Verilog HDL

[As per Choice Based Credit System (CBCS) scheme]

B. E. (EC / TC) Choice Based Credit System (CBCS) and Outcome Based Education (OBE) SEMESTER – V			
Verilog HDL			
Course Code	**18EC56**	**IA Marks**	**40**
Number of Lecture Hours/Week	**03**	**Exam Marks**	**60**
Total Number of Lecture Hours	**40 (08 Hours per Module)**	**Exam Hours**	**03**
CREDITS– 03			

Course Learning Objectives:

- Learn different Verilog HDL constructs.
- Familiarize the different levels of abstraction in Verilog.
- Understand Verilog Tasks, Functions and Directives.
- Understand timing and delay Simulation.
- Understand the concept of logic synthesis and its impact in verification

Module 1	RBT Level
Overview of Digital Design with Verilog HDL: Evolution of CAD, emergence of HDLs, typical HDL-flow, why Verilog HDL?, trends in HDLs. **Hierarchical Modeling Concepts:** Top-down and bottom-up design methodology, differences between modules and module instances, parts of a simulation, design block, stimulus block.	L1,L2,L3
Module 2	
Basic Concepts: Lexical conventions, data types, system tasks, compiler directives. **Modules and Ports:** Module definition, port declaration, connecting ports, hierarchical name referencing.	L1,L2,L3
Module 3	
Gate-Level Modeling: Modeling using basic Verilog gate primitives, description of and/or and buf/not type gates, rise, fall and turn-off delays, min, max, and typical delays. **Dataflow Modeling:** Continuous assignments, delay specification, expressions, operators, operands, operator types.	L1,L2,L3
Module 4	
Behavioral Modeling: Structured procedures, initial and always, blocking and non-blocking statements, delay control, generate statement, event control, conditional statements, Multiway branching, loops, sequential and parallel blocks. **Tasks and Functions:** Differences between tasks and functions, declaration, invocation, automatic tasks and functions.	L1,L2,L3
Module 5	
Useful Modeling Techniques: Procedural continuous assignments, overriding parameters, conditional compilation and execution, useful system tasks. **Logic Synthesis with Verilog:** Logic Synthesis, Impact of logic synthesis, Verilog HDL Synthesis, Synthesis design flow, Verification of Gate-Level Netlist. **(Chapter 14 till 14.5 of Text).**	L1,L2,L3

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

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

Question paper pattern:

- Examination will be conducted for 100 marks with question paper containing 10 full questions, each of 20 marks.
- Each full question can have a maximum of 4 sub questions.
- There will be 2 full questions from each module covering all the topics of the module.
- Students will have to answer 5 full questions, selecting one full question from each module.

• The total marks will be proportionally reduced to 60 marks as SEE marks is 60.
Text Book: Samir Palnitkar, **"Verilog HDL: A Guide to Digital Design and Synthesis"**, Pearson Education, Second Edition.
Reference Books: 1. Donald E. Thomas, Philip R. Moorby, "The Verilog Hardware Description Language", Springer Science+Business Media, LLC, Fifth edition. 2. Michael D. Ciletti, "Advanced Digital Design with the Verilog HDL" Pearson (Prentice Hall), Second edition. 3. Padmanabhan, Tripura Sundari, "Design through Verilog HDL", Wiley, 2016 or earlier.

Module-1

Overview of Digital Design with Verilog HDL

Evolution of CAD, emergence of HDLs, typical HDL-flow, why Verilog HDL?, trends in HDLs. (Text1)

Hierarchical Modeling Concepts

Top-down and bottom-up design methodology, differences between modules and module instances, parts of a simulation, design block, stimulus block. (Text1)

Overview of Digital Design with Verilog HDL

Evolution of CAD:

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.

Emergence of HDLs:

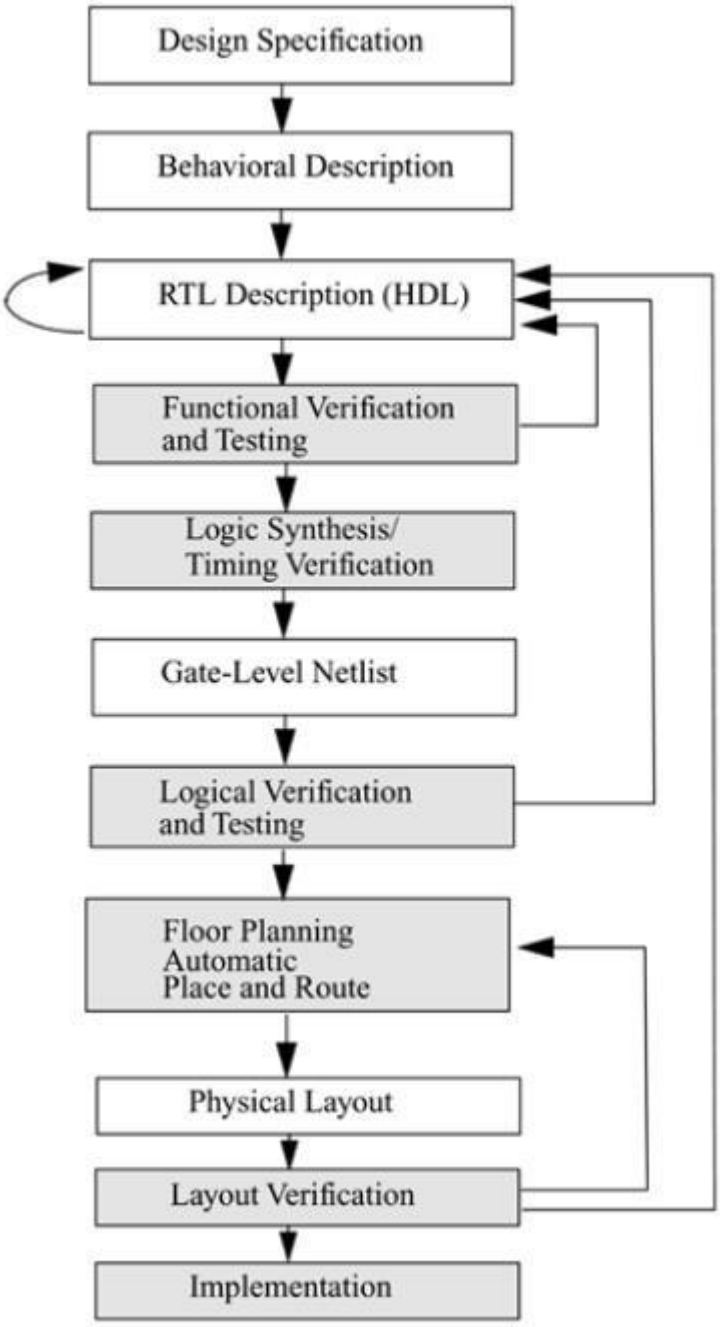
In the field of digital design, the complexity in designing a circuit gave birth to standard languages to describe digital circuits (ie. Hardware Description Languages - HDL). HDL is a Computer Aided design (CAD) tool for the modern design and synthesis of digital systems. HDLs were been used to model 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 synthesised 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 a IEEE standard - IEEE 1364-2001.

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

Typical HDL flow:



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

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 netlist 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.

Why Verilog HDLs?

HDLs have many advantages that helps in developing large digital circuits reaching the optimised 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 postlogic 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.

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.

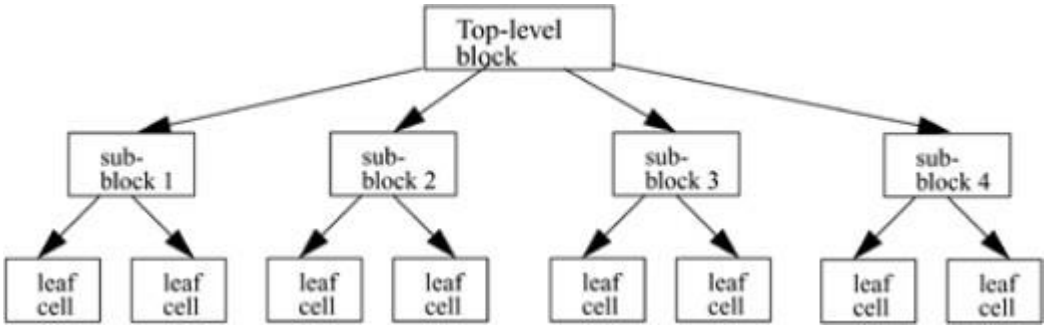
Digital circuits are designed in HDL at an RTL level, so that logic synthesis tools can create gate

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 optimise 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.

Design Methodologies:

There are two types of design methodologies: Top down and bottom-up.

Top-down design methodology:

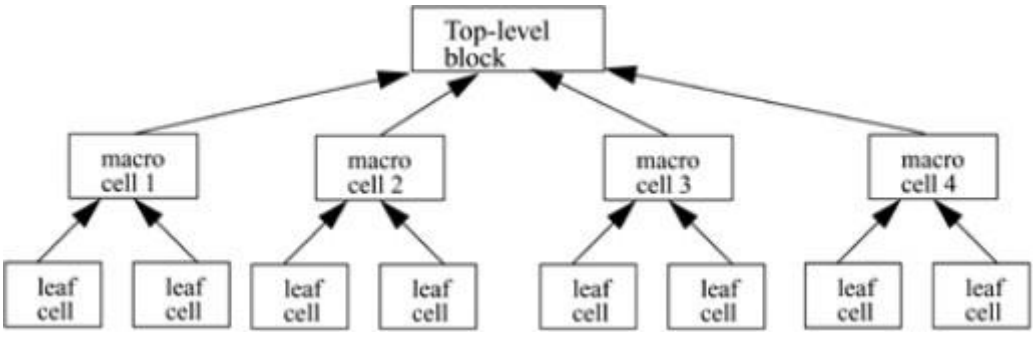


This designing approach allows early testing, easy change of different technologies, a well structures system design and offers many other advantages.

In this method, top-level block is defined and sub-blocks necessary to build the top-level block are identified.

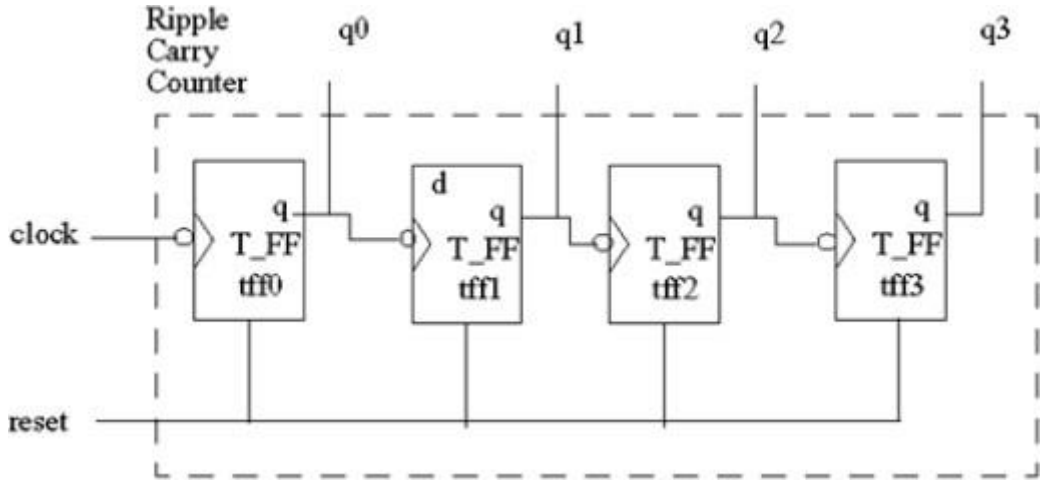
We further subdivide, sub-blocks until cells cannot be further divided, we call these cells as leaf cells.

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. 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.

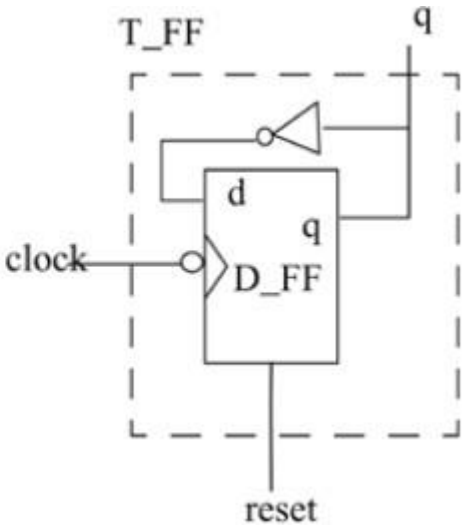
Illustration of hierarchical modelling concepts:

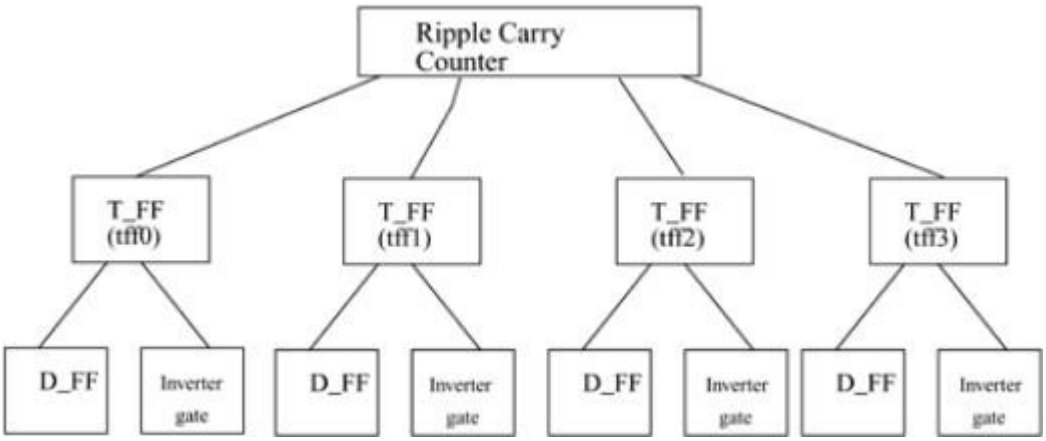


Example 1. 4 bit Ripple carry adder.

It is a circuit used to add two, 4 bit numbers can be designed using a adder that can add two, 1 bit numbers. The design is as shown in figure, using the concept of sub dividing the blocks. hence 4 bit Ripple carry Vader is built in a hierarchical fashion by using building blocks as follows.

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





Differences between modules and module instances

We shall now study what module in verily and what is all about module instantiation.

Module:

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.

Each module must have a module_name, which is the identifier for the module, and a module_terminal_list, which describes the input and output terminals of the module.

Ripple carry counter, T_FF, D_FF are examples of modules.

module <module_name> (<module_terminal_list>);

...

<module internals>

...

...

endmodule

Example:

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 module behaves identically with the external environment irrespective of the level of abstraction at which the module is described. The internals of the module are hidden from the environment.

Thus, the level of abstraction to describe a module can be changed without any change in the environment.

- 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.

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 2, 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: Module Instantiation

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops. Interconnections are // shown in Section 2.2, 4-bit Ripple Carry Counter.
module ripple_carry_counter(q, clk, reset);
output [3:0] q; //I/O signals and vector declarations
        //will be explained later.
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 2-4 for interconnections.
module T_FF(q, clk, reset);
//Declarations to be explained later
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. Explained later.
endmodule
```

In Verilog, it is illegal to nest modules. One module definition cannot contain another module definition within the module and endmodule statements. Instead, a module definition can incorporate copies of other modules by instantiating them.

Module definitions simply specify how the module will work, its internals, and its interface. Modules must be instantiated for use in the design.

Example 2-2 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
```

Parts 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, the stimulus block becomes the top-level block. It manipulates signals clk and reset, and it checks and displays output signal q. 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 . 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.

Design Block

We use a top-down design methodology. First, we write the Verilog description of the top-level design block

Example 2-3 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 (Example 2-4) the internals of the module T_FF, which was shown in Figure 2-4.

Example 2-4 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 2-5) the internals of 35

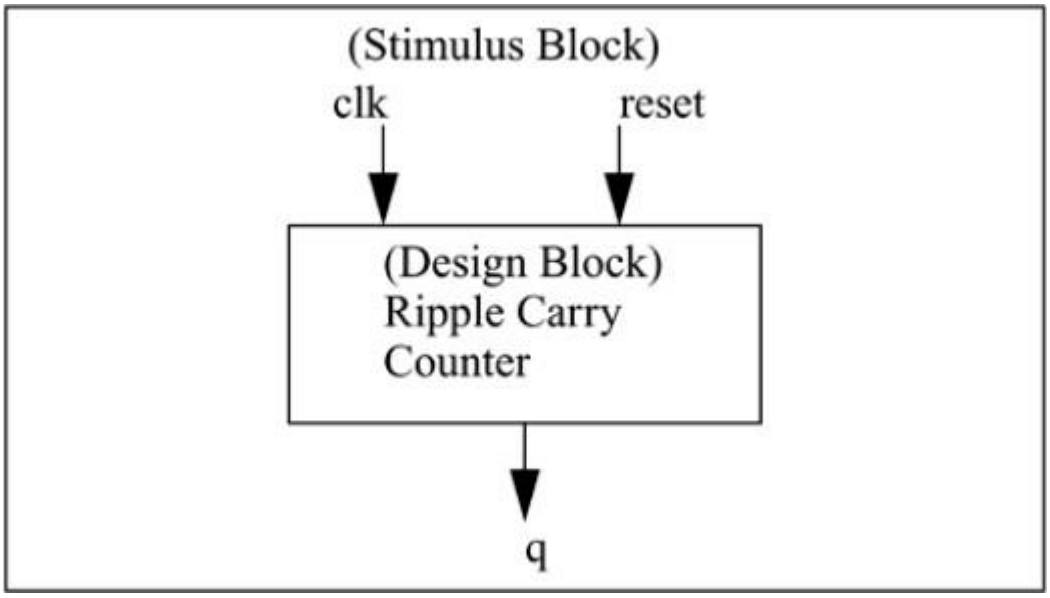
---

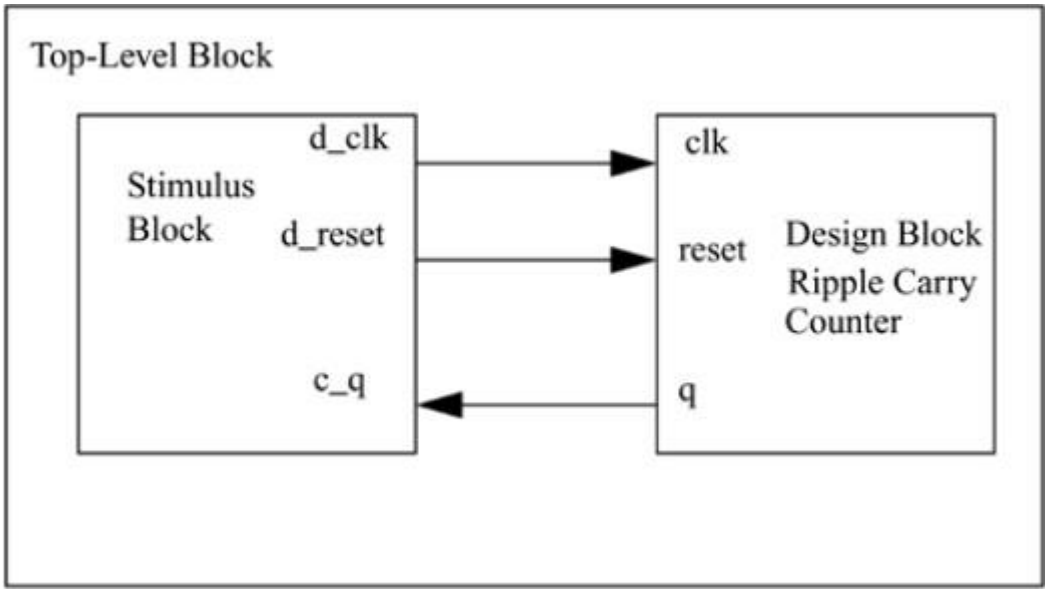
module D_FF. We assume asynchronous reset for the D_FFF.

Example 2-5 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.

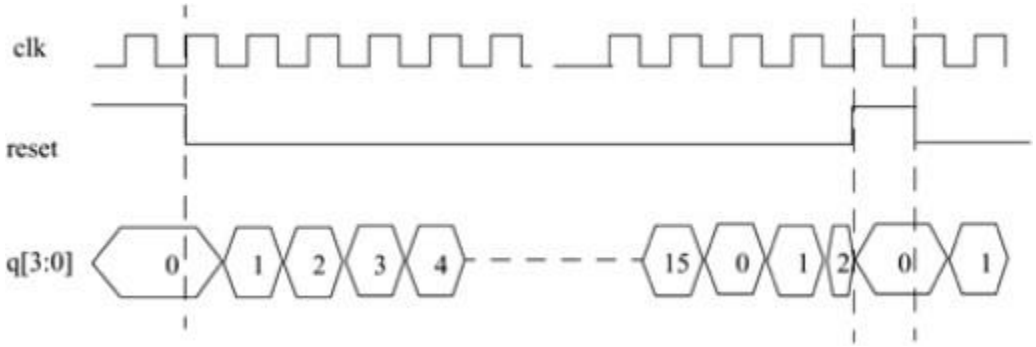


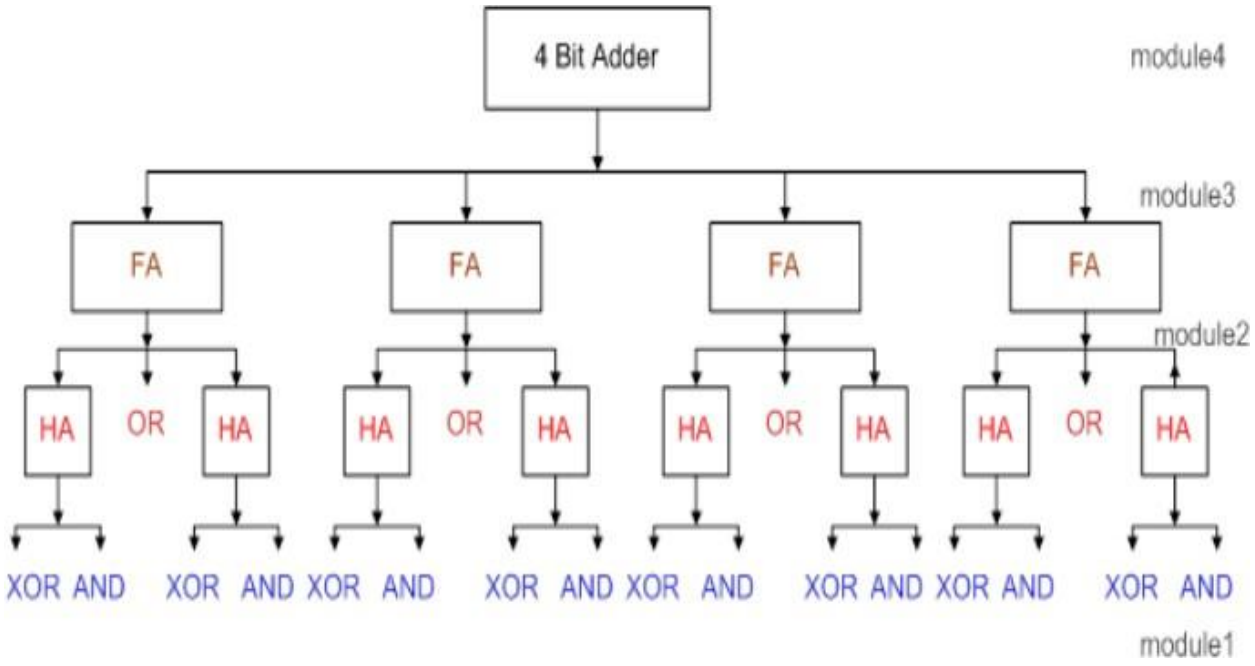


We must now write the stimulus block to check if the ripple carry counter design is functioning correctly. In this case, we must control the signals clk and reset so that the regular function of the ripple carry counter and the asynchronous reset mechanism are both tested. We use the waveforms shown in Figure 2-8 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.

We are now ready to write the stimulus block (see Example 2-6) that will create the above waveforms. We will use the stimulus style shown in Figure 2-6. Do not worry about the Verilog syntax at this point. Simply concentrate on how the design block is instantiated in the stimulus block.

Example 2-6 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 2-7.

Basic Concepts

Lexical conventions, data types, system tasks, compiler directives. (Text1)

Modules and Ports

Module definition, port declaration, connecting ports, hierarchical name referencing. (Text1)

Module definition, port declaration, connecting ports, hierarchical name referencing. (Text1)

Lexical conventions

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

Whitespace

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

Comments

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

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

Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```

Number Specification

There are two types of number specification in Verilog: sized and unsized.

Sized numbers

Sized numbers are represented as <size> '<base format> <number>.

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

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

'hc3 // This is a 32-bit

'o21 // This is a 32-bit

X or Z values

decimal number by default

hexadecimal number

octal number

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.

12'h13x // This is a 12-bit hex number; 4 least significant bits

unknown

6'hx // This is a 6-bit hex number

32'bz // This is a 32-bit high impedance number

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

Negative numbers

Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>. An optional signed specifier can be added for signed arithmetic.

-6'd3 // 8-bit negative number stored as 2's complement of 3

-6'sd3 // Used for performing signed integer math

4'd-2 // Illegal specification

Underscore characters and question marks

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

Strings

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

"Hello Verilog World" // is a string

"a / b" // is a string

3.1.6 Identifiers and Keywords

Keywords are special identifiers reserved to define the language constructs. Keywords are in lowercase. A list of all keywords in Verilog is contained in Appendix C, List of Keywords, System Tasks, and Compiler Directives.

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, which are explained later in the book).

reg value; // reg is a keyword; value is an identifier

input clk; // input is a keyword, clk is an identifier

3.1.7 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

Data Types

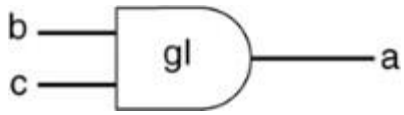
This section discusses the data types used in Verilog.

3.2.1 Value Set

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table 3-1.

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 Table 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 3-1 net a is connected to the output of and gate g1. Net a will continuously assume the value computed at the output of gate g1, which is b & c.

Nets are declared primarily with the keyword wire. Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably. The default value of a net is z (except the trireg net, which defaults to x). Nets get the output value of their drivers. If a net has no driver, it gets the value z.

wire a; // Declare net a for the above circuit

wire b,c; // Declare two wires b,c for the above circuit

wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.

Registers

Registers represent data storage elements. Registers retain value until another value is placed onto them. Do not confuse the term registers in Verilog with hardware registers built from edge-triggered flipflops in real circuits. 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 // this construct will be discussed later

begin

```
  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 3-2 Signed Register Declaration

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

Vectors

Nets or reg data types can be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit).

```
wire a; // scalar net variable, 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 should
 // always be on the left of a range specification
virtual_addr[0:1] // Two most significant bits of vector virtual_addr
```

Variable Vector Part Select

Another ability provided in Verilog HDl is to have variable part selects of a vector. This allows part selects to be put in for loops to select various parts of the vector. There are two special part-select operators:

[<starting_bit>+:width] - part-select increments from starting bit [<starting_bit>-:width] - part-select decrements from starting bit

The starting bit of the part select can be varied, but the width has to be constant. The following example shows the use of variable vector part select:

```
reg [255:0] data1; //Little endian notation
```

```
reg [0:255] data2; //Big endian notation
reg [7:0] byte;
//Using a variable part select, one can choose parts
byte = data1[31-:8]; //starting bit = 31, width =8 => data[31:24]
byte = data1[24+:8]; //starting bit = 24, width =8 => data[31:24]
byte = data2[31-:8]; //starting bit = 31, width =8 => data[24:31]
byte = data2[24+:8]; //starting bit = 24, width =8 => data[24:31]
//The starting bit can also be a variable. The width has
//to be constant. Therefore, one can use the variable part select //in a loop to select all bytes of the
vector.
for (j=0; j<=31; j=j+1)
   byte = data1[(j*8)+:8]; //Sequence is [7:0], [15:8]... [255:248]
//Can initialize a part of the vector
data1[(byteNum*8)+:8] = 8'b0; //If byteNum = 1, clear 8 bits [15:8]
```

3.2.5 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 x 106). 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
```

```
array_4d[0][0][0][0][15:0] = 0; //Clear bits 15:0 of the register
                                //accessed by indices [0][0][0][0]
port_id = 0; // Illegal syntax - Attempt to write the entire array
matrix [1] = 0; // Illegal syntax - Attempt to write [1][0]..[1][255]
```

3.2.7 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.
```

3.2.8 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
```

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
```

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

3.3 System Tasks and Compiler Directives

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

3.3.1 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. We will discuss only the most useful system tasks. Other tasks are listed in Verilog manuals provided by your simulator vendor or in the IEEE Standard Verilog Hardware Description Language specification.

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);

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

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.

Compiler Directives

Compiler directives are provided in Verilog. All compiler directives are defined by using the `<keyword> construct. We deal with the two most useful compiler directives.

`define

The `define directive is used to define text macros in Verilog (see Example 3-7). 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).

`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.

Two other directives, `ifdef and `timescale, are used frequently.

Modules

We shall study the internals of modules and concentrate on how modules are defined and instantiated. 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.

To understand the components of a module shown above, let us consider a simple example of an SR latch, as shown in Figure 4-2.

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 4-1.

Example 4-1 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
wire q, qbar;
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

```
//Display the string in quotes
$display("Hello Verilog World");
-- Hello Verilog World

//Display value of current simulation time 230
$display($time);
-- 230

//Display value of 41-bit virtual address 1fe0000001c at time 200
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c

//Display value of port_id 5 in binary
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
-- ID of the port is 00101

//Display x characters
//Display value of 4-bit bus 10xx (signal contention) in binary
reg [3:0] bus;
$display("Bus value is %b", bus);
-- Bus value is 10xx

//Display the hierarchical name of instance p1 instantiated under
//the highest-level module called top. No argument is required. This
//is a useful feature)
$display("This string is displayed from %m level of hierarchy");
-- This string is displayed from top.p1 level of hierarchy


     //Display special characters, newline and %
     $display("This is a \n multiline string with a %% sign");
     -- This is a
     -- multiline string with a % sign

     //Display other special characters


//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



// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time 1000.
initial // to be explained later. time = 0
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
```

```
//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]
// you can then define a 32-bit register as 'WORD_REG reg32;

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

Notice the following characteristics about the modules defined above:

- In the SR latch definition above , notice that all components described in Figure 4-1 need not be present in a module. We do not find variable declarations, dataflow (assign) statements, or behavioral blocks (always or initial).

- However, the stimulus block for the SR latch contains module name, wire, reg, and variable declarations, instantiation of lower level modules, behavioral block (initial), and endmodule statement but does not contain port list, port declarations, and data flow (assign) statements.

- Thus, all parts except module, module name, and endmodule are optional and can be mixed and matched as per design needs.

. **Ports**

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

4.2.1 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 4-3.

Figure 4-3. I/O Ports for Top and Full Adder

Notice that in 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 Example 4-2.

Example 4-2 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
```

4.2.2 Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

Each port in the port list is defined as input, output, or inout, based on the direction of the port signal.

Example 4-3 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
```

Note that 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.

Example 4-4 Port Declarations for DFF

```
module DFF(q, d, clk, reset);
output q;
reg q; // Output port q holds value; therefore it is declared as reg. input d, clk, reset;
...
...
endmodule
```

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.

Example 4-5 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
```

Port Connection Rules

One can visualize 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 Figure 4-4.

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 inter- module 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 in Example 4-3 is instantiated in the stimulus block Top. Example 4-6 shows an illegal port connection.

Example 4-6 Illegal Port Connection

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

This problem is rectified if the variable SUM is declared as a net (wire).

Connecting Ports to External Signals

There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition. These two methods cannot be mixed. These methods are discussed in the following sections.

Connecting by ordered list

Connecting by ordered list is the most intuitive method for most beginners. 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. To connect signals in module Top by ordered list, the Verilog code is shown in Example 4-7. 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 4-7 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.

Hierarchical Names

We described earlier how Verilog supports a hierarchical design methodology. 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.

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.

Gate-Level Modeling

Modeling using basic Verilog gate primitives, description of and/or and buf/not type gates, rise, fall and turn-off delays, min, max, and typical delays. (Text1)

Dataflow Modeling

Continuous assignments, delay specification, expressions, operators, operands, operator types. (Text1)

Modeling using basic Verilog gate primitives, description of and/or and buf/not type gates, rise, fall and turn-off delays, min, max, and typical delays.

Modeling using basic Verilog gate primitives:

Most digital design is now done at gate level or higher levels of abstraction. At gate level, the circuit is described in terms of gates (e.g., and, nand). Hardware design at this level is intuitive for a user with a basic knowledge of digital logic design because it is possible to see a one-to-one correspondence between the logic circuit diagram and the Verilog description.

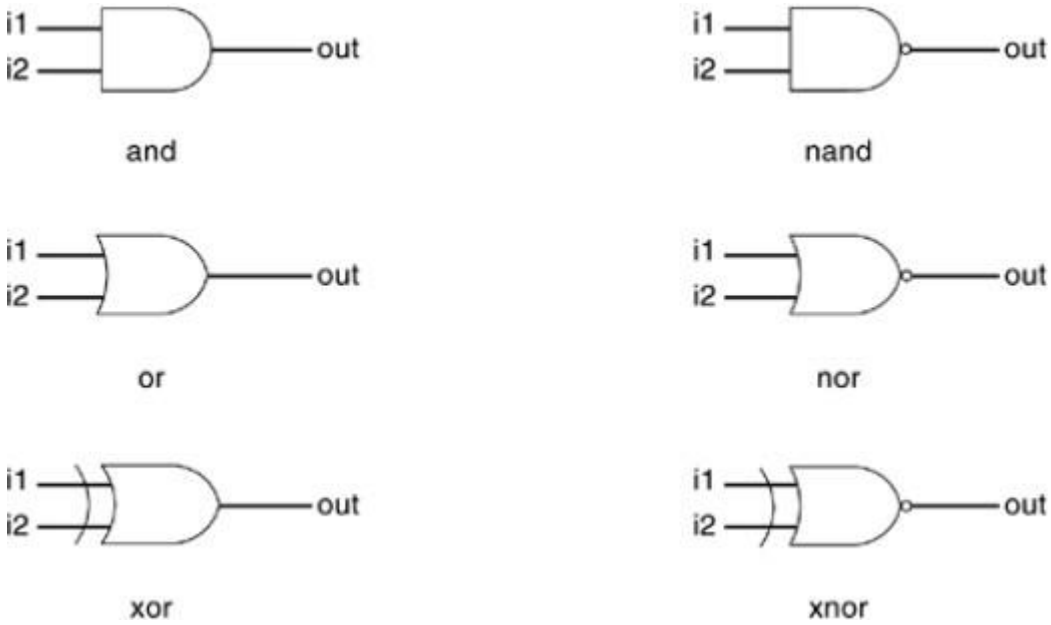
Description of Gate Types:

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

And/Or Gates

And/or gates have one scalar output and multiple scalar inputs. The first terminal in the list of gate terminals is an output and the other terminals are inputs. The output of a gate is evaluated as soon as one of the inputs changes. The and/or gates available in Verilog are shown below.

and, nand, or, nor, xor, xnor.



The corresponding logic symbols for these gates are shown in Figure 1.1. We consider 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 1.1, 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.

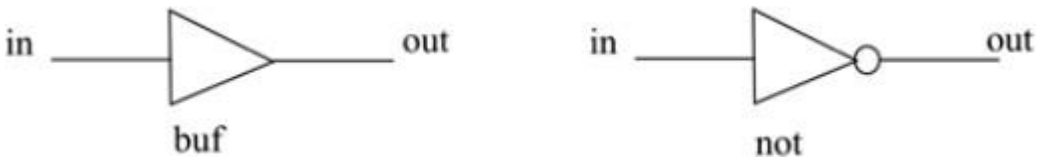
Example 1.1 Gate Instantiation of And/Or Gates

```
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 1.1. Outputs of gates with more than two inputs are computed by applying the truth table iteratively.

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.

buf not

The symbols for these logic gates are shown in Figure 1.2.

These gates are instantiated in Verilog as shown Example 1.2. Notice that these gates can have multiple outputs but exactly one input, which is the last terminal in the port list.

Example 1.2 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
```

The truth tables for these gates are very simple. Truth tables for gates with one input and one output are shown in Table 1.2.

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

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

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

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

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

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

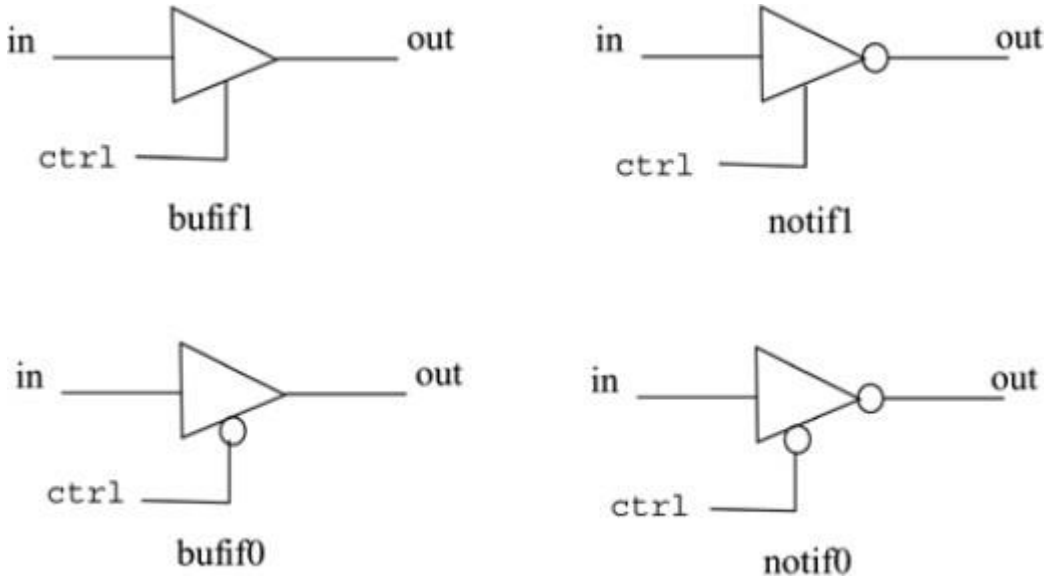
Bufif/notif

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 1.3.



The truth tables for these gates are shown in Table 1.3. 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.

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

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

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

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

Example 5-3 Gate Instantiations of Bufif/Notif Gates

```
//Instantiation of bufif gates.
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);
//Instantiation of notif gates
```

notif1 n1 (out, in, ctrl);

notif0 n0 (out, in, ctrl);

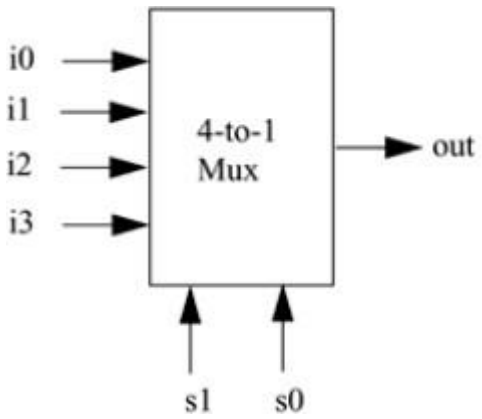
Array of Instances

There are many situations when repetitive instances are required. These instances differ from each other only by the index of the vector to which they are connected. To simplify specification of such instances, Verilog HDL allows an array of primitive instances to be defined. Example 1.4 shows an example of an array of instances.

Example 1.4 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]);
```

Examples



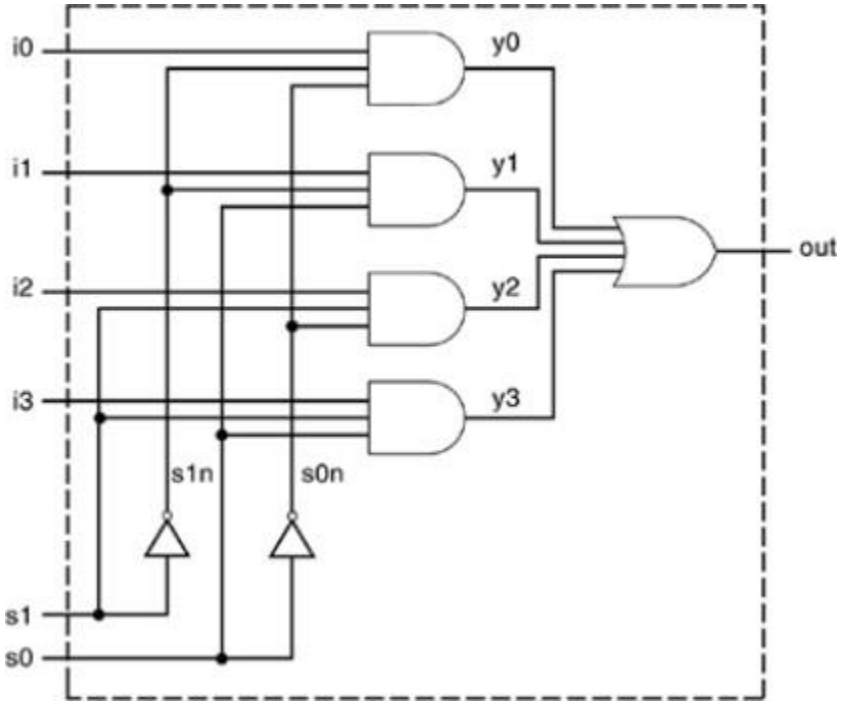
s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Having understood the various types of gates available in Verilog, we will discuss a real example that illustrates design of gate-level digital circuits.

Gate-level multiplexer

We will design a 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 logic diagram has a one-to-one correspondence with the Verilog description. The Verilog description for the multiplexer is shown in Example 1.5. 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 5-5 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 1.6. 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 1.6 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
  // choose IN1
```

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

The output of the simulation is shown below. Each combination of the select signals is tested.

IN0= 1, IN1= 0, IN2= 1, IN3= 0

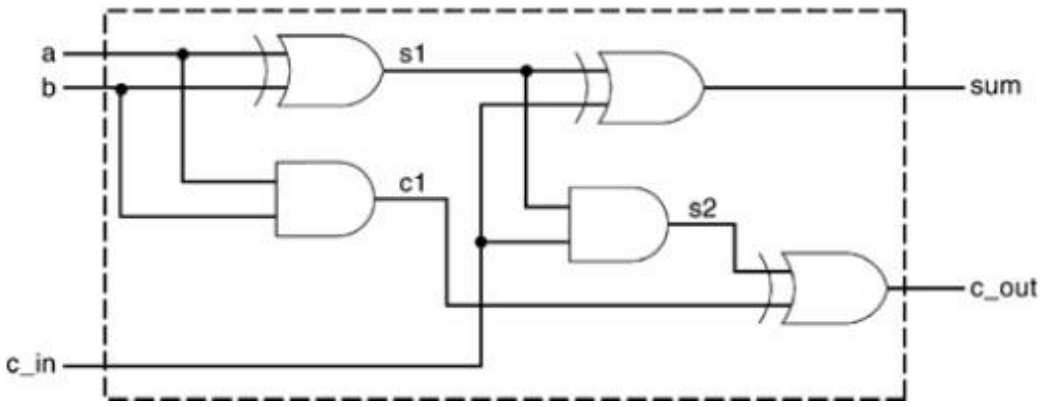
S1 = 0, S0 = 0, OUTPUT = 1

S1 = 0, S0 = 1, OUTPUT = 0

S1 = 1, S0 = 0, OUTPUT = 1

S1 = 1, S0 = 1, OUTPUT = 0

4-bit Ripple Carry Full Adder



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 1.6.

This logic diagram for the 1-bit full adder is converted to a Verilog description, shown in Example 1.7.

Example 1.7 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 1.7. Notice that fa0, fa1, fa2, and fa3 are instances of the module fulladd (1-bit full adder).

Example 1.8 Verilog Description for 4-bit Ripple Carry Full Adder

```
// Define a 4-bit full adder
module fulladd4(sum, c_out, a, b, c_in);
// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;
// Internal nets
wire c1, c2, c3;
// Instantiate four 1-bit full adders.
```

```
fulladd fa0(sum[0], c1, a[0], b[0], c_in);
fulladd fa1(sum[1], c2, a[1], b[1], c1);
fulladd fa2(sum[2], c3, a[2], b[2], c2);
fulladd fa3(sum[3], c_out, a[3], b[3], c3);
endmodule
```

Example 1.9 Stimulus for 4-bit Ripple Carry Full Adder

```
// Define the stimulus (top level module)
module stimulus;
// Set up variables
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
// Instantiate the 4-bit full adder. call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);
// Set up the monitoring for the signal values
initial
begin
$monitor($time," A= %b, B=%b, C_IN= %b, --- C_OUT= %b, SUM= %b\n", A, B, C_IN, C_OUT,
SUM);
end
// Stimulate inputs
initial
begin
  A = 4'd0; B = 4'd0; C_IN = 1'b0;
  #5 A = 4'd3; B = 4'd4;
  #5 A = 4'd2; B = 4'd5;
  #5 A = 4'd9; B = 4'd9;
  #5 A = 4'd10; B = 4'd15;
  #5 A = 4'd10; B = 4'd5; C_IN = 1'b1;
end
endmodule
```

The output of the simulation is shown below.

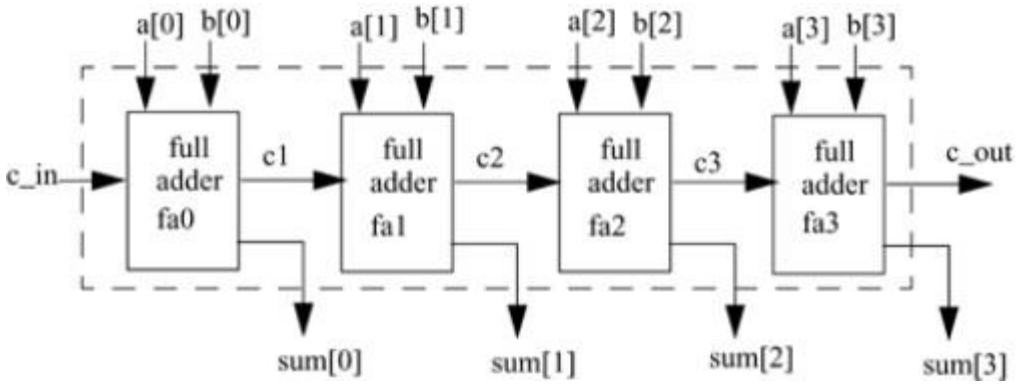
```
0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000
5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111
```

10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111

15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010

20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001

25 A= 1010, B=0101, C_IN= 1,, C_OUT= 1, SUM= 0000



Gate Delays

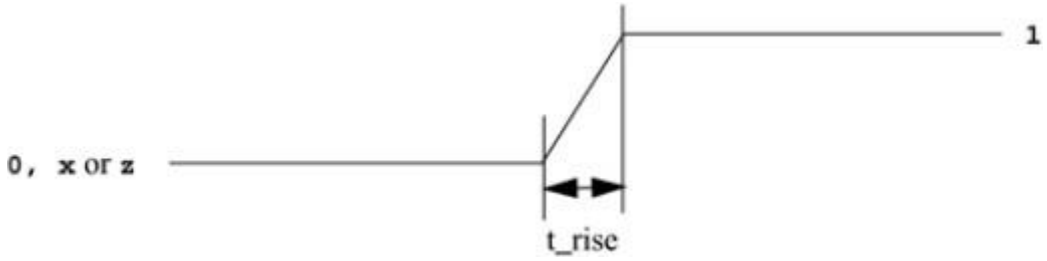
Until now, we described circuits without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

Rise, Fall, and Turn-off Delays

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

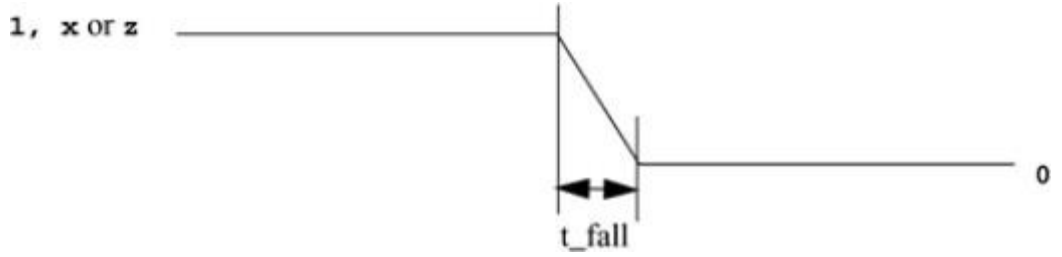
Rise delay

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 1.10.

Example 1.10 Types of Delay Specification

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

Examples of delay specification are shown below.

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

Min/Typ/Max Values

Verilog provides an additional level of control for each type of delay mentioned above. For each type of delay?rise, fall, and turn-off?three values, min, typ, and max, can be specified. Any one value can be chosen at the start of the simulation. Min/typ/max values are used to model devices whose delays vary within a minimum and maximum range 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 Example 1.11.

Example 1.11 Min, Max, and Typical Delay Values

```
// One delay
// if +mindelays, delay= 4
// if +typdelays, delay= 5
// if +maxdelays, delay= 6
and #(4:5:6) a1(out, i1, i2);
// Two delays
// if +mindelays, rise= 3, fall= 5, turn-off = min(3,5)
// if +typdelays, rise= 4, fall= 6, turn-off = min(4,6)
// if +maxdelays, rise= 5, fall= 7, turn-off = min(5,7)
and #(3:4:5, 5:6:7) a2(out, i1, i2);
// Three delays
// if +mindelays, rise= 2 fall= 3 turn-off = 4
// if +typdelays, rise= 3 fall= 4 turn-off = 5
// if +maxdelays, rise= 4 fall= 5 turn-off = 6
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
```

Examples of invoking the Verilog-XL simulator with the command-line options are shown below. Assume that the module with delays is declared in the file test.v.

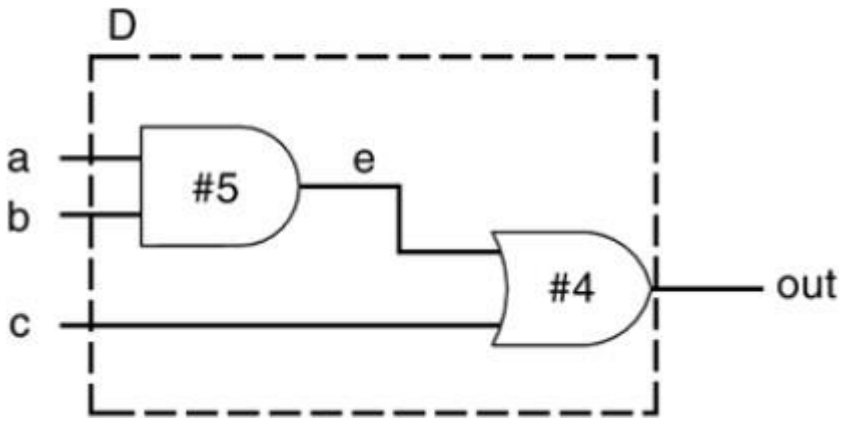
```
//invoke simulation with maximum delay
> verilog test.v +maxdelays
//invoke simulation with minimum delay
> verilog test.v +mindelays
//invoke simulation with typical delay
> verilog test.v +typdelays
```

Delay Example

Let us consider a simple example to illustrate the use of gate delays to model timing in the logic circuits. A simple module called D implements the following logic equations:

out=(a b)+c

The gate-level implementation is shown in Module D (Figure 1.8). The module contains two gates with delays of 5 and 4 time units.



The module D is defined in Verilog as shown in Example 1.12.

Example 1.12 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
```

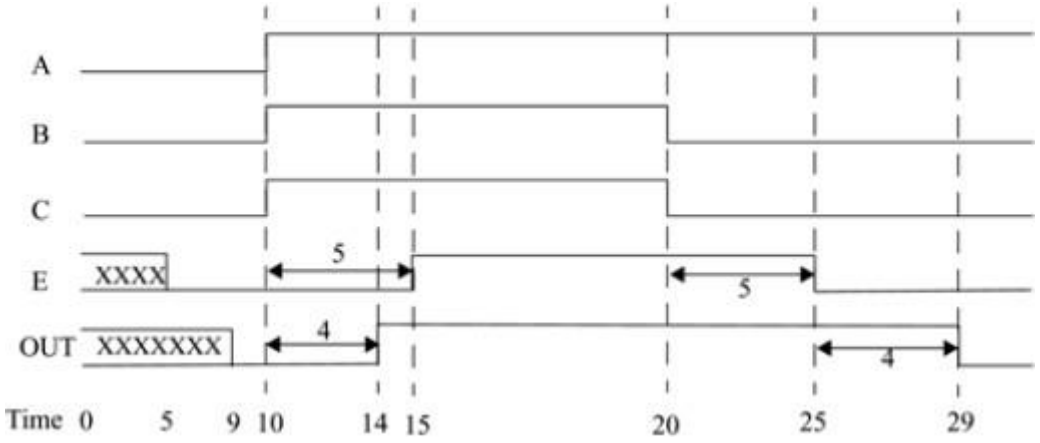
This module is tested by the stimulus file shown in Example 1.13. **Example 1.13 Stimulus for Module D with Delay**

```
// 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 1.9 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.



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.

Continuous assignments, delay specification, expressions, operators, operands, operator types. (Text1)

With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance. No longer can companies devote engineering resources to handcrafting entire designs with gates. Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis. Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated.

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

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 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;
```

We now discuss a shorthand method of placing a continuous assignment on a net.

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
```

Delays

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

Regular Assignment Delay

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation 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 recomputation 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

The waveform in Figure 2.1 is generated by simulating the above assign statement. It shows the delay on signal out. Note the following change:

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 recomputation, 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 not propagated to the output.

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;
```

Having discussed continuous assignments and delays, let us take a closer look at expressions, operators, and operands that are used inside continuous assignments.

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

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

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 (functions are discussed later).

```
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.

```
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 2.1 shows the complete listing of operator symbols classified by category.

Behavioral Modeling

Structured procedures, initial and always, blocking and non-blocking statements, delay control, generate statement, event control, conditional statements, multiway branching, loops, sequential and parallel blocks. (Text1)

With the increasing complexity of digital design, it has become vitally important to make wise design decisions early in a project. Designers need to be able to evaluate the trade- offs of various architectures and algorithms before they decide on the optimum architecture and algorithm to implement in hardware. Thus, architectural evaluation takes place at an algorithmic level where the designers do not necessarily think in terms of logic gates or data flow but in terms of the algorithm they wish to implement in hardware. They are more concerned about the behavior of the algorithm and its performance. Only after the high-level architecture and algorithm are finalized, do designers start focusing on building the digital circuit to implement the algorithm.

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.

7.1.1 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.

Example 7-1 initial Statement

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

```
end
initial
#50 $finish;
endmodule
```

In the above example, the three initial statements start to execute in parallel at time 0. If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time. Thus, the execution sequence of the statements inside the initial blocks will be as follows.

time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.

Combined Variable Declaration and Initialization

Variables can be initialized when they are declared. Example 7-2 shows such a declaration.

Example 7-2 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 7-3 shows such a declaration.

Example 7-3 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

input

input

--

--

endmodule

co = 0; //Initialize 1 bit output co

[7:0] a, b;

ci;

Combined ANSI C Style Port Declaration and Initialization

ANSI C style port declaration can also be combined with an initialization. Example 7-4 shows such a declaration.

Example 7-4 Combined ANSI C Port Declaration and Variable Initialization

module adder (output reg [7:0] sum = 0, //Initialize 8 bit output

---

--

--

endmodule

7.1.2 always Statement

output reg

input

input

);

co = 0, //Initialize 1 bit output co

[7:0] a, b,

ci

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. Example 7-5 illustrates one method to model a clock generator in Verilog.

Example 7-5 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
```

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).

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 discussed in Chapter 6, 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 listed in Table 6-1 on page 96 can be used in behavioral expressions. There are two types of procedural assignment statements: blocking and nonblocking.

7.2.1 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 7-6 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
to end
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
                    // part select of a vector
count = count + 1; //Assignment to an integer (increment)
```

In Example 7-6, 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

7.2.2 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.

Example 7-7 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)

Delay Controls

Various behavioral timing control constructs are available in Verilog. 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.

7.3.1 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.

```
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 7-10.

Example 7-10 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
y. end
#(latency + delta) p = 1; // Delay control with expression
#y x = x + 1; // Delay control with identifier. Take value of
#(4:5:6) q = 0; // Minimum, typical and maximum delay values.
        //Discussed in gate-level modeling chapter.
```

In Example 7-10, 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 7-11 shows the contrast between intra-assignment delays and regular delays.

Example 7-11 Intra-assignment Delays

```
//define register variables
reg x, y, z;
```

```
//intra assignment delays
initial
begin
    x = 0; z = 0;
end
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.
//Equivalent method with temporary variables and regular delay control
initial
begin
    x = 0; z = 0;
z
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
     //might change between 0 and 5,
end
//the value assigned to y at time 5 is unaffected.
```

Note the difference between intra-assignment delays and regular delays. Regular delays defer the execution of the entire assignment. Intra-assignment delays compute the right- hand-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 7-12 illustrates zero delay control.

Example 7-12 Zero Delay Control

```
initial
begin
x = 0;
```

```
y = 0; end
initial
begin
   #0 x = 1; //zero delay control
#0 y = 1; end
```

In Example 7-12, 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.

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 mentioned in Table 6-1 on page 96. 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 7-18 Conditional Statement Examples

```
//Type 1 statements
if(!lock) buffer = data;
```

```
if(enable) out = in;
//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
begin
end else
data_queue = data;
number_queued = number_queued + 1;
$display("Queue Full. Try again");
//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
y = x + z;
else if(alu_control == 1)
      y = x - z;
else if(alu_control == 2)
      y = x * z;
else
$display("Invalid ALU control signal");
```

Multiway Branching

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.

7.5.1 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. 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 7-18.

```
//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. 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 7-19 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 7- 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 7-20 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.

7.5.2 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. The use of casez is similar. Only one bit is considered to determine the next state and the other bits are ignored.

Example 7-21 casex Use

reg [3:0] encoding;
- integer state;
-

casex (encoding) //logic value x represents a don't care bit.
- 4'b1xxx : next_state = 3;
- 4'bx1xx : next_state = 2;
- 4'bxx1x : next_state = 1;
-

4'bxxx1 : next_state = 0;
- default : next_state = 0;
- endcase
-

Thus, an input encoding = 4'b10xz would cause next_state = 3 to be executed.

Loops

There are four types of looping statements in Verilog: while, for, repeat, and forever. The syntax of these loops is very similar to the syntax of loops in the C programming language. All looping statements can appear only inside an initial or always block. Loops may contain delay expressions.

7.6.1 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. If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end.

Example 7-22 While Loop

//Illustration 1: Increment count from 0 to 127. Exit at count 128.

//Display the count variable.

integer count;

```
initial
begin
end
count = 0;
while (count < 128) //Execute loop till count is 127.
                //exit at count 128
begin
       $display("Count = %d", count);
      count = count + 1;
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;
  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
```

7.6.2 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 7-22 can be coded as a for loop (Example 7-23). 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 7-23 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.

7.6.3 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.

Example 7-24 Repeat Loop

```
//Illustration 1 : increment and display count from 0 to 127
integer count;
initial
```

```
begin
  count = 0;
  repeat(128)
  begin
      $display("Count = %d", count);
      count = count + 1;
  end
end
//Illustration 2 : Data buffer module example
//After it receives a data_start signal.
//Reads data for next 8 cycles.
module data_buffer(data_start, data, clock);
parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;
reg [15:0] buffer [0:7];
integer i;
always @(posedge clock)
begin
 if(data_start) //data start signal is true
 begin
   i = 0;
   repeat(cycles) //Store data at the posedge of next 8 clock
//cycles
   begin
     @(posedge clock) buffer[i] = data; //waits till next
i = i + 1; end
end end
endmodule
```

7.6.4 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

Example 7-25 Forever Loop

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

7.7 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.

7.7.1 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 7-26. 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 7-26 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
end
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
```

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.

Example 7-27 Parallel Blocks:

```
//Example 1: Parallel blocks with delay.
reg x, y;
```

```
reg [1:0] z, w;
    initial fork
    join
```

```
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
```

Parallel blocks provide a mechanism to execute statements in parallel. 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.

```
//Parallel blocks with deliberate race condition
reg x, y;
reg [1:0] z, w;
initial fork
join
x = 1'b0;
y = 1'b1;
z = {x, y};
w = {y, x};
```

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.

7.7.2 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 7-28.

Example 7-28 Nested Blocks

```
//Nested blocks
initial
begin
x = 1'b0;
```

```
end
fork
join
#20 w = {y, x};
#5 y = 1'b1;
#10 z = {x, y};
```

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 7-29 shows naming of blocks and hierarchical naming of blocks.

Example 7-29 Named Blocks

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

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. The difference is that a break statement can break the current loop only, whereas the keyword disable allows disabling of any named block in the design.

Consider the illustration in Example 7-22 on page 142, which looks for the first true bit in the flag. The while loop can be recoded, using the disable statement as shown in Example 7-30. The disable statement terminates the while loop as soon as a true bit is seen.