Table 3.4
Even-Parity-Checker Truth Table

Four Bits Received				Parity Error Check
x	y	z	P	C
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

represents an odd function. The parity checker can be implemented with exclusive-OR gates:

$$C = x \oplus y \oplus z \oplus P$$

The logic diagram of the parity checker is shown in Fig. 3.34(b).

It is worth noting that the parity generator can be implemented with the circuit of Fig. 3.34(b) if the input P is connected to logic 0 and the output is marked with P. This is because $z \oplus 0 = z$, causing the value of z to pass through the gate unchanged. The advantage of this strategy is that the same circuit can be used for both parity generation and checking.

It is obvious from the foregoing example that parity generation and checking circuits always have an output function that includes half of the minterms whose numerical values have either an odd or even number of 1's. As a consequence, they can be implemented with exclusive-OR gates. A function with an even number of 1's is the complement of an odd function. It is implemented with exclusive-OR gates, except that the gate associated with the output must be an exclusive-NOR to provide the required complementation.

3.9 HARDWARE DESCRIPTION LANGUAGE

Manual methods for designing logic circuits are feasible only when the circuit is small. For anything else (i.e., a practical circuit), designers use computer-based design tools. Coupled with the correct-by-construction methodology, computer-based design tools leverage the creativity and the effort of a designer and reduce the risk of producing a flawed design. Prototype integrated circuits are too expensive and time consuming to build, so all modern design tools rely on a hardware description language to describe, design, and test a circuit in software before it is ever manufactured.

A *hardware description language* (HDL) is a computer-based language that describes the hardware of digital systems in a textual form. It resembles an ordinary computer programming language, such as C, but is specifically oriented to describing hardware structures and the behavior of logic circuits. It can be used to represent logic diagrams, truth tables, Boolean expressions, and complex abstractions of the behavior of a digital system. One way to view an HDL is to observe that it describes a relationship between signals that are the inputs to a circuit and the signals that are the outputs of the circuit. For example, an HDL description of an AND gate describes how the logic value of the gate's output is determined by the logic values of its inputs.

As a *documentation* language, an HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers. The language content can be stored, retrieved, edited, and transmitted easily and processed by computer software in an efficient manner.

HDLs are used in several major steps in the design flow of an integrated circuit: design entry, functional simulation or verification, logic synthesis, timing verification, and fault simulation.

Design entry creates an HDL-based description of the functionality that is to be implemented in hardware. Depending on the HDL, the description can be in a variety of forms: Boolean logic equations, truth tables, a netlist of interconnected gates, or an abstract behavioral model. The HDL model may also represent a partition of a larger circuit into smaller interconnected and interacting functional units.

Logic simulation displays the behavior of a digital system through the use of a computer. A simulator interprets the HDL description and either produces readable output, such as a time-ordered sequence of input and output signal values, or displays waveforms of the signals. The simulation of a circuit predicts how the hardware will behave before it is actually fabricated. Simulation detects functional errors in a design without having to physically create and operate the circuit. Errors that are detected during a simulation can be corrected by modifying the appropriate HDL statements. The stimulus (i.e., the logic values of the inputs to a circuit) that tests the functionality of the design is called a *test bench.* Thus, to simulate a digital system, the design is first described in an HDL and then verified by simulating the design and checking it with a test bench, which is also written in the HDL. An alternative and more complex approach relies on formal mathematical methods to prove that a circuit is functionally correct. We will focus exclusively on simulation.

Logic synthesis is the process of deriving a list of physical components and their interconnections (called a *netlist*) from the model of a digital system described in an HDL. The netlist can be used to fabricate an integrated circuit or to lay out a printed circuit board with the hardware counterparts of the gates in the list. Logic synthesis is similar to compiling a program in a conventional high-level language. The difference is that, instead of producing an object code, logic synthesis produces a database describing the elements and structure of a circuit. The database specifies how to fabricate a physical integrated circuit that implements in silicon the functionality described by statements made in an HDL. Logic synthesis is based on formal exact procedures that implement digital circuits and addresses that part of a digital design which can be automated with computer software. The design of today's large, complex circuits is made possible by logic synthesis software.

Timing verification confirms that the fabricated, integrated circuit will operate at a specified speed. Because each logic gate in a circuit has a propagation delay, a signal transition at the input of a circuit cannot immediately cause a change in the logic value of the output of a circuit. Propagation delays ultimately limit the speed at which a circuit can operate. Timing verification checks each signal path to verify that it is not compromised by propagation delay. This step is done after logic synthesis specifies the actual devices that will compose a circuit and before the circuit is released for production.

In VLSI circuit design, *fault simulation* compares the behavior of an ideal circuit with the behavior of a circuit that contains a process-induced flaw. Dust and other particulates in the atmosphere of the clean room can cause a circuit to be fabricated with a fault. A circuit with a fault will not exhibit the same functionality as a fault-free circuit. Fault simulation is used to identify input stimuli that can be used to reveal the difference between the faulty circuit and the fault-free circuit. These test patterns will be used to test fabricated devices to ensure that only good devices are shipped to the customer. Test generation and fault simulation may occur at different steps in the design process, but they are always done before production in order to avoid the disaster of producing a circuit whose internal logic cannot be tested.

Companies that design integrated circuits use proprietary and public HDLs. In the public domain, there are two standard HDLs that are supported by the IEEE: VHDL and Verilog. VHDL is a Department of Defense–mandated language. (The *V* in VHDL stands for the first letter in VHSIC, an acronym for very high-speed integrated circuit.) Verilog began as a proprietary HDL of Cadence Design Systems, but Cadence transferred control of Verilog to a consortium of companies and universities known as Open Verilog International (OVI) as a step leading to its adoption as an IEEE standard. VHDL is more difficult to learn than Verilog. Because Verilog is an easier language than VHDL to describe, learn, and use, we have chosen it for this book. However, the Verilog HDL descriptions listed throughout the book are not just about Verilog, but also serve to introduce a design methodology based on the concept of computer-aided modeling of digital systems by means of a typical hardware description language. Our emphasis will be on the modeling, verification, and synthesis (both manual and automated) of Verilog models of circuits having specified behavior. The Verilog HDL was initially approved as a standard HDL in 1995; revised and enhanced versions of the language were approved in 2001 and 2005. We will address only those features of Verilog, including the latest standard, that support our discussion of HDL-based design methodology for integrated circuits.

Module Declaration

The language reference manual for the Verilog HDL presents a syntax that describes precisely the constructs that can be used in the language. In particular, a Verilog model is composed of text using keywords, of which there are about 100. Keywords are predefined lowercase identifiers that define the language constructs. Examples of keywords are **module**, **endmodule**, **input**, **output**, **wire**, **and**, **or**, and **not**. For clarity, keywords will be displayed in boldface in the text in all examples of code and wherever it is appropriate to call attention to their use. Any text between two forward slashes (//) and the end of the line is interpreted as a comment and will have no effect on a simulation using the model. Multiline comments begin with /* and terminate with */. Blank spaces are ignored, but they may not appear within the text of a keyword, a user-specified identifier, an operator, or the representation of a number. Verilog is case sensitive, which means that uppercase and lowercase letters are distinguishable (e.g., **not** is not the same as NOT). The term *module* refers to the text enclosed by the keyword pair **module** . . . **endmodule**. A module is the fundamental descriptive unit in the Verilog language. It is declared by the keyword **module** and must always be terminated by the keyword **endmodule**.

Combinational logic can be described by a schematic connection of gates, by a set of Boolean equations, or by a truth table. Each type of description can be developed in Verilog. We will demonstrate each style, beginning with a simple example of a Verilog gate-level description to illustrate some aspects of the language.

The HDL description of the circuit of Fig. 3.35 is shown in HDL Example 3.1. The first line of text is a comment (optional) providing useful information to the reader. The second line begins with the keyword **module** and starts the declaration (description) of the module; the last line completes the declaration with the keyword **endmodule**. The keyword **module** is followed by a name and a list of ports. The name (*Simple_Circuit* in this example) is an identifier. Identifiers are names given to modules, variables (e.g., a signal), and other elements of the language so that they can be referenced in the design. In general, we choose meaningful names for modules. Identifiers are composed of alphanumeric characters and the underscore (_), and are case sensitive. Identifiers must start with an alphabetic character or an underscore, but they cannot start with a number.

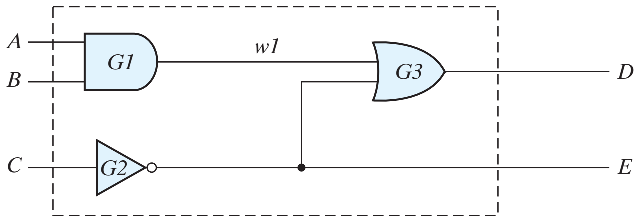

FIGURE 3.35
Circuit to demonstrate an HDL

HDL Example 3.1 (Combinational Logic Modeled with Primitives)

```
// Verilog model of circuit of Figure 3.35. IEEE 1364–1995 Syntax

module  Simple_Circuit (A, B, C, D, E);
 output          D, E;
 input           A, B, C;
 wire            w1;

 and             G1 (w1, A, B); // Optional gate instance name
 not             G2 (E, C);
 or              G3 (D, w1, E);
endmodule
```

The *port list* of a module is the interface between the module and its environment. In this example, the ports are the inputs and outputs of the circuit. The logic values of the inputs to a circuit are determined by the environment; the logic values of the outputs are determined within the circuit and result from the action of the inputs on the circuit. The port list is enclosed in parentheses, and commas are used to separate elements of the list. The statement is terminated with a semicolon (;). In our examples, all keywords (which must be in lowercase) are printed in bold for clarity, but that is not a requirement of the language. Next, the keywords **input** and **output** specify which of the ports are inputs and which are outputs. Internal connections are declared as wires. The circuit in this example has one internal connection, at terminal *w1*, and is declared with the keyword **wire.** The structure of the circuit is specified by a list of (predefined) *primitive* gates, each identified by a descriptive keyword (**and, not, or**). The elements of the list are referred to as *instantiations* of a gate, each of which is referred to as a *gate instance*. Each *gate instantiation* consists of an optional name (such as *G1, G2*, etc.) followed by the gate output and inputs separated by commas and enclosed within parentheses. The output of a primitive gate is always listed first, followed by the inputs. For example, the OR gate of the schematic is represented by the **or** primitive, is named *G3*, and has output *D* and inputs *w1* and *E*. (*Note*: The output of a primitive must be listed first, but the inputs and outputs of a module may be listed in any order.) The module description ends with the keyword **endmodule.** Each statement must be terminated with a semicolon, but there is no semicolon after **endmodule.**

It is important to understand the distinction between the terms *declaration* and *instantiation*. A Verilog module is declared. Its declaration specifies the input–output behavior of the hardware that it represents. Predefined primitives are not declared, because their definition is specified by the language and is not subject to change by the user. Primitives are used (i.e., instantiated), just as gates are used to populate a printed circuit board. We'll see that once a module has been declared, it may be used (instantiated) within a design. Note that *Simple_Circuit* is not a computational model like those developed in an ordinary programming language: The sequential ordering of the statements instantiating gates in the model has no significance and does not specify a sequence of computations. A Verilog model is a *descriptive* model. *Simple_Circuit* describes what primitives form a circuit and how they are connected. The input–output behavior of the circuit is

Table 3.5
Output of Gates after Delay

	Time Units (ns)	Input ABC	Output E w1 D
Initial	—	0 0 0	1 0 1
Change	—	1 1 1	1 0 1
	10	1 1 1	0 0 1
	20	1 1 1	0 0 1
	30	1 1 1	0 1 0
	40	1 1 1	0 1 0
	50	1 1 1	0 1 1

implicitly specified by the description because the behavior of each logic gate is defined. Thus, an HDL-based model can be used to simulate the circuit that it represents.

Gate Delays

All physical circuits exhibit a propagation delay between the transition of an input and a resulting transition of an output. When an HDL model of a circuit is simulated, it is sometimes necessary to specify the amount of delay from the input to the output of its gates. In Verilog, the propagation delay of a gate is specified in terms of *time units* and by the symbol #. The numbers associated with time delays in Verilog are dimensionless. The association of a time unit with physical time is made with the `**timescale** compiler directive. (Compiler directives start with the (`) back quote, or grave accent, symbol.) Such a directive is specified before the declaration of a module and applies to all numerical values of time in the code that follows. An example of a timescale directive is

```
`timescale 1ns/100ps
```

The first number specifies the unit of measurement for time delays. The second number specifies the precision for which the delays are rounded off, in this case to 0.1 ns. If no timescale is specified, a simulator may display dimensionless values or default to a certain time unit, usually 1 ns ($=10^{-9}$ s). Our examples will use only the default time unit.

HDL Example 3.2 repeats the description of the simple circuit of Example 3.1, but with propagation delays specified for each gate. The **and**, **or**, and **not** gates have a time delay of 30, 20, and 10 ns, respectively. If the circuit is simulated and the inputs change from $A, B, C = 0$ to $A, B, C = 1$, the outputs change as shown in Table 3.5 (calculated by hand or generated by a simulator). The output of the inverter at E changes from 1 to 0 after a 10-ns delay. The output of the AND gate at $w1$ changes from 0 to 1 after a 30-ns delay. The output of the OR gate at D changes from 1 to 0 at $t = 30$ ns and then changes back to 1 at $t = 50$ ns. In both cases, the change in the output of the OR gate results from a change in its inputs 20 ns earlier. It is clear from this result that although output D eventually returns to a final value of 1 after the input changes, the gate delays produce a negative spike that lasts 20 ns before the final value is reached.

HDL Example 3.2 (Gate-Level Model with Propagation Delays)

```
// Verilog model of simple circuit with propagation delay

module Simple_Circuit_prop_delay (A, B, C, D, E);
 output D, E;
 input  A, B, C;
 wire   w1;

 and          #(30) G1 (w1, A, B);
 not          #(10) G2 (E, C);
 or           #(20) G3 (D, w1, E);
endmodule
```

In order to simulate a circuit with an HDL, it is necessary to apply inputs to the circuit so that the simulator will generate an output response. An HDL description that provides the stimulus to a design is called a *test bench.* The writing of test benches is explained in more detail at the end of Section 4.12. Here, we demonstrate the procedure with a simple example without dwelling on too many details. HDL Example 3.3 shows a test bench for simulating the circuit with delay. (Note the distinguishing name *Simple_Circuit_prop_delay*.) In its simplest form, a test bench is a module containing a signal generator and an instantiation of the model that is to be verified. Note that the test bench (*t_Simple_Circuit_prop_delay*) has no input or output ports, because it does not interact with its environment. In general, we prefer to name the test bench with the prefix *t_* concatenated with the name of the module that is to be tested by the test bench, but that choice is left to the designer. Within the test bench, the inputs to the circuit are declared with keyword **reg** and the outputs are declared with the keyword **wire**. The module *Simple_Circuit_prop_delay* is instantiated with the instance name M1. Every instantiation of a module must include a unique instance name. Note that using a test bench is similar to testing actual hardware by attaching signal generators to the inputs of a circuit and attaching

HDL Example 3.3 (Test Bench)

```
// Test bench for Simple_Circuit_prop_delay

module t_Simple_Circuit_prop_delay;
 wire   D, E;
 reg    A, B, C;

Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

 initial #200 $finish;
endmodule
```

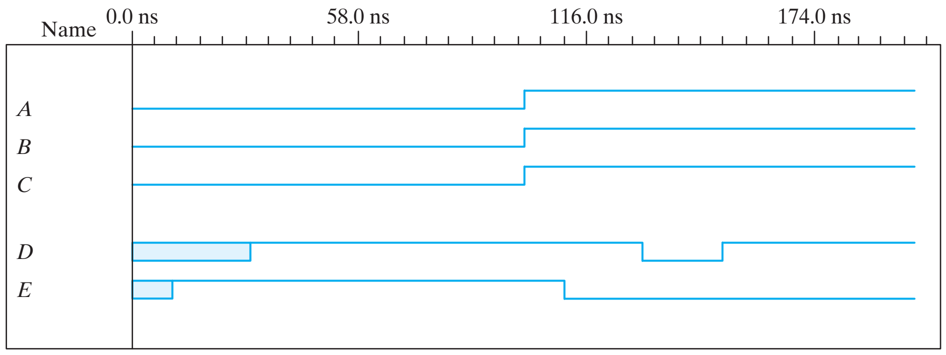

FIGURE 3.36
Simulation output of HDL Example 3.3

probes (wires) to the outputs of the circuit. (The interaction between the signal generators of the stimulus module and the instantiated circuit module is illustrated in Fig. 4.36.)

Hardware signal generators are not used to verify an HDL model: The entire simulation exercise is done with software models executing on a digital computer under the direction of an HDL simulator. The waveforms of the input signals are abstractly modeled (generated) by Verilog statements specifying waveform values and transitions. The **initial** keyword is used with a set of statements that begin executing when the simulation is initialized; the signal activity associated with **initial** terminates execution when the last statement has finished executing. The **initial** statements are commonly used to describe waveforms in a test bench. The set of statements to be executed is called a *block statement* and consists of several statements enclosed by the keywords **begin** and **end**. The action specified by the statements begins when the simulation is launched, and the statements are executed in sequence, left to right, from top to bottom, by a simulator in order to provide the input to the circuit. Initially, $A, B, C = 0$. (*A, B,* and *C* are each set to 1′b0, which signifies one binary digit with a value of 0.) After 100 ns, the inputs change to $A, B, C = 1$. After another 100 ns, the simulation terminates at time 200 ns. A second **initial** statement uses the **$finish** system task to specify termination of the simulation. If a statement is preceded by a delay value (e.g., #100), the simulator postpones executing the statement until the specified time delay has elapsed. The timing diagram of waveforms that result from the simulation is shown in Figure 3.36. The total simulation generates waveforms over an interval of 200 ns. The inputs *A, B,* and *C* change from 0 to 1 after 100 ns. Output *E* is unknown for the first 10 ns (denoted by shading), and output *D* is unknown for the first 30 ns. Output *E* goes from 1 to 0 at 110 ns. Output *D* goes from 1 to 0 at 130 ns and back to 1 at 150 ns, just as we predicted in Table 3.5.

Boolean Expressions

Boolean equations describing combinational logic are specified in Verilog with a continuous assignment statement consisting of the keyword **assign** followed by a Boolean expression. To distinguish arithmetic operators from logical operators, Verilog uses the symbols (&), (/), and (~) for AND, OR, and NOT (complement), respectively. Thus, to describe the simple circuit of Fig. 3.35 with a Boolean expression, we use the statement

assign D = (A && B)||(!C);

HDL Example 3.4 describes a circuit that is specified with the following two Boolean expressions:

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

The equations specify how the logic values E and F are determined by the values of A, B, C, and D.

HDL Example 3.4 (Combinational Logic Modeled with Boolean Equations)

```
// Verilog model: Circuit with Boolean expressions

module  Circuit_Boolean_CA (E, F, A, B, C, D);
 output      E, F;
 input       A, B, C, D;

 assign E = A || (B && C) || ((!B) && D);
 assign F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```

The circuit has two outputs E and F and four inputs A, B, C, and D. The two **assign** statements describe the Boolean equations. The values of E and F during simulation are determined dynamically by the values of A, B, C, and D. The simulator detects when the test bench changes a value of one or more of the inputs. When this happens, the simulator updates the values of E and F. The continuous assignment mechanism is so named because the relationship between the assigned value and the variables is permanent. The mechanism acts just like combinational logic, has a gate-level equivalent circuit, and is referred to as *implicit combinational logic*.

We have shown that a digital circuit can be described with HDL statements, just as it can be drawn in a circuit diagram or specified with a Boolean expression. A third alternative is to describe combinational logic with a truth table.

User-Defined Primitives

The logic gates used in Verilog descriptions with keywords **and, or,** etc., are defined by the system and are referred to as *system primitives. (Caution: Other languages may use these words differently.)* The user can create additional primitives by defining them in tabular form. These types of circuits are referred to as *user-defined primitives* (UDPs). One way of specifying a digital circuit in tabular form is by means of a truth table. UDP descriptions do not use the keyword pair **module** . . . **endmodule.** Instead, they are declared with the keyword pair **primitive** . . . **endprimitive.** The best way to demonstrate a UDP declaration is by means of an example.

HDL Example 3.5 defines a UDP with a truth table. It proceeds according to the following general rules:

- It is declared with the keyword **primitive**, followed by a name and port list.
- There can be only one output, and it must be listed first in the port list and declared with keyword **output**.
- There can be any number of inputs. The order in which they are listed in the **input** declaration must conform to the order in which they are given values in the table that follows.
- The truth table is enclosed within the keywords **table** and **endtable**.
- The values of the inputs are listed in order, ending with a colon (:). The output is always the last entry in a row and is followed by a semicolon (;).
- The declaration of a UDP ends with the keyword **endprimitive**.

HDL Example 3.5 (User-Defined Primitive)

```
// Verilog model: User-defined Primitive
primitive  UDP_02467 (D, A, B, C);
 output  D;
 input    A, B, C;
//Truth table for D 5 f (A, B, C) 5 Σ(0, 2, 4, 6, 7);
 table
//      A      B      C      :      D        // Column header comment
        0      0      0      :      1;
        0      0      1      :      0;
        0      1      0      :      1;
        0      1      1      :      0;
        1      0      0      :      1;
        1      0      1      :      0;
        1      1      0      :      1;
        1      1      1      :      1;
 endtable
endprimitive

// Instantiate primitive

// Verilog model: Circuit instantiation of Circuit_UDP_02467

module  Circuit_with_UDP_02467 (e, f, a, b, c, d);
 output           e, f;
 input            a, b, c, d

 UDP_02467                (e, a, b, c);
 and                      (f, e, d);            // Option gate instance name omitted
endmodule
```

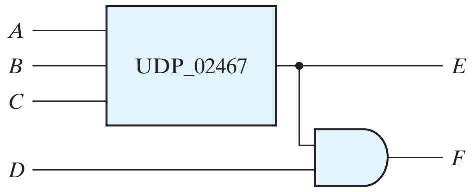

FIGURE 3.37
Schematic for *Circuit with_UDP_02467*

Note that the variables listed on top of the table are part of a comment and are shown only for clarity. The system recognizes the variables by the order in which they are listed in the input declaration. A user-defined primitive can be instantiated in the construction of other modules (digital circuits), just as the system primitives are used. For example, the declaration

```
Circuit_with_UDP_02467 (E, F, A, B, C, D);
```

will produce a circuit that implements the hardware shown in Figure 3.37.

Although Verilog HDL uses this kind of description for UDPs only, other HDLs and computer-aided design (CAD) systems use other procedures to specify digital circuits in tabular form. The tables can be processed by CAD software to derive an efficient gate structure of the design. None of Verilog's predefined primitives describes sequential logic. The model of a sequential UDP requires that its output be declared as a **reg** data type, and that a column be added to the truth table to describe the next state. So the columns are organized as inputs : state : next state.

In this section, we introduced the Verilog HDL and presented simple examples to illustrate alternatives for modeling combinational logic. A more detailed presentation of Verilog HDL can be found in the next chapter. The reader familiar with combinational circuits can go directly to Section 4.12 to continue with this subject.

PROBLEMS

(Answers to problems marked with * appear at the end of the text.)

3.1* Simplify the following Boolean functions, using three-variable maps:
(a) $F(x, y, z) = \Sigma(0, 2, 4, 5)$ (b) $F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$
(c) $F(x, y, z) = \Sigma(0, 1, 2, 3, 5)$ (d) $F(x, y, z) = \Sigma(1, 2, 3, 7)$

3.2 Simplify the following Boolean functions, using three-variable maps:
(a)* $F(x, y, z) = \Sigma(0, 1, 5, 7)$ (b)* $F(x, y, z) = \Sigma(1, 2, 3, 6, 7)$
(c) $F(x, y, z) = \Sigma(2, 3, 4, 5)$ (d) $F(x, y, z) = \Sigma(1, 2, 3, 5, 6, 7)$
(e) $F(x, y, z) = \Sigma(0, 2, 4, 6)$ (f) $F(x, y, z) = \Sigma(3, 4, 5, 6, 7)$

3.3* Simplify the following Boolean expressions, using three-variable maps:
(a)* $xy + x'y'z' + x'yz'$ (b)* $x'y' + yz + x'yz'$
(c)* $F(x, y, z) = x'y + yz' + y'z'$ (d) $F(x, y, z) = x'yz + xy'z' + xy'z$

3.4 Simplify the following Boolean functions, using *Karnaugh* maps:

(a)* $F(x, y, z) = \Sigma(2, 3, 6, 7)$ (b)* $F(A, B, C, D) = \Sigma(4, 6, 7, 15)$

(c)* $F(A, B, C, D) = \Sigma(3, 7, 11, 13, 14, 15)$ (d)* $F(w, x, y, z) = \Sigma(2, 3, 12, 13, 14, 15)$

(e) $F(w, x, y, z) = \Sigma(11, 12, 13, 14, 15)$ (f) $F(w, x, y, z) = \Sigma(8, 10, 12, 13, 14)$

3.5 Simplify the following Boolean functions, using four-variable maps:

(a)* $F(w, x, y, z) = \Sigma(1, 4, 5, 6, 12, 14, 15)$

(b) $F(A, B, C, D) = \Sigma(2, 3, 6, 7, 12, 13, 14)$

(c) $F(w, x, y, z) = \Sigma(1, 3, 4, 5, 6, 7, 9, 11, 13, 15)$

(d)* $F(A, B, C, D) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$

3.6 Simplify the following Boolean expressions, using four-variable maps:

(a)* $A'B'C'D' + AC'D' + B'CD' + A'BCD + BC'D$

(b)* $x'z + w'xy' + w(x'y + xy')$

(c) $A'B'C'D + AB'D + A'BC' + ABCD + AB'C$

(d) $A'B'C'D' + BC'D + A'C'D + A'BCD + ACD'$

3.7 Simplify the following Boolean expressions, using four-variable maps:

(a)* $w'z + xz + x'y + wx'z$

(b) $AD' + B'C'D + BCD' + BC'D$

(c)* $AB'C + B'C'D' + BCD + ACD' + A'B'C + A'BC'D$

(d) $wxy + xz + wx'z + w'x$

3.8 Find the minterms of the following Boolean expressions by first plotting each function in a map:

(a)* $xy + yz + xy'z$ (b)* $C'D + ABC' + ABD' + A'B'D$

(c) $wyz + w'x' + wxz'$ (d) $A'B + A'CD + B'CD + BC'D'$

3.9 Find all the prime implicants for the following Boolean functions, and determine which are essential:

(a)* $F(w, x, y, z) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$

(b)* $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$

(c) $F(A, B, C, D) = \Sigma(2, 3, 4, 5, 6, 7, 9, 11, 12, 13)$

(d) $F(w, x, y, z) = \Sigma(1, 3, 6, 7, 8, 9, 12, 13, 14, 15)$

(e) $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 7, 8, 9, 10, 13, 15)$

(f) $F(w, x, y, z) = \Sigma(0, 1, 2, 5, 7, 8, 10, 15)$

3.10 Simplify the following Boolean functions by first finding the essential prime implicants:

(a) $F(w, x, y, z) = \Sigma(0, 2, 5, 7, 8, 10, 12, 13, 14, 15)$

(b) $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$

(c)* $F(A, B, C, D) = \Sigma(1, 3, 4, 5, 10, 11, 12, 13, 14, 15)$

(d) $F(w, x, y, z) = \Sigma(0, 1, 4, 5, 6, 7, 9, 11, 14, 15)$

(e) $F(A, B, C, D) = \Sigma(0, 1, 3, 7, 8, 9, 10, 13, 15)$

(f) $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 7, 10, 15)$

3.11 Convert the following Boolean function from a sum-of-products form to a simplified product-of-sums form.

$$F(x, y, z) = \Sigma(0, 1, 2, 5, 8, 10, 13)$$

3.12 Simplify the following Boolean functions:

(a)* $F(A, B, C, D) = \Pi(1, 3, 5, 7, 13, 15)$

(b) $F(A, B, C, D) = \Pi(1, 3, 6, 9, 11, 12, 14)$

3.13 Simplify the following expressions to (1) sum-of-products and (2) products-of-sums:

(a)* $x'z' + y'z' + yz' + xy$

(b) $ACD' + C'D + AB' + ABCD$

(c) $(A' + B + D')(A' + B' + C')(A' + B' + C)(B' + C + D')$

(d) $BCD' + ABC' + ACD$

3.14 Give three possible ways to express the following Boolean function with eight or fewer literals:

$$F = A'BC'D + AB'CD + A'B'C' + ACD'$$

3.15 Simplify the following Boolean function F, together with the don't-care conditions d, and then express the simplified function in sum-of-minterms form:

(a) $F(x, y, z) = \Sigma(0, 1, 4, 5, 6)$
$d(x, y, z) = \Sigma(2, 3, 7)$

(b)* $F(A, B, C, D) = \Sigma(0, 6, 8, 13, 14)$
$d(A, B, C, D) = \Sigma(2, 4, 10)$

(c) $F(A, B, C, D) = \Sigma(5, 6, 7, 12, 14, 15,)$
$d(A, B, C, D) = \Sigma(3, 9, 11, 15)$

(d) $F(A, B, C, D) = \Sigma(4, 12, 7, 2, 10,)$
$d(A, B, C, D) = \Sigma(0, 6, 8)$

3.16 Simplify the following functions, and implement them with two-level NAND gate circuits:

(a) $F(A, B, C, D) = AC'D' + A'C + ABC + AB'C + A'C'D'$

(b) $F(A, B, C, D) = A'B'C'D + CD + AC'D$

(c) $F(A, B, C) = (A' + C' + D')(A' + C')(C' + D')$

(d) $F(A, B, C, D) = A' + B + D' + B'C$

3.17* Draw a NAND logic diagram that implements the complement of the following function:

$$F(A, B, C, D) = \Sigma(0, 1, 2, 3, 6, 10, 11, 14)$$

3.18 Draw a logic diagram using only two-input NOR gates to implement the following function:

$$F(A, B, C, D) = (A \oplus B)'(C \oplus D)$$

3.19 Simplify the following functions, and implement them with two-level NOR gate circuits:

(a)* $F = wx' + y'z' + w'yz'$

(b) $F(w, x, y, z) = \Sigma(0, 3, 12, 15)$

(c) $F(x, y, z) = [(x + y)(x = z)]'$

3.20 Draw the multiple-level NOR circuit for the following expression:

$$CD(B + C)A + (BC' + DE')$$

3.21 Draw the multiple-level NAND circuit for the following expression:

$$w(x + y + z) + xyz$$

3.22 Convert the logic diagram of the circuit shown in Fig. 4.4 into a multiple-level NAND circuit.

3.23 Implement the following Boolean function F, together with the don't-care conditions d, using no more than two NOR gates:

$$F(A, B, C, D) = \Sigma(2, 4, 10, 12, 14,)$$
$$d(A, B, C, D) = \Sigma(0, 1, 5, 8)$$

Assume that both the normal and complement inputs are available.

3.24 Implement the following Boolean function F, using the two-level forms of logic (a) NAND-AND, (b) AND-NOR, (c) OR-NAND, and (d) NOR-OR:

$$F(A, B, C, D) = \Sigma(0, 4, 8, 9, 10, 11, 12, 14)$$

3.25 List the eight degenerate two-level forms and show that they reduce to a single operation. Explain how the degenerate two-level forms can be used to extend the number of inputs to a gate.

3.26 With the use of maps, find the simplest sum-of-products form of the function $F = fg$, where

$$f = abc' + c'd + a'cd' + b'cz'$$

and

$$g = (a + b + c' + d')(b' + c' + d)(a' + c + d')$$

3.27 Show that the dual of the exclusive-OR is also its complement.

3.28 Derive the circuits for a three-bit parity generator and four-bit parity checker using an odd parity bit.

3.29 Implement the following four Boolean expressions with three half adders:

$$D = A \oplus B \oplus C$$
$$E = A'BC + AB'C$$
$$F = ABC' + (A' + B')C$$
$$G = ABC$$

3.30* Implement the following Boolean expression with exclusive-OR and AND gates:

$$F = AB'CD' + A'BCD' + AB'C'D + A'BC'D$$

3.31 Write a Verilog gate-level description of the circuit shown in
(a) Fig. 3.20(a) (b) Fig. 3.20(b) (c) Fig. 3.21(a)
(d) Fig. 3.21(b) (e) Fig. 3.24 (f) Fig. 3.25

3.32 Using continuous assignment statements, write a Verilog description of the circuit shown in
(a) Fig. 3.20(a) (b) Fig. 3.20(b) (c) Fig. 3.21(a)
(d) Fig. 3.21(b) (e) Fig. 3.24 (f) Fig. 3.25

3.33 The exclusive-OR circuit of Fig. 3.30(a) has gates with a delay of 3 ns for an inverter, a 6 ns delay for an AND gate, and a 8 ns delay for an OR gate. The input of the circuit goes from $xy = 00$ to $xy = 01$.
(a) Determine the signals at the output of each gate from $t = 0$ to $t = 50$ ns.
(b) Write a Verilog gate-level description of the circuit, including the delays.
(c) Write a stimulus module (i.e., a test bench similar to HDL Example 3.3), and simulate the circuit to verify the answer in part (a).

3.34 Using continuous assignments, write a Verilog description of the circuit specified by the following Boolean functions:

$$Out\_1 = (A + B')C'(C + D)$$
$$Out\_2 = (C'D + BCD + CD')(A' + B)$$
$$Out\_3 = (AB + C)D + B'C$$

Write a test bench and simulate the circuit's behavior.

3.35* Find the syntax errors in the following declarations (note that names for primitive gates are optional):

```
module Exmpl-3(A, B, C, D, F)           // Line 1
 inputs      A, B, C, Output D, F,      // Line 2
 output      B                          // Line 3
 and         g1(A, B, D);               // Line 4
 not         (D, A, C),                 // Line 5
 OR          (F, B; C);                 // Line 6
endmodule;                              // Line 7
```

3.36 Draw the logic diagram of the digital circuit specified by the following Verilog description:

(a)
```
module Circuit_A (A, B, C, D, F);
  input      A, B, C, D;
  output     F;
  wire       w, x, y, z, a, d;
  or         (x, B, C, d);
  and        (y, a ,C);
  and        (w, z ,B);
  and        (z, y, A);
  or         (F, x, w);
  not        (a, A);
  not        (d, D);
endmodule
```

(b)
```
module Circuit_B (F1, F2, F3, A0, A1, B0, B1);
  output     F1, F2, F3;
  input      A0, A1, B0, B1;
  nor        (F1, F2, F3);
  or         (F2, w1, w2, w3);
  and        (F3, w4, w5);
  and        (w1, w6, B1);
  or         (w2, w6, w7, B0);
  and        (w3, w7, B0, B1);
  not        (w6, A1);
  not        (w7, A0);
  xor        (w4, A1, B1);
  xnor       (w5, A0, B0);
endmodule
```

(c)
```
module Circuit_C (y1, y2, y3, a, b);
  output y1, y2, y3;
  input a, b;

  assign y1 = a || b;
  and (y2, a, b);
  assign y3 = a && b;
endmodule
```

3.37 A majority logic function is a Boolean function that is equal to 1 if the majority of the variables are equal to 1, equal to 0 otherwise.
(a) Write a truth table for a four-bit majority function.
(b) Write a Verilog user-defined primitive for a four-bit majority function.

3.38 Simulate the behavior of *Circuit_with_UDP_02467*, using the stimulus waveforms shown in Fig. P3.38.

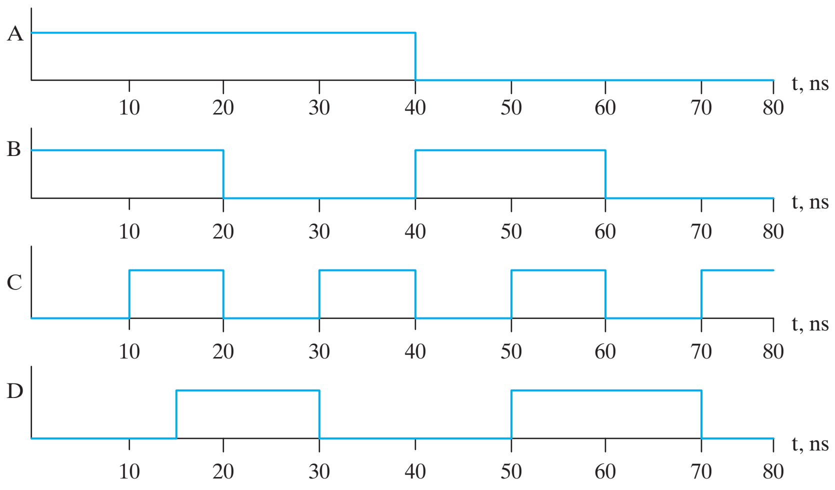

FIGURE P3.38
Stimulus waveforms for Problem 3.38

3.39 Using primitive gates, write a Verilog model of a circuit that will produce two outputs, s and c, equal to the sum and carry produced by adding two binary input bits a and b (e.g., $s = 1$ and $c = 0$ if $a = 0$ and $b = 1$). (*Hint:* Begin by developing a truth table for s and c.)

REFERENCES

1. Bhasker, J. 1997. *A Verilog HDL Primer*. Allentown, PA: Star Galaxy Press.
2. Ciletti, M. D. 1999. *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL*. Upper Saddle River, NJ: Prentice Hall.
3. Hill, F. J., and G. R. Peterson. 1981. *Introduction to Switching Theory and Logical Design*, 3rd ed. New York: John Wiley.
4. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std. 1364-1995). 1995. New York: The Institute of Electrical and Electronics Engineers.
5. Karnaugh, M. A Map Method for Synthesis of Combinational Logic Circuits. *Transactions of AIEE, Communication and Electronics*. 72, part I (Nov. 1953): 593–99.
6. Kohavi, Z. 1978. *Switching and Automata Theory*, 2nd ed. New York: McGraw-Hill.