CHAPTER 2

Introduction to Verilog®

As integrated circuit technology has improved to allow more and more components on a chip, digital systems have continued to grow in complexity. While putting a few transistors on an integrated circuit (IC) was a miracle when it happened, technology improvements have advanced the **VLSI** (very large scale integration) field continually. The early integrated circuits belonged to **SSI** (small-scale integration), **MSI** (medium-scale integration), or **LSI** (large-scale integration) categories depending on the density of integration. SSI referred to ICs with 1 to 20 gates, MSI referred to ICs with 20 to 200 gates, and LSI referred to devices with 200 to a few thousand gates. Many popular building blocks such as adders, multiplexers, decoders, registers, and counters are available as MSI standard parts. When the term VLSI was coined, devices with 10,000 gates were called VLSI chips. The boundaries between the different categories are fuzzy today. Many modern microprocessors contain more than 100 million transistors. Compared to what was referred to as VLSI in its initial days, modern integration capability could be described as ULSI (ultra large scale integration). Despite the changes in integration ability and the fuzzy definition, the term VLSI remains popular.

As digital systems have become more complex, detailed design of the systems at the gate and flip-flop level has become very tedious and time-consuming. Two or three decades ago, digital systems were created using hand-drawn schematics, bread-boards, and wires, which were connected to the bread-board. Now, hardware design often involves no-hands-on tasks with bread-boards and wires.

In this chapter, first we present an introduction to computer-aided design. Then, an introduction to hardware description languages is presented. Basic features of Verilog® are presented, and examples are provided to illustrate how digital hardware is described, simulated, and synthesized using Verilog. Advanced features of Verilog are presented later in Chapter 8.

2.1 Computer-Aided Design

Computer-aided design (CAD) tools have advanced significantly during the past decade, and nowadays digital design is performed using a variety of software tools. Prototypes or even final designs can be created without discrete components and interconnection wires.

Figure 2-1 illustrates the steps in modern digital system design. Like any engineering design, the first step in the design flow is formulating the problem, stating the **design requirements,** and arriving at the **design specification**. The next step is to **formulate the design** at a conceptual level, either at a block diagram level or at an algorithmic level.

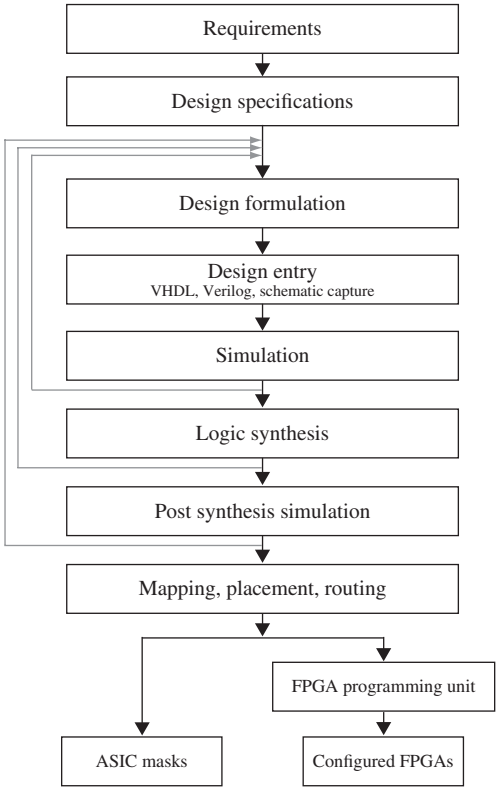


FIGURE 2-1: Design Flow in Modern Digital System Design

Design entry is the next step in the design flow. Previously, this would have been a hand-drawn schematic or blueprint. Now with CAD tools, the design conceptualized in the previous step needs to be entered into the CAD system in an appropriate manner. Designs can be entered in multiple forms. A few years ago, CAD tools used to provide a graphical method to enter designs. This was called **schematic capture**. The schematic editors typically were supplemented with a library of standard digital building blocks such as gates, flip-flops, multiplexers, decoders,

counters, registers, and so forth. ORCAD (a company that produced design automation tools) provided a very popular schematic editor. Nowadays, **hardware description languages** (HDLs) are used to enter designs in textual form. Two popular HDLs are VHDL and Verilog.

A hardware description language (HDL) allows a digital system to be designed and debugged at a higher level of abstraction than schematic capture. In schematic capture, a designer inputs a schematic with gates, flip-flops, and standard MSI building blocks. However, with HDLs, the details of the gates and flip-flops do not need to be handled during early phases of design. A design can be entered in what is called a **behavioral description** of the design. In a behavioral HDL description, one specifies only the general working of the design at a flow-chart or algorithmic level without associating to any specific physical parts, components, or implementations. Another method to enter a design in VHDL and Verilog is the **structural description** entry. In structural design, specific components or specific implementations of components are associated with the design. A structural VHDL or Verilog model of a design can be considered as a textual description of a schematic diagram that you would have drawn interconnecting specific gates, flip-flops, and other modules.

Once the design has been entered, it is important to simulate it to confirm that the conceptualized design does function correctly. Initially, one should perform the **simulation** at the high-level behavioral model. This early simulation unveils problems in the initial design. If problems are discovered, the designer goes back and alters the design to meet the requirements.

Once the functionality of the design has been verified through simulation, the next step is **synthesis**. Synthesis means conversion of the higher-level abstract description of the design to actual components at the gate and flip-flop levels. Use of computer-aided design tools to do this conversion, also called synthesis, is standard practice in the industry now. The output of the synthesis tool, consisting of a list of gates and a list of interconnections, specifying how to interconnect them, is often referred to as a **netlist**. Synthesis is analogous to writing software programs in a high-level language such as C and then using a compiler to convert the programs to machine language. Just as a C compiler can generate optimized or unoptimized machine code, a synthesis tool can generate optimized or unoptimized hardware. The synthesis software generates different hardware implementations, depending on algorithms embedded in the software to perform the translation and optimization. A synthesis tool is nothing but a compiler to convert design descriptions to hardware, and it is not unusual to name synthesis packages with phrases such as “design compiler,” “silicon compiler,” and the like.

The next step in the design flow is **post-synthesis simulation**. The earlier simulation at a higher level of abstraction does not take into account specific implementations of the hardware components that the design is using. If post-synthesis simulation unveils problems, one should go back and modify the design to meet timing requirements. Arriving at a proper design implementation is an iterative process.

Next, a designer moves into specific realizations of the design. A design can be implemented in several different target technologies. It could be a completely custom IC, or it could be implemented in a standard part that is easily available from

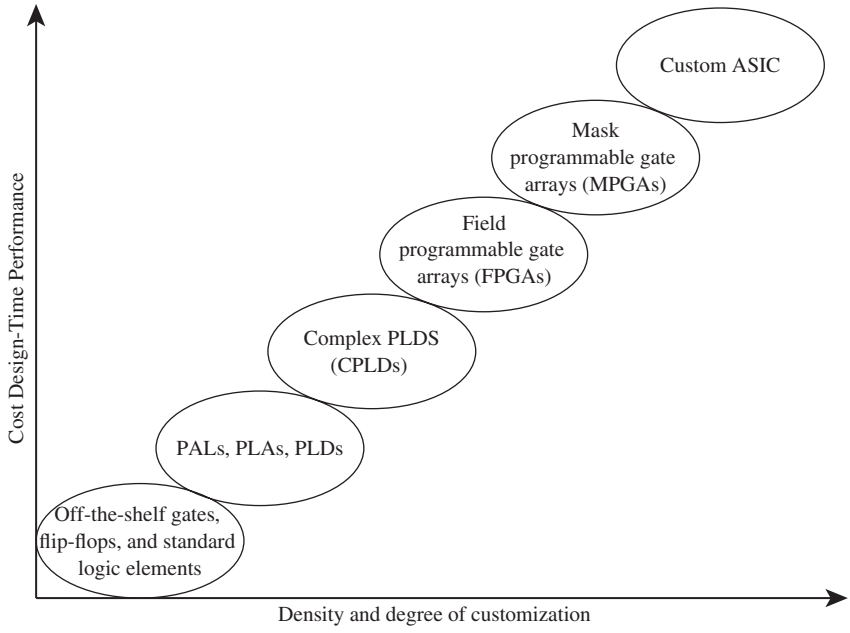


FIGURE 2-2: Spectrum of Design Technologies

a vendor. The target technologies that are commonly available now are illustrated in Figure 2-2.

At the lowest level of sophistication and density is an old-fashioned printed circuit board with off-the-shelf gates, flip-flops, and other standard logic-building blocks. Slightly higher in density are programmable logic arrays (PLAs), programmable array logic (PALs), and simple programmable logic devices (SPLDs). PLDs with higher density and gate count are called complex programmable logic devices (CPLDs). In addition, there are the popular field programmable gate arrays (FPGAs) and mask programmable gate arrays (MPGAs), or simply gate arrays. The highest level of density and performance is a fully custom application-specific integrated circuit (ASIC).

Two most common target technologies currently are FPGAs and ASICs. The initial steps in the design flow are largely the same for either realization. Towards the final stages in the design flow, different operations are performed depending on the target technology. This is indicated in Figure 2-1. The design is **mapped** into specific target technology and **placed** into specific parts in the target ASIC or FPGA. The paths taken by the connections between components are decided during the **routing**. If an ASIC is being designed, the routed design is used to generate a photomask that will be used in the integrated circuit (IC) manufacturing process. If a design is to be implemented in an FPGA, the design is translated to a format specifying what is to be done to various programmable points in the FPGA. In modern FPGAs, programming simply involves writing a sequence of 0s and 1s into the programmable cells in the FPGA, and no specific programming unit other than a personal computer (PC) is required to program an FPGA.

2.2 Hardware Description Languages

Hardware description languages (HDLs) are a popular mode of design entry for digital circuits and systems. There are two popular HDLs—VHDL and Verilog. Before the advent of HDLs, designers used graphical schematics and schematic capture tools to document and simulate digital circuits. A need was felt to create a textual method of documenting circuits and feeding them into simulators in the textual form as opposed to a graphic form. This book uses the Verilog language for illustrating principles of modern digital system design. Those interested in VHDL can find the equivalent material in Roth and John, *Digital Systems Design Using VHDL*, 2nd ed. (Cengage Learning, 2008).

Verilog is a hardware description language used to describe the behavior and/or structure of digital systems. Verilog is a general-purpose hardware description language that can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. While the competing language VHDL was originally developed under funding from the Department of Defense (DoD), Verilog was developed by the industry. It was initially developed as a proprietary language by a company called Gateway Design Automation around 1984.

In 1990, Cadence acquired Gateway Design Automation and became the owner of Verilog. Cadence marketed it as a language and as a simulator, but it remained proprietary. At this time, Synopsis was promoting the concept of top-down design using Verilog. Cadence realized that it needed to make Verilog open in order to prevent the industry from shifting to VHDL and hence opened up the language. An organization called Open Verilog International (OVI) was formed, which helped to create a vendor-independent language specification for Verilog, clarifying many of the confusions around the proprietary specification. This was followed by an effort to create an IEEE standard for Verilog. The first Verilog IEEE Standard was created in 1995, which was revised in 2001 and 2005. Synopsis created synthesis tools for Verilog around 1988.

HDLs can describe a digital system at several different levels—behavioral, data flow, and structural. For example, a binary adder could be described at the behavioral level in terms of its function of adding two binary numbers without giving any implementation details. The same adder could be described at the data flow level by giving the logic equations for the adder. Finally, the adder could be described at the structural level by specifying the gates and the interconnections between the gates that comprise the adder.

HDLs lead naturally to a top-down design methodology, in which the system is first specified at a high level and tested using a simulator. After the system is debugged at this level, the design can gradually be refined, eventually leading to a structural description closely related to the actual hardware implementation. HDLs are designed to be technology independent. If a design is described in HDL and implemented in today's technology, the same HDL description could be used as a starting point for a design in some future technology.

Verilog has its syntactic roots in C whereas VHDL has its syntactic roots in ADA. Some find Verilog easier or less intimidating to learn due to its similarity

with C, while many find VHDL to be excellent for supporting design and documentation of large systems. VHDL and Verilog enjoy approximately 50/50 global market share. Both languages can accomplish most requirements for digital design rather easily. But Verilog is touted by many to be slightly more supportive for synthesis and VHDL is touted to be slightly more elegant for simulation of very large systems at a higher level of abstraction. Often design companies continue to use what they are used to; hence, Verilog users continue to use Verilog and VHDL users continue to use VHDL. If one knows one of these languages, it is not difficult to transition to the other. VHDL has more conceptual elegance for the higher-level abstractions, while Verilog has enjoyed its popularity from its syntactic similarity to C.

More recently, there also have been efforts in system design languages such as System C, Handel-C, and System Verilog. System C is created as an extension to C++; hence, some who are more comfortable with general-purpose software find it less intimidating. These languages are primarily targeted at describing large digital systems at a higher level of abstraction. They are primarily used for verification and validation. When different parts of a large system are designed by different teams, one team can use a system-level behavioral description of the block being designed by the other team during the initial design process. Problems that might otherwise become obvious only during system integration may become evident in early stages, reducing the design cycle time for large systems. System-level simulation languages are used during the design of large systems. Efforts to synthesize hardware from high level-languages are steadily progressing, and tools to convert models written in languages such as C++ to hardware are emerging.

Learning a Language

There are several challenges when you learn a new language, whether it be a language for common communication (English, Spanish, French, etc.), a computer language such as C, or a special-purpose language such as Verilog. If it is not your first language, you typically have a tendency to compare it with a language you know. In the case of Verilog, if you already know another hardware description language, it is good to compare it with Verilog, but you should be careful when comparing it with languages such as C. VHDL and Verilog have a very different purpose from languages such as C, and a comparison with C is not a meaningful activity. We will be describing the language assuming it is your first HDL; however, we will assume basic knowledge of computer languages such as C and the basic compilation and execution flow.

When one learns a new language, one needs to study the alphabet of the new language, its vocabulary, grammar, syntax rules, and semantics of language descriptions. The process of learning Verilog is not much different. One needs to learn the alphabet, vocabulary or lexical elements of the language, syntax (grammar and rules), and semantics (meaning of descriptions). The lexical elements of the language include various **identifiers, reserved words**, special symbols, and literals. We have listed these in Appendix B. The syntax or grammar determines what combinations of lexical elements can be combined to make valid Verilog descriptions. These are the rules that govern the use of different Verilog constructs. Then one needs to understand the semantics or meaning of Verilog descriptions. It is here that one understands what descriptions represent combinational hardware versus sequential

hardware. And just as fluency in a natural language comes by speaking, reading, and writing the language, mastery of Verilog comes by repeated use of the language to create models for various digital systems.

Since Verilog is a hardware description language, it differs from an ordinary programming language in several ways. Most importantly, Verilog has statements that execute concurrently since they must model real hardware in which the components are all in operation at the same time. It is popularly used for the purposes of describing, documenting, simulating, and automatically generating hardware. Hence, its constructs are tailored for these purposes. We will present the various methods to model different kinds of digital hardware using examples in the following sections.

Common Abbreviations

VHDL:	VHSIC hardware description language
VHSIC:	very-high-speed integrated circuit
HDL:	hardware description language
CAD:	computer-aided design
EDA:	electronic design automation
LSI:	large-scale integration
MSI:	medium-scale integration
SSI:	small-scale integration
VLSI:	very-large-scale integration
ULSI:	ultra-large-scale integration
ASCII:	American standard code for information interchange
ISO:	International Standards Organization
ASIC:	application-specific integrated circuit
FPGA:	field-programmable gate array
PLA:	programmable logic array
PAL:	programmable array logic
PLD:	programmable logic device
CPLD:	complex programmable logic device
STA:	static timing analysis

2.3 Verilog Description of Combinational Circuits

The biggest difficulty in modeling hardware using a general-purpose computer language is representing concurrently operating hardware. Computer programs that you are normally accustomed to are sequences of instructions with a well-defined order. At any point of time during execution, the program is at a specific point in its flow, and it encounters and executes different parts of the program sequentially. In order to model combinational circuits that have several gates (all of which are always working simultaneously), one needs to be able to "simulate" the execution of several parts of the circuit at the same time.

For both built-in and user-defined primitives, `instance_name` is optional. Primitives can be instantiated only within modules. The first port of an instantiated primitive is always an output and the other ports are all inputs. The details of UDPs will be dealt with in Chapter 5.

1.2.7 Attributes

An attribute is a mechanism provided by Verilog HDL for annotating information about objects, statements and groups of statements in the source description to be used by various related tools. Almost all statements can be attributed in Verilog HDL. There are no standardized attributes. An attribute can be attached as a prefix to a declaration, a module item, a statement or a port connection and as a suffix to an operator or a function name in an expression. Attributes have the following syntax:

```
(*attr_spec {, attr_spec}*)
```

where `attr_spec` can be either `attr_name` or `attr_name = const_expr`. If an attribute is not assigned to a value, its value defaults to 1. The following are some examples:

```
// example 1: attach full_case attribute only
(* full_case=1, parallel_case = 0 *)
case (selection)
  <rest_of_case_statement>
endcase

// example 2: attach an attribute to a module
(* dont_touch *) module array_multiplier
     (x, y, product);
  <the body of array_multiplier>
endmodule

// example 3: attach an attribute to a reg variable
(* fsm_state *)   reg [1:0] state1;
(* fsm_state=1 *) reg [1:0] state2, state3;

// example 4: attach an attribute to a function call
x = add(* mode = "cla" *)(y, z);

// example 5: attach an attribute to an operator
x = y + (* mode = "cla" *) z;
```

Review Questions

Q1.6 How would you describe a bundle of signals in Verilog HDL when it is used to describe hardware modules?

Q1.7 How many possible values may a net or `reg` variable have?

Q1.8 How many data types are included in net data types? What are these?

Q1.9 How many data types are included in variable data types? What are these?

Q1.10 Define the following two terms: *scalar* and *vector*, in the sense of Verilog HDL.

Q1.11 What kinds of primitives are provided in Verilog HDL?

1.3 MODULE MODELING STYLES

In order to understand the design methodology of modern digital systems, it is instructive to distinguish among *design*, *model*, *synthesis*, *implementation* or *realization*. Design is a series of transformations from one representation of a system to another until a representation that can be fabricated exists; that is, it can be created, fashioned, executed or constructed according to a plan. Model is a process that converts a specification document into a HDL module. Model is a system of postulates, data and inferences presented as a mathematical description of an entity or the state of affair. Synthesis is a process that converts HDL modules into a structural representation. Synthesis can be divided into logic synthesis and high-level synthesis. Logic synthesis is a process that converts an RTL description into a gate-level netlist, while high-level synthesis is a process that converts a high-level description (i.e. specification) into RTL results. Implementation (or realization) is the process of transforming design abstraction into physical hardware components such as FPGAs or cell-based ICs (integrated circuits).

1.3.1 Modules

As mentioned previously, a Verilog HDL module consists of two major parts: the interface and the internal.

1.3.1.1 Modeling the Internal of a Module For each module in Verilog HDL, the internal (or body) can be modeled as one of the following styles:

1. *Structural style*. A design is described as a set of interconnected components. The components can be modules, UDPs, primitive gates and/or primitive switches.
   (a) Gate level. A design is said to be modeled at gate-level when it only comprises a set of interconnected gate primitives.
   (b) Switch level. A design is said to be modeled at switch-level when it only consists of a set of interconnected switch primitives.

   The structural style is a mechanism that can be used to construct a hierarchical design for a large digital system.
2. *Dataflow style*. The module is described by specifying the dataflow (i.e. data dependence) between registers and how the data is processed.
   (a) A module is specified as a set of continuous assignment statements.

3. *Behavioral* or *algorithmic style*. The design is described in terms of the desired design algorithm without concerning the hardware implementation details.
   (a) Designs can be described in any high-level programming languages.
4. *Mixed style*. The design is described in terms of the mixing use of above three modeling styles.
   (a) Mixed styles are most commonly used in modeling large designs.

In industry, the term *register-transfer level* (RTL) is often used to mean a structure that combines both behavioral and dataflow constructs and can be acceptable by logic synthesis tools.

1.3.1.2 Port Declaration The interface signals (including supply and ground) of any Verilog HDL module can be cast into one of the following three types:

(1) `input`: Declare a group of signals as input ports.
(2) `output`: Declare a group of signals as output ports.
(3) `inout`: Declare a group of signals as bidirectional ports; that is, they can be used as input or output ports but not at the same time.

The simplest way to describe the complete interface of a module is to divide it into three parts: port list, port declaration and data type declaration of each port. Usually, an output port, except that it is a net data type, must be declared a data type associated with it. However, input ports are often left with their data types undeclared. For example:

```
// port list style
module adder(x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output [3:0] sum;
output c_out;
reg [3:0] sum;
reg c_out;
```

This style of port declaration is known as *port list style*. The declaration of a port and its associated data type can be combined into a single line. Based on this idea, the above interface portion of the module can be rewritten as follows:

```
// port list style
module adder(x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output reg [3:0] sum;
output reg c_out;
```

Of course, port list, port declarations and their associated data types can also be put together into a single list. This style is often called *port list declaration style*. Hence, the above module interface can be rewritten as follows:

```
// port list declaration style
module adder(input [3:0] x, y,
             input c_in,
             output reg [3:0] sum,
             output reg c_out
); // sometimes called ANSI C style
```

This style is the same as that of ANSI C programming language so that we often name it as the ANSI C style. Note that all of the above interface styles are valid in Verilog HDL.

1.3.1.3 Port Connection Rules The port connection (also called port association) rules of Verilog HDL modules are consistent with those of actual hardware modules. That is, Verilog HDL allows ports to remain unconnected and with different sizes. In addition, unconnected inputs are driven to the "z" state; unconnected outputs are not used. Connecting ports to external signals can be done by one of the following two methods:

- *Named association*. The ports to be connected to external signals are specified by listing their names. The port order is not important.
- *Positional association*. The ports are connected to external signals by an ordered list. The signals to be connected must have the same order as the ports in the port list, leaving the unconnected port blank.

However, these two methods cannot be mixed in the same module. Moreover, Verilog HDL primitives (including both built-in and user-defined) can only be connected by positional association.

The operation to "call" (much like the *macro expansion* in assembly language) a built-in primitive, a user-defined primitive or the other module is called *instantiation* and each copy of the called primitive or module is called an *instance*. Figure 1.4 shows how to instantiate gate primitives and user-defined modules, as well as how to connect their ports through nets and input/output ports. The built-in primitives can only be connected by using positional association. They cannot be connected through named association. In addition, as mentioned before, the instance names of these primitives are optional.

The module `full_adder` depicted in Figure 1.4 shows how to instantiate an already defined module and how to connect their ports through nets. The user-defined modules can be connected by using either named association or positional association. In addition, the instance names of these instantiations are necessary.

Coding Styles

1. *A module cannot be declared within another module.*
2. *A module can instantiate other modules.*
3. *A module instantiation must have a module identifier (instance name) except for built-in primitives, gate and switch primitives and user-defined primitives (UDPs).*

```
module half_adder (x, y, s, c);
input  x, y;
output s, c;
// -- half adder body-- //
// instantiate primitive gates
   xor xor1 (s, x, y);
   and and1 (c, x, y);
endmodule

module full_adder (x, y, cin, s, cout);
input  x, y, cin;
output s, cout;
wire   s1,c1,c2;  // outputs of both half adders
// -- full adder body-- //
// instantiate the half adder
   half_adder ha_1 (x, y, s1, c1);
   half_adder ha_2 (.x(cin), .y(s1), .s(s), .c(c2));
   or (cout, c1, c2);
endmodule
```

Can only be connected by using positional association

Instance name is optional

Connecting by using positional association

Connecting by using named association

Instance name is necessary

FIGURE 1.4 Port connection rules

4. *It should use named association at the top-level modules to avoid confusion that may arise from synthesis tools.*

1.3.2 Structural Modeling

As mentioned previously, the structural modeling of a design is by connecting required instantiations of built-in primitives, user-defined primitives or other (user-defined) modules through nets. The following example shows several examples of structural modeling:

EXAMPLE 1.4 ***An Example of Structural Modeling at Gate Level***

The `half_adder` instantiates two gate primitives and the `full_adder` instantiates two `half_adder` modules and one gate primitive. Finally, the `four_bit_adder` is constructed by four `full_adder` instances in turn.

```
// gate-level hierarchical description of 4-bit adder
// gate-level description of half adder
module half_adder (x, y, s, c);
input  x, y;
output s, c;
// half adder body
// instantiate primitive gates
   xor (s,x,y);
   and (c,x,y);
endmodule
```



```
// gate-level description of full adder
module full_adder (x, y, cin, s, cout);
input  x, y, cin;
output s, cout;
wire   s1, c1, c2;  // outputs of both half adders
// full adder body
// instantiate the half adder
   half_adder ha_1 (x, y, s1, c1);
   half_adder ha_2 (cin, s1, s, c2);
   or (cout, c1, c2);
endmodule

// gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input  [3:0] x, y;
input  c_in;
output [3:0] sum;
output c_out;
wire   c1, c2, c3; // intermediate carries
// four_bit adder body
// instantiate the full adder
   full_adder fa_1 (x[0], y[0], c_in, sum[0], c1);
   full_adder fa_2 (x[1], y[1], c1, sum[1], c2);
   full_adder fa_3 (x[2], y[2], c2, sum[2], c3);
   full_adder fa_4 (x[3], y[3], c3, sum[3], c_out);
endmodule
```

In fact, the structural style is one way to model a complex digital system in a hierarchical manner. An example of a 4-bit adder constructed in a hierarchical manner is depicted in Figure 1.5. Here, the 4-bit adder is composed of four full-adders and then each full-adder is built by basic logic gates in turn. Although this example is quite simple, it manifests several important features when designing a large digital system.

1.3.3 Dataflow Modeling

The essential structure used to model a design in the dataflow style is the continuous assignment. In a continuous assignment, a value is assigned onto a net. It must be a net because continuous assignments are used to model the behavior of combinational logic circuits. A continuous assignment starts with the keyword `assign` and has the syntax:

```
assign [delay] l_value = expression;
```

Anytime the value of an operand used in the `expression` changes, the `expression` is evaluated and the result is assigned to `l_value` after the specified `delay`. The

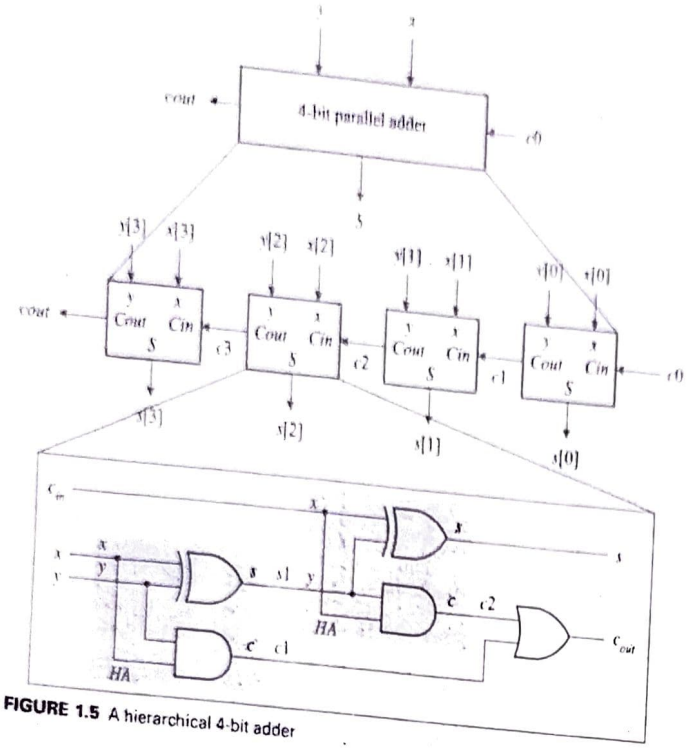


FIGURE 1.5 A hierarchical 4-bit adder

`delay` specifies the amount of time between a change of operand used in the expression and the assignment to `l_value`. If no `delay` is specified, the default is zero delay. Continuous assignments in a module execute concurrently regardless of the order they appear.

The following example illustrates how a continuous assignment is used to describe the 1-bit adder (i.e. full adder) depicted in Figure 1.6.

EXAMPLE 1.5 *A Full Adder Modeling in Dataflow Style*

In this example, we assume that the adder requires 5 time units to complete its operations. The delay will be ignored by the synthesis tools when the module is

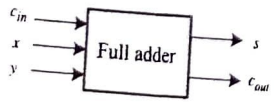


FIGURE 1.6 The block diagram of a full adder

synthesized because the delay will be replaced by the actual delays of the gates used to realize the adder.

```
module full_adder_dataflow(x, y, c_in, sum, c_out);
// I/O port declarations
input  x, y, c_in;
output sum, c_out;

// specify the function of a full adder
  assign #5 {c_out, sum} = x + y + c_in;
endmodule
```

■

1.3.4 Behavioral Modeling

The behavioral style uses the following two procedural constructs: `initial` and `always`. The `initial` statement can only be executed once and therefore is usually used to set up initial values of variable data types whereas the `always` statement, as the name implies, is executed repeatedly. The `always` statements are used to model combinational or sequential logic. Each `always` corresponds to a piece of logic.

The `l_value` used in an expression within an `initial` or `always` statement must be a variable data type, which retains its value until a new value is assigned. All `initial` statements and `always` statements begin their execution at simulation time 0 concurrently.

The following example illustrates how a procedural construct is used to describe the 1-bit adder (i.e. full adder) depicted in Figure 1.6.

EXAMPLE 1.6 *A Full Adder Modeling in Behavioral Style*

Basically, the expression used to describe the operations of a 1-bit full adder in behavioral style is the same as that of the dataflow style except that it needs to be put inside an `always` statement. In addition, the `@(x, y, c_in)` is used to sensitize the changes of input signals.

```
module full_adder_behavioral(x, y, c_in, sum, c_out);
// I/O port declarations
input  x, y, c_in;
output sum, c_out;
reg    sum, c_out;  // sum and c_out need to be declared
                    //  as reg types.

// specify the function of a full adder
always @(x, y, c_in)// can also use always @(*) or
                    // always@(x or y or c_in)
  #5 {c_out, sum} = x + y + c_in;
endmodule
```

■

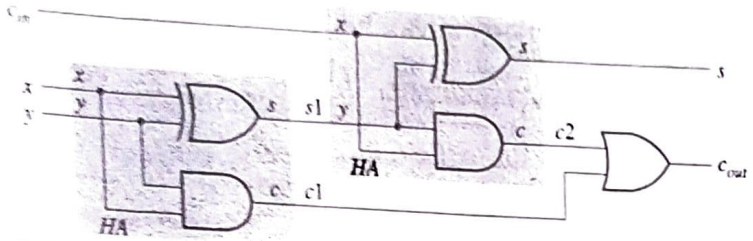


FIGURE 1.7 A full adder constructed with basic logic gates

1.3.5 Mixed-Style Modeling

As mentioned above, the mixed-style modeling is usually used to construct a hierarchical design in large systems. However, we are still able to model a simple design in mixed style. For example, as depicted in Figure 1.7 the full adder is constructed with two half adders and an OR gate. An example of the full adder being modeled in mixed style is shown in the following example.

EXAMPLE 1.7 *A Full Adder Module in Mixed Style*

The first half adder is modeled in structural style, the second half adder in dataflow style and the OR gate in behavioral style.

```
module full_adder_mixed_style(x, y, c_in, s, c_out);
// I/O port declarations
input  x, y, c_in;
output s, c_out;
reg    c_out;
wire   s1, c1, c2;
// structural modeling of HA 1.
   xor xor_ha1 (s1, x, y);
   and and_ha1(c1, x, y);
// dataflow modeling of HA 2.
   assign s = c_in ^ s1;
   assign c2 = c_in & s1;
// behavioral modeling of output OR gate.
always @(c1, c2) // can also use always @(*)
   c_out = c1 | c2;
endmodule
```

■

Review Questions

Q1.12 What are the differences of design, model, synthesis and implementation (also called realization)?

Q1.13 Describe the features of structural style.

Q1.14 What is the basic statement used in dataflow style?

Q1.15 What are the basic statements used in behavioral style?

Q1.16 Can we write a module by mixing the use of various modeling styles?

1.4 SIMULATION

For a design to be useful, it must be verified so as to make sure that it can correctly operate according to the requirement. Verilog HDL not only provides capabilities to model a design, but also provides facilities to generate and control stimuli, monitor and store responses, and check the results. In this section, we use a 4-bit adder as an example to illustrate how to verify a design entirely through the mechanism provided by Verilog HDL.

1.4.1 Basic Simulation Constructs

Two basic simulation structures in Verilog HDL are shown in Figure 1.8. The first structure is to take the unit under test (UUT) as an instantiated module in the stimulus module. This is often used in simple or small projects since it is intuitively simple to write. The second construct considers both stimulus block and UUT as the separate instantiated module at the top-level module. This is suitable for large projects. In this book, we will use the first structure when writing a test bench.

In general, a test bench comprises several basic parts: an instantiation of the UUT, stimulus generation and control, response monitoring and storing, and result checking. Except for the instantiation of the UUT, the rest of the parts are usually modeled in behavioral style. In the rest of this section, we deal with these features of a typical test

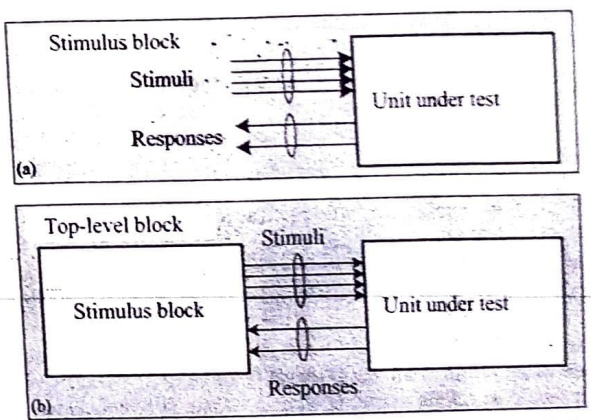


FIGURE 1.8 Two basic constructs for a simulation: (a) stimulus block at the top-level module; (b) stimulus block considered as a separate module

hardware. And just as fluency in a natural language comes by speaking, reading, and writing the language, mastery of Verilog comes by repeated use of the language to create models for various digital systems.

Since Verilog is a hardware description language, it differs from an ordinary programming language in several ways. Most importantly, Verilog has statements that execute concurrently since they must model real hardware in which the components are all in operation at the same time. It is popularly used for the purposes of describing, documenting, simulating, and automatically generating hardware. Hence, its constructs are tailored for these purposes. We will present the various methods to model different kinds of digital hardware using examples in the following sections.

Common Abbreviations

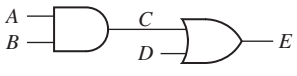
VHDL:	VHSIC hardware description language
VHSIC:	very-high-speed integrated circuit
HDL:	hardware description language
CAD:	computer-aided design
EDA:	electronic design automation
LSI:	large-scale integration
MSI:	medium-scale integration
SSI:	small-scale integration
VLSI:	very-large-scale integration
ULSI:	ultra-large-scale integration
ASCII:	American standard code for information interchange
ISO:	International Standards Organization
ASIC:	application-specific integrated circuit
FPGA:	field-programmable gate array
PLA:	programmable logic array
PAL:	programmable array logic
PLD:	programmable logic device
CPLD:	complex programmable logic device
STA:	static timing analysis

2.3 Verilog Description of Combinational Circuits

The biggest difficulty in modeling hardware using a general-purpose computer language is representing concurrently operating hardware. Computer programs that you are normally accustomed to are sequences of instructions with a well-defined order. At any point of time during execution, the program is at a specific point in its flow, and it encounters and executes different parts of the program sequentially. In order to model combinational circuits that have several gates (all of which are always working simultaneously), one needs to be able to "simulate" the execution of several parts of the circuit at the same time.

Verilog models combinational circuits by what are called **concurrent statements or continuous assignments**. Concurrent statements (continuous assignments) are statements that are always ready to execute. These are statements that are evaluated any time and every time a signal on the right side of the statement changes.

We will start by describing a simple gate circuit in Verilog. If each gate in the circuit of Figure 2-3 has a 5 ns propagation delay, the circuit can be described by two Verilog statements as shown, where *A*, *B*, *C*, *D*, and *E* are signals. A signal in Verilog usually corresponds to a signal in a physical system. The symbol "&&" represents the AND gate and the symbol "||" represents the OR. The #5 indicates a delay symbol of 5 ns. The symbol "=" is the signal assignment operator, which indicates that the value computed on the right side is assigned to the signal on the left side. The **assign** statement is used to assign a value, as shown in Figure 2-3. When the statements in Figure 2-3 are simulated, the first statement will be evaluated any time *A* or *B* changes, and the second statement will be evaluated any time *C* or *D* changes. Suppose that initially $A = 1$ and $B = C = D = E = 0$. If *B* changes to 1 at time 0, *C* will change to 1 at time = 5 ns. Then *E* will change to 1 at time = 10 ns (assuming timescale is 1 ns).




```
assign #5 C = A && B;
assign #5 E = C || D;
```

FIGURE 2-3: A Simple Gate Circuit

Verilog signal assignment statements like the ones in the foregoing example are examples of concurrent statements or continuous assignments. The Verilog simulator monitors the right side of each concurrent statement, and any time a signal changes, the expression on the right side is immediately reevaluated. The new value is assigned to the signal on the left side after an appropriate delay. This is exactly the way the hardware works. Any time a gate input changes, the gate output is recomputed by the hardware and the output changes after the gate delay.

Unlike a sequential program, the order of the above concurrent statements is unimportant. If we write

```
assign #5 E = C || D;
assign #5 C = A && B;
```

the simulation results would be exactly the same as before. In general, a signal assignment statement has the form

```
assign [#delay] signal_name = expression;
```

The expression is evaluated when the statement is executed, and the signal on the left side is scheduled to change after `delay`. The square brackets indicate that `#delay` is optional; they are not part of the statement. If `#delay` is omitted, then the signal is updated immediately. Unlike in VHDL, Verilog simulators do not display delta delays for continuous assign statements. The delta delay is an infinitesimally

small delay used to maintain/indicate sequentiality between dependent events happening at the same time. Note that the time at which the statement executes and the time at which the signal is updated are not the same if delay is specified.

Even if a Verilog program has no explicit loops, concurrent statements execute repeatedly as if they were in a loop. Figure 2-4 shows an inverter with the output connected back to the input. If the output is 0, then this 0 feeds back to the input and the inverter output changes to 1 after the inverter delay, which is assumed to be 10 ns. Then the 1 feeds back to the input, and the output changes to 0 after the inverter delay. The signal *CLK* will continue to oscillate between 0 and 1 as shown in the waveform. The corresponding concurrent Verilog statement will produce the same result. If *CLK* is initialized to 0 the statement executes and *CLK* changes to 1 after 10 ns. Since *CLK* has changed, the statement executes again, and *CLK* will change back to 0 after another 10 ns. This process will continue indefinitely.

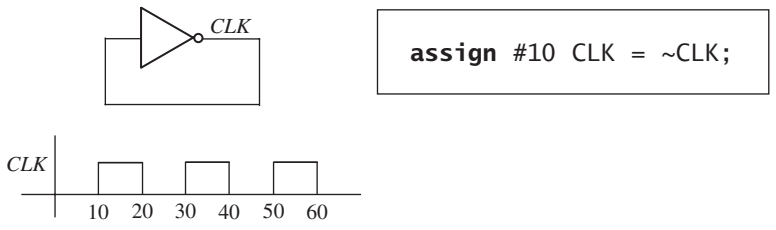


FIGURE 2-4: Inverter with Feedback

The statement in Figure 2-4 generates a clock waveform with a half period of 10 ns. On the other hand, if the concurrent statement

```
assign CLK = ~CLK;
```

is used, time will never advance to 1 ns.

In general, **Verilog is case sensitive**; that is, capital and lower-case letters are treated as different by the compiler and by the simulator. Thus, the statements

```
assign #10 Clk = ~Clk;
```

and

```
assign #10 CLK = ~CLK;
```

would result in two different clocks. Signal names and other **Verilog identifiers** may contain letters, numbers, the underscore character (_), and the dollar sign ($). An identifier must start with a letter or underscore character, and it cannot start with a number or a $ sign. The dollar sign ($) is reserved as the first character for **system tasks**. The following are valid identifiers:

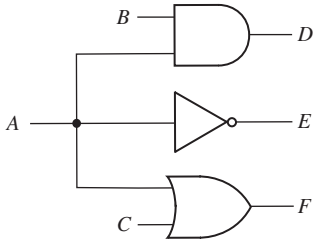
```
adder
Mux_input
_error_code
Index_bit
vector_sz
_$five
Count
XOR
```

The following are invalid identifiers:

```
4bitadder
$error_code
```

Every Verilog statement must be terminated with a semicolon. Spaces, tabs, and carriage returns are treated in the same way. This means that a Verilog statement can be continued over several lines, or several statements can be placed on one line. In a line of Verilog code, anything following a double slash (//) is treated as a comment to the end of the line. Comments for more than one line start with "/*" and end with "*/". Words such as **and, or**, and **always** are reserved words (or keywords) that have a special meaning to the Verilog compiler. In this text, we will put all reserved words in boldface type. Verilog reserved words (keywords) are shown in Appendix B.

Figure 2-5 shows three gates that have the signal *A* as a common input and the corresponding Verilog code. The three concurrent statements execute simultaneously whenever *A* changes, just as the three gates start processing the signal change at the same time. However, if the gates have different delays, the gate outputs can change at different times. If the gates have delays of 2 ns, 1 ns, and 3 ns, respectively, and *A* changes at time 5 ns, then the gate outputs *D*, *E*, and *F* can change at times 7 ns, 6 ns, and 8 ns, respectively. The Verilog statements work in the same way. Even though the statements execute simultaneously at 5 ns, the signals *D*, *E*, and *F* are updated at times 7 ns, 6 ns, and 8 ns.




```
// when A changes, these concurrent
// statements all execute at the
// same time
  assign #2 D = A && B;
  assign #1 E = ~A;
  assign #3 F = A || C;
```

FIGURE 2-5: Three Gates with a Common Input and Different Delays

In the foregoing examples, every signal is of type **wire** (or *net*), and it generally has a value of 0 or 1 (or 1'b0, 1'b1). In general, the net values in Verilog are represented as *<number of bits>'<base><value>*. The values on nets can be represented as binary, decimal, or hexadecimal indicated by `b`, `d`, and `h` respectively.

In digital design, we often need to perform the same operation on a group of signals. A one-dimensional array of bit signals is referred to as a vector. If a 4-bit vector named *B* has an index range 0 through 3, then the 4 elements of the **wire** or **reg** data types are designated *B[0], B[1], B[2], and B[3]*. One can declare a multiple bit wire using a statement such as

```
wire B[3:0];
```

The statement `B = 4'b1100` assigns 1 to *B[3]*, 1 to *B[2]*, 0 to *B[1]*, and 0 to *B[0]*.

Figure 2-6 shows an array of four AND gates. The inputs are represented by 4-bit vectors *A* and *B*, and the output by 4-bit vector *C,* where the && (logical AND operator) is used. Although we can write four Verilog statements to represent the four gates, it is much more efficient to write a single Verilog statement that performs the & (bitwise AND operator) operation on the vectors *A* and *B*. When applied to vectors, the & operator performs the bitwise AND operation on corresponding pairs of elements.

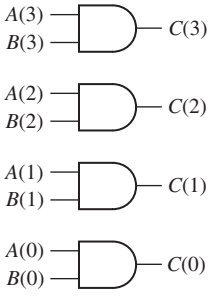


FIGURE 2-6: Array of AND Gates

```
// the hard way
assign C[3] = A[3] && B[3];
assign C[2] = A[2] && B[2];
assign C[1] = A[1] && B[1];
assign C[0] = A[0] && B[0];
```

```
// the easy way assuming C, A and
// B are 4-bit vectors
assign C = A & B;
```

2.4 Verilog Modules

The general structure of a Verilog code is a module description. A module is a basic building block that declares the input and output signals and specifies the internal operation of the module. As an example, consider Figure 2-7. The **module** declaration has the name `two_gates` and specifies the inputs and outputs. *A*, *B*, and *D* are input signals, and *E* is an output signal. The signal *C* is declared within the module as a **wire** since it is an internal signal. The two concurrent statements that describe the gates are placed and the module ends with **endmodule**. All the input and output signals are listed in the module statement without specifying whether they are input or output.

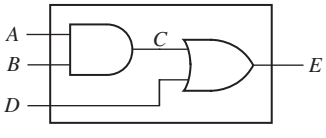


FIGURE 2-7: Verilog Module with Two Gates

```
module two_gates (A, B, D, E);
output E;

input A, B, D;

wire C;

  assign C = A && B; // concurrent
  assign E = C || D; // statements
endmodule
```

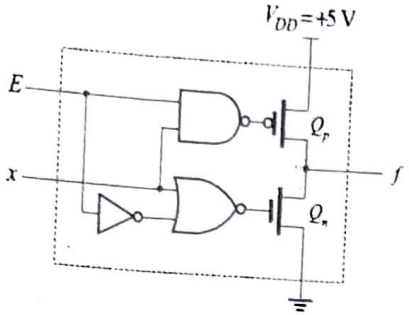


FIGURE 2.32 A logic diagram for problem 2.9

(a) Draw the logic circuit.

(b) Model the logic circuit at switch level.

2.9 Model the logic circuit shown in Figure 2.32 at gate level in structural style.

2.10 A pseudo-nMOS logic circuit implements the following switching expression:

$$f(x, y, z) = (x + yz)'$$

(a) Draw the logic circuit.

(b) Model the logic circuit at switch level.

2.11 A pseudo-nMOS logic circuit implements the following switching expression:

$$f(w, x, y, z) = [w(x + y) + z]'$$

(a) Draw the logic circuit.

(b) Model the logic circuit at switch level.

2.12 Use ideal MOS switches to model a switch with both rise and fall times of 0 and a turn-off time of 2 time units.

CHAPTER 3

DATAFLOW MODELING

THE RATIONALE behind dataflow modeling is on the observation that any digital system can be constructed by interconnecting registers and a combinational logic put between them for performing the required functions. A design modeled on the dataflow style has the following features:

1. Dataflow provides a powerful way to implement a design.
2. Automated tools are used to create a gate-level circuit from the description of a dataflow design. This process is often called *logic synthesis*.
3. RTL (register-transfer level) is a combination of dataflow and behavioral modeling.

A continuous assignment is the most basic statement of dataflow modeling. It is used to continuously drive a value onto a net. A continuous assignment begins with the keyword `assign` and is always active. It is usually used to model a combinational logic.

3.1 DATAFLOW MODELING

The essential element of dataflow modeling is the continuous assignment. Hence, in this section, we introduce the basic syntax of continuous assignment and give some insight into it.

3.1.1 Continuous Assignment

A continuous assignment is the most basic statement of dataflow modeling. It continuously drives a value onto a net. The assignment begins with the keyword `assign` and has the following syntax:

```
assign net_lvalue = expression;
assign net1 = expr1, net2 = expr2, ..., netn = exprn;
```

where `net_lvalue` is a scalar or vector net, or their concatenation. The second statement is a compact form of n continuous assignments. The operands used in the

Digital System Designs and Practices Ming-Bo Lin

expression can be variables or nets or function calls. Variables or nets can be scalar or vectors. Any logic function can be realized with continuous assignments. Continuous assignments can only update values of net data types such as `wire`, `triand` and so on.

Continuous assignments are always active. That is, whenever an operand in the right-hand-side expression changes value, the expression is evaluated and the new value is assigned to `net_lvalue`. Thus, it is quite appropriate to model the behavior of combinational logic. For example:

```
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

The above continuous assignment carries out a 4-bit addition with carry input. The braces ({ }) in the left-hand side is a concatenation operator, which will be described in detail later.

3.1.1.1 Net Declaration Assignment

Usually, we declare a net and then use a continuous assignment to assign a value onto it. In Verilog HDL, there is a shortcut for this. A continuous assignment can be placed on a net when it is declared – we call it *net declaration assignment* for convenience, such as the following example:

```
wire out;                  // normal continuous assignment
assign out = in1 & in2;
```

which is equivalent to

```
wire out = in1 & in2;    // net declaration assignment
```

Like regular continuous assignment, net declaration assignment is always active. Moreover, there can only be one net declaration assignment per net since a net can only be declared once.

3.1.1.2 Implicit Net Declaration

Implicit net declaration is a feature of Verilog HDL. An implicit net declaration will be inferred for a signal name when it is used on the left-hand side of a continuous assignment. For example:

```
wire in1, in2;
assign out = in1 & in2;
```

Note that `out` is not declared as a `wire`, but an implicit `wire` declaration for `out` is done by the simulator. It is a good practice to declare the net type explicitly and use regular continuous assignment to avoid any unintentional errors. However, implicit net declarations are often used with input and output ports of modules (see Section 6.1.1).

3.1.2 Expressions

The essence of dataflow modeling is using expressions that consist of sub-expressions, operators and operands, instead of primitive gates, or instantiations of modules or UDPs

used in structural style. Expressions are constructs that combine operators and operands to produce results. In general, an expression has the following format:

```
expression = operators + operands
```

where operands can be any of the allowed data types and operators act on the operands to product the desired results.

The allowed operands in an expression can be one of the following: constants, net, `reg`, `integer`, `time`, `real`, `realtime`, bit-select, part-select, array element and function calls. We will describe these operands in the next section in greater detail.

Like C programming language, Verilog HDL has a rich set of operators, which include arithmetic, shift, bitwise, case equality, reduction, logic, relational, equality and miscellaneous operators. We will discuss these operators in a dedicated section of this chapter. Table 3.1 summarizes the operators, which are grouped together as similar operators, in Verilog HDL.

The precedence of operators in Verilog HDL is listed in Table 3.2. Operators on the same row have the same precedence. Rows are arranged in order of decreasing precedence for the operators. For example, *, / and % operators have the same precedence and are higher than the binary + and − operators. The left-to-right associativity applies to all operators except the conditional operator, which associates from right to left. Associativity refers to the order in which the operators having the same precedence are evaluated. Thus, in the following example y is added to x and then z is subtracted from the result of $x + y$:

$$x + y - z$$

When operators differ in precedence, the operators with the higher precedence associate first. In the following example, y is divided by z (division has higher precedence than

TABLE 3.1 The summary of operators in Verilog HDL

Arithmetic	Bitwise	Reduction	Relational
+: add	~: NOT	&: AND	>: greater than
−: subtract	&: AND	|: OR	<: less than
*: multiply	|: OR	~ &: NAND	>=: greater than or equal
/: divide	^: XOR	~ |: NOR	<=: less than or equal
%: modulus	~^, ^~: XNOR	^: XOR	**Miscellaneous**
**: exponent	**Case equality**	~^, ^~ : XNOR	{,}: concatenation
Shift	===: equality	**Logical**	{c{ }}: replication
<<: left shift	!==: inequality	&&: AND	? : conditional
>>: right shift	**Equality**	||: OR	
<<<: arithmetic left shift	==: equality	!: NOT	
>>>: arithmetic right shift	!=: inequality		

TABLE 3.2 The precedence of operators in Verilog HDL

Operators	Symbols	Precedence
Unary	+ ! ~	Highest
Exponent	**	
Multiply, divide, modulus	* / %	
Add, subtract	+-	
Shift	<< >> <<< >>>	
Relational	< <= > >=	
Equality	== ! = === ! ==	
Reduction	& ~ &	
	^ ^ ~	
	| ~|	
Logical	&&	
	||	
Conditional	?:	Lowest

addition) and then the result is added to x:

$$x + y/z$$

Parentheses can be used to change the operator precedence. For instance, $(x + y)/z$ is not the same as $x + y/z$. It is good practice to use parentheses whenever there may be ambiguities.

3.1.3 Delays

The time between when any operand in the right-hand side changes and when the new value of the right-hand side is assigned to the left-hand side is controlled by the delay value. The delay value is specified after the keyword `assign`. The inertial delay model is used as the default model. The delay value of a continuous assignment can be specified by using the following syntax:

```
assign #delay net_lvalue = expression;
assign #delay1 net1 = expr1,
       #delay2 net2 = expr2,
       ...,
       #delayn netn = exprn;
```

where `#delay`, `#delay1`, ... `#delayn` are specified exactly in the same way as that of gate primitives. That is, there may be zero-value, one-value, two-value or three-value delay specifications using a delay specifier in the form of `min:typ:max`.

For example:

```
wire in1, in2, out;
assign #10 out = in1 & in2; // delay is 10 time units
```

A net declaration assignment is used to specify both the delay and assignment on the net. In this case, the delay is inertial delay rather than transport delay because it is equivalent to first declaring a net and then doing a continuous assignment with regular delay.

```
// net declaration continuous assignment with delay
wire #10 out = in1 & in2;

// regular continuous assignment with delay
wire out;
assign #10 out = in1 & in2;
```

A net can be declared associated with a delay value. The transport delay model is used by default in this case. Net declaration delays can also be used in gate-level modeling.

```
// net delays
wire #10 out;
assign out = in1 & in2;

// regular assignment delay
wire out;
assign #10 out = in1 & in2;

// regular assignment delay
wire #5 out;
assign #10 out = in1 & in2;
```

Review Questions

Q3.1 Explain the meaning of dataflow.

Q3.2 What is net declaration assignment?

Q3.3 What is the basic statement in dataflow style?

Q3.4 Describe how to assign delays to continuous assignments.

Q3.5 Explain the meaning of implicit net declaration.

3.2 OPERANDS

The operands in an expression can be any of constants, parameters, nets, variables (`reg`, `integer`, `time`, `real`, `realtime`), bit-select, part-select, array element and function calls. The various net data types have been discussed in the previous chapters and so we do not repeat them here again. A parameter is like a constant and is declared using a `parameter` or `localparam` declaration (referring to Section 6.1.2 for details). A function call may also be used as an operand in an expression. It can be either a user-defined function or a system function (see Sections 5.2 and 5.3 for details). In this section, we address constants, variable data types, bit-selects and part-selects, array and memory elements.

3.2.1 Constants

There are three types of constant in Verilog HDL. These are *integer*, *real* and *string*.

3.2.1.1 Integer Constants Integer constants can be specified in decimal, hexadecimal, octal or binary format. There are two forms to express integer constants: *simple decimal form* and *base format notation*.

Simple Decimal Form In the simple decimal form, a number is specified as a sequence of digits 0 through 9, optionally beginning with a plus (+) or minus (−) unary operator. For example:

```
-123   // is decimal -123
12345  // is decimal 12345
```

An integer value in this form represents a signed number. A negative number is represented in two's complement form.

Base Format Notation When the base format notation is used, a number is composed of up to three parts: an optional size constant, a single quote followed by a base format character and the digits representing the value of the number.

```
[size]'[s/S][base_format]base_value
```

where " ' " is a single quote. The `size` specifies the size of the constant in number of bits. It is specified as a non-zero unsigned decimal number. For example, the size specification for two hexadecimal digits is 8, because one hexadecimal digit requires 4 bits.

The `base_format` consists of a case-insensitive character specifying the base for the number, optionally preceded by the single character s (or S) to indicate a signed (in two's complement form) quantity, preceded by the single quote character ('). The allowed bases are decimal (d or D), hexadecimal (h or H), octal (o or O) and binary (b or B).

The `base_value` consists of digits (0 to 9 and a to f) that are legal for the specified base format. The `base_value` should immediately follow the base format, optionally preceded by white space. The hexadecimal digits a(A) to f(F) are case-insensitive. To represent a signed (two's complement) number, a single character "s or S" must be preceded with the `base_format`.

```
 4'b1001     // a 4-bit binary number
16'habcd     // a 16-bit hexadecimal number
   2006      // unsized number -- a 32-bit decimal
             // number by default
  'habc      // unsized number -- a 32-bit hexadecimal
             // number
 4'sb1001    // a 4-bit signed number, it represents
             // -7.
-4'sb1001    // a 4-bit signed number, it represents
             // -(-7) = 7.
```

In order to improve readability, Verilog HDL allows us to use the underscore character (_) anywhere in a number except as the first character. The underscore character is ignored.

```
16'b1101_1001_1010_0000 // a 16-bit number in binary form
8'b1001_0001            // an 8-bit number in binary form
```

An x represents the unknown value and a z represents the high-impedance value. An x (z) represents 4 x (z) bits in the hexadecimal base, 3 x (z) bits in the octal base, and a 1 x (z) bit in the binary base. In Verilog HDL, the question mark (?) can be used to improve readability in the case where the high-impedance value is a 'don't-care' condition. Hence, when used in a number, the question-mark character is an alternative for the z character.

```
16'hxxbc          // the number is 16'bxxxx_xxxx_1011_1100
16'hzzbc          // the number is 16'bzzzz_zzzz_1011_1100
16'b01??_1001_11?0_??00 //a 16-bit number in binary form
8'b01??_11??            //equivalent to an 8'b01zz_11zz
```

If the size specified for the constant is larger than the size of the `base_value`, the `base_value` will be padded to the left with zeros or a sign bit, depending on whether the `base_value` is an unsigned or a signed quantity. If the left-most bit in the `base_value` is an x or a z, then the left bits of the `base_value` will be padded with xs or zs.

```
 8'bx001          // the number is 8'bxxxx_x001
 8'bzz011         // the number is 8'bzzzz_z011
16'hx8            // the number is 16'bxxxx_xxxx_xxxx_1000
```

```
16'b1101_1001    // the number is 16'b0000_0000_1101_1001
16'sb1101_1001   // the number is 16'b1111_1111_1101_1001
```

In the last one, the left bits are padded with sign bit ("1") since the `base_value` is a two's complement number, declared by `s`.

If the size specified for the constant is smaller than the size of the `base_value`, the left-most bits of `base_value` are truncated in order to fit the specified size.

```
 8'b1110_1101_1001   // the number is 8'b1101_1001
10'sb1001_1001_0001  // the number is 10'b01_1001_0001
```

For the unsized constant, the size of the `base_value` is by default at least 32 bits. Hence, the `base_value` will be padded to the left with something like that for when the size is specified.

```
'b1110_1101_1001    // 32'b0000_0000_0000_0000_0000_
                    // 1110_1101_1001
'sb1001_1001_0001   // 32'b1111_1111_1111_1111_1111_
                    // 1001_1001_0001
```

3.2.1.2 Real Constants Real numbers can be specified in either decimal notation or in scientific notation. Real numbers expressed with a decimal point must have at least one digit on each side of the decimal point.

```
1.5          //
 .3          // illegal ---
1294.872     //
1.44E9       // the exponent symbol can be e or E
1.50e-7
0.1e-0
15E12
32E-6
26.176_45_e-12  // underscores are ignored)
```

The conversion of real numbers to integers is implicitly defined by the language. Real numbers are converted to integers by rounding the real number to the nearest integer. Implicit conversion takes place when a real number is assigned to an integer. For examples:

```
  1.5     // yields 2 when converted into an integer
  0.3     // yields 0 when converted into an integer
 23.445   // yields 23 when converted into an integer
-245.56   // yields -246 when converted into an
          // integer
```


3.2.1.3 String Constants A string is a sequence of characters enclosed by double quotes (" ") and may not be split into multiple lines. One character is represented as an 8-bit ASCII code. Hence, a string is considered as an unsigned integer constant represented by a sequence of 8-bit ASCII codes.

String variables are variables of `reg` type with widths equal to eight times the number of characters in the string, such as in the following example.

EXAMPLE 3.1 ***An Example of String Manipulation***

In this example, assume that a string `str1` is employed to store the string "`Welcome to Digital World!`". Hence, it requires a `reg` 25*8 or a width of 200 bits. Another string `str2` is used to store the string "`Hello!`". It needs a `reg` 6*8 or a width of 48 bits. After the execution of the program, the output is:

```
Welcome to Digital World!
Hello! is stored as: 48656c6c6f21
```

```
module string_test;
// internal signal declarations:
reg [25*8:1] str1;
reg [6*8:1] str2;
initial begin
    str1 = ``Welcome to Digital World!'';
    $display(``%s\n'', str1);
    str2 = ``Hello!'';
    $display(``%s is stored as: %h\n'', str2, str2);
end
endmodule
```

■

The backslash (\) character can be used to escape certain special characters.

```
\n    \\ new line character
\t    \\ tab character
\\    \\ \ character
\''   \\ '' character
\ddd  \\ a character specified in 3 octal digits.
```

Review Questions

Q3.6 What is the meaning of a parameter?

Q3.7 Can a function call be used as an operand in the expression of a continuous assignment?

Q3.8 What are the three constants provided by Verilog HDL?

Q3.9 Explain the meaning of an `s` qualifier in base format notation of integer constants.

Q3.10 At least how many bits are defaulted when a constant is unsized?

3.2.2 Data Types

Verilog HDL has two classes of data types: *nets* and *variables*. Nets mean any hardware connection points and variables represent any data storage elements. Note that net is not a reserved word but represents a class of data types, as we have described in the previous chapter. A net data type can be referenced anywhere in a module and must be driven by a primitive, continuous assignment `assign`, `force/release` statement or module ports. The details of net data types have been described in Chapter 2; hence, we omit them here.

3.2.2.1 *Variable Data Types*

A variable represents a data storage element. It stores a value from one assignment to the next. The variable data types include five different kinds: `reg`, `integer`, `time`, `real` and `realtime`. All `reg`, `time` and `integer` variable data types are initialized with an unknown value, `x`, and `real` as well as `realtime` variables data types are initialized with 0.0.

The `reg` Variable A `reg` variable holds a value between assignments. It may be used to model hardware registers such as edge-sensitive (i.e. flip-flops) and level-sensitive (i.e. latches) storage elements. However, a `reg` variable need not actually represent a hardware storage element because it can also be used to represent combinatorial logic. The declaration of a `reg` variable may use the following form:

```
reg [signed] [[msb:lsb]] id1[= const_expr1],...,
   idn[= const_exprn];
```

where the initialization part `const_expr` is optional. A `reg` variable may be declared to store a signed value by using the keyword `signed` following `reg`. The default value of a `reg` variable is unsigned; `[msb:lsb]` is an optional part and is used to specify the range of the `reg` variable. The use and meaning of `msb` and `lsb` are the same as in the case of net so we do not repeat here. If no range is specified, it defaults to a 1-bit reg.

```
reg a, b, c_in;      // reg a, b, and c_in are 1-bit reg
                     // variables
reg [7:0] data_a;    // data_a is an 8-bit reg, the msb
                     // is bit 7
reg [0:7] data_b;    // data_b is an 8-bit reg, the msb
                     // is bit 0
reg signed [7:0] d;// d is an 8-bit signed reg
```

Vector Versus Scalar It is more convenient to describe a bundle of signals as a basic unit when describing a hardware module. This bundle of signals is usually called a vector (multiple bit width) in Verilog HDL. All net data types the `reg` variable can be declared as vectors. The default type of net and `reg` variable data types is a 1-bit vector or is called scalar.

The `integer` Variable An `integer` variable contains integer values and can be used as a general-purpose variable used for modeling high-level behavior. The syntax for declaring `integer` is as follows:

```
integer id1[= const_expr1],..., idn[= const_exprn];
```

where the initialization part `const_expr` is optional. The `integer` variables use the same assignment rules as `reg` variables. An `integer` variable is treated as a signed `reg` variable with the lsb being bit 0 and has at least 32 bits, regardless of implementations. Arithmetic operations performed on integer variables produce two's complement results.

```
integer  i,j;        // declare two integer variables
integer  data[7:0]; // array of integer
```

The `time` Variable. A `time` variable is used for storing and manipulating simulation time quantities. It is typically used in conjunction with the `$time` system task. The syntax for declaring `time` is as follows:

```
time id1[= const_expr1],..., idn[= const_exprn];
```

where the initialization part `const_expr` is optional. A `time` variable holds only unsigned value and is at least 64 bits, with the least significant bit being bit 0.

```
time events;         // hold one time value
time current_time;   // hold one time value
```

The `real` and `realtime` Variables The Verilog HDL supports `real` and `realtime` variable data types in addition to `integer` and `time`. The `real` and `realtime` are identical and can be used interchangeably. The syntax for declaring `real` or `realtime` is as follows:

```
real|realtime identifier1, ... ,identifiern;
```

Both `real` and `realtime` variables cannot use range declaration and their initial values are defaulted to zero (0.0).

```
real     events;        // declare a real variable
realtime current_time;  // hold current time as real
```

Review Questions

Q3.11 What can drive a net?

Q3.12 What variable data type can be used to model a hardware storage element?

Q3.13 What are the differences between `reg` and `integer` variables?

Q3.14 What are the features of `time` variable?

Q3.15 Describe the features of `real` and `realtime` variables.

3.2.3 Bit-Select and Part-Select

A bit-select extracts a particular bit from a vector. As mentioned previously, all net data types and the `reg` variable data type can be declared as vectors. Although `integer` and `time` are not allowed to be declared as vectors, they can also be accessed by bit-select or part-select. However, bit-select or part-select of `real` and `realtime` is not allowed. The bit-select has the form:

```
vector_name [bit_select_expr]
```

where `vector_name` can be any vector of net, `reg`, `integer` and `time` data types. The `bit_select_expr` can be an expression. If the `bit_select_expr` evaluates to an `x` or a `z`, or if it is out of bounds, then the bit-select value is an `x`.

In a part-select, a contiguous sequence of bits of a vector is selected. There are two forms of part-select: a *constant* part-select and an *indexed* part-select. A constant part-select has the following form:

```
vector_name [msb_const_expr:lsb_const_expr]
```

where `vector_name` can be any vector of net, `reg`, `integer` and `time` data types. The `msb_const_expr` and `lsb_const_expr` must be constant expressions. The following are some examples:

```
reg  [15:0] data_bus; // declarations
wire [7:0] a;
integer response_time;

data_bus[3:0]       // reg variable part-select
a [4:3]             // net part-select
response_time[5:0] // integer variable part-select
```

An indexed part-select has the following form:

```
vector_name [<starting_bit>+:const_width]
vector_name [<starting_bit>-:const_width]
```

where `vector_name` can be any vector of net, `reg`, `integer` and `time` data types. The `starting_bit` may be a variable but the `width` has to be constant. The range of bits selected is the index specified by `starting_bit` plus or minus the numbers specified by `const_width`; "`+:`" indicates that part-select increases from the `starting_bit`; "`-:`" indicates that part-select decreases from the `starting_bit`.

For example:

```
data_bus[8+:8]    // select data_bus[15:8]
data_bus[15-:4]   // select data_bus[15:12]
```

Like the case of bit-select, if either the range index is out of bounds or evaluates to an `x` or a `z`, the part-select value is an `x`.

The following example shows a simple application of the indexed part-select.

EXAMPLE 3.2 ***Conversion of a Big Endian to a Little Endian***

By properly setting the starting bits and ranges, the following module converts a big endian into its corresponding little endian and vice versa.

```
module swap_bytes (in, out);
input  [31:0] in;
output [31:0] out;
// using indexed part-select
assign out [31 -: 8] = in [0  +: 8],
       out [23 -: 8] = in [8  +: 8],
       out [15 -: 8] = in [16 +: 8],
       out [7  -: 8] = in [24 +: 8];
endmodule
```

■

Note that a net or variable may be declared with the keyword `signed` as a signed data type. The default is unsigned. However, bit-select and part-select results are unsigned regardless of the operands.

Vectored and Scalared Usually, a vector net can be accessed by bit-select and part-select. However, when a vector net does not allow it to be accessed by bit-select or part-select, the keyword `vectored` may be specified in the net declaration. The syntax is as follows:

```
net_name [vectored|scalared] [signed] [range] [delay]
  identifiers;
```

When no keyword `vectored` is specified, the default is `scalered`, which is the same as when the keyword `scalared` is specified, and bit-selects and part-selects are permitted to access the vector net. For example:

```
wire scalared [63:0] bus64; // a bus that will be
                            // expanded
tri  vectored [31:0] data;  // data is dealt with as
                            // a unit
```

Review Questions

Q3.16 Explain the meanings of bit-select and part-select.

Q3.17 Can we disable a vector to be accessed by bit-select or part-select?

Q3.18 What is a constant part-select?

Q3.19 What is an indexed part-select?

Q3.20 Can bit-select and part-select be applied to `integer` and `time` variables?

Q3.21 Can bit-select and part-select be applied to `real` and `realtime` variables?

3.2.4 Array and Memory Elements

Arrays can be used to group elements into multi-dimensional objects. Although only nets and the `reg` variable can be declared as vectors, all net and variable data types are allowed to be declared as multi-dimensional arrays. A multi-dimensional array is declared by specifying the address ranges after the declared identifier, called *dimension*, one for each. In addition, an array element can be a scalar or a vector if the element is a net or `reg` data type. The vector size (defined by `range`) specifies the number of bits in each element and the dimensions (i.e. address ranges) specify the number of elements in each dimension of the array. The syntax is as follows:

```
net_type [signed][range]id[msb:lsb]{[msb:lsb]};
reg      [signed][range]id[msb:lsb]{[msb:lsb]};
integer  id[msb:lsb]{[msb:lsb]};
time     id[msb:lsb]{[msb:lsb]};
```

where `msb` and `lsb` are constant-valued expressions that indicate the range of indices of a dimension. If no dimensions are specified, each net or variable only stores a value. For example:

```
wire    a[3:0]; // a scalar wire array of 4 elements
reg     d[7:0]; // a scalar reg array of 8 elements
wire    [7:0]  x[3:0];   // an 8-bit wire array of
                         // 4 elements
reg     [31:0] y[15:0];  // a 32-bit reg array of
                         // 16 elements
integer states [3:0];    // an integer array of
                         // 4 elements
time    current[5:0];    // a time array of 6 elements
```

To access an element from an array, we use the following format:

```
array_name[addr_expr]{[addr_expr]}
```

where `addr_expr` can be any expression. As with bit-select or part-select, if `addr_expr` is out of bounds, or if any bit in the `addr_expr` is an `x` or a `z`, then the reference value is an `x`. Only an element of an array can be assigned a value in a single assignment; an entire or partial array dimensions cannot be assigned to another using a single assignment. However, a bit-select or part-select of an element of an array may be accessed and assigned. To assign a value to an element of an array, it needs to specify

an index for every dimension. The index can be an expression.

```
states[3] = 33559; // assign decimal number to integer
                   // in array
current[t_index] = $time; // assign current simulation
                          // time to element indexed
                          // by integer index
```

3.2.4.1 Memory Memory is a basic module in any digital system; in Verilog HDL, it is simply declared as a one-dimensional `reg` array. A memory can be used to model a read-only memory (ROM), a random access memory (RAM) and a register file. Reference to a memory may be made to a whole word or a portion of a word of memory.

To access a memory word, we use the following format:

```
mem_name[addr_expr]
```

where `addr_expr` can be any expression. For example:

```
reg  [7:0] mema [7:0];       // one-dimensional array
                             // of 8-bit vector
reg  [7:0] memb [3:0][3:0]; // two-dimensional array
                             // of 8-bit vector
wire sum [7:0][3:0];         // two-dimensional array
                             // of scalar wire

mema = 0;     // illegal -- attempt to write to entire
              // array
memb[1] = 0; // illegal -- it needs two indices
memb[1][3:1] = 0; // illegal -- attempt to write to
                  // partial array

mema[1] = 0;      // assigns 0 to the second element
                  // of mema
memb[1][0] = 3;   // assigns 3 to the element[1][0]
```

Bit-select and part-select of a memory element or an array element of an array are allowed. To do this, the desired element is first selected by using normal array access. Then bit-select and part-select are applied to the selected element in the same manner as in the cases of vectors.

```
mema[4][3]       // the 3rd bit of the 4th element
mema[5][7:4]     // the higher four bits of the 5th
                 // element
```

```
memb[3][1][1:0]   // the lower two bits of the [3][1]th
                  // element
sum[5][0]         // the [5][0]th element
```

Note that a memory of n 1-bit `reg` variables is different from an n-bit vector `reg`.

```
reg [1:n] rega; // an n-bit register is not the same
reg mema [1:n]; // as a memory of n 1-bit registers
```

Memory indirection is allowed and can be specified in a single expression, for example:

```
mem_name[mem_name[23]] // use memory indirections
```

In this example, `mem_name[23]` addresses the 23rd word of the memory mem_name. The value at the word is then used as the address to access mem_name.

Review Questions

Q3.22 What kinds of data types can be declared as an array?

Q3.23 How would you declare a multi-dimensional array in Verilog HDL?

Q3.24 How would you distinguish between vector size and address range in an array declaration?

Q3.25 What is a memory from the viewpoint of Verilog HDL?

Q3.26 Can an array be assigned to another in a single statement?

Q3.27 How would you access a bit in a word of a memory?

3.3 OPERATORS

As shown in Table 3.1, Verilog HDL has a rich set of operators, including arithmetic operators, bit-wise operators, reduction operators, concatenation and replication operators, logical operators, relational operators, equality operators, shift operators and conditional operators. In this section, we describe these operators in detail and give some examples showing how to use them to model actual logic circuits.

3.3.1 Bit-wise Operators

Bit-wise operators perform a bit-by-bit operation on two operands and produce a vector result. In bit-wise operations, a `z` is treated as an unknown `x`. When two operands are not of equal length, the shorter operand is zero-extended to match the length of the longer operand.

TABLE 3.3 The bitwise operators

Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^,^~	Bitwise exclusive nor

The bit-wise operators include five operators: & (and), | (or), ^ (xor), ^~(xnor) and ~(negation), as shown in Table 3.3. The functions of these operators are as follows:

- & (and): if any bit is 0, the result is 0, or else if both bits are 1, then the result is 1; otherwise the result is an x.
- | (or): if any bit is 1, the result is 1, or else if both bits are 0, the result is 0; otherwise the result is an x.
- ^ (xor): if one bit is 1 and the other is 0, the result is 1, or else if both bits are 0 or 1, the result is 0; otherwise the result is an x.
- ^~(xnor): if one bit is 1 and the other is 0, the result is 0, or else if both bits are 0 or 1, the result is 1; otherwise the result is an x.
- ~(negation): if the input bit is 1 the result is 0, or else if the input bit is 0, the result is 1; otherwise the result is an x.

The following module uses bit-wise operators to model a 4-to-1 multiplexer.

EXAMPLE 3.3 ***A Dataflow Model for a 4-to-1 Multiplexer***

In this example, we use a continuous assignment to directly realize the switching expression derived from Figure 2.5. The resulting expression uses & (and), | (or) and ~(negation) bit-wise operators.

```
module mux41_dataflow(i0, i1, i2, i3, s1, s0, out);
// port declarations
input i0, i1, i2, i3;
input s1, s0;
output out;
// using basic and, or, not logic operators.
   assign out = (~s1 & ~s0 & i0) |
                (~s1 &  s0 & i1) |
                ( s1 & ~s0 & i2) |
                ( s1 &  s0 & i3) ;
endmodule
```

■

The synthesized result of the above module `mux41_dataflow` is shown in Figure 3.1.

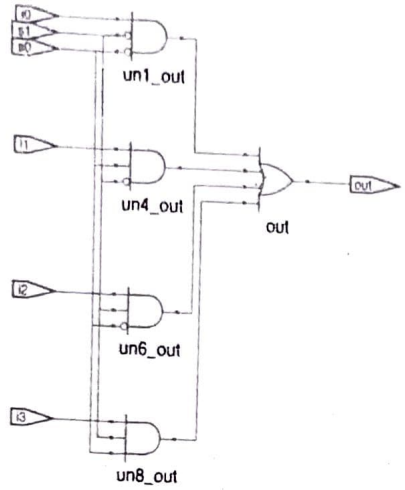


FIGURE 3.1 A dataflow model for a 4-to-1 MUX

Review Questions

Q3.28 Describe the basic operations of the set of bit-wise operators.

Q3.29 What operations are taken when two operands are of unequal length?

Q3.30 Describe the operation of & (and) using the four-value set.

Q3.31 Describe the operation of ^ (xor) using the four-value set.

Q3.32 Describe the operation of ~(negation) using the four-value set.

3.3.2 Arithmetic Operators

The set of arithmetic operators contains six members {+, −, *, /, %, **}, as shown in Table 3.4. The operators + and − perform addition and subtraction of two numbers, respectively. In Verilog HDL, negative numbers are represented in two's complement form. The operators + and − can also be used as unary operators to represent signed numbers. The operators * and / compute the multiplication and division of two numbers, respectively. The integer division (/) truncates any fractional part toward zero, while the modulus operator (%) produces the remainder from the division of two numbers.

TABLE 3.4 The arithmetic operators

Symbol	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponent (power)
%	Modulus

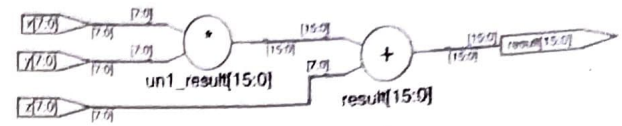


FIGURE 3.2 A multiply and accumulate unit

The exponent operator computes the power of a number. The result of the exponent operator ** is `real` if either operand is real, integer or signed.

The following example is a simple application of arithmetic operators.

EXAMPLE 3.4 ***A Multiply and Accumulate Unit***

In this example, we directly use the arithmetic operators, + and *, to construct a multiply and accumulate unit. Depending on the synthesis tool and the target technology library, the multiply operator * may be synthesized by using a hardware macro, basic gates or cells organized by an internal algorithm inherent to the tool.

```
// an example to illustrate arithmetic operators
module multiplier_accumulator(x, y, z, result);
input  [7:0] x, y, z;
output [15:0] result;
   assign result = x * y + z ;
endmodule
```

■

The synthesized result of the preceding example is shown in Figure 3.2.

The following example is a simple application of divide arithmetic operator.

EXAMPLE 3.5 ***An Unsigned Divider***

In this example, we directly use the divide arithmetic operator to construct an unsigned divider. Depending on the synthesis tool and the target technology library, the divide operator / may be synthesized by using a hardware macro, basic gates or cells organized by an internal algorithm inherent to the tool.

```
// an example to illustrate the divide operator
module divide_operator(x, y, result);
input  [7:0] x, y;
output [7:0] result;
   assign result = x / y;
endmodule
```

■

The following module segment shows the use of the exponent operator (**):

```
parameter ADDR_SIZE = 4;
localparam ROM_SIZE = 2 ** ADDR_SIZE - 1;
```

The exponent (power) operator is usually not supported by synthesis tools except that it can be evaluated to an integer power of 2. The details of `parameter` and `localparam` will be dealt with in Section 6.1.2.

A `reg` variable is treated as an unsigned value unless explicitly declared to be signed. An `integer` variable is treated as signed. In addition, for an arithmetic operation, if any bit of an operand is an `x` or a `z`, then the entire result value would be an `x`.

The size of the result of an arithmetic expression is determined by the size of the largest operand and the left-hand-side target as well if it is an assignment. This rule is also applied equally well to all intermediate results of an expression.

EXAMPLE 3.6 ***The Result Size of an Expression***

Both sizes of result and intermediate results of the first continuous assignment are determined by inputs `a` and `b`, and net variable `c`, which is 4 bits. The result size of the second continuous assignment is 8 bits because the largest operand is 8 bits.

```
// an example to illustrate arithmetic operators
module arithmetic_operator(a, b, e, c, d);
input  [3:0] a, b;
input  [6:0] e;
output [3:0] c;
output [7:0] d;
   assign c = a + b;
   assign d = a + b + e;
endmodule
```

■

The synthesized result of the preceding example is shown in Figure 3.3.

In general, an unsigned value is stored in a net, a `reg` variable, and an integer in base format without a signed qualifier (the `s`). A signed value is stored in a signed net, a signed `reg` variable, an `integer` variable, an integer in decimal form and an integer in base format with a signed qualifier (the `s`).

In an expression with mixed signed and unsigned operands, all operands are converted to unsigned before any operation take places. The conversion of a number between signed and unsigned format can also be accomplished by system functions: `$signed` and `$unsigned`, which are dealt with in Section 5.3.4.

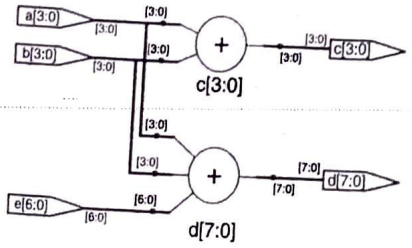


FIGURE 3.3 The synthesized result of the `arithmetic_operator` module

EXAMPLE 3.7 ***Mixed Signed and Unsigned Operands***

The first expression is carried out in two's complement but the second expression is performed in unsigned numbers because input `e` is an unsigned number.

```
// an example to illustrate arithmetic operators
module arithmetic_signed(a, b, e, c, d);
input  signed [3:0] a, b;
input  [6:0] e;
output signed [3:0] c;
output [7:0] d;

   assign c = a + b;
   assign d = a + b + e;
endmodule
```

■

Review Questions

Q3.33 Describe the operation of the `%` (modulus) arithmetic operator.

Q3.34 Describe the operation of the `**` (exponent) arithmetic operator.

Q3.35 Which data types are unsigned by default?

Q3.36 What will be the result type (signed or unsigned) when mixed signed and unsigned operands are used in an expression?

Q3.37 How would you determine the result size of an arithmetic expression?

3.3.3 Concatenation and Replication Operators

A concatenation operator is expressed by { and }, with commas separating the expressions within it, as shown in Table 3.5. The operands of a concatenation operator must be sized. Operands can be scalar nets or `reg` variables, vector nets or `reg` variables, bit-select, part-select or sized constants. For example:

```
y = {a, b[0], c[1]};
```

TABLE 3.5 The concatenation and replication operators

Symbol	Operation
{,}	Concatenation
{const_expr {}}	Replication

concatenates a, b[0] and c[1] into a group in that order and assigns to y. Another example is as follows:

```
y = {a, b[0], 4'b1};
```

which concatenates a, b[0] and 4'b1 into a group and assigns to y. Here, 4'b1 corresponds to 1'b1, 1'b1, 1'b1 and 1'b1 (see Section 3.2.1).

Another form of concatenation is the replication operation, which is expressed as {const_expr{}}, as shown in Table 3.5. The first expression, const_expr, must be a non-zero, non-x and non-z constant expression, while the second expression follows the rules for concatenation. For instance, {4{a}} replicates "a" 4 times. A replication operator specifies how many times to replicate the number inside the braces. For example:

```
y = {a, {4{b[0]}}, c[1]};
```

The right-hand side has six bits in total.

The following example shows how the concatenation operator is used to concatenate a scalar and a vector net operand.

EXAMPLE 3.8 ***A 4-bit Adder Illustrates the Use of the Concatenation Operator***

In this example, a 4-bit adder is described in dataflow style. The left-hand side of the continuous assignment is a concatenation of a scaler c_out and a vector sum and forms a 5-bit result.

```
module four_bit_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input  [3:0] x, y;  // declare as a 4-bit array
input  c_in;
output [3:0] sum;   // declare as a 4-bit array
output c_out;

// specify the function of a 4-bit adder.
   assign {c_out, sum} = x + y + c_in;
endmodule
```

The following example deals with the use of a replication operator.

EXAMPLE 3.9 ***A 4-bit Two's Complement Adder***

This example uses a replication operator to make four copies of input carry c_in, then combines with an input operand y through an xor operator to form a true/one's complement generation circuit. By using this circuit, a 4-bit two's complement adder is constructed.

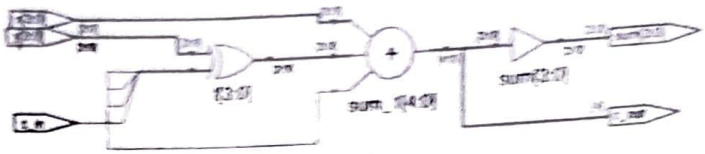


FIGURE 3.4 *A two's complement 4-bit adder*

```
module twos_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input  [3:0] x, y;  // declare as a 4-bit array
input  c_in;
output [3:0] sum;   // declare as a 4-bit array
output c_out;
wire   [3:0] t;     // outputs of xor gates

// specify the function of a two's complement adder
   assign t = y ^ {4{c_in}};
   assign {c_out, sum} = x + t + c_in;
endmodule
```

The synthesized result of the above twos_adder module is shown in Figure 3.4.

3.3.4 Reduction Operators

The set of unary reduction operators carries out a bit-wise operation on a single vector operand and yields a 1-bit result. Reduction operators only perform on one vector operand and work in a bit-by-bit way from right to left. The reduction operators are shown in Table 3.6.

A simple application of reduction operator for modeling the 9-bit parity generator described in the previous chapter is as in the following example.

EXAMPLE 3.10 ***A 9-bit Parity Generator Using a Reduction Operator***

As mentioned above, the operand of a reduction operator must be a vector. In this example, the vector x is reduced to a 1-bit result by the reduction operator ^ (xor). The synthesized result is shown in Figure 3.5.

TABLE 3.6 The reduction operators

Symbol	Operation
&	Reduction and
~&	Reduction nand
|	Reduction or
~|	Reduction nor
^	Reduction exclusive or
~^, ^~	Reduction exclusive nor

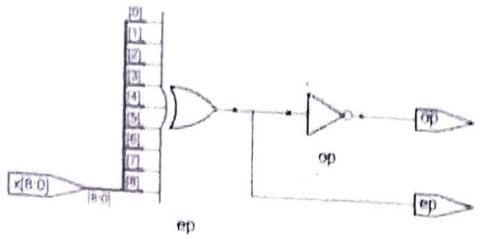


FIGURE 3.5 A 9 bit parity generator using a reduction operator

```
module parity_gen_9b_reduction(x, ep,op);
// I/O port declarations
input  [8:0] x;
output ep, op;
// dataflow modeling using reduction operator
   assign ep = ^x;   // even parity generator
   assign op = ~ep;  // odd parity generator
endmodule
```

■

From the preceding example, we can see the power of reduction operators. By using these operations, we may write a very compact and readable module. Another simple but useful application is to detect whether all bits in a byte are zeros or ones. In the following example, we assume that both results are needed.

EXAMPLE 3.11 ***An All-bit-zero/One Detector***

The outputs `zero` and `one` are assigned to 1 if all bits of the input vector `x` are zeros and ones, respectively. These two detectors are easily implemented by the reduction operators | (or) & (and). The synthesized result is shown in Figure 3.6.

```
module all_bit_01_detector_reduction(x, zero,one);
// I/O port declarations
input   [7:0] x;
output  zero, one;
// dataflow modeling
   assign zero = ~(|x);  // all-bit zero detector
   assign one = &x;      // all-bit one detector
endmodule
```

■

Review Questions

Q3.38 Describe the operation of the concatenation operator {}.

Q3.39 Describe the operation of the replication operator {const_expr{}}.

Q3.40 What are the differences between bit-wise operators and reduction operators?

Q3.41 Describe the operation of the reduction operator & (and).

Q3.42 Describe the operation of the reduction operator ^ (xor).

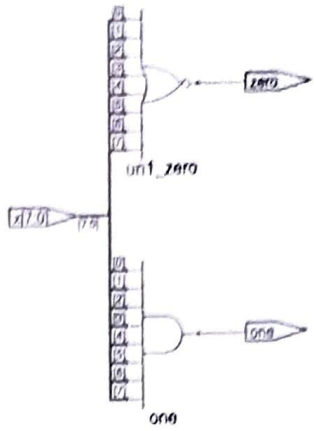


FIGURE 3.6 An all-bit-zero/one detector

3.3.5 Logical Operators

There are three logical operators, (&&) (and), (||) (or) and ! (negation), as shown in Table 3.7. They operate on the logical values 0 or 1 and produce a 1-bit value 0, 1 or an x. If any operand bit is x or z, it is treated to x and is treated as a false condition by simulators. For vector operands, a non-zero vector is treated as a 1.

For example, in the following example, if `reg c` holds the integer value `123` and `d` holds the value 0, then `a` is `1` and `b` is `0`.

```
reg a, b;
reg [7:0] c, d;

a = c || d;  // a is set to 1
b = c && d;  // b is set to 0
```

A comment about ! (negation) is that it often uses a construct like `if (!reset)` to represent an equivalent one: `if (reset == 0)`.

3.3.6 Relational Operators

Relational operators include four operators, > (greater than), < (less than), >= (greater than or equal to) and <= (less than or equal to), as shown in Table 3.8. Relational operators return the logical value 1 if the expression is true and 0 if the expression is

TABLE 3.7 The logical operators

Symbol	Operation
!	Logical negation
&&	Logical and
||	Logical or

TABLE 3.8 The relational operators

Symbol	Operation
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

false. The expression results in a value x if there are any unknown (x) or z bits in the operands.

When two operands are not of equal length and any operand is unsigned, the smaller operand is zero-extended to match the size of the larger operand and the operation is performed between two unsigned values. If both operands are signed, the smaller operand is sign-extended and the operation is performed between signed values. When either operand is a 'real', then the other operand is converted to an equivalent real value and the operation is performed between two real values.

All relational operators have the same precedence and their precedence is lower than that of arithmetic operators.

3.3.7 Equality Operators

The set of equality operators include four operators, == (logical equality), != (logical inequality), === (case equality) and !== (case inequality), as shown in Table 3.9. Equality operators return the logical value 1 if the expression is true and 0 if the expression is false. These operators compare the two operands bit by bit, zero-extended if the operands are not of equal length. The logical equality operators (==, !=) yield an unknown x if either operand has x or z in its bits. The case equality operators (===, !==) yield a 1 if the two operands match exactly and 0 if the two operands do not match exactly.

A magnitude comparator is often used in digital systems to compare two unsigned numbers. In the following example, we employ relational and equality operators to describe a 4-bit magnitude comparator in dataflow style.

EXAMPLE 3.12 ***A 4-bit Magnitude Comparator***

Because the operation of a magnitude comparator is to compare and indicate the relative magnitude of two input numbers, logical equality (==) and relational operators, greater than (>) and less than (<) are used to construct the desired circuit. In order to be

TABLE 3.9 The equality operators

Symbol	Operation
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

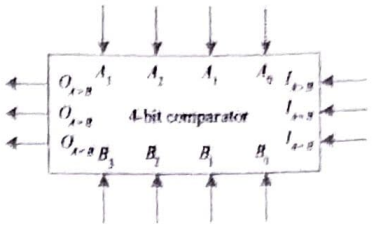


FIGURE 3.7 A 4-bit comparator

cascadable, three additional inputs are also incorporated into the module, as shown in Figure 3.7. Consequently, the overall result is equality if both results from the current and the preceding modules are equal. The overall result is greater (less) than if the current result is greater (less) than or the current result is equal to but the result from the preceding modules is greater (less) than.

```
module four_bit_comparator(Iagtb, Iaeqb, Ialtb, a, b,
 Oagtb, Oaeqb, Oaltb);
// I/O port declarations
input  [3:0] a, b;
input  Iagtb, Iaeqb, Ialtb;
output Oagtb, Oaeqb, Oaltb;
// dataflow modeling using relation operators
   assign Oaeqb = (a == b) && (Iaeqb == 1);  // equality
   assign Oagtb = (a > b) || ((a == b) && (Iagtb == 1));
                             // greater than
   assign Oaltb = (a < b) || ((a == b) && (Ialtb == 1));
                             // less than
endmodule
```

The synthesized result of the above `four_bit_comparator` module is shown in Figure 3.8.

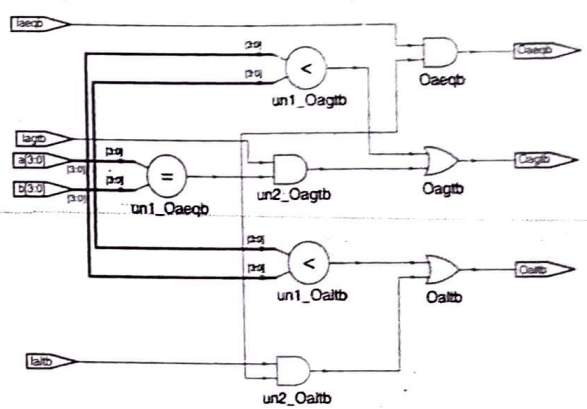


FIGURE 3.8 The synthesized result of a cascadable 4-bit comparator

TABLE 3.10 The shift operators

Symbol	Operation
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift

Although we may depend on the precedence of operators and write a correct expression, such as in the following example:

```
a < s - 1 && b != c && d != f;
```

it is good practice to use parentheses to indicate very clearly the precedence intended for improving readability, as in the following rewriting example:

```
(a < s - 1) && (b != c) && (d != f);
```

3.3.8 Shift Operators

There are two types of shift operators, the logical shift operators, << and >>, and the arithmetic shift operators, <<< and >>>, as shown in Table 3.10. Operators << and >> are logical left and right shifts, respectively, while operators <<< and >>> are arithmetic left and right shifts, respectively. The shift operation shifts the left operand by the number specified by the right operand. The vacant bit positions are filled with zeros for logical and arithmetic left-shift operations and filled with the MSBs (sign bits) for arithmetic right-shift operation.

The following example shows a simple application of a logical left-shift operator.

EXAMPLE 3.13 ***A Simple Application of a Logical Left-shift Operator***

In this example, both `start` and `result` are declared as `reg` variables and a logical left-shift operator is applied to `start`. The `result` variable is assigned the binary value `0100`, which is `0001` shifted to the left two positions and zero-filled.

```
module logical_shift;
reg [3:0] start, result;
initial begin
   start = 4'b0001;
   result = (start << 2);
end
endmodule
```

■

The following example shows a simple application of an arithmetic right-shift operator.

EXAMPLE 3.14 ***A Simple Application of an Arithmetic Right-shift Operator***

In this example, both `result` and `start` are declared as `signed reg` variables and the arithmetic right-shift operator is applied to `start`. The `result` variable is assigned the binary value `1110`, which is `1000` shifted to the right two positions and sign-extended.


```
module arithmetic_shift;
reg signed [3:0] start, result;
initial begin
   start = 4'b1000;
   result = (start >>> 2);
end
endmodule
```

■

The following example compares the differences between logical and arithmetic right shifts. You are encouraged to write a test bench, simulate the program and see what happens.

EXAMPLE 3.15 ***An Example Which Illustrates Logical and Arithmetic Shifts***

This example explains the differences between logical and arithmetic right shifts. The logical right shift fills the vacant bits with 0 and the arithmetic right shift fills the vacant bits with the sign bit, known as a sign-bit extension.

```
// an example illustrates logical and arithmetic shifts
module arithmetic_shift(x, y, z);
input  signed [3:0] x;
output [3:0] y;
output signed [3:0] z;
   assign y = x >> 1;   // logical right shift
   assign z = x >>> 1;  // arithmetic right shift
endmodule
```

■

Note that the net variables x and z must be declared with the keyword `signed`. It is advised to replace the net variable with an unsigned net (i.e. remove the keyword `signed`) and see what happens.

3.3.9 Conditional Operator

The conditional operator selects an expression based on the value of the condition expression. It has the form:

```
condition_expr ? true_statement: false_statement
```

where `condition_expr` is evaluated first. If the result is true then the `true_statement` is executed; otherwise, the `false_statement` is evaluated. This operator is equivalent to the following:

```
if  (condition_expr) true_statement
else false_statement
```

A simple example for describing a 2-to-1 multiplexer is as follows:

```
assign out = selection ? input1: input0;
```

The input `input1` is passed to the output end `out` if the selection is 1; otherwise, the input `input0` is passed to the output end `out`.

A more complicated example of using a conditional operator to model a 4-to-1 multiplexer is shown in the following example.

EXAMPLE 3.16 ***A 4-to-1 Multiplexer Constructed by Using the Conditional Operator***

In this example, we use nested conditional operators; two are inside the outmost conditional operator. You are asked to check the correctness of this module by using a synthesizer and observing the synthesized result. Although a construct like this is quite concise, it is not easy to understand for a naive reader.

```
module mux4_to_1_cond (i0, i1, i2, i3, s1, s0, out);
// port declarations from the I/O diagram
input  i0, i1, i2, i3;
input  s1, s0;
output out;
// using conditional operator (?:)
   assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```

■

The following example rewrites the nested conditional operator used in the preceding example in another more readable form. You may compare both constructs and see the difference between them. Actually, both constructs produce exactly the same gate-level circuits. Check them on your system.

EXAMPLE 3.17 ***A 4-to-1 Multiplexer Constructed by Using the Case Equality and Conditional Operators***

In this example, we explicitly use the case equality operator (`===`) to determine which input will be routed to the ouput. The selection signal `sel` is compared with the constant *i*, where *i* runs from 0 to 3, in order. When the comparison is exactly equal to each other, the input is assigned to `out`. Indeed, this is exactly the same as in the use of the `case` statement, which will be introduced in Section 4.4.2. If we change the case equality operator (`===`) into a logical equality (`==`), what will happen?

```
module mux41_equality(i0, i1, i2, i3, sel, out);
// port declarations
input  i0, i1, i2, i3;
input  [1:0] sel;
output out;
// using case and conditional operators.
assign out = (sel === 0) ? i0 :
             (sel === 1) ? i1 :
             (sel === 2) ? i2 :
             (sel === 3) ? i3 : 4'bz;
endmodule
```

■

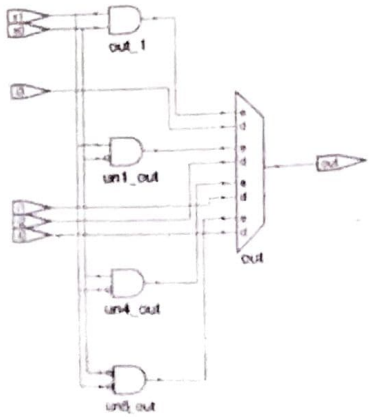


FIGURE 3.9 A 4-to-1 MUX constructed by using the conditional operator

The synthesized results of both modules described previously are the same, as depicted in Figure 3.9.

Review Questions

Q3.43 Describe the operation of logical operators.

Q3.44 Describe the operation of relational operators.

Q3.45 Describe the operation of the set of equality operators.

Q3.46 What is the basic difference between logical equality (`==`) and case equality (`===`)?

Q3.47 What is the basic difference between an arithmetic right shift (`>>>`) and a logical right shift (`>>`)?

Q3.48 Describe the operations of the conditional operator (`? :`).

SUMMARY

The rationale behind dataflow modeling is on the observation that any digital system can be constructed by interconnecting registers and put between them a combinational logic for performing the desired functions. A continuous assignment is the most basic statement of dataflow modeling. It is used to continuously drive a value onto a net. Continuous assignments are always active.

A continuous assignment begins with the keyword `assign` followed by an expression. An expression is a construct that combines operators and operands to produce a result. An expression consists of sub-expressions, operators and operands. The operands in an expression can be any of the following: constants, parameters, nets, variables (`reg`, `integer`, `time`, `real`, `realtime`), bit-select,

part-select, array element and function calls. The set of operators includes arithmetic, shift, bit-wise, case equality, reduction, logic, relational, equality and miscellaneous operators.

Most of the operators can be accepted by synthesis tools. Only a few of them cannot, or only a limited form can be accepted. The exponent and modulus might be two such examples – check on your system. Through using the dataflow style, a combinational logic circuit can be modeled by using continuous assignments rather than structural connections between instances of modules.

REFERENCES

1. J. Bhasker, *A Verilog HDL Primer*, 3rd Edn, Star Galaxy Publishing, 2005.
2. S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd Edn, SunSoft Press, 2003.
3. IEEE 1364-2001 Standard, *IEEE Standard Verilog Hardware Description Language*, 2001.

PROBLEMS

3.1 Simplify the following switching expression and use bit-wise operators to model it:

$$f(w, x, y, z) = \Sigma(5, 6, 7, 9, 10, 11, 13, 14, 15)$$

3.2 Simplify the following switching expression and use bit-wise operators to model it:

$$f(w, x, y, z) = \Sigma(0, 4, 5, 7, 8, 9, 13, 15)$$

3.3 Using bit-wise operators, model the logic circuit shown in Figure 3.10.

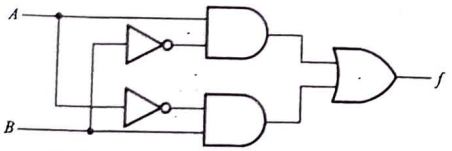


FIGURE 3.10 A logic diagram for problem 3.3

3.4 Using bit-wise operators, model the logic circuit shown in Figure 3.11.

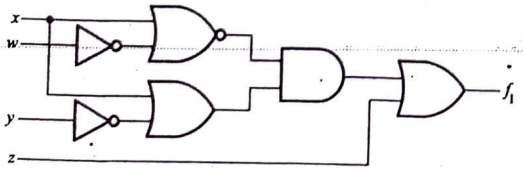


FIGURE 3.11 A logic diagram for problem 3.4

3.5 Using the multiply (*) operator, write a module to compute $x * y$, where x and y are two signed numbers. Synthesize the module on your system and check the synthesized result. Write a test bench to verify whether the module behaves correctly.

3.6 Using the divide (/) operator, write a module to compute x/y, where x and y are two signed numbers. Synthesize the module on your system and check the synthesized result. Write a test bench to verify whether the module behaves correctly.

3.7 Using the modulus (%) operator, write a module to compute $x\%y$, where x and y are two unsigned numbers. Synthesize the module on your system and check the synthesized result. Write a test bench to verify whether the module behaves correctly.

3.8 By assuming that both numbers are signed, re-do problem 3.7.

3.9 Suppose that an arithmetic and logic unit (ALU) has the function shown in Table 3.11. The output is a function of six mode-selection lines, $m5$ to $m0$.

TABLE 3.11 A function table for problem 3.9

m5	m4	m3	m2	m1	m0	ALU output
0	0	0	1	0	0	A .and B
0	0	1	0	0	0	A .and .not B
0	0	1	0	0	1	A .xor B
0	1	1	0	0	1	A plus .not B plus carry
0	1	0	1	1	0	A plus B plus carry
1	1	0	1	1	0	.not A plus B plus carry
0	0	0	0	0	0	A
0	0	0	0	0	1	A .or B
0	0	0	1	0	1	B
0	0	1	0	1	0	.not B
0	0	1	1	0	0	zero

(a) Derive the 1-bit slice switching expression of the output as the function of the mode selection lines $m5$ to $m0$.

(b) Write a module at the dataflow level to perform the output function. Write a test bench to verify whether the module behaves correctly.

(c) By instantiating the above module, construct a 4-bit ALU. Write a test bench to verify whether the module behaves correctly.

3.10 Using the conditional operator, write a module to shift the input `data` right logically by the number of positions specified by another input `shift`, ranging from 0 to 3.

3.11 Using the conditional operator, write a module to shift the input `data` right arithmetically by the number of positions specified by another input `shift`, ranging from 0 to 3.

3.12 Using the conditional operator, write a two's complement adder. Write a test bench to verify whether the module behaves correctly.

Figure 2-6 shows an array of four AND gates. The inputs are represented by 4-bit vectors *A* and *B*, and the output by 4-bit vector *C,* where the && (logical AND operator) is used. Although we can write four Verilog statements to represent the four gates, it is much more efficient to write a single Verilog statement that performs the & (bitwise AND operator) operation on the vectors *A* and *B*. When applied to vectors, the & operator performs the bitwise AND operation on corresponding pairs of elements.

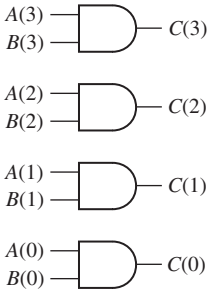


FIGURE 2-6: Array of AND Gates

```
// the hard way
assign C[3] = A[3] && B[3];
assign C[2] = A[2] && B[2];
assign C[1] = A[1] && B[1];
assign C[0] = A[0] && B[0];
```

```
// the easy way assuming C, A and
// B are 4-bit vectors
assign C = A & B;
```

2.4 Verilog Modules

The general structure of a Verilog code is a module description. A module is a basic building block that declares the input and output signals and specifies the internal operation of the module. As an example, consider Figure 2-7. The **module** declaration has the name `two_gates` and specifies the inputs and outputs. *A*, *B*, and *D* are input signals, and *E* is an output signal. The signal *C* is declared within the module as a **wire** since it is an internal signal. The two concurrent statements that describe the gates are placed and the module ends with **endmodule**. All the input and output signals are listed in the module statement without specifying whether they are input or output.

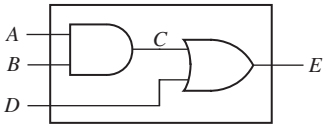


FIGURE 2-7: Verilog Module with Two Gates

```
module two_gates (A, B, D, E);
output E;

input A, B, D;

wire C;

  assign C = A && B; // concurrent
  assign E = C || D; // statements
endmodule
```

The **module** I/O declaration part can be considered as the black box picture of the module being designed and its external interface; that is, it represents the interconnections from this module to the external world as in Figure 2-8.

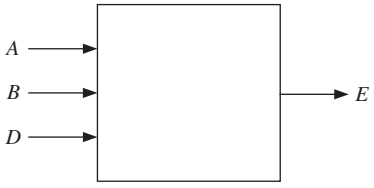


FIGURE 2-8: Black Box View of the 2-Gate Module

Just as in this simple example, when we describe a system in Verilog, we must specify input and output signals and also specify the functionalities of the module that are part of the system (see Figure 2-9). Each module declaration includes a list of interface signals that can be used to connect to other modules or to the outside world. We will use module declarations of the form:

```
module module-name (module interface list);
[list-of-interface-ports]
...
[port-declarations]
...
[functional-specification-of-module]
...
endmodule
```

The items enclosed in square brackets are optional. The `list-of-interface-ports` normally has the following form:

```
type-of-port list-of-interface-signals
{; type-of-port list-of-interface-signals};
```

The curly brackets indicate zero or more repetitions of the enclosed clause. Type-of-port indicates the direction of information; whether information is flowing into the port or out of it. Input port signals are of keyword **input**, output port signals are of keyword **output**, and bidirectional signals are of keyword **inout**. Also, list-of-ports can be combined with the module interface list.

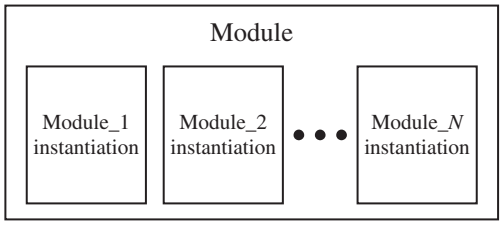
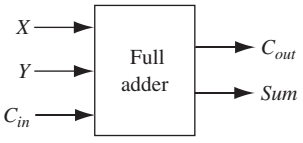


FIGURE 2-9: Verilog Program Structure

In the `port-declarations` section, we can declare internal signals that are used within the module. The module contains other module instances that describe the operation of the module.

Next, we will write a Verilog module for a full adder. A full adder adds two bit inputs and a carry input to generate a sum bit and a carry output bit. As shown in Figure 2-10, the port declaration specifies that X, Y, and C_{in} are input signals of type bit and that C_{out} and *Sum* are output signals of type bit.

FIGURE 2-10: Verilog Module for a Full Adder




```
module FullAdder(X, Y, Cin, Cout, Sum);
output Cout, Sum;
input X, Y, Cin;
  assign #10 Sum = X ^ Y ^ Cin;
  assign #10 Cout = (X && Y) || (X && Cin) || (X && Cin);
endmodule
```

In this example, the Verilog assignment statements for *Sum* and C_{out} represent the logic equations for the full adder. The specified equations are

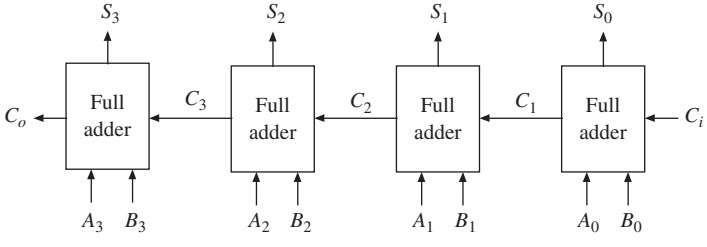
$$\texttt{Sum} = \texttt{X} \oplus \texttt{Y} \oplus \texttt{Cin}$$
$$\texttt{Cout} = \texttt{XY} \oplus \texttt{YCin} + \texttt{XCin}$$

Several other architectural descriptions such as a truth table or an interconnection of gates could have been used instead. In the C_{out} equation, parentheses are required around `(X && Y)` since Verilog does not specify an order of precedence for the logic operators except the NOT operator.

Four-Bit Full Adder

Next, we will show how to use the `FullAdder` module defined previously as a **module** in a system, which consists of four full adders connected to form a 4-bit binary adder (see Figure 2-11). We first declare the 4-bit adder as another module (see Figure 2-12). Since the inputs and the sum output are four bits wide, we declare them as a 4-bit vector and they are dimensioned `[3:0]`. (We could have used a range `[1:4]` instead).

FIGURE 2-11: 4-Bit Binary Adder



Next, we instantiate the `FullAdder` module within the module of `Adder4` (Figure 2-12).

Following the I/O port declaration, we declare a 3-bit internal carry signal C as a data type **wire**. After that, we create several instances of the `FullAdder`

component. (In CAD jargon, we "instantiate" four copies of the FullAdder.) Each copy of FullAdder has a port map. The port map corresponds one-to-one with the signals in the component port. Thus, *A[0]*, *B[0]*, and *Ci* correspond to the inputs *X*, *Y*, and C_{in}, respectively. *C[1]* and *S[0]* correspond to the C_{out} and *Sum* outputs of the adder for least significant bit. Unconnected ports can be omitted. In case the signals are not connected to the ports by name, the order of the signals in the port map must be the same as the order of the signals in the port of the module declaration. In this example, we use the ports in order, a method called **positional association**. The other method called **named association** is described in Chapter 8. Note that the order of the signals in named association can be in any order as long as the signals in the module are connected to the ports by name.

FIGURE 2-12: Structural Description of a 4-Bit Adder

```
module Adder4 (S, Co, A, B, Ci);
output [3:0] S;
output Co;
input  [3:0] A, B;
input  Ci;

wire   [3:1] C;  // C is an internal signal

// instantiate four copies of the FullAdder

FullAdder FA0 (A[0], B[0], Ci,   C[1], S[0]);
FullAdder FA1 (A[1], B[1], C[1], C[2], S[1]);
FullAdder FA2 (A[2], B[2], C[2], C[3], S[2]);
FullAdder FA3 (A[3], B[3], C[3], Co,   S[3]);

endmodule
```

In preparation for simulation, we can place the modules for the FullAdder and for Adder4 together in one project and compile. Some tools may require them to be in the same file.

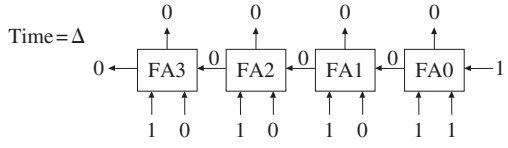
All of the simulation examples in this text use the **ModelSim Verilog simulator** from Mentor Graphics. Most other Verilog simulators use similar command files and can produce output in a similar format. The simulator command file is usually called the **do file**. We will use the following simulator commands to test Adder4:

```
add list A B Co C Ci S  // put these signals on the output list
force A 1111            // set the A inputs to 1111
force B 0001            // set the B inputs to 0001
force Ci 1              // set Ci to 1
run 50ns                // run the simulation for 50ns
force Ci 0
force A 0101
force B 1110
run 50ns
```

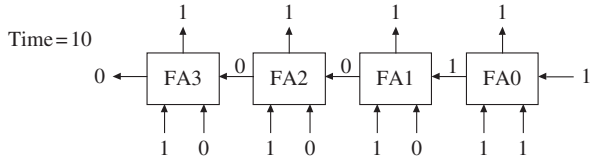
We have chosen to run the simulation for 50 ns for each input set, since this is more than enough time for the carry to propagate through all of the full adders. The simulation results for the above command list are:

ns	delta	a	b	co	c	ci	s
0	+0	1111	0001	x	xxx	1	xxxx
10	+0	1111	0001	x	xx1	1	xxx1
20	+0	1111	0001	x	x11	1	xx01
30	+0	1111	0001	x	111	1	x001
40	+0	1111	0001	1	111	1	0001
50	+0	0101	1110	1	111	0	0001
60	+0	0101	1110	1	110	0	0101
70	+0	0101	1110	1	100	0	0111
80	+0	0101	1110	1	100	0	0011

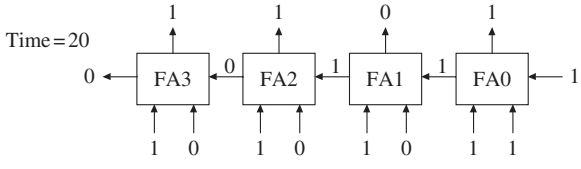
The listing shows how the carry propagates one position every 10 ns. The x values can be avoided by initializing the `S`, `C`, and `Co` values to 0. Under that condition, the simulation progresses as follows



The sum and carry are computed by each FA and appear at the FA outputs 10 ns later:



Since the inputs to FA1 have changed, the outputs change 10 ns later:



The final simulation results are:

1111 + 0001 + 1 = 0001 with a carry of 1 (at time = 40 ns) and
0101 + 1110 + 0 = 0011 with a carry of 1 (at time = 80 ns).

The simulation stops at 80 ns since no further changes occur after that time.

Use of "Inout" Mode

Let us consider the example in Figure 2-13. Assume that all variables are 0 @ 0 ns, but *A* changes to 1 @ 10 ns.

FIGURE 2-13: Verilog Code Illustrating Use of Output as an Input Signal

```
module gates (A, B, C, D, E);
input  A, B, C;
output D, E;

assign #5 D = A || B;  // statement 1
assign #5 E = C || D;  // statement 2 uses D as an input

endmodule
```

The code in Figure 2-13 will actually compile, simulate, or synthesize in most tools even though *D* is declared only as an output. Statement 2 uses *D* as an input. In VHDL, *D* should be strictly in either **in** or **inout** mode, but Verilog is not that strict in compilation. The **output** mode can also be used as an input in a statement inside the same module, but **inout** has to be used as in Figure 2-14 if *D* has to be used as input and output by other modules.

FIGURE 2-14: Verilog Code Illustrating Use of Mode Inout

```
module gates (A, B, C, D, E);
input  A, B, C;
output E;
inout  D;

assign #5 D = A || B;  // statement 1
assign #5 E = C || D;  // statement 2

endmodule
```

All signals remain at 0 until time 10 ns. The change in *A* at 10 ns results in statement 1 reevaluating. The value of *D* becomes 1 at time equal to 15 ns. The change in *D* at time 15 ns results in statement 2 reevaluating. Signal *E* changes to 1 at time 20 ns. The description represents TWO gates, each with a delay of 5 ns.

2.5 Verilog Assignments

There are two types of assignment in the Verilog: continuous assignments and procedural assignments. Continuous assignments are used to assign values for combinational logic circuits. The **assign** keyword can be used after the net is separately declared, which is referred to as explicit continuous assignments. Implicit continuous assignments assign the value in declaration without using the **assign** keyword. The following examples show the difference between explicit and implicit continuous assignments.

```
wire C;
assign C = A || B;    // explicit continuous assignment
wire D = E && F;      // implicit continuous assignment
```

Procedural assignments are used to model registers and finite state machines using the **always** keyword. More details on procedural assignment are explained in the following section.

2.6 Procedural Assignments

The concurrent statements from the previous section are useful in modeling combinational logic. Combinational logic constantly reacts to input changes. In contrast, synchronous sequential logic responds to changes dependent on the clock. Many input changes might be ignored since output and state changes occur only at valid conditions of the clock. Modeling sequential logic requires primitives to model selective activity conditional on clock, edge-triggered devices, sequence of operations, and so forth. There are two types of procedural assignments in Verilog. **Initial** blocks execute only once at time zero, whereas **always** blocks loop to execute over and over again. In other words, the initial block execution and always block execution starts at time 0. Always block waits for the event, whereas initial block just executes all the statements without waiting. Verilog also has a procedural assign statement that can be used inside the always block, but we do not use them in this book.

In this unit, we will learn **initial** and **always** statements, which help to model sequential logic. Initial blocks are useful in simulation and verification, but only always blocks are synthesized.

Initial Statements

An initial statement has the following basic form:

```
initial
begin
  sequential-statements
end
```

Always Statements

An always statement has the following basic form:

```
always @(sensitivity-list)
begin
  sequential-statements
end
```

When an always statement is used, the statements between the **begin** and the **end** are executed sequentially rather than concurrently. The expression in parentheses after the word **always** is called a sensitivity list, and the process executes whenever any signal in the sensitivity list changes. The symbol "@" should be used before the sensitivity list. For example, if the always statement has the sensitivity list `@(A, B, C)`, then it executes whenever any one of *A*, *B*, or *C* changes. Whenever one of the signals in the sensitivity list changes, the sequential statements in the always block are executed in sequence one time. In earlier versions of Verilog, "or" is used to specify more than one element in the sensitivity list. In Verilog 2001, comma (,) is

also used in the sensitivity list. Starting with Verilog 2001, an always statement can be used with a * in the sensitivity list to cause the always block to execute whenever any signal changes. When a process finishes executing, it goes back to the beginning and waits for a signal on the sensitivity list to change again.

The variables on the left-hand side element of an = or <= in an always block should be defined as **reg** data type. Any other data type including **wire** is illegal.

The assignment operator "=" indicates concurrent execution when used outside an always block. When the statements

```
C = A && B;  // concurrent statements
E = C || D;  // when used outside always block
```

are used outside an always block, the order of the statements does not matter. But when used in an always block, they become sequential statements executed in the order they are written.

Blocking and Non-Blocking Assignments

Sequential statements can be evaluated in two different ways in Verilog—blocking assignments and non-blocking assignments. A **blocking statement** must complete the evaluation of the right-hand side of a statement before the next statements in a sequential block are executed. Operator "=" is used for representing the blocking assignment. The meaning of "blocking" is that a blocking assignment has to complete before the next statement starts execution (i.e., it *blocks* the next assignment in the sequential block from starting evaluation). A **non-blocking statement** allows assignment evaluation without blocking the sequential flow. In other words, several assignments can be evaluated at the same time. Operator "<=" is used for representing the non-blocking assignment.

For instance, consider the situation if they are in an always block, as shown here:

```
always @(A, B, D)
begin
  C = A && B;  // Blocking operator is used
  E = C || D;  // Statements execute sequentially
end
```

The block executes once when any of the signals *A*, *B*, or *D* changes. The first statement updates the value of *C* before the second statement starts execution; hence, the second statement uses the new value of *C* as input. If *C* or *E* changes when the block executes, then the always block will not execute a second time because *C* is not on the sensitivity list. Operator "=" is used for representing what Verilog calls the **blocking assignment**, which *blocks* the next assignment in the sequential block. It should be noticed that the assignment operator "=" has a blocking nature inside the always block but a non-blocking or concurrent nature outside the always block.

The operator "<=" is to evaluate several assignments at the same time without blocking the sequential flow. Consider the following code:

```
always @(A, B, D)
begin
  C <= A && B;  // Statements execute simultaneously because
  E <= C || D;  // non-blocking operator is used
end
```

The block executes once when any of the signals *A*, *B*, or *D* changes. Both statements execute simultaneously with the values of *A*, *B, C,* and *D* at the beginning of the always block. The first statement does not update the value of *C* before the second statement starts execution; hence, the second statement uses the old value of *C* as input. If *C* changes when the block executes, then the always block will not execute a second time because *C* is not on the sensitivity list. Operator "<=" is used for representing what Verilog calls the **non-blocking assignment** inside an always statement. It should be noticed that the concurrent operations occur with "=" outside the always block, but with "<=" inside the always block. *C* and *E* should be defined as **reg** data type since **reg** is the only legal type on the left-hand side element of an = or <= in an always block.

Figure 2-15 shows a comparison of blocking assignments and non-blocking assignments. The **posedge** keyword of Verilog is used for an edge-triggered functionality in the sensitivity list. Signals *A* and *B* are used for a blocking statement, and *C* and *D* are applied for a non-blocking statement. Assume the initial values of input signals are *A*=*C*=1'b1 and *B*=*D*=1'b0. In the case of blocking assignments, both *A* and *B* will become 1'b0. Since the second assignment will not be evaluated until the completion of the first assignment, the newly evaluated *A* signal will be assigned to *B* in the second assignment. On the other hand, non-blocking assignments will start evaluating all statements in a sequential block at the same time; the result will be independent of the assignment order. Therefore, the result of non-blocking assignments will be *C*=1'b0 and *D*=1'b1. So the signals swap in the case of *C* and *D*, but not in the case of *A* and *B*.

Always statements can be used for modeling combinational logic and sequential logic; however, always statements are not necessary for modeling combinational logic. They are, however, required for modeling sequential logic. One should be very careful when using always statements to represent combinational logic. If

FIGURE 2-15: Blocking and Non-Blocking Assignments

```
module sequential_module (A, B, C, D, clk);
input  clk;
output A, B, C, D;
reg    A, B, C, D;

always @(posedge clk)
begin
  A = B;      // blocking statement 1
  B = A;      // blocking statement 2
end

always @(posedge clk)
begin
  C <= D;     // non-blocking statement 1
  D <= C;     // non-blocking statement 2
end

endmodule
```

any of the input signals are accidentally omitted from the sensitivity list, there can be mismatches between synthesis and simulation and a lot of confusion. Hence, the common practice starting with Verilog 2001 of using the **always @*** statement if a combinational circuit is desired, which avoids accidental errors such as these. If the sensitivity list is "*" then the block will get triggered for any input signal changes.

Consider the code in Figure 2-16, where an always statement is used to model two cascaded gates. *D* and *E* should be defined as **reg** since **reg** is the only legal type on the left-hand side element of an = or <= in an always block. Also, *D* should be defined as output if the output of the first gate is desired externally. If inout is used for *D*, you will have a compile error, since *D* is of **reg** type. Normally, **input** and **inout** ports can be only net (or wire) data type. The statement order is important here because blocking assignment is used.

FIGURE 2-16: Verilog Code for Combinational Logic with Blocking Assignments in an Always Block

```
module two_gates (A, B, C, D, E);
input  A, B, C;
output D, E;

reg D, E;

always @(*)
begin
  #5 D = A || B;  // blocking statement 1
  #5 E = C || D;  // blocking statement 2
end

endmodule
```

Let us assume that all variables are 0 @ 0 ns. Then, *A* changes to 1 @ 10 ns. That causes the module to execute. The statements inside the always statement execute once sequentially. *D* becomes 1 @ 15 ns, and *E* becomes 1 @ 20 ns.

Section 2.14 has additional examples illustrating the distinction between blocking and non-blocking operators inside always statements. While sequential logic can be modeled using the blocking operator "=," it is generally advised not to do so. A good coding practice while writing synthesizable code is to use non-blocking assignments (i.e., "<=") in always blocks intended to create sequential logic and the blocking operator "=" in always blocks intended to create combinational logic.

Another rule to remember is not to mix blocking and non-blocking assignments in the same always block. When each always block is written, think whether you want sequential logic or combinational logic and then use blocking assignments if combinational logic is desired.

Sensitivity List

Both combinatorial always blocks and sequential always blocks have a sensitivity list that includes a list of events. An always block will be activated if one of the events occurs. In the combinatorial logic, the sensitivity list includes all signals that are used in the condition statement and all signals on the right-hand side of the assignment.

On the other hand, the sensitivity list in sequential circuit contains three kinds of edge-triggered events: clock, reset, and set signal event. The sensitivity list can be specified using **@(*)** if a combinational circuit is desired, indicating that the block must be triggered for any input signal changes. If sensitivity list is omitted at the always keyword, delays or time-controlled events must be specified inside the always block. More details on this form of always block are presented in Section 2.8.

Wire and Reg

The two Verilog data types that we have used so far are **wire** and **reg** (more on data types is presented in Section 2.11). The **wire** acts as real wires in circuit designs. The **reg** is similar to wires, but can store information just like registers. The declarations for **wire** and **reg** signals should be done inside a module but outside any initial or always block. The initial value of a wire is *z* (high impedance), and the initial value of a reg is *x* (unknown).

The **wires** are either single bit or multiple bits in Verilog. The wires cannot store any information. They can be used only in modeling combinational logic and must be driven by something. The wires are a data type that can be used on the left-hand side of an assign statement but cannot be used on the left-hand side of = or <= in an always @ block.

The data type **reg** is used where the assigned data needs to be stored until the next assignment. If you want to assign your output in sequential code (within an always block), you should declare it as a **reg**. Otherwise, it should be a **wire** by default. One can use **reg** to model both combinational and sequential logic. Data type **reg** is the only legal type on the left-hand side element of an = or <= in an always block or initial block (normally used in test benches). It cannot be used on the left-hand side of an assign statement. Section 2.11 presents more on data types.

The default Verilog HDL data value set is a 4-value system consisting of four basic values:

0 represents a logic zero, or a false condition.

1 represents a logic one, or a true condition.

x represents an unknown logic value.

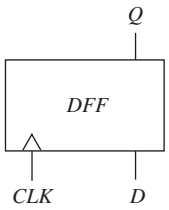
z represents a high-impedance state (often called the tristated value).

The 4-valued logic is described in more detail in Chapter 8.

For better understanding of sequential statements and operation of always statements, several more examples will be presented. In the following section, we explain how simple flip-flops can be modeled using always statements, and then we explain the basics of the Verilog simulation process.

2.7 Modeling Flip-Flops Using Always Block

A flip-flop can change state either on the rising or on the falling edge of the clock input. This type of behavior is modeled in Verilog by an always block. For a simple D flip-flop with a Q output that changes on the rising edge of *CLK*, the corresponding code is given in Figure 2-17.




```
always @ (posedge CLK)
begin
  Q <= D;
end
```

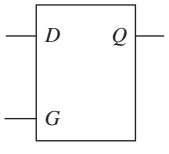
FIGURE 2-17: Verilog Code for a Simple D Flip-Flop

In Figure 2-17, on the rising edge of *CLK*, the always block executes once through and then waits at the start of the block until *CLK* changes again. The **sensitivity list** of the always block tests for a rising edge of the clock, and *Q* is set equal to *D* when a rising edge occurs. The expression `posedge (or negedge)` is used to accomplish the functionality of an edge-triggered device. If `CLK is changed from 0 to 1` it is a rising edge. If `CLK changes from 1 to 0`, it indicates a falling edge.

If the flip-flop has a delay of 5 ns between the rising edge of the clock and the change in the *Q* output, we would replace the statement `Q <= D;` with `Q <= #5 D;` in the foregoing always block.

The statements between **begin** and **end** in an always block operate as sequential statements. In the previous example, `Q <= D;` is a sequential statement that executes only following the rising edge of *CLK*. In contrast, the concurrent statement assign `Q = D;` executes whenever *D* changes. If we synthesize the foregoing code, the synthesizer infers that *Q* must be a flip-flop since it changes only on the rising edge of *CLK*. If we synthesize the concurrent statement assign `Q = D;` the synthesizer will simply connect *D* to *Q* with a wire or a buffer.

In Figure 2-17, note that *D* is not on the sensitivity list because changing *D* will not cause the flip-flop to change state. Figure 2-18 shows a transparent latch and its Verilog representation. Both *G* and *D* are on the sensitivity list since if $G = 1$, a change in *D* causes *Q* to change. If *G* changes to 0, the always block executes, but *Q* does not change. For the sensitivity list, both (*G* or *D*) and (*G*, *D*) are acceptable.




```
always @ (G or D)
begin
 if (G)
   Q <= D;
end
```

FIGURE 2-18: Verilog Code for a Transparent Latch

If a flip-flop has an active-low asynchronous clear input (*ClrN*) that resets the flip-flop independently of the clock, then we must modify the code of Figure 2-17 so that it executes when either *CLK* or *ClrN* changes. To do this, we add *ClrN* to the sensitivity list. The Verilog code for a D flip-flop with asynchronous clear is given in Figure 2-19. Since the asynchronous *ClrN* signal overrides *CLK*, *ClrN* is tested first and the flip-flop is cleared if *ClrN* is 0. Otherwise, *CLK* is tested, and *Q* is updated if a rising edge has occurred.

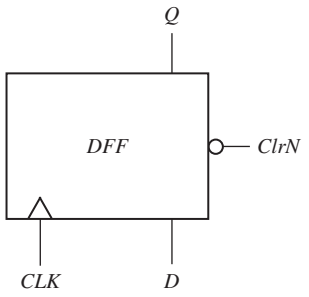


FIGURE 2-19: Verilog Code for a D Flip-Flop with Asynchronous Clear

```
always @ (posedge CLK or negedge ClrN)
begin
  if (~ClrN)
    Q <= 0;
  else
    Q <= D;
end
```

In the foregoing examples, we have used two types of sequential statement: signal assignment statements and **if** statements. The basic **if** statement has the form

```
if (condition)
  sequential statements1
else
  sequential statements2
```

The condition is a Boolean expression that evaluates to TRUE or FALSE. If it is TRUE, `sequential statements1` are executed; otherwise, `sequential statements2` are executed.

Verilog **if** statements are sequential statements that can be used within an always block (or an initial block), but they cannot be used as concurrent statements outside of an always block. The most general form of the **if** statement is

```
if (condition)
  sequential statements
  // 0 or more else if clauses may be included
else if (condition)
  sequential statements}
[else  sequential statements]
```

The curly brackets indicate that any number of **else if** clauses may be included, and the square brackets indicate that the **else** clause is optional. The example of Figure 2-20 shows how a flow chart can be represented using nested **if**s or the

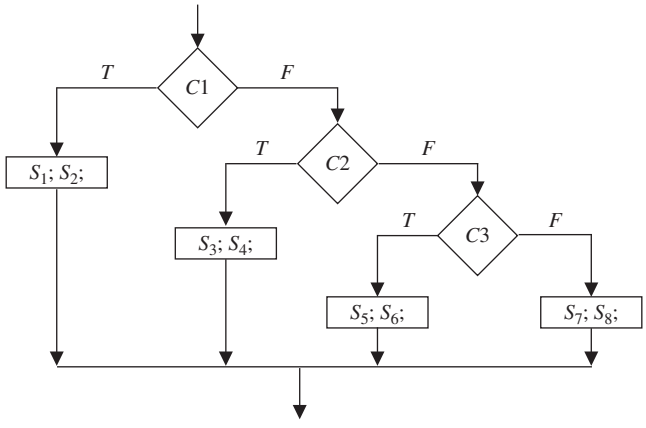
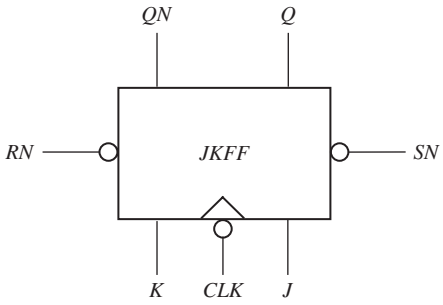


FIGURE 2-20: Equivalent Representations of a Flow Chart Using Nested Ifs and Else Ifs

```
if (C1)
begin
  S1; S2;
end
else if (C2)
begin
  S3; S4;
end
else if (C3)
begin
  S5; S6;
end
else
begin
  S7; S8;
end
```

FIGURE 2-21: J-K Flip-Flop



equivalent using **else if**s. In this example, *C1*, *C2*, and *C3* represent conditions that can be true or false, and $S_1, S_2, \ldots, S_8$ represent sequential statements. If more than one statement needs to be in an **if** block, **begin** and **end** should be used.

Next, we will write a Verilog module for a J-K flip-flop (Figure 2-21). This flip-flop has active-low asynchronous preset (*SN*) and clear (*RN*) inputs. State changes related to *J* and *K* occur on the falling edge of the clock. In this chapter, we use a suffix N to indicate an active-low (negative-logic) signal. For simplicity, we will assume that the condition $SN = RN = 0$ does not occur.

The Verilog code for the J-K flip-flop is given in Figure 2-22. The input and output signals are listed after the **module** statement. We define a **reg** *Qint* as an internal signal that represents the state of the flip-flop internal to the module. The two concurrent statements, statement4 and statement5, transmit this internal signal to the *Q* and *QN* outputs of the flip-flop. Because the flip-flop can change state in response to changes in *SN*, *RN*, and *CLK*, these three signals are in the sensitivity list of the always statement. Both *RN* and *SN* are active low signals. If $RN = 0$, the flip-flop is reset, and if $SN = 0$, the flip-slop is set. Since *RN* and *SN,* reset and set the flip-flop independently of the clock, they are tested first. If *RN* and *SN* are both 1, we test for the falling edge of the clock. In the **if** statement, both (~RN) and (RN == 1'b0) are acceptable.

FIGURE 2-22: J-K Flip-Flop Model

```
module JKFF (SN, RN, J, K, CLK, Q, QN);
input  SN, RN, J, K, CLK;
output Q, QN;

reg Qint;

always @(negedge CLK or RN or SN)
begin
  if (~RN)
    #8  Qint <= 0;                                  // statement1
  else if (~SN)
    #8  Qint <= 1;                                  // statement2
  else
    Qint <= #10 ((J && ~Qint) || (~K && Qint)); // statement3
end

assign Q = Qint;                                    // statement4
assign QN = ~Qint;                                  // statement5

endmodule
```

The condition (negedge CLK) is TRUE only if *CLK* has just changed from 1 to 0. The next state of the flip-flop is determined by its characteristic equation:

$$Q^+ = JQ' + K'Q$$

The 8 ns delay represents the time it takes to set or clear the flip-flop output after *SN* or *RN* changes to 0. The 10 ns delay represents the time it takes for *Q* to change after the falling edge of the clock.

2.8 Always Blocks Using Event Control Statements

An alternative form for an always block uses wait or event control statements instead of a sensitivity list. If a sensitivity list is omitted at the always keyword, delays or time-controlled events must be specified inside the always block. For example,

```
always
begin
#10 clk <= ~clk;
end
```

will work as long as non-zero delay is specified.

An always block cannot have both wait statements and a sensitivity list. An always block with wait statements may have the form

```
always
begin
  sequential-statements
  wait-statement
  sequential-statements
  wait-statement
  . . .
end
```

Such an always block could look like

```
always
begin
  rst = 1; // sequential statements
  @(posedge CLK); //wait until posedge CLK
  // more sequential statements
end
```

This always block will execute the `sequential-statements` until a wait (event control) statement is encountered. Then it will wait until the specified condition is satisfied. It will then execute the next set of `sequential-statements` until another wait is encountered. It will continue in this manner until the end of the always block is reached. Then it will start over again at the beginning of the block.

The wait statement is used as a level-sensitive event control. The general syntax of the wait statement is

```
wait (Boolean-expression)
```

A procedural statement waits when the Boolean expression is FALSE. When the expression is TRUE, the statement is executed. The logic values 0, 'x', and 'z' are treated as FALSE. Logic 1 is TRUE. The following example will block the flow of the procedural block when the condition of the wait statement is FALSE. The wait statement can also be used to handshake or synchronize two concurrent processes, as illustrated in the following sequence:

```
always
begin
  wait (WR)
    MEM = DATA_IN;
  wait (~WR)
    DATA_OUT = MEM;
end
```

When the WR signal becomes true, DATA_IN gets written into MEM but as soon as the WR signal becomes false, the MEM value becomes available on DATA_OUT.

Example

For a half adder, sum and carry can be found using the equations sum = x XOR y ; carry = x AND y. What is wrong with the following code for a half adder that must add if add signal equals 1?

```
always @(*)
begin
if (add == 1)
   sum = x ^ y;
   carry = x & y;
end
```

(a) It will compile but not simulate correctly
(b) It will compile and simulate correctly but not synthesize correctly
(c) It will work correctly in simulation and synthesis
(d) It will not even compile

Answer: (a). This code will compile but will not simulate correctly. The if statement is missing begin and end. Currently only the sum is part of the if statement. The carry statement will get executed regardless of the add signal. This can be corrected by adding begin and end for the if statement. That will result in correct simulation. It can still lead to latches in synthesis. Latches can be avoided by adding else clause or by initializing sum and carry to 0 at the beginning of the always statement.

Example

What is wrong with the following code for a half adder that must add if add signal equals 1?

```
always @(*)
begin
if (add == 1)
   sum = x ^ y;
   carry = x & y;
 else
   sum = 0;
   carry = 0;
end
```

(a) It will compile but not simulate correctly
(b) It will compile and simulate correctly but not synthesize correctly
(c) It will work correctly in simulation and synthesis
(d) It will not even compile

Answer: (d). This code will not even compile due to the missing begin and end inside the if statement. When the compiler gets to the else, it finds that the corresponding if statement is missing. Both if and else clauses need begin and end. Once that is corrected, both simulation and synthesis will work correctly.

2.9 Delays in Verilog

In one of the initial examples in this chapter, we used the statement

```
assign #5 D = A && B;
```

to model an AND gate with a propagation delay of 5 ns (assuming its time unit is ns). The foregoing statement will model the AND gate's delay; however, it also introduces some complication, which many readers will not normally expect. If you simulate this AND gate with inputs that change very often in comparison to the gate delay (e.g., at 1 ns, 2 ns, 3 ns, etc.), the simulation output will not show the changes. This is due to the way Verilog delays work.

Basically, delays in Verilog can be categorized into two models: inertial delay and transport delay. The inertial delay for combinational blocks can be expressed in the following three ways:

```
// explicit continuous assignment
wire D;
assign #5 D = A && B;

// implicit continuous assignment
wire #5 D = A && B;

// net declaration
wire #5 D;
assign D = A && B;
```

Any changes in *A* and *B* will result in a delay of 5 ns before the change in output is visible. If values in *A* or *B* are changed 5 ns before the evaluation of *D* output, the change in values will be propagated. However, an input pulse that is shorter than the delay of the assignment does not propagate to the output. This feature is called **inertial delay**. Inertial delay is intended to model gates and other devices that do not propagate short pulses from the input to the output. If a gate has an ideal inertial delay *T*, in addition to delaying the input signals by time *T*, any pulse with a width less than *T* is rejected. For example, if a gate has an inertial delay of 5 ns, a pulse of width 5 ns would pass through, but a pulse of width 4.999 ns would be rejected.

Transport delay is intended to model the delay introduced by wiring; it simply delays an input signal by the specified delay time. In order to model this delay, a delay value must be specified on the right-hand side of the statement. Figure 2-23 illustrates transport delay and inertial delay in Verilog. Consider the following code:

```
always @ (X)
begin
  Z1 <= #10 (X);      // transport delay
end
assign #10 Z2 = X;   // inertial delay
```

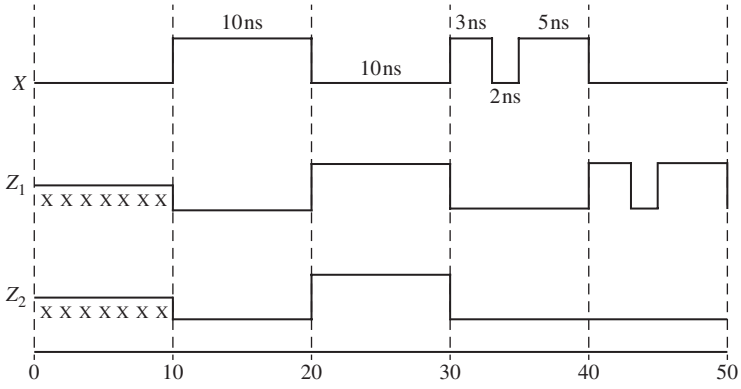


FIGURE 2-23: Inertial and Transport Delays

The first statement has transport delay while the second one has inertial delay. As shown in Figure 2-23, if the delay is shorter than 10 ns, the input signal will not be propagated to the output in the second statement. Only one pulse (between 10 ns and 20 ns) on input *X* is propagated to the output Z_2, since it has 10 ns pulse width. All other pulses are not propagated to the output Z_2. But Z_1 has transport delay and hence propagates all pulses. It is assumed that the output Z_1 and Z_2 are initialized to 0 at 0 ns. The delay in the statement `Z1 <= #10 X;` is called **intra-assignment delay**. The expression on the right hand side is evaluated but not assigned to Z_1 until the delay has elapsed (also called **delayed assignment**). However, in a statement like `#10 Z1 <= X;` the delay of #10 elapses first and then the expression is evaluated and assigned to Z_1 (also called **delayed evaluation**).

The placement of the delay on the right-hand side cannot be done with continuous assign statements. Hence the following statement is illegal. It will produce a compile-time error.

```
assign a = #10 b;
```

Verilog also has a type of delay called **net delay**. Consider the code

```
wire C1;
wire #10 C2; // net delay on wire C2

assign #30 C1 = A || B;   // statement 1 – inertial delay
assign #20 C2 = A || B;   // statement 2 – inertial delay
                             will be
                          // added to net delay before being
                          // assigned to wire C2
```

The wire C_2 has a **net delay** of 10 ns associated with it, specified in its declaration whereas C_1 has no such net delay. Net delay refers to the time it takes from any driver on the net to change value to the time when the net value is updated and propagated further. There are inertial delays of 30 ns for C_1 in statement 1 and 20 ns for C_2 in statement 2, typically representative of gate delays. After statement 2 processes its delay of 20 ns, the net delay of 10 ns is added to it. Figure 2-24(a) indicates the difference between C_1 and C_2. C_1 rejects all narrow pulses less than 30 ns, whereas C_2 rejects only pulses less than 20 units.

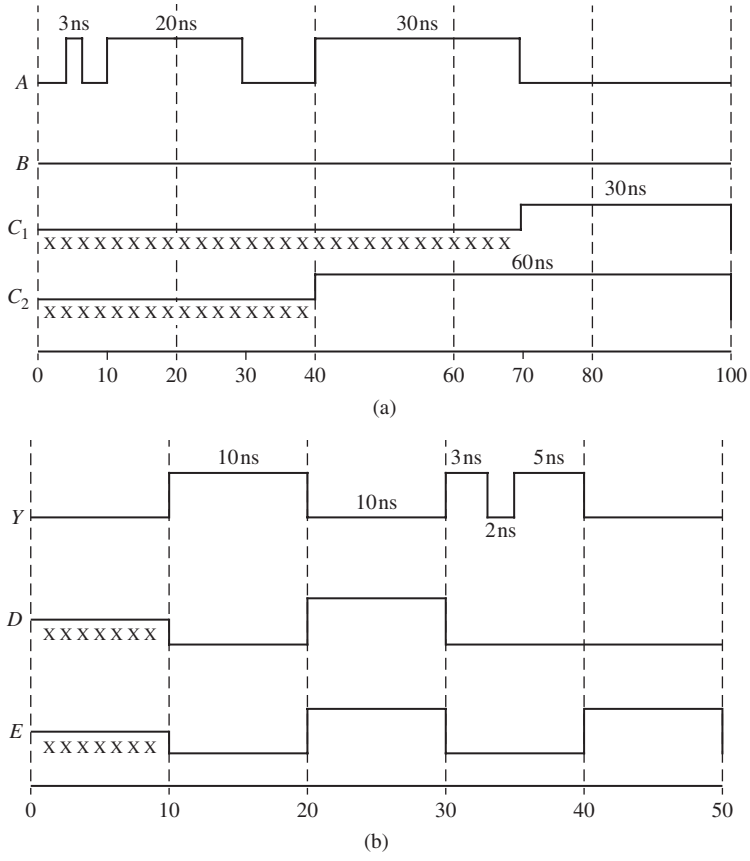


FIGURE 2-24: Example of Net Delays

Now consider the following two statement pairs with the Y waveform as shown in Figure 2-24(b).

```
wire #3 D; // net delay on wire D
assign #7 D = Y; // statement 1 – inertial delay

wire #7 E; // net delay on wire E
assign #3 E = Y; // statement 1 – inertial delay
```

The assign statement for *D* works with a 7 ns inertial delay and rejects any pulse below 7 ns. Hence *D* rejects the 3 ns, 2 ns and 5 ns pulses in *Y*. The 3 ns net delay from the wire statement is added to the signal that comes out from the assign statement. In the case of *E*, pulses below 3 ns are rejected. Hence the 3 ns pulse in *Y* passes through the assign statement for *E*, the 2 ns pulse is rejected and the 5 ns pulse is accepted. Hence the 3 ns and 5 ns pulses get combined in the absence of the 2 ns pulse to yield output on *E* appears as a big 10 ns pulse. The 7 ns net delay from the wire statement is added to the signal that comes out from the assign statement. If any pulses less than 7 ns are encountered at the net delay phase, they will be rejected. Figure 2-24(b) illustrates the waveforms for *D* and *E*.

Note that these delays are relevant only for simulation. Understanding how inertial delay works can remove a lot of frustration in your initial experience with Verilog simulation. The pulse rejection associated with inertial delay can inhibit many output changes. In simulations with basic gates and simple circuits, one should make sure that test sequences that you apply are wider than the inertial delays of the modeled devices. The focus of this book is synthesizeable Verilog where all specified delays are ignored. Hence we do not further dwell on simulator behavior for delays.

2.10 Compilation, Simulation, and Synthesis of Verilog Code

After describing a digital system in Verilog, simulation of the Verilog code is important for two reasons. First, we need to verify that the Verilog code correctly implements the intended design, and second, we need to verify that the design meets its specifications. We first simulate the design and then synthesize it to the target technology (FPGA or custom ASIC). In this section, first we describe steps in simulation and then introduce synthesis. As illustrated in Figure 2-25, there are three phases in the simulation of Verilog code: **analysis (compilation), elaboration, and simulation**.

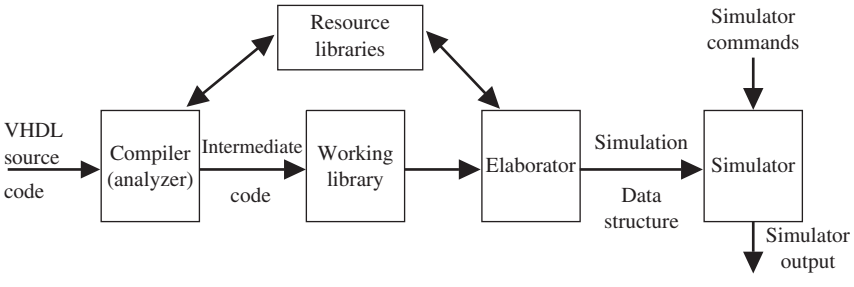


FIGURE 2-25: Compilation, Elaboration, and Simulation of Verilog Code

Before the Verilog model of a digital system can be simulated, the Verilog code must first be compiled. The Verilog compiler, also called an **analyzer**, first checks the Verilog source code to see that it conforms to the syntax and semantic rules of Verilog. If there is a syntax error, such as a missing semicolon, or if there is a semantic error, such as trying to add two signals of incompatible types, the compiler will output an error message. The compiler also checks to see that references to libraries are correct. If the Verilog code conforms to all of the rules, the compiler generates intermediate code, which can be used by a simulator or by a synthesizer.

After a Verilog design has been parsed but before simulation begins, the design must have the modules being instantiated linked to the modules being defined, the parameters propagated among the various modules, and hierarchical references resolved. This phase in understanding a Verilog description is referred to as **elaboration**. During elaboration, a *driver* is created for each signal. Each driver holds the current value of a signal and a queue of future signal values. Each time a signal is scheduled to change in the future, the new value is placed in the queue along with the time at which the change is scheduled. In addition, memory storage is allocated for the required signals; the interconnections among the port signals are specified; and a mechanism is established for executing the Verilog statements in the proper sequence. The resulting data structure represents the digital system being simulated.

The simulation process consists of an **initialization phase** and actual **simulation**. The simulator accepts simulation commands, which control the simulation of the digital system and which specify the desired simulator output. Verilog simulation uses what is known as **discrete event simulation**. The passage of time is simulated in discrete steps in this method of simulation. The initialization phase is used to give an initial value to the signal. To facilitate correct initialization, the initial value can be specified in the Verilog model. In the absence of any specifications of the initial values, some simulator packages may assign an initial value depending on the type of the signal. Please note that this initialization is only for simulation and not for synthesis.

A design consists of connected threads of execution or processes. Processes are objects that can be evaluated, that may have state, and that can respond to changes on their inputs to produce outputs. Processes include modules, initial and always procedural blocks, continuous assignments, procedural assignment statements, system tasks, and so forth.

Every change in value of a net or variable in the circuit being simulated is considered an **update event**. Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are evaluated in an arbitrary order. The evaluation of a process is also an event, known as an **evaluation event**. The term **simulation time** is used to refer to the time value maintained by the simulator to model the actual time it would take for the circuit being simulated.

Events can occur at different times. In order to keep track of the events and to make sure they are processed in the correct order, the events are kept on an **event queue**, ordered by simulation time. Putting an event on the queue is called **scheduling an event**.

The Verilog event queue is logically segmented into five different regions:

i. **Active event region:** Events that occur at the current simulation time are in this region. Events can be added to any of the five regions but can be removed only

from this region (i.e., the *active* region). Events can be processed in any order from within this region. (This freedom to choose any active event for immediate processing is an essential source of non-determinism in the Verilog HDL.)

ii. **Inactive event region:** Events that occur at the current simulation time but that shall be processed after all the active events are processed are in this region. Blocking assignments with zero delays are in this region until they get moved later to the active region.

iii. **Non-blocking assign update region:** Events that have been evaluated during some previous simulation time but that shall be assigned at this simulation time after all the active and inactive events are processed are in this region.

iv. **Monitor event region:** Events that shall be processed after all the active, inactive, and non-blocking assign update events are processed are in this region. These are the *monitor* events.

v. **Future event region:** Events that occur at some future simulation time are in this region. These are the *future* events. Future events are divided into *future inactive events* and *future non-blocking assignment update events*.

When each Verilog statement is processed, events are added to the various queue regions according to the following convention for each type of statement:

i. **Continuous assignment**—evaluate RHS and add to active region as an active update event.

ii. **Procedural continuous assign**—evaluate RHS and add to active region as an update event.

iii. **Blocking assignment with delay**—compute RHS and put into future event region for time after delay.

iv. **Blocking assignment with no delay**—compute RHS and put into inactive region for current time.

v. **Non-blocking assignment with no delay**—compute RHS and schedule as non-blocking assign update event for current time if zero delay.

vi. **Non-blocking assignment with delay**—compute RHS and schedule as non-blocking assign update event for future time if zero delay.

vii. **$monitor** and **$strobe** system tasks—create monitor events for these system tasks. (These events are continuously reenabled in every successive time step.)

The processing of all the active events is called a **simulation cycle**.

For each simulation time, the following actions are performed in order:

i. Process all active update events. (Whenever there is an active update event, the corresponding object is modified and new events are added to the various event queue regions for other processes sensitive to this update.)

ii. Then activate all inactive events for that time (and process them because now they are active).

iii. Then activate all non-blocking assign update events and process them.

iv. Then activate all monitor events and process them.

v. Advance time to the next event time and repeat from step i.

All of these five steps happen at the same time, but the events occur in the order active, inactive, non-blocking update, and monitor events.

VHDL uses the concept of an infinitesimal delay called delta (Δ) delay to explicitly indicate the various update times within the same simulation time, however the Verilog Language Reference Model (LRM) does not mention delta delays. In the Verilog simulators we experimented with, there is a delta delay indicated in non-blocking procedural assignments with zero delay, but no delta delays were observed in blocking or continuous assign statements with zero delay. However, implicitly the simulation uses the ordering between the aforementioned 5 event queue regions to yield the correct ordering. Blocking assignments with zero delay are in the inactive queue first, and happen after continuous assignments but no delta delay is shown to indicate the delay. Non-determinism is usually avoided except that the freedom to choose any active event for immediate processing from the active queue region contributes to some non-determinism in the Verilog HDL.

Basically, the simulator works as follows with "<=": whenever a component input changes, the output is scheduled to change after the specified delay or after Δ if no delay is specified. When all events for the current time have been processed, simulated time is advanced to the next time at which an event is specified. When time is advanced by a finite amount (1 ns for example), the Δ counter is reset and simulation resumes. Real time does not advance again until all events associated with the current simulation time have been processed.

If two non-blocking updates are made to the same variable in the same time step, the second one dominates by the end of the time step. For example, in Figure 2-26(a) events are added to the event queue in source code order because of

FIGURE 2-26: Illustration of Non-Determinism

```
module determinate;
reg a;

initial a = 0;

always begin
a <= #5 0;
a <= #5 1;
end
// The assigned value of a is deterministic
// because of ordering from begin to end
endmodule
```

(a)

```
module nondeterminate;
reg a;
initial a = 0;

always a <= #5 0;
always a <= #5 1;
// The assigned value of a is non-deterministic
Endmodule
```

The IEEE 1364 Standard Verilog Language Reference Manual (LRM) [1] provides definitions and interpretations for the various Verilog constructs, which all compliant simulators shall implement. However, there is a great deal of choice in the definitions, and some differences in the details of execution are to be expected between different simulators.

Those who are accustomed to the elegant delta delay conventions in VHDL may be disappointed with Verilog simulation outputs. Verilog was created primarily with circuit synthesis in mind and Verilog simulation may not clearly indicate the precise ordering of multiple events happening at the same simulation time. The IEEE 1364 Standard Verilog Language Reference Manual (LRM) does not even use the word 'delta' in it.

the **begin . . . end**, and the two updates are performed in source order as well. Hence, the variable a will be assigned 0 first and then 1 in that order. There is no non-determinism in this code. However, for the code in Figure 2-26(b), the two **always** blocks are concurrent with respect to each other and there is no ordering between them. Hence the assigned value of a is non-deterministic.

2.10.1 Simulation with Multiple Processes (Initial or Always Blocks)

If a model contains more than one process, all processes execute concurrently with other processes. If there are concurrent statements outside always statements, they also execute concurrently. Statements inside of each always block execute sequentially. A process takes no time to execute unless it has wait statements in it. As an example of simulation of multiple processes, we trace execution of the Verilog code shown in Figure 2-27.

FIGURE 2-27: Verilog Code to Illustrate Process Simulation

```
module twoprocess
  reg A,B;

initial
begin
  A = 0;
  B = 0;
end

// process P1
always @(B)
begin
  A <= 1;
  A <= #5 0;
end

// process P2
always @(A)
begin
   if (A)
        B <= #10 ~B;
end
```

Figure 2-28 shows the drivers for the signals *A* and *B* as the simulation progresses. In the absence of an initial block, each driver would hold *x*, since this is the default initial value for a signal. When simulation begins, initialization takes place and each driver holds 0 since an initial block is included in the provided code. Both always statements wait until a signal on the sensitivity list changes. With the initial block here, the signal changes from initialization lead to the execution of the always statements. In the absence of the initial block, one can force input changes using simulation commands. When process *P1* executes at zero time, two changes in *A* are scheduled (*A* changes to 1 at time Δ and back to 0 at time = 5 ns). Meanwhile, process *P2* executes at zero time, but no change in *B* occurs since

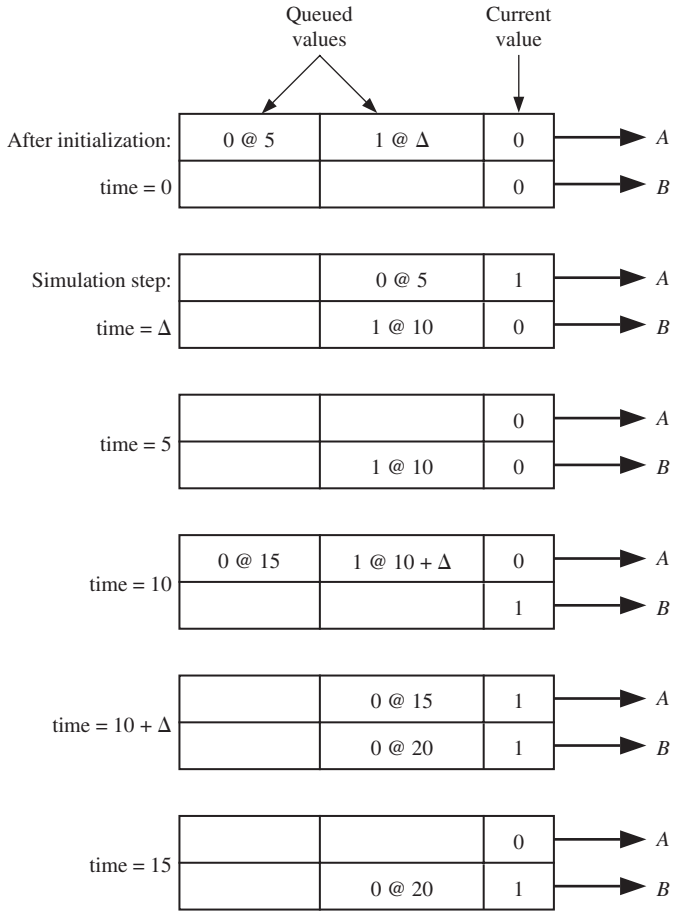


FIGURE 2-28: Signal Drivers for Simulation Example

A is still 0 during execution at time 0 ns. Time advances to Δ, and *A* changes to 1. The change in *A* causes process *P2* to execute, and since $A = 1$, *B* is scheduled to change to 1 at time 10 ns. The next scheduled change occurs at time = 5 ns, when *A* changes to 0. This change causes *P2* to execute, but *B* does not change. *B* changes to 1 at time = 10 ns. The change in *B* causes *P1* to execute, and two changes in *A* are scheduled. When *A* changes to 1 at time 10 + Δ, process *P2* executes, and *B* is scheduled to change at time 20 ns. Then *A* changes at time 15 ns, and the simulation continues in this manner until the runtime limit is reached. It should be understood that *A* changes at 15 ns and not at 15 + Δ. The Δ delay comes into the picture only when no time delay is specified.

Verilog simulators use event-driven simulation, as illustrated in the preceding example. A change in a signal is referred to as an *event*. Each time an event occurs, any processes that have been waiting on the event are executed in zero time, and any resulting signal changes are queued up to occur at some future time. When all the active processes are finished executing, simulation time is advanced to the time for which the next event is scheduled, and the simulator processes that event. This continues until either no more events have been scheduled or the simulation time limit is reached.

Inertial delays can now be explained in the following manner. Each input change causes the simulator to schedule a change, which is scheduled to occur after the specified delay; however, if another input change happens before the specified delay has elapsed, the first change is dequeued from the simulation driver queue. Hence only pulses wider than the specified delay appear at the output.

One of the most important uses of Verilog is to synthesize or automatically create hardware from a Verilog description. The **synthesis** software for Verilog translates the Verilog code to a circuit description that specifies the needed components and the connections between the components. The initial steps (analysis and elaboration) in Figure 2-25 are common whether Verilog is used for simulation or synthesis. The simulation and synthesis processes are shown in Figure 2-29.

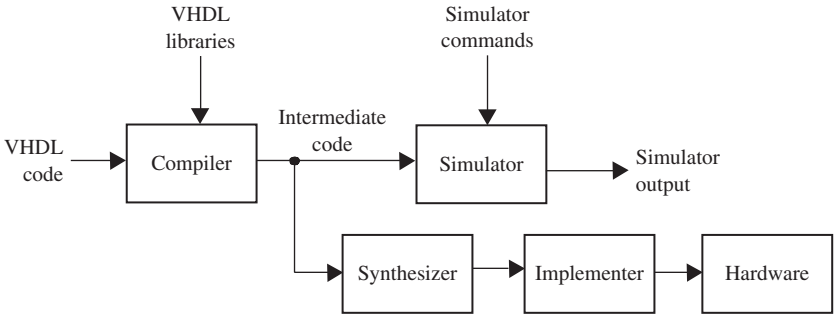


FIGURE 2-29: Compilation, Simulation, and Synthesis of Verilog Code

Although synthesis can be done in parallel to simulation, practically it follows simulation because designers would normally want to catch errors first before attempting to synthesize. After the Verilog code for a digital system has been simulated to verify that it works correctly, the Verilog code can be synthesized to produce a list of required components and their interconnections, typically called the **netlist**. The synthesizer output can then be used to implement the digital system using specific hardware, such as a CPLD or an FPGA or as an ASIC. The CAD software used for implementation generates the necessary information to program the CPLD or FPGA hardware. Synthesis and implementation of digital logic from Verilog code is discussed in more detail in Chapter 6.

2.11 Verilog Data Types and Operators

2.11.1 Data Types

Verilog has two main groups of data types: the variable data types and the net data types. These two groups differ in the way that they are assigned and hold values. They also represent different hardware structures.

The *net* data types can represent physical connections between structural entities, such as gates. Generally, it does not store values. Instead, its value is determined by the values of its drivers, such as a continuous assignment or a gate. A very popular *net* data type is the **wire**. There are also several other predefined data types

But if A is initialized to `8'shA5` as in

```
integer A = 8'shA5;
```

A >>> 4 yields 11111010 (shift right signed by 4, A's sign bit is 1). The `'sh` indicates that the value is signed hex.

However, in **reg** declarations, if a signed register is desired, it should be explicitly mentioned. For instance,

```
reg [7:0] A = 8'shA5
```

does not make the register signed. It should be declared as **`reg signed [7:0] A = 8'hA5`**

The + and − operators can be applied to any types, including integer or real numeric operands. When types are mixed, the expression self-evaluates to a type according to the types of the operands. If a and b are 16 bits each, (a + b) will evaluate to 16 bits. However, (a + b + 0) will evaluate to integer. If any operand is real, the result is real. **If any operand is unsigned, the result is unsigned, regardless of the operator**.

When expressions are evaluated, if the operands are of unequal bit lengths and if one or both operands are unsigned, the smaller operand shall be zero-extended to the size of the larger operand. If both operands are signed, the smaller operand shall be sign-extended to the size of the larger operand. If constants need to be extended, signed constants are sign-extended and unsigned constants are zero-extended.

The * and / operators perform multiplication and division on integer or floating-point operands. The ** operator raises an integer or floating-point number to an integer power. The % (modulus) operator calculates the remainder for integer operands.

> According to the Verilog Language Reference Manual (LRM), the result of the modulus operator takes the sign of the first operand. For example,
>
> −10 % 3
>
> results in –1 because the result takes the sign of the first operand. However, number theory books define **mod m** as a function from the set of integers to the set of {0,1,2,….,m−1) and hence –10 mod 3 is equal to 2. However, there is no reason for serious concern because –1 mod 3 is 2 according to the **mod m** definition. Hence -1 as given by Verilog and 2 as given by their definition are in fact the same number. VHDL has two separate operators, one for modulus and one for remainder. The VHDL **remainder** is signed according to the sign of the first operand whereas the modulus follows the **mod m** definition from number theory.

2.12 Simple Synthesis Examples

Synthesis tools try to infer the hardware components needed by "looking" at the Verilog code. In order for code to synthesize correctly, certain conventions must be followed. When writing Verilog code, you should always keep in mind that you are

designing hardware, not simply writing a computer program. Each Verilog statement implies certain hardware requirements. Consequently, poorly written Verilog code may result in poorly designed hardware. Even if Verilog code gives the correct result when simulated, it may not result in hardware that works correctly when synthesized. Timing problems may prevent the hardware from working properly even though the simulation results are correct.

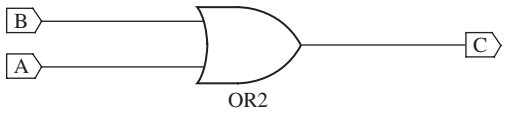
Consider the Verilog code in Figure 2-30. (Note that *B* is missing from the sensitivity list in the **always** statement.) This code will simulate as follows. Whenever *A* changes, it will cause the process to execute once. The value of *C* will reflect the values of *A* and *B* when the process began. If *B* changes now, that will not cause the process to execute.

FIGURE 2-30: Verilog Code Example Where Simulation and Synthesis Results in Different Outputs

```
module Q1 (A, B, C);
input   A;
input   B;
output  C;

reg     C;

always @(A)
  C = #5 A | B;
endmodule
```

If this code is synthesized, most synthesizers will output an OR gate as in Figure 2-31. The synthesizer will warn you that B is missing from the sensitivity list in always statement, but will go ahead and synthesize the code properly. The synthesizer will also ignore the 5 ns delay on the preceding statement. If you want to model an exact 5 ns delay, you will have to use counters. The simulator output will not match the synthesizer's output since the always statement will not execute when B changes. This is an example of where the synthesizer guessed a little more than what you wrote; it assumed that you probably meant an OR gate and created that circuit (accompanied by a warning). But this circuit functions differently from what was simulated before synthesis. It is important that you always check for synthesizer warnings of missing signals in the sensitivity list. Perhaps the synthesizer helped you; perhaps it created hardware that you did not intend to.

FIGURE 2-31: Synthesize Output for Code in Figure 2-30



Now, consider the Verilog code in Figure 2-32. What hardware will you get if you synthesized this code?

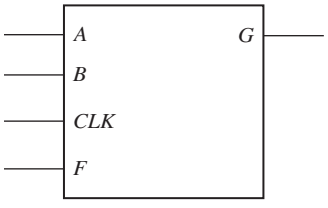
Let us think about the block diagram of the circuit represented by this code without worrying about the details inside. The block diagram is as shown in

FIGURE 2-32: Example Verilog Code

```
module Q3 (A, B, F, CLK, G);
input   A;
input   B;
input   F;
input   CLK;
output  G;

reg     G;
reg     C;

always @(posedge CLK)
begin
  C <= A & B;  // statement 1
  G <= C | F;  // statement 2
end

endmodule
```

Figure 2-33. The ability to hide details and use abstractions is an important part of good system design.

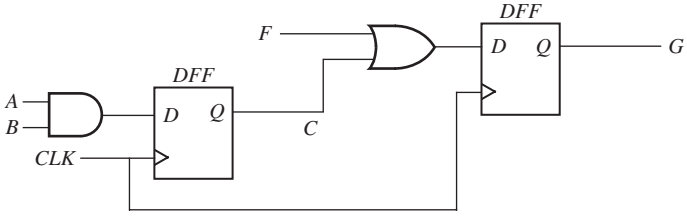
FIGURE 2-33: Block Diagram for Verilog Code in Figure 2-32



Note that *C* is an internal signal, and therefore, it does not show up in the block diagram.

Now, let us think about the details of the circuit inside this block. This circuit is not two cascaded gates; the signal assignment statements are in a process (an always statement). An edge-triggered clock is implied by the use of **posedge** or **negedge** in the clock statement preceding the signal assignment. Since the values of *C* and *G* need to be retained after the clock edge, flip-flops are required for both *C* and *G*. Please note that a change in the value of *C* from statement 1 will not be considered during the execution of statement 2 in that pass of the process. It will be considered only in the next pass, and the flip-flop for *C* makes this happen in the hardware also. Hence the code implies hardware shown in Figure 2-34.

FIGURE 2-34: Hardware Corresponding to Verilog Code in Figure 2-32



We saw earlier that the following code represents a D-latch:

```
always @(G or D)
begin
 if (G)  Q <= D;
end
```

Let us understand why this code does not represent an AND gate with G and D as inputs. If $G = 1$, an AND gate will result in the correct output to match the **if** statement. However, what happens if currently $Q = 1$ and then G changes to 0? When G changes to 0', an AND gate would propagate that to the output; however, the device we have modeled here should not. It is expected to make no changes to the output if G is not equal to 1. Hence, it is clear that this device has to be a D-latch and not an AND gate.

In order to infer flip-flops or registers that change state on the rising edge of a clock signal, most synthesizers require that the sensitivity list in an always statement should include an edge-triggered signal as in

```
always @(posedge CLK)
```

For every assignment statement in an always statement, a signal on the left side of the assignment will cause creation of a register or flip-flop. The moral to this story is that if you do not want to create unnecessary flip-flops, do not put the signal assignments in a clocked always statement. If `clock` is omitted in the sensitivity list of an always statement, the synthesizer may produce latches instead of flip-flops.

Now consider the Verilog code in Figure 2-35. If you attempt to synthesize this code, the synthesizer will generate an empty block diagram. This is because D, the output of the block shown in the Figure, is never assigned. The code assigns the new value to C, which is never brought out to the outside world. It will generate warnings that

```
Input <CLK> is never used.
Input <A> is never used.
Input <B> is never used.
Output <D> is never assigned.
```

FIGURE 2-35: Example Verilog Code That Will Not Synthesize

```
module no_syn (A, B, CLK, D);
input   A;
input   B;
input   CLK;
output  D;

reg     C;

always @(posedge CLK)
  C <= A & B;

endmodule
```