.13 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 shows how an automatic function is defined to compute a factorial.

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
```

4.14 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 9-11 shows how a constant function can be used to compute the width of the address bus in a module.

Example 9-11. Constant Functions

```
//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
```

4.15 Summary

In this chapter, we discussed tasks and functions used in behavior Verilog modeling.

- *Tasks* and *functions* are used to define common Verilog functionality that is used at many places in the design. Tasks and functions help to make a module definition more readable by breaking it up into manageable subunits. Tasks and functions serve the same purpose in Verilog as subroutines do in C.
- Tasks can take any number of **`input`**, **`inout`**, or **`output`** arguments. Delay, event, or timing control constructs are permitted in tasks. Tasks can enable other tasks or functions.
- Re-entrant tasks defined with the keyword **`automatic`** allow each task call to operate in an independent space. Therefore, re-entrant tasks work correctly even with concurrent tasks calls.
- Functions are used when exactly one return value is required and at least one input argument is specified. Delay, event, or timing control constructs are not permitted in functions. Functions can invoke other functions but cannot invoke other tasks.
- A register with name as the function name is declared implicitly when a function is declared. The return value of the function is passed back in this register.
- Recursive functions defined with the keyword **`automatic`** allow each function call to operate in an independent space. Therefore, recursive or concurrent calls to such functions will work correctly.
- Tasks and functions are included in a design hierarchy and can be addressed by hierarchical name referencing.

Exercises

1: Define a **`function`** to calculate the *factorial* of a 4-bit number. The output is a 32-bit value. Invoke the function by using stimulus and check results.

2: Define a **`function`** to multiply two 4-bit numbers *a* and *b*. The output is an 8-bit value. Invoke the function by using stimulus and check results.

3: Define a **`function`** to design an 8-function ALU that takes two 4-bit numbers *a* and *b* and computes a 5-bit result *out* based on a 3-bit *select* signal. Ignore overflow or underflow bits.

Select Signal	Function Output
3'b000	a
3'b001	a + b
3'b010	a - b

3'b011	a / b
3'b100	a % 1 (remainder)
3'b101	a << 1
3'b110	a >> 1
3'b111	(a > b) (magnitude compare)

4: Define a **`task`** to compute the factorial of a 4-bit number. The output is a 32-bit value. The result is assigned to the output after a delay of 10 time units.

5: Define a **`task`** to compute even parity of a 16-bit number. The result is a 1-bit value that is assigned to the output after three positive edges of clock. (Hint: Use a **`repeat`** loop in the task.)

6: Using named events, tasks, and functions, design the traffic signal controller in Traffic Signal Controller on page 160.

4.11 Outcomes

After completion of the module the students are able to:

- Explain the significance of structured procedures always and initial in behavioral modeling.
- Define blocking and nonblocking procedural assignments.
- Understand delay-based timing control mechanism in behavioral modeling. Use regular delays, intra-assignment delays, and zero delays.
- Describe event-based timing control mechanism in behavioral modeling. Use regular event control, named event control, and event OR control.
- Use level-sensitive timing control mechanism in behavioral modeling.
- Explain conditional statements using if and else.
- Describe multiway branching, using case, casex, and casez statements.
- Understand looping statements such as while, for, repeat, and forever.
- Define sequential and parallel blocks.

4.12 : 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

MODULE -5

Useful Modeling Techniques

5.1 : 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.

5.2 : Procedural Continuous Assignments

We learned the basic features of Verilog in the preceding modules. In this module we will discuss additional features that enhance the Verilog language, making it powerful and flexible for modeling and analyzing a design.

We studied procedural assignments in Section 7.2, *Procedural 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 an useful extension to the regular procedural assignment statement.

5.3 : 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.A simple example is the negative edge-triggered D-flipflop with asynchronous reset that we modeled in Example 6-8. In Example 5-1, we now model the same D_FF, using **assign** and **deassign** statements.

Example 5-1. 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
```

In Example 5-1, we overrode the assignment on *q* and *qbar* and assigned new values to them when the *reset* signal went high. The register variables retain the continuously assigned value after the **deassign** until they are changed by a future procedural assignment. The **assign** and **deassign** constructs are now considered to be a bad coding style and it is recommended that alternative styles be used in Verilog HDL code.

5.4 : 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.

5.5: 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

Parameter values can be changed in any module instance in the design with the keyword **defparam**. The hierarchical name of the module instance can be used to override parameter values. Consider Example 9-2, which uses **defparam** to override the parameter values in module instances.

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*.

Example 5-3. ANSI C Style Parameter Declaration

```
//Define a module hello_world
module hello_world #(parameter id_num = 0) ;//ANSI C Style Parameter

initial //display the module identification number
      $display("Displaying hello_world id number = %d", id_num);
endmodule
```

Module_Instance Parameter Values

Parameter values can be overridden when a module is instantiated. To illustrate this, we will use Example 5-2 and modify it a bit. The new parameter values are passed during module instantiation. The top-level module can pass parameters to the instances *w1* and *w2*, as shown below. Notice that **defparam** is not needed. The simulation output will be identical to the output obtained with the **defparam** statement.

```
//define top-level module
module top;

//instantiate two hello_world modules; pass new parameter values
//Parameter value assignment by ordered list
hello_world #(1) w1; //pass value 1 to module w1

//Parameter value assignment by name
hello_world #(.id_num(2)) w2; //pass value 2 to id_num parameter
                              //for module w2

endmodule
```

If multiple parameters are defined in the module, during module instantiation, they can be overridden by specifying the new values in the same order as the parameter declarations in the module. If an overriding value is not specified, the default parameter declaration values are taken. Alternately, one can override specific values by naming the parameters and the corresponding values. This is called parameter value assignment by name. Consider Example 9-4.

Example 9-4. Module Instance Parameter Values

```
//define module with delays
```

```
module bus_master;
parameter delay1 = 2;
parameter delay2 = 3;
parameter delay3 = 7;
...
<module internals>
...
endmodule

//top-level module; instantiates two bus_master modules
module top;

//Instantiate the modules with new delay values

//Parameter value assignment by ordered list
bus_master #(4, 5, 6) b1(); //b1: delay1 = 4, delay2 = 5, delay3 = 6
bus_master #(9, 4) b2(); //b2: delay1 = 9, delay2 = 4, delay3 = 7(default)

//Parameter value assignment by name
bus_master #(.delay2(4), delay3(7)) b3(); //b2: delay2 = 4, delay3 = 7
                                          //delay1=2 (default)
// It is recommended to use the parameter value assignment by name
// This minimizes the chance of error and parameters can be added
// or deleted without worrying about the order.

endmodule
```

Module-instance parameter value assignment is a very useful method used to override parameter values and to customize module instances.

5.6 : 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.

Consider Example 5-6, which illustrates conditional execution with **$test$plusargs**.

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
```

The variables are displayed only if the flag *DISPLAY_VAR* is set at run time. Flags can be set at run time by specifying the option *+DISPLAY_VAR* at run time.

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
```

5.5: Time Scales

Often, in a single simulation, delay values in one module need to be defined by using certain time unit, e.g., 1 µs, 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.

Example 5-8. Time Scales

```
//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 µs
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 µs  = 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 µs. 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*.

5.7 Useful System Tasks

In this section, we discuss the system tasks that are useful for a variety of purposes in Verilog. 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.

Example 9-9. 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
```

The advantage of multichannel descriptors is that it is possible to selectively write to multiple files at the same time. This is explained below in greater detail.

Writing to files

The system tasks **$fdisplay**, **$fmonitor**, **$fwrite**, and **$fstrobe** are used to write to files.[4] Note that these tasks are similar in syntax to regular system tasks **$display**, **$monitor**, etc., but they provide the additional capability of writing to files.

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. We will use the file descriptors defined in Example 9-9 to illustrate the use of the **$fdisplay** and **$fmonitor** tasks.

```
//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.

Example 5-10. 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. See Example 5-11.

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. All bits, bit-selects, or part-selects of the 32-bit random number can be used (see Example 5-12).

Example 5-12. 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
```

The random number generator is able to generate signed integers. Therefore, depending on the way the **$random** task is used, it can generate positive or negative integers. Example 9-13 shows an example of such generation.

Example 5-13. 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.
```

Note that the algorithm used by **$random** is standardized. Therefore, the same simulation test run on different simulators will generate consistent random patterns for the same seed value.

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>*. Example 9-14 illustrates how memory is initialized.

Example 9-14. 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

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. The use of VCD file in the debug process is shown in Figure 9-1.

Figure 9-1. 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.

Example 9-15. VCD File System Tasks

//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]

Postprocessing tools with graphical displays are commercially available and are now an important part of the simulation and debug process. For large simulation runs, it is very difficult for the designer to analyze the output from **$display** or **$monitor** statements. It is more intuitive to analyze results from graphical waveforms. Formats other than VCD have also emerged, but VCD still remains the popular dump format for Verilog simulators.

However, it is important to note that VCD files can become very large (hundreds of megabytes for large designs). It is important to selectively dump only those signals that need to be examined.

5.8 Logic Synthesis with Verilog HDL

Advances in logic synthesis have pushed HDLs into the forefront of digital design technology. Logic synthesis tools have cut design cycle times significantly. Designers can design at a high level of abstraction and thus reduce design time. In this chapter, we discuss logic synthesis with Verilog HDL. Synopsys synthesis products were used for the examples in this chapter, and results for individual examples may vary with synthesis tools. However, the concepts discussed in this chapter are general enough to be applied to any logic synthesis tool.[1] This chapter is intended to give the reader a basic understanding of the mechanics and issues involved in logic synthesis. It is not intended to be comprehensive material on logic synthesis. Detailed knowledge of logic synthesis can be obtained from reference manuals, logic synthesis books, and by attending training classes.

Learning Objectives

- 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.

5.9 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. Thus, the *designer's mind* was used as the logic synthesis tool, as illustrated in Figure 14-1

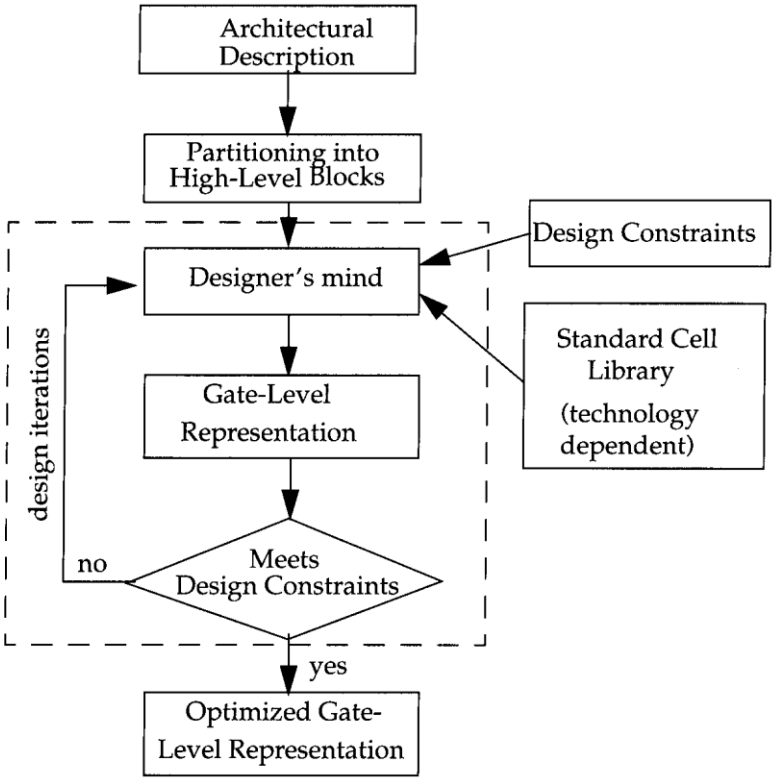


Figure 14-1. Designer's Mind as the Logic Synthesis Tool

The advent of *computer-aided logic synthesis tools* has automated the process of converting the high-level description to logic gates. Instead of trying to perform logic synthesis in their minds, designers can now concentrate on the architectural trade-offs, high-level description of the design, accurate design constraints, and optimization of cells in the standard cell library. These are fed to the computer-aided logic synthesis tool, which performs several iterations internally and generates the optimized gate-level description. Also, instead of drawing the high-level description on a screen or a piece of paper, designers describe the high-level design in terms of HDLs. Verilog HDL has become one of the popular HDLs for the writing of high-level descriptions. Figure 14-2 illustrates the process.

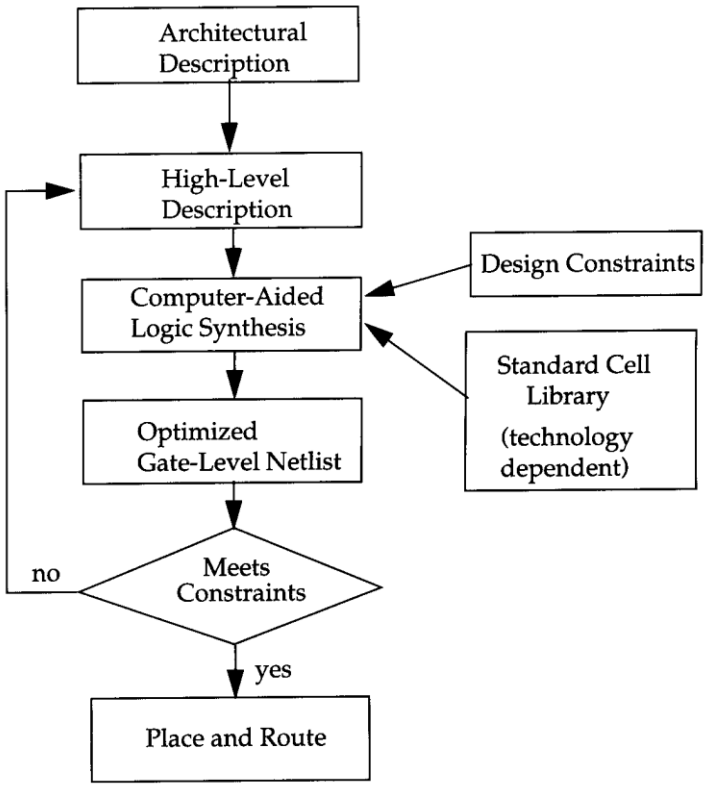


Figure 14-2. Basic Computer-Aided Logic Synthesis Process

Automated logic synthesis has significantly reduced time for conversion from high-level design representation to gates. This has allowed designers to spend more time on designing at a higher level of representation, because less time is required for converting the design to gates.

5.10 Impact of Logic Synthesis

Logic synthesis has revolutionized the digital design industry by significantly improving productivity and by reducing design cycle time. Before the days of automated logic synthesis, when designs were converted to gates manually, the design process had the following limitations:

- For large designs, manual conversion was prone to human error. A small gate missed somewhere could mean redesign of entire blocks.
- The designer could never be sure that the design constraints were going to be met until the gate-level implementation was completed and tested.
- A significant portion of the design cycle was dominated by the time taken to convert a high-level design into gates.

- If the gate-level design did not meet requirements, the turnaround time for redesign of blocks was very high.
- *What-if* scenarios were hard to verify. For example, the designer designed a block in gates that could run at a cycle time of 20 ns. If the designer wanted to find out whether the circuit could be optimized to run faster at 15 ns, the entire block had to be redesigned. Thus, redesign was needed to verify *what-if* scenarios.
- Each designer would implement design blocks differently. There was little consistency in design styles. For large designs, this could mean that smaller blocks were optimized, but the overall design was not optimal.
- If a bug was found in the final, gate-level design, this would sometimes require redesign of thousands of gates.
- Timing, area, and power dissipation in library cells are fabrication-technology specific. Thus if the company changed the IC fabrication vendor after the gate-level design was complete, this would mean redesign of the entire circuit and a possible change in design methodology.
- Design reuse was not possible. Designs were technology-specific, hard to port, and very difficult to reuse.

Automated logic synthesis tools addressed these problems as follows:

- High-level design is less prone to human error because designs are described at a higher level of abstraction.
- High-level design is done without significant concern about design constraints. Logic synthesis will convert a high-level design to a gate-level netlist and ensure that all constraints have been met. If not, the designer goes back, modifies the high-level design and repeats the process until a gate-level netlist that satisfies timing, area, and power constraints is obtained.
- Conversion from high-level design to gates is fast. With this improvement, design cycle times are shortened considerably. What took months before can now be done in hours or days.
- Turnaround time for redesign of blocks is shorter because changes are required only at the register-transfer level; then, the design is simply resynthesized to obtain the gate-level netlist.
- *What-if* scenarios are easy to verify. The high-level description does not change. The designer has merely to change the timing constraint from 20 ns to 15 ns and resynthesize the design to get the new gate-level netlist that is optimized to achieve a cycle time of 15 ns.
- Logic synthesis tools optimize the design as a whole. This removes the problem with varied designer styles for the different blocks in the design and suboptimal designs.

- If a bug is found in the gate-level design, the designer goes back and changes the high-level description to eliminate the bug. Then, the high-level description is again read into the logic synthesis tool to automatically generate a new gate-level description.
- Logic synthesis tools allow *technology-independent* design. A high-level description may be written without the IC fabrication technology in mind. Logic synthesis tools convert the design to gates, using cells in the standard cell library provided by an IC fabrication vendor. If the technology changes or the IC fabrication vendor changes, designers simply use logic synthesis to retarget the design to gates, using the standard cell library for the new technology.
- Design reuse is possible for technology-independent descriptions. For example, if the functionality of the I/O block in a microprocessor does not change, the RTL description of the I/O block can be reused in the design of derivative microprocessors. If the technology changes, the synthesis tool simply maps to the desired technology.

5.11 Verilog HDL Synthesis

For the purpose of logic synthesis, designs are currently written in an HDL at a *register transfer level (RTL)*. The term RTL 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

Not all constructs can be used when writing a description for a logic synthesis tool. In general, any construct that is used to define a cycle-by-cycle RTL description is acceptable to the logic synthesis tool. A list of constructs that are typically accepted by logic synthesis tools is given in Table 14-1. The capabilities of individual logic synthesis tools may vary. The constructs that are typically acceptable to logic synthesis tools are also shown.

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)

Remember that we are providing a cycle-by-cycle RTL description of the circuit. Hence, there are restrictions on the way these constructs are used for the logic synthesis tool. For example, the **`while`** and **`forever`** loops must be broken by a *@ (posedge clock*) or *@ (negedge clock)* statement to enforce cycle-by-cycle behavior and to prevent combinational feedback. Another restriction is that logic synthesis ignores all timing delays specified by **#**<*delay*> construct. Therefore, pre- and post-synthesis Verilog simulation results may not match. The designer must use a description style that eliminates these mismatches. Also, the **`initial`** construct is not supported by logic synthesis tools. Instead, the designer must use a reset mechanism to initialize the signals in the circuit.

It is recommended that all signal widths and variable widths be explicitly specified. Defining unsized variables can result in large, gate-level netlists because synthesis tools can infer unnecessary logic based on the variable definition.

5.12 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.

Table 14-2. Verilog HDL Operators for Logic Synthesis

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	multiply
	/	divide
	+	add
	–	subtract
	%	modulus
	+	unary plus
	–	unary minus
Logical	!	logical negation
	&&	logical and
	||	logical or
Relational	>	greater than
	<	less than
	>=	greater than or equal
	<=	less than or equal
Equality	==	equality
	!=	inequality
Bit-wise	~	bitwise negation
	&	bitwise and
	|	bitwise or
	^	bitwise ex-or
	^~ or ~^	bitwise ex-nor
Reduction	&	reduction and
	~&	reduction nand
	|	reduction or
	~|	

Operator Type	Operator Symbol	Operation Performed
	^	reduction nor
	^~ or ~^	reduction ex-or
		reduction ex-nor
Shift	>>	right shift
	<<	left shift
	>>>	arithmetic right shift
	<<<	arithmetic left shift
Concatenation	{ }	concatenation
Conditional	?:	conditional

Interpretation of a Few Verilog Constructs

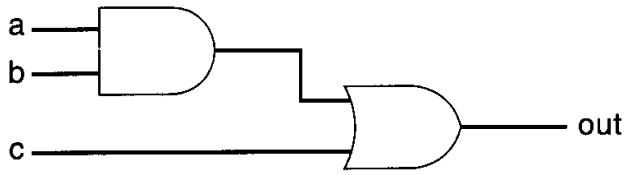
Having described the basic Verilog constructs, let us try to understand how logic synthesis tools frequently interpret these constructs and translate them to logic gates.

The assign statement

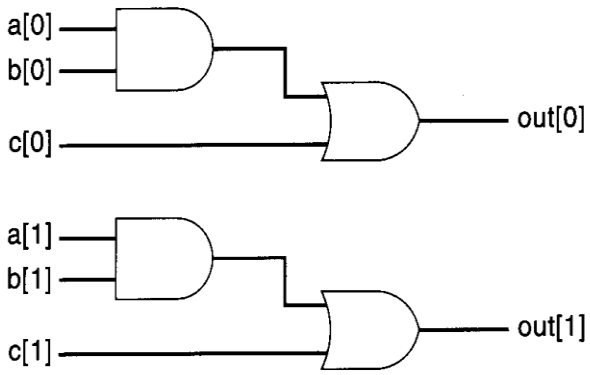
The **assign** construct is the most fundamental construct used to describe combinational logic at an RTL level. Given below is a logic expression that uses the **assign** statement.

```
assign out = (a & b) | c;
```

This will frequently translate to the following gate-level representation:



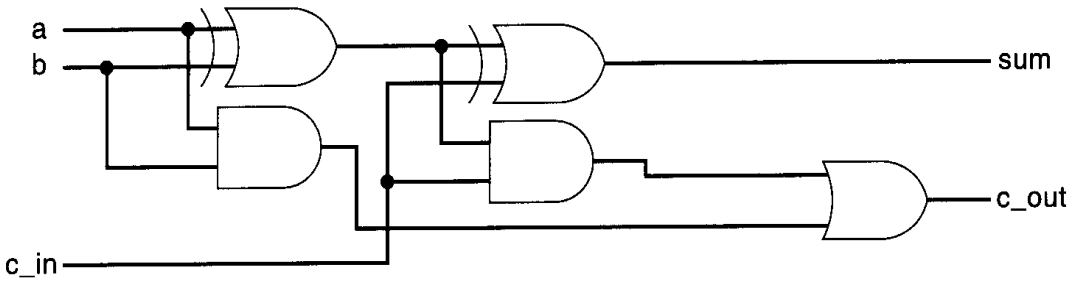
If *a, b, c,* and *out* are 2-bit vectors *[1:0]*, then the above **assign** statement will frequently translate to two identical circuits for each bit.



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.

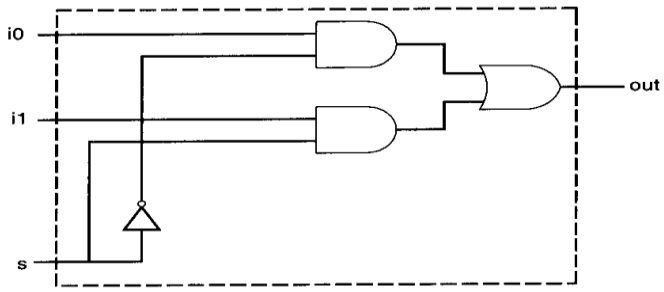


Figure 14-3 Multiplexer Description

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 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:

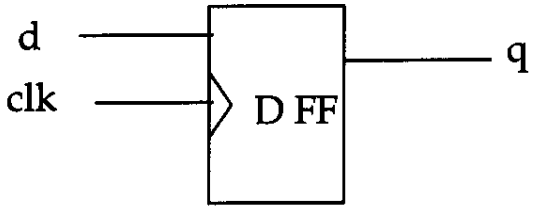
```
c = c_in;
for(i=0; i <=7; i = i + 1)
    {c, sum[i]} = a[i] + b[i] + c; // builds an 8-bit ripple adder
c_out = c;
```

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;
```

For combinational logic, the **always** statement must be triggered by a signal other than the *clk*, *reset*, or *preset*. For example, the following block will be interpreted as a 1-bit full adder:

```
always @(a or b or c_in)
            {c_out, sum} = a + b + c_in;
```

The function statement

Functions synthesize to combinational blocks with one output variable. The output might be scalar or vector. A 4-bit full adder is implemented as a function in the Verilog description below. The most significant bit of the function is used for the carry bit.

```
function [4:0] fulladd;
input [3:0] a, b;
input c_in;
begin
    fulladd = a + b + c_in; //bit 4 of fulladd for carry, bits[3:0] for sum.
end
endfunction
```

Synthesis Design Flow

Having understood how basic Verilog constructs are interpreted by the logic synthesis tool, let us now discuss the synthesis design flow from an RTL description to an optimized gate-level description.

RTL to Gates

To fully utilize the benefits of logic synthesis, the designer must first understand the flow from the high-level RTL description to a gate-level netlist. Figure 14-4 explains that flow.

RTL description

The designer describes the design at a high level by using RTL constructs. The designer spends time in functional verification to ensure that the RTL description functions correctly. After the functionality is verified, the RTL description is input to the logic synthesis tool.

Translation

The RTL description is converted by the logic synthesis tool to an unoptimized, intermediate, internal representation. This process is called *translation*. Translation is relatively simple and uses techniques similar to those discussed in Section 14.3.3, *Interpretation of a Few Verilog Constructs*. The translator understands the basic primitives and operators in the Verilog RTL description. Design constraints such as area, timing, and power are not considered in the translation process. At this point, the logic synthesis tool does a simple allocation of internal resources.

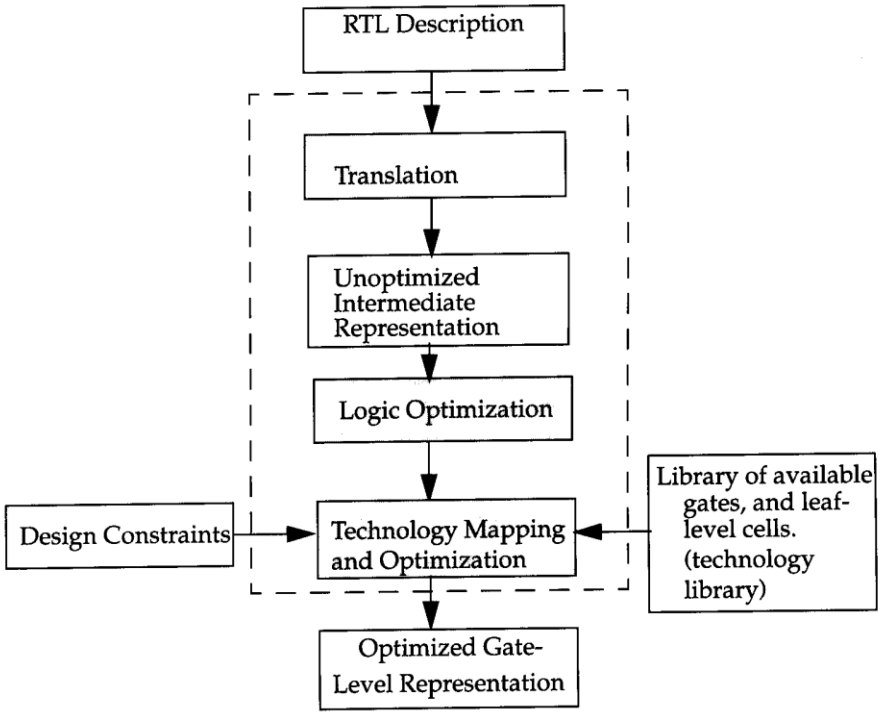


Figure 14-4. Logic Synthesis Flow from RTL to Gates

Unoptimized intermediate representation

The translation process yields an *unoptimized intermediate representation* of the design. The design is represented internally by the logic synthesis tool in terms of internal data structures. The unoptimized intermediate representation is incomprehensible to the user.

Logic optimization

The logic is now optimized to remove redundant logic. Various technology independent boolean logic optimization techniques are used. This process is called *logic optimization*. It is a very important step in logic synthesis, and it yields an *optimized internal representation* of the design.

Technology mapping and optimization

Until this step, the design description is independent of a specific *target technology*. In this step, the synthesis tool takes the internal representation and implements the representation in gates, using the cells provided in the technology library. In other words, the design is *mapped* to the desired *target technology*.

Suppose you want to get your IC chip fabricated at ABC Inc. ABC Inc. has 0.65 micron CMOS technology, which it calls *abc_100* technology. Then, abc_100 becomes the target technology. You must therefore implement your internal design representation in gates, using the cells provided in abc_100 technology library. This is called *technology mapping*. Also, the implementation should satisfy such design constraints as timing, area, and power. Some local optimizations are done to achieve the best results for the target technology. This is called *technology optimization* or *technology-dependent optimization*.

Technology library

The *technology library* contains *library cells* provided by ABC Inc. The term *standard cell library* used earlier in the chapter and the term *technology library* are identical and are used interchangeably.

To build a technology library, ABC Inc. decides the range of functionality to provide in its library cells. As discussed earlier, library cells can be basic logic gates or macro cells such as adders, ALUs, multiplexers, and special flip-flops. The library cells are the basic building blocks that ABC Inc. will use for IC fabrication. Physical layout of library cells is done first. Then, the area of each cell is computed from the cell layout. Next, modeling techniques are used to estimate the timing and power characteristics of each library cell. This process is called *cell characterization*.

Finally, each cell is described in a format that is understood by the synthesis tool. The cell description contains information about the following:

- Functionality of the cell
- Area of the cell layout
- Timing information about the cell
- Power information about the cell

A collection of these cells is called the *technology library*. The synthesis tool uses these cells to implement the design. The quality of results from synthesis tools will typically be dominated by the cells available in the technology library. If the choice of cells in the technology library is limited, the synthesis tool cannot do much in terms of optimization for timing, area, and power.

Design constraints

Design constraints typically include the following:

- ***Timing*—.** The circuit must meet certain timing requirements. An internal static timing analyzer checks timing.
- ***Area*—.** The area of the final layout must not exceed a limit.
- ***Power*—.** The power dissipation in the circuit must not exceed a threshold.

In general, there is an inverse relationship between area and timing constraints. For a given technology library, to optimize timing (faster circuits), the design has to be parallelized, which typically means that larger circuits have to be built. To build smaller circuits, designers must generally compromise on circuit speed. The inverse relationship is shown in Figure 14-5.

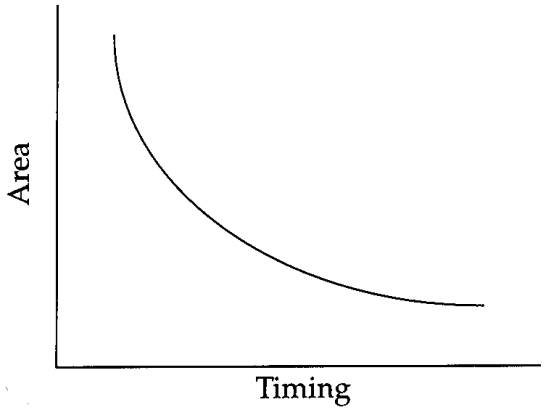


Figure 14-5. Area vs. Timing Trade-off

On top of design constraints, *operating environment factors*, such as input and output delays, drive strengths, and loads, will affect the optimization for the target technology. Operating environment factors must be input to the logic synthesis tool to ensure that circuits are optimized for the required operating environment.

Optimized gate-level description

After the technology mapping is complete, an optimized gate-level netlist described in terms of target technology components is produced. If this netlist meets the required constraints, it is handed to ABC Inc. for final layout. Otherwise, the designer modifies the RTL or reconstrains the design to achieve the desired results. This process is iterated until the netlist meets the required constraints. ABC Inc. will do the layout, do timing checks to ensure that the circuit meets required timing after layout, and then fabricate the IC chip for you.

There are three points to note about the synthesis flow.

1. For very high speed circuits like microprocessors, vendor technology libraries may yield nonoptimal results. Instead, design groups obtain information about the fabrication process used by the vendor, for example, 0.65 micron CMOS process, and build their own technology library components. Cell characterization is done by the designers. Discussion about building technology libraries and cell characterization is beyond the scope of this book.
2. Translation, logic optimization, and technology mapping are done *internally* in the logic synthesis tool and are not visible to the designer. The technology library is given to the designer. Once the technology is chosen, the designer can control only the input RTL description and design constraint specification. Thus, writing efficient RTL descriptions, specifying design constraints accurately, evaluating design trade-offs, and having a good technology library are very important to produce optimal digital circuits when using logic synthesis.
3. For submicron designs, interconnect delays are becoming a dominating factor in the overall delay. Therefore, as geometries shrink, in order to accurately model interconnect delays, synthesis tools will need to have a tighter link to layout, right at the RTL level. Timing analyzers built into synthesis tools will have to account for interconnect delays in the total delay calculation.

An Example of RTL-to-Gates

Let us discuss synthesis of a 4-bit magnitude comparator to understand each step in the synthesis flow. Steps of the synthesis flow such as translation, logic optimization, and technology mapping are not visible to us as designers. Therefore, we will concentrate on the components that are visible to the designer, such as the RTL description, technology library, design constraints, and the final, optimized, gate-level description.

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
```

Notice that the RTL description is very concise.

Technology library

We decide to use the 0.65 micron CMOS process called *abc_100* used by ABC Inc. to make our IC chip. ABC Inc. supplies a technology library for synthesis. The library contains the following library cells. The library cells are defined in a format understood by the synthesis tool.

```
//Library cells for abc_100 technology

VNAND//2-input nand gate
VAND//2-input and gate
VNOR//2-input nor gate
VOR//2-input or gate
VNOT//not gate
VBUF//buffer
NDFF//Negative edge triggered D flip-flop
PDFF//Positive edge triggered D flip-flop
```

Functionality, timing, area, and power dissipation information of each library cell are specified in the technology library.

Design constraints

According to the specification, the design should be as fast as possible for the target technology, abc_100. There are no area constraints. Thus, there is only one design constraint.

- Optimize the final circuit for fastest timing

Logic synthesis

The RTL description of the magnitude comparator is read by the logic synthesis tool. The design constraints and technology library for abc_100 are provided to the logic synthesis tool. The logic synthesis tool performs the necessary optimizations and produces a gate-level description optimized for abc_100 technology.

Final, Optimized, Gate-Level Description

The logic synthesis tool produces a final, gate-level description. The schematic for the gate-level circuit is shown in Figure 14-6.

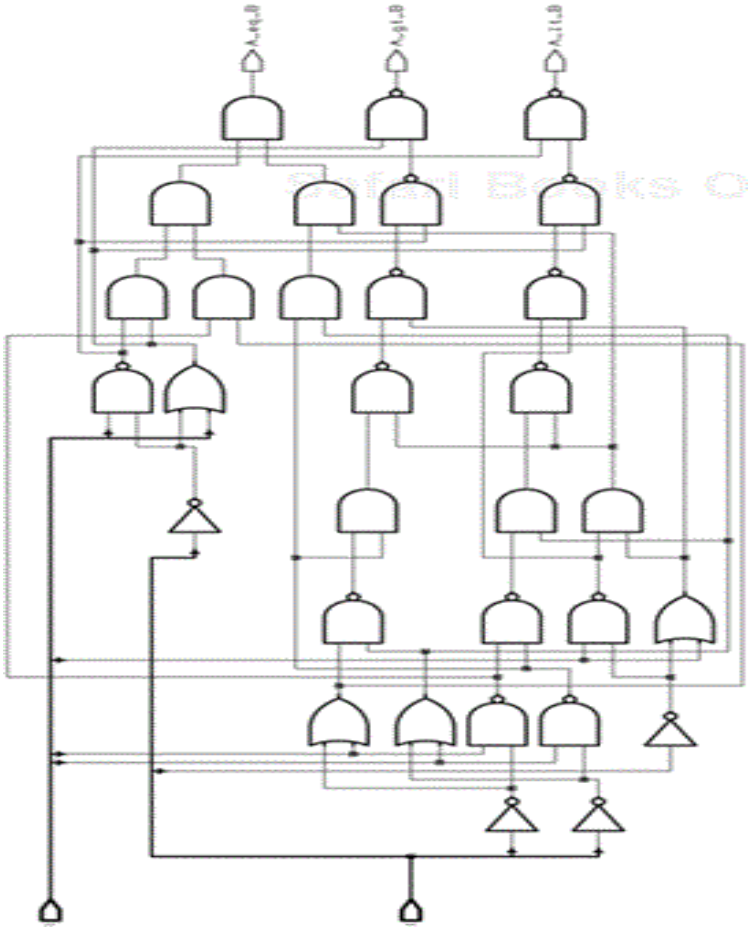


Figure 14-6. Gate-Level Schematic for the Magnitude Comparator

The gate-level Verilog description produced by the logic synthesis tool for the circuit is shown below. Ports are connected *by name*.

Example 14-2. Gate-Level Description for the Magnitude Comparator

```
module magnitude_comparator ( A_gt_B, A_lt_B, A_eq_B, A, B );
input  [3:0] A;
input  [3:0] B;
output A_gt_B, A_lt_B, A_eq_B;
    wire n60, n61, n62, n50, n63, n51, n64, n52, n65, n40, n53,
         n41, n54, n42, n55, n43, n56, n44, n57, n45, n58, n46,
         n59, n47, n48, n49, n38, n39;
    VAND U7 ( .in0(n48), .in1(n49), .out(n38) );
    VAND U8 ( .in0(n51), .in1(n52), .out(n50) );
    VAND U9 ( .in0(n54), .in1(n55), .out(n53) );
    VNOT U30 ( .in(A[2]), .out(n62) );
    VNOT U31 ( .in(A[1]), .out(n59) );
    VNOT U32 ( .in(A[0]), .out(n60) );
    VNAND U20 ( .in0(B[2]), .in1(n62), .out(n45) );
    VNAND U21 ( .in0(n61), .in1(n45), .out(n63) );
    VNAND U22 ( .in0(n63), .in1(n42), .out(n41) );
    VAND U10 ( .in0(n55), .in1(n52), .out(n47) );
    VOR U23 ( .in0(n60), .in1(B[0]), .out(n57) );
```

```
    VAND U11 ( .in0(n56), .in1(n57), .out(n49) );
    VNAND U24 ( .in0(n57), .in1(n52), .out(n54) );
    VAND U12 ( .in0(n40), .in1(n42), .out(n48) );
    VNAND U25 ( .in0(n53), .in1(n44), .out(n64) );
    VOR U13 ( .in0(n58), .in1(B[3]), .out(n42) );
    VOR U26 ( .in0(n62), .in1(B[2]), .out(n46) );
    VNAND U14 ( .in0(B[3]), .in1(n58), .out(n40) );
    VNAND U27 ( .in0(n64), .in1(n46), .out(n65) );
    VNAND U15 ( .in0(B[1]), .in1(n59), .out(n55) );
    VNAND U28 ( .in0(n65), .in1(n40), .out(n43) );
    VOR U16 ( .in0(n59), .in1(B[1]), .out(n52) );
    VNOT U29 ( .in(A[3]), .out(n58) );
    VNAND U17 ( .in0(B[0]), .in1(n60), .out(n56) );
    VNAND U18 ( .in0(n56), .in1(n55), .out(n51) );
    VNAND U19 ( .in0(n50), .in1(n44), .out(n61) );
    VAND U2 ( .in0(n38), .in1(n39), .out(A_eq_B) );
    VNAND U3 ( .in0(n40), .in1(n41), .out(A_lt_B) );
    VNAND U4 ( .in0(n42), .in1(n43), .out(A_gt_B) );
    VAND U5 ( .in0(n45), .in1(n46), .out(n44) );
    VAND U6 ( .in0(n47), .in1(n44), .out(n39) );
endmodule
```

If the designer decides to use another technology, say, *xyz_100* from XYZ Inc., because it is a better technology, the RTL description and design constraints do not change. Only the technology library changes. Thus, to map to a new technology, a logic synthesis tool simply reads the unchanged RTL description, unchanged design constraints, and new technology library and creates a new, optimized, gate-level netlist.

Note that if automated logic synthesis were not available, choosing a new technology would require the designer to redesign and reoptimize by hand the gate-level netlist in Example 14-2.

IC Fabrication

The gate-level netlist is verified for functionality and timing and then submitted to ABC Inc. ABC Inc. does the chip layout, checks that the post-layout circuit meets timing requirements, and then fabricates the IC chip, using abc_100 technology.

Verification of Gate-Level Netlist

The optimized gate-level netlist produced by the logic synthesis tool must be verified for functionality. Also, the synthesis tool may not always be able to meet both timing and area requirements if they are too stringent. Thus, a separate timing verification can be done on the gate-level netlist.

Functional Verification

Identical stimulus is run with the original RTL and synthesized gate-level descriptions of the design. The output is compared to find any mismatches. For the magnitude comparator, a sample stimulus file is shown below.

Example 14-3. Stimulus for Magnitude Comparator

```
module stimulus;

reg [3:0] A, B;
```

```
wire A_GT_B, A_LT_B, A_EQ_B;

//Instantiate the magnitude comparator
magnitude_comparator MC(A_GT_B, A_LT_B, A_EQ_B, A, B);

initial
  $monitor($time," A = %b, B = %b, A_GT_B = %b, A_LT_B = %b, A_EQ_B = %b",
        A, B, A_GT_B, A_LT_B, A_EQ_B);

//stimulate the magnitude comparator.
initial
begin
  A = 4'b1010; B = 4'b1001;
  # 10 A = 4'b1110; B = 4'b1111;
  # 10 A = 4'b0000; B = 4'b0000;
  # 10 A = 4'b1000; B = 4'b1100;
  # 10 A = 4'b0110; B = 4'b1110;
  # 10 A = 4'b1110; B = 4'b1110;
end

endmodule
```

The same stimulus is applied to both the RTL description in Example 14-1 and the synthesized gate-level description in Example 14-2, and the simulation output is compared for mismatches. However, there is an additional consideration. The gate-level description is in terms of library cells *VAND*, *VNAND*, etc. Verilog simulators do not understand the meaning of these cells. Thus, to simulate the gate-level description, a *simulation library*, *abc_100.v*, must be provided by ABC Inc. The simulation library must describe cells *VAND*, *VNAND*, etc., in terms of Verilog HDL primitives **and**, **nand**, etc. For example, the *VAND* cell will be defined in the simulation library as shown in Example 14-4.

Example 14-4. Simulation Library

```
//Simulation Library abc_100.v. Extremely simple. No timing checks.

module VAND (out, in0, in1);
input in0;
input in1;
output out;

//timing information, rise/fall and min:typ:max
specify
(in0 => out) = (0.260604:0.513000:0.955206, 0.255524:0.503000:0.936586);
(in1 => out) = (0.260604:0.513000:0.955206, 0.255524:0.503000:0.936586);
endspecify

//instantiate a Verilog HDL primitive
and (out, in0, in1);
endmodule
...
//All library cells will have corresponding module definitions
//in terms of Verilog primitives.
...
```

Stimulus is applied to the RTL description and the gate-level description. A typical invocation with a Verilog simulator is shown below.

```
//Apply stimulus to RTL description
> verilog stimulus.v mag_compare.v

//Apply stimulus to gate-level description.
//Include simulation library "abc_100.v" using the -v option
> verilog stimulus.v mag_compare.gv -v abc_100.v
```

The simulation output must be identical for the two simulations. In our case, the output is identical. For the example of the magnitude comparator, the output is shown in Example 14-5.

Example 14-5. Output from Simulation of Magnitude Comparator

```
 0 A = 1010, B = 1001, A_GT_B = 1, A_LT_B = 0, A_EQ_B = 0
10 A = 1110, B = 1111, A_GT_B = 0, A_LT_B = 1, A_EQ_B = 0
20 A = 0000, B = 0000, A_GT_B = 0, A_LT_B = 0, A_EQ_B = 1
30 A = 1000, B = 1100, A_GT_B = 0, A_LT_B = 1, A_EQ_B = 0
40 A = 0110, B = 1110, A_GT_B = 0, A_LT_B = 1, A_EQ_B = 0
50 A = 1110, B = 1110, A_GT_B = 0, A_LT_B = 0, A_EQ_B = 1
```

If the output is not identical, the designer needs to check for any potential bugs and rerun the whole flow until all bugs are eliminated.

Comparing simulation output of an RTL and a gate-level netlist is only a part of the functional verification process. Various techniques are used to ensure that the gate-level netlist produced by logic synthesis is functionally correct. One technique is to write a high-level architectural description in C++. The output obtained by executing the high-level architectural description is compared against the simulation output of the RTL or the gate-level description. Another technique called *equivalence checking* is also frequently used.

Timing verification

The gate-level netlist is typically checked for timing by use of *timing simulation* or by a *static timing verifier*. If any timing constraints are violated, the designer must either redesign part of the RTL or make trade-offs in design constraints for logic synthesis. The entire flow is iterated until timing requirements are met. Details of static timing verifiers are beyond the scope of this book.

Summary

In this chapter, we discussed the following aspects of Verilog:

- *Procedural continuous assignments* can be used to override the assignments on registers and nets. **assign** and **deassign** can override assignments on registers. **force** and **release** can override assignments on registers and nets. **assign** and **deassign** are used in the actual design. **force** and **release** are used for debugging.
- Parameters defined in a module can be overridden with the **defparam** statement or by passing a new value during *module instantiation*. During module instantiation, parameter values can be assigned by ordered list or by name. It is recommended to use parameter assignment by name.
- Compilation of parts of the design can be made conditional by using the **'ifdef**, **'ifndef**, **'elsif**, **'else**, and **'endif** directives. Compilation flags are defined at compile time by using the **`define** statement.
- Execution is made conditional in Verilog simulators by means of the **$test$plusargs** system task. The execution flags are defined at run time by +<*flag_name*>.
- Up to 30 files can be opened for writing in Verilog. Each file is assigned a bit in the *multichannel descriptor*. The multichannel descriptor concept can be used to write to multiple files. The *IEEE Standard Verilog Hardware Description Language* document describes more advanced ways of doing file I/O.
- Hierarchy can be displayed with the *%m* option in any display statement.
- *Strobing* is a way to display values at a certain time or event after all other statements in that time unit have executed.
- Random numbers can be generated with the system task **$random**. They are used for random test vector generation. $**random** task can generate both positive and negative numbers.
- Memory can be initialized from a data file. The data file contains addresses and data. Addresses can also be specified in memory initialization tasks.
- *Value Change Dump* is a popular format used by many designers for debugging with postprocessing tools. Verilog allows all or selected module variables to be dumped to the VCD file. Various system tasks are available for this purpose.

Exercises

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.

3: A 1-bit full adder FA is defined with gates and with delay parameters as shown below.

```
// Define a 1-bit full adder
module fulladd(sum, c_out, a, b, c_in);
parameter d_sum = 0, d_cout = 0;

// 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 #(d_sum) (sum, s1, c_in); //delay on output sum is d_sum
and (c2, s1, c_in);

or  #(d_cout) (c_out, c2, c1); //delay on output c_out is d_cout

endmodule
```

Define a 4-bit full adder *fulladd4* as shown in Example 5-8 on page 77, but pass the following parameter values to the instances, using the two methods discussed in the book:

Instance	Delay Values
fa0	d_sum=1, d_cout=1
fa1	d_sum=2, d_cout=2
fa2	d_sum=3, d_cout=3
fa3	d_sum=4, d_cout=4

1. Build the *fulladd4* module with **defparam** statements to change instance parameter values. Simulate the 4-bit full adder using the stimulus shown in Example 5-9 on page 77. Explain the effect of the full adder delays on the times when outputs of the adder appear. (Use delays of 20 instead of 5 used in this stimulus.)
2. Build the *fulladd4* with delay values passed to instances *fa0*, *fa1*, *fa2*, and *fa3* during instantiation. Resimulate the 4-bit adder, using the stimulus above. Check if the results are identical.

4: Create a design that uses the full adder example above. Use a conditional compilation (**`ifdef**). Compile the *fulladd4* with **defparam** statements if the text macro *DPARAM* is defined by the **`define** statement; otherwise, compile the *fulladd4* with module instance parameter values.

5: 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

//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
```

6: 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
```

7: Consider the 4-bit full adder in Example 6-4 on page 108. Write a stimulus file to do random testing of the full adder. Use a random number generator to generate a 32-bit random number. Pick bits *3:0* and apply them to input *a*; pick bits *7:4* and apply them to input *b*. Use bit *8* and apply it to *c_in*. Apply 20 random test vectors and observe the output.

8: Use the 8-byte memory initialization example in Example 9-14 on page 205. Modify the file to read data in *hexadecimal*. Write a new data file with the following addresses and data values. Unspecified locations are not initialized.

Location Address	Data
1	33
2	66
4	z0
5	0z
6	01

9: Write an **initial** block that controls the VCD file. The **initial** block must do the following:

- Set *myfile.dmp* as the output VCD file.
- Dump all variables two levels deep in module instance *top.a1.b1.c1*.
- Stop dumping to VCD at time 200.
- Start dumping to VCD at time 400.
- Stop dumping to VCD at time 500.
- Create a checkpoint. Dump the current value of all VCD variables to the dumpfile.