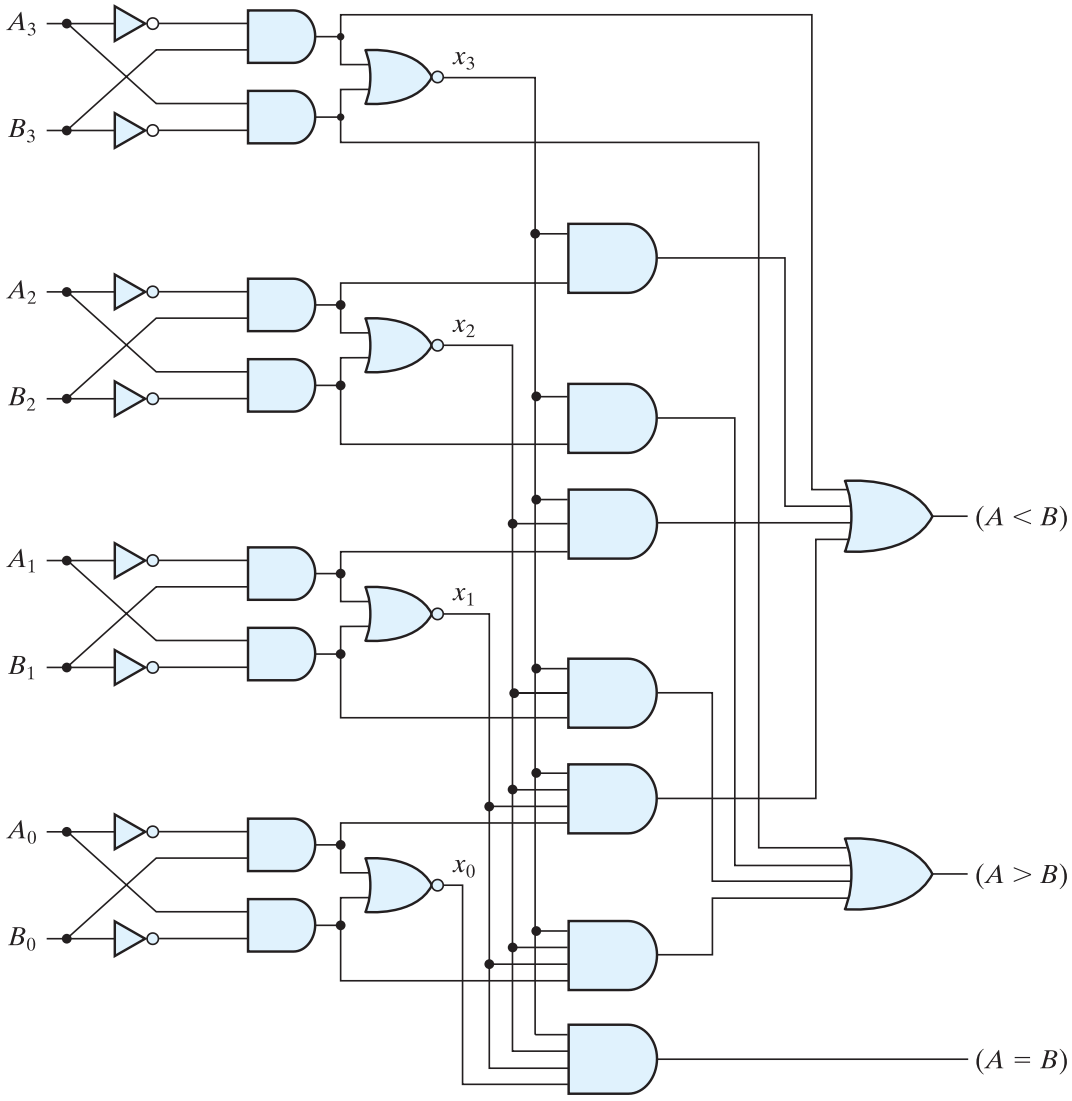


FIGURE 4.17
Four-bit magnitude comparator

with exclusive-NOR circuits and are applied to an AND gate to give the output binary variable $(A = B)$. The other two outputs use the x variables to generate the Boolean functions listed previously. This is a multilevel implementation and has a regular pattern. The procedure for obtaining magnitude comparator circuits for binary numbers with more than four bits is obvious from this example.

4.9 DECODERS

Discrete quantities of information are represented in digital systems by binary codes. A binary code of n bits is capable of representing up to 2^n distinct elements of coded information. A *decoder* is a combinational circuit that converts binary information from

n input lines to a maximum of 2^n unique output lines. If the n-bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.

The decoders presented here are called n-to-m-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables. Each combination of inputs will assert a unique output. The name *decoder* is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

As an example, consider the three-to-eight-line decoder circuit of Fig. 4.18. The three inputs are decoded into eight outputs, each representing one of the minterms of the three input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms. A particular application of this decoder is binary-to-octal conversion. The input variables represent a binary number, and the outputs represent the eight digits of a number in the octal number system. However, a three-to-eight-line decoder can be used for decoding *any* three-bit code to provide eight outputs, one for each element of the code.

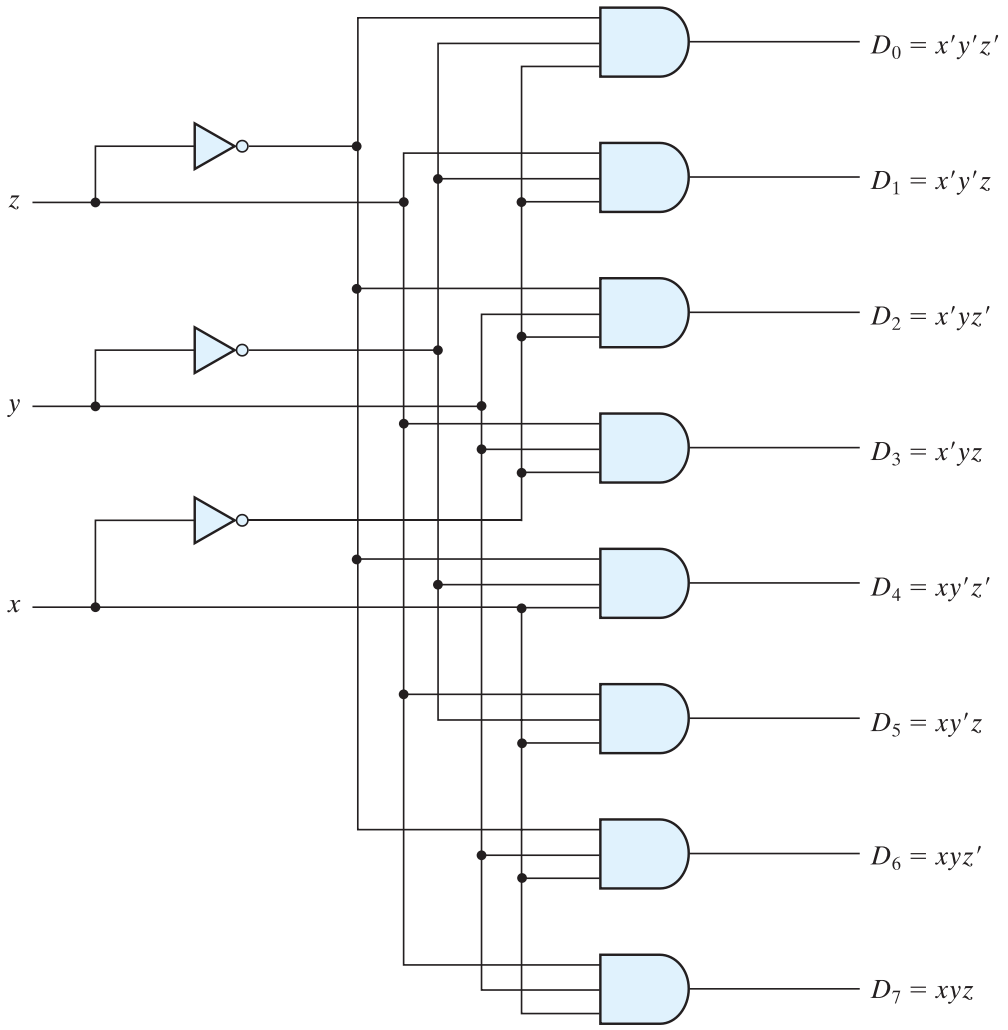


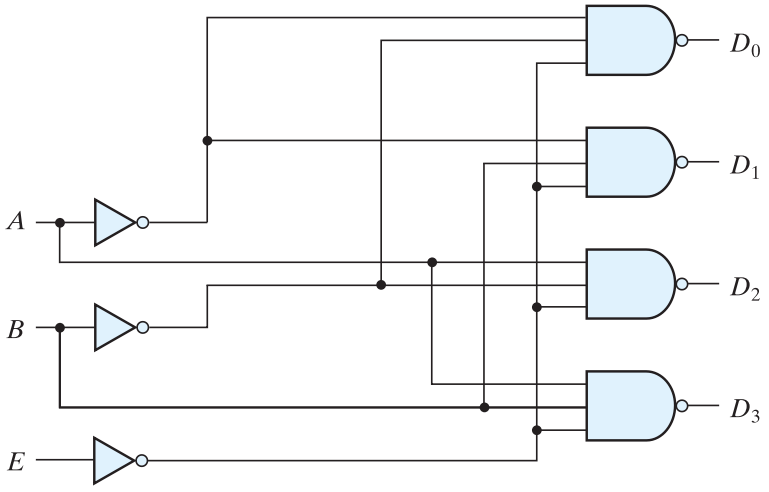
FIGURE 4.18
Three-to-eight-line decoder

Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

Inputs			Outputs							
x	*y*	*z*	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

The operation of the decoder may be clarified by the truth table listed in Table 4.6. For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output whose value is equal to 1 represents the minterm equivalent of the binary number currently available in the input lines.

Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Furthermore, decoders include one or more *enable* inputs to control the circuit operation. A two-to-four-line decoder with an enable input constructed with NAND gates is shown in Fig. 4.19. The circuit operates with complemented outputs and a complement enable input. The decoder is enabled when E is equal to 0 (i.e., active-low enable). As indicated by the truth table, only one



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	1

(b) Truth table

FIGURE 4.19
Two-to-four-line decoder with enable input

output can be equal to 0 at any given time; all other outputs are equal to 1. The output whose value is equal to 0 represents the minterm selected by inputs A and B. The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs. When the circuit is disabled, none of the outputs are equal to 0 and none of the minterms are selected. In general, a decoder may operate with complemented or uncomplemented outputs. The enable input may be activated with a 0 or with a 1 signal. Some decoders have two or more enable inputs that must satisfy a given logic condition in order to enable the circuit.

A decoder with enable input can function as a *demultiplexer*—a circuit that receives information from a single line and directs it to one of 2^n possible output lines. The selection of a specific output is controlled by the bit combination of n selection lines. The decoder of Fig. 4.19 can function as a one-to-four-line demultiplexer when E is taken as a data input line and A and B are taken as the selection inputs. The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary combination of the two selection lines A and B. This feature can be verified from the truth table of the circuit. For example, if the selection lines $AB = 10$, output D_2 will be the same as the input value E, while all other outputs are maintained at 1. Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a *decoder–demultiplexer*.

Decoders with enable inputs can be connected together to form a larger decoder circuit. Figure 4.20 shows two 3-to-8-line decoders with enable inputs connected to form a 4-to-16-line decoder. When $w = 0$, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111. When $w = 1$, the enable conditions are reversed: The bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all 0's. This example demonstrates the usefulness of enable inputs in decoders and other

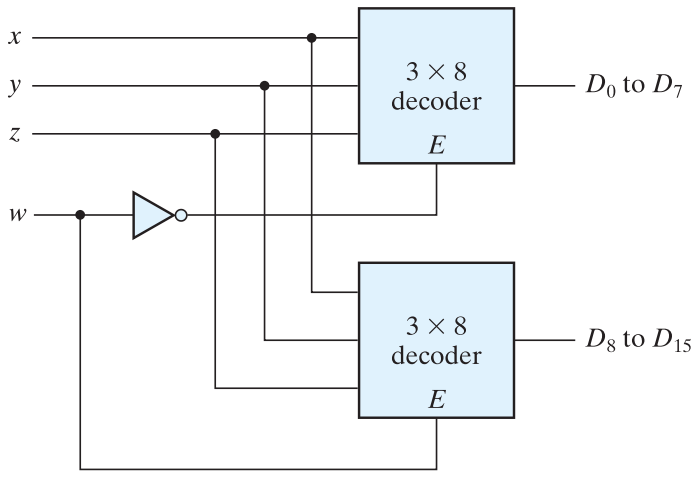


FIGURE 4.20
4 × 16 decoder constructed with two 3 × 8 decoders

combinational logic components. In general, enable inputs are a convenient feature for interconnecting two or more standard components for the purpose of combining them into a similar function with more inputs and outputs.

Combinational Logic Implementation

A decoder provides the 2^n minterms of n input variables. Each asserted output of the decoder is associated with a unique pattern of input bits. Since any Boolean function can be expressed in sum-of-minterms form, a decoder that generates the minterms of the function, together with an external OR gate that forms their logical sum, provides a hardware implementation of the function. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n-line decoder and m OR gates.

The procedure for implementing a combinational circuit by means of a decoder and OR gates requires that the Boolean function for the circuit be expressed as a sum of minterms. A decoder is then chosen that generates all the minterms of the input variables. The inputs to each OR gate are selected from the decoder outputs according to the list of minterms of each function. This procedure will be illustrated by an example that implements a full-adder circuit.

From the truth table of the full adder (see Table 4.4), we obtain the functions for the combinational circuit in sum-of-minterms form:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$

$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

Since there are three inputs and a total of eight minterms, we need a three-to-eight-line decoder. The implementation is shown in Fig. 4.21. The decoder generates the eight minterms for x, y, and z. The OR gate for output S forms the logical sum of minterms 1, 2, 4, and 7. The OR gate for output C forms the logical sum of minterms 3, 5, 6, and 7.

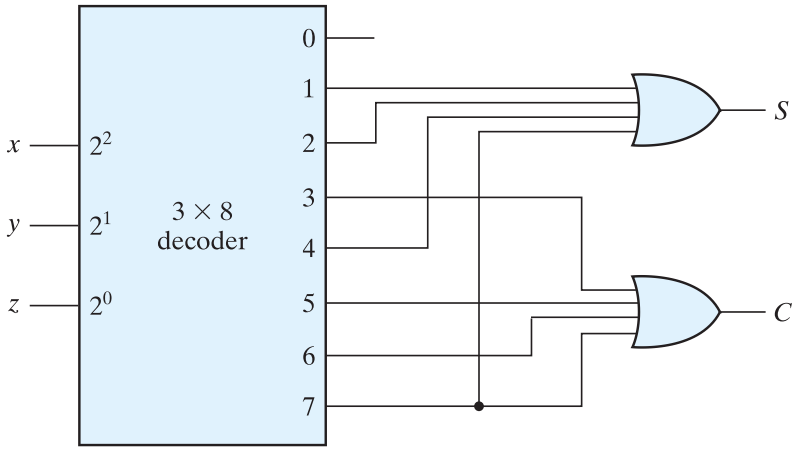


FIGURE 4.21
Implementation of a full adder with a decoder

A function with a long list of minterms requires an OR gate with a large number of inputs. A function having a list of k minterms can be expressed in its complemented form F' with $2^n - k$ minterms. If the number of minterms in the function is greater than $2^n/2$, then F' can be expressed with fewer minterms. In such a case, it is advantageous to use a NOR gate to sum the minterms of F'. The output of the NOR gate complements this sum and generates the normal output F. If NAND gates are used for the decoder, as in Fig. 4.19, then the external gates must be NAND gates instead of OR gates. This is because a two-level NAND gate circuit implements a sum-of-minterms function and is equivalent to a two-level AND–OR circuit.

4.10 ENCODERS

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in Table 4.7. It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1, 3, 5, or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7. These conditions can be expressed by the following Boolean output functions:

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

The encoder can be implemented with three OR gates.

Table 4.7
Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

The encoder defined in Table 4.7 has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. The output 111 does not represent either binary 3 or binary 6. To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110 because D_6 has higher priority than D_3.

Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; but this output is the same as when D_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

Priority Encoder

A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given in Table 4.8. In addition to the two outputs x and y, the circuit has a third output designated by V; this is a *valid* bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V is equal to 0. The other two outputs are not inspected when V equals 0 and are specified as don't-care conditions. Note that whereas X's in output columns represent don't-care conditions, the X's in the input columns are useful for representing a truth table in condensed form. Instead of listing all 16 minterms of four variables, the truth table uses an X to represent either 1 or 0. For example, X100 represents the two minterms 0100 and 1100.

According to Table 4.8, the higher the subscript number, the higher the priority of the input. Input D_3 has the highest priority, so, regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3). D_2 has the next priority level. The output is 10 if $D_2 = 1$, provided that $D_3 = 0$, regardless of the values of the other two lower priority inputs. The output for D_1 is generated only if higher priority inputs are 0, and so on down the priority levels.

Table 4.8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

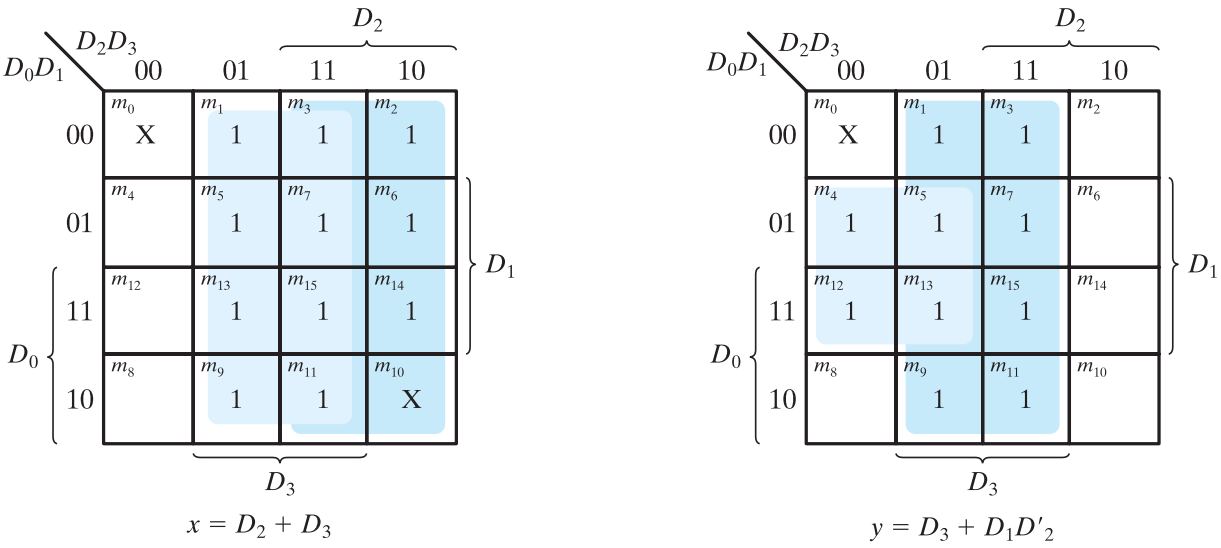


FIGURE 4.22
Maps for a priority encoder

The maps for simplifying outputs x and y are shown in Fig. 4.22. The minterms for the two functions are derived from Table 4.8. Although the table has only five rows, when each X in a row is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the fourth row in the table, with inputs XX10, represents the four minterms 0010, 0110, 1010, and 1110. The simplified Boolean expressions for the priority encoder are obtained from the maps. The condition for output V is an OR function of all the input variables. The priority encoder is implemented in Fig. 4.23 according to the following Boolean functions:

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D'_2$$
$$V = D_0 + D_1 + D_2 + D_3$$

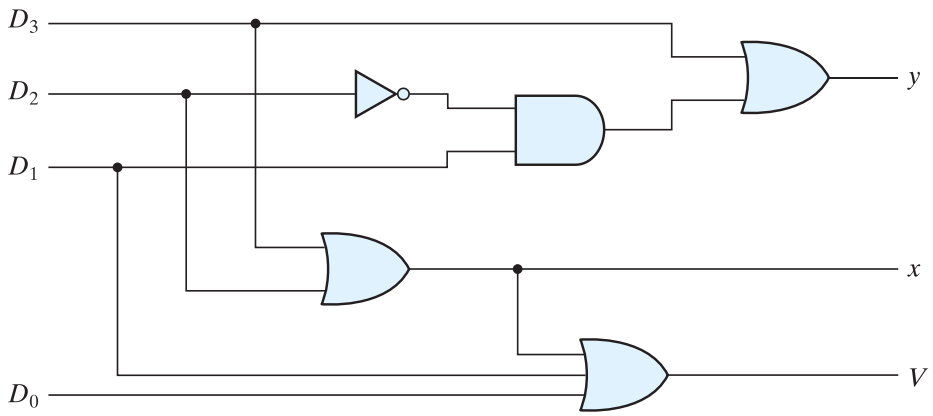


FIGURE 4.23
Four-input priority encoder

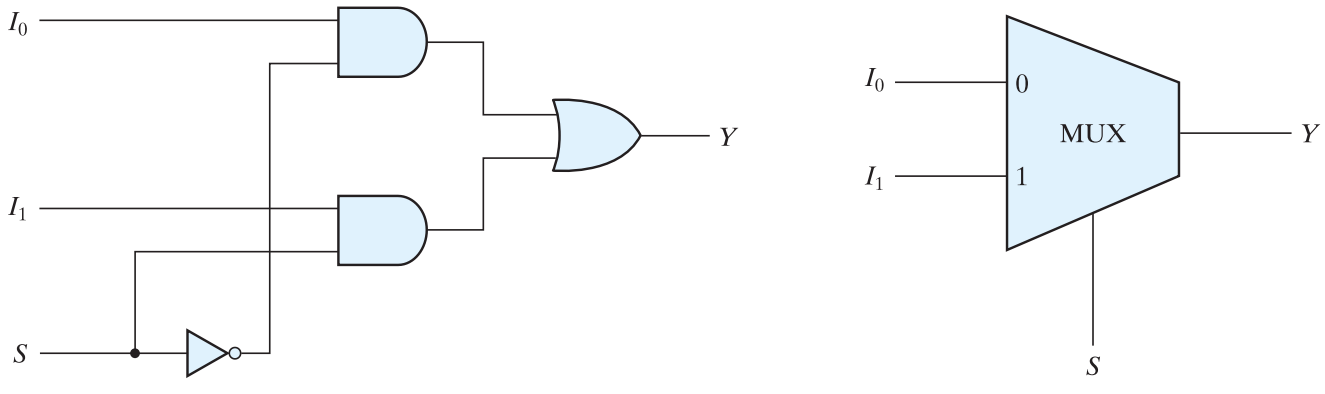
4.11 MULTIPLEXERS

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Fig. 4.24. The circuit has two data input lines, one output line, and one selection line S. When $S = 0$, the upper AND gate is enabled and I_0 has a path to the output. When $S = 1$, the lower AND gate is enabled and I_1 has a path to the output. The multiplexer acts like an electronic switch that selects one of two sources. The block diagram of a multiplexer is sometimes depicted by a wedge-shaped symbol, as shown in Fig. 4.24(b). It suggests visually how a selected one of multiple data sources is directed into a single destination. The multiplexer is often labeled "MUX" in block diagrams.

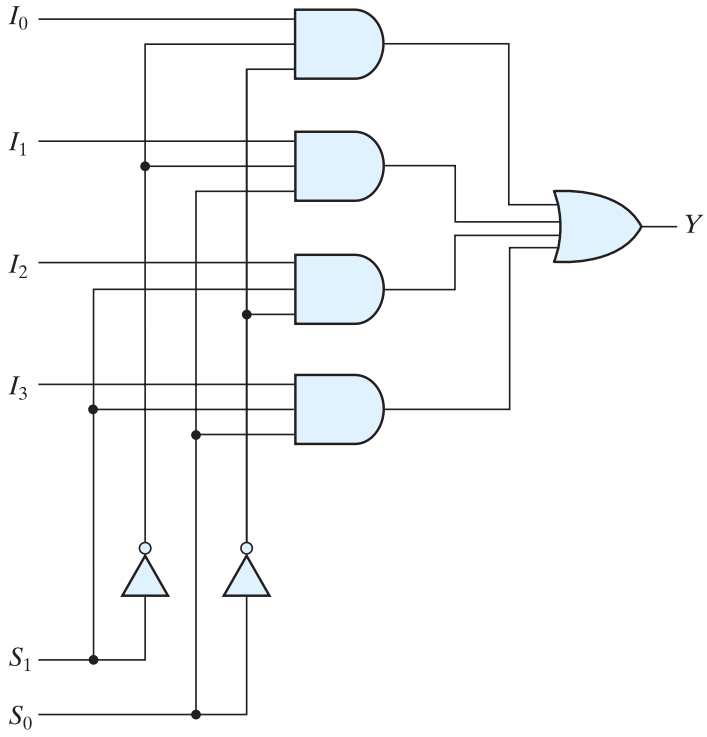
A four-to-one-line multiplexer is shown in Fig. 4.25. Each of the four inputs, I_0 through I_3, is applied to one input of an AND gate. Selection lines S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate that provides the one-line output. The function table lists the input that is passed to the output for each combination of the binary selection values. To demonstrate the operation of the circuit, consider the case when $S_1S_0 = 10$. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The output of the OR gate is now equal to the value of I_2, providing a path from the selected input to the output. A multiplexer is also called a *data selector*, since it selects one of many inputs and steers the binary information to the output line.

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed, they decode the selection input lines. In general, a 2^n-to-1-line multiplexer is constructed from an n-to-2^n decoder by adding 2^n input lines to it, one to each AND gate. The outputs of the AND gates are applied to a single OR gate. The size of a multiplexer is specified by



(a) Logic diagram (b) Block diagram

FIGURE 4.24
Two-to-one-line multiplexer



(a) Logic diagram

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

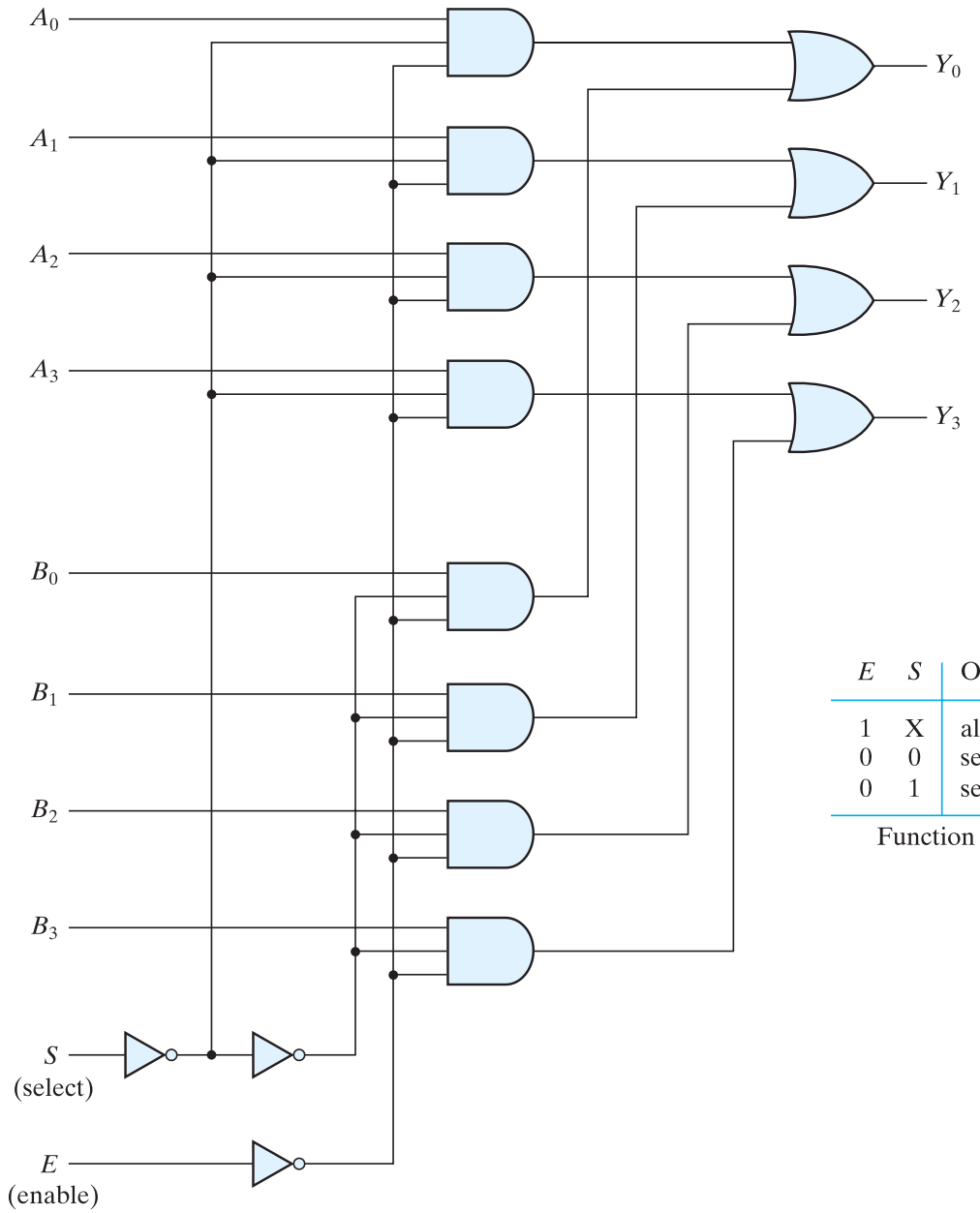
FIGURE 4.25
Four-to-one-line multiplexer

the number 2^n of its data input lines and the single output line. The n selection lines are implied from the 2^n data lines. As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer.

Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. As an illustration, a quadruple 2-to-1-line multiplexer is shown in Fig. 4.26. The circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either input A_0 or input B_0. Similarly, output Y_1 may have the value of A_1 or B_1, and so on. Input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active (i.e., asserted) for normal operation. Although the circuit contains four 2-to-1-line multiplexers, we are more likely to view it as a circuit that selects one of two 4-bit sets of data lines. As shown in the function table, the unit is enabled when $E = 0$. Then, if $S = 0$, the four A inputs have a path to the four outputs. If, by contrast, $S = 1$, the four B inputs are applied to the outputs. The outputs have all 0's when $E = 1$, regardless of the value of S.

Boolean Function Implementation

In Section 4.9, it was shown that a decoder can be used to implement Boolean functions by employing external OR gates. An examination of the logic diagram of a multiplexer reveals that it is essentially a decoder that includes the OR gate within the unit. The



E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Function table

FIGURE 4.26
Quadruple two-to-one-line multiplexer

minterms of a function are generated in a multiplexer by the circuit associated with the selection inputs. The individual minterms can be selected by the data inputs, thereby providing a method of implementing a Boolean function of n variables with a multiplexer that has n selection inputs and 2^n data inputs, one for each minterm.

We will now show a more efficient method for implementing a Boolean function of n variables with a multiplexer that has $n - 1$ selection inputs. The first $n - 1$ variables of the function are connected to the selection inputs of the multiplexer. The remaining single variable of the function is used for the data inputs. If the single variable is denoted

by z, each data input of the multiplexer will be z, z', 1, or 0. To demonstrate this procedure, consider the Boolean function

$$F(x, y, z) = \Sigma(1, 2, 6, 7)$$

This function of three variables can be implemented with a four-to-one-line multiplexer as shown in Fig. 4.27. The two variables x and y are applied to the selection lines in that order; x is connected to the S_1 input and y to the S_0 input. The values for the data input lines are determined from the truth table of the function. When $xy = 00$, output F is equal to z because $F = 0$ when $z = 0$ and $F = 1$ when $z = 1$. This requires that variable z be applied to data input 0. The operation of the multiplexer is such that when $xy = 00$, data input 0 has a path to the output, and that makes F equal to z. In a similar fashion, we can determine the required input to data lines 1, 2, and 3 from the value of F when $xy = 01, 10$, and 11, respectively. This particular example shows all four possibilities that can be obtained for the data inputs.

The general procedure for implementing any Boolean function of n variables with a multiplexer with $n - 1$ selection inputs and 2^{n-1} data inputs follows from the previous example. To begin with, Boolean function is listed in a truth table. Then first $n - 1$ variables in the table are applied to the selection inputs of the multiplexer. For each combination of the selection variables, we evaluate the output as a function of the last variable. This function can be 0, 1, the variable, or the complement of the variable. These values are then applied to the data inputs in the proper order.

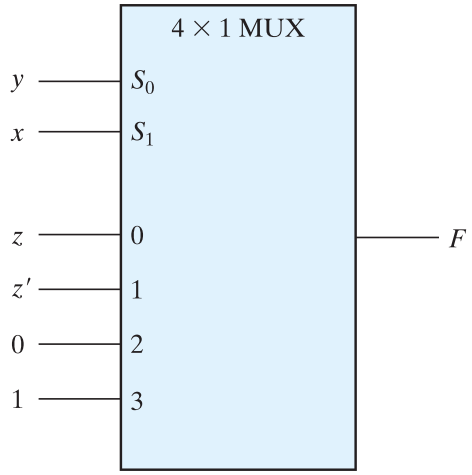
As a second example, consider the implementation of the Boolean function

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

This function is implemented with a multiplexer with three selection inputs as shown in Fig. 4.28. Note that the first variable A must be connected to selection input S_2 so that A, B, and C correspond to selection inputs S_2, S_1, and S_0, respectively. The values for the

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

FIGURE 4.27
Implementing a Boolean function with a multiplexer

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

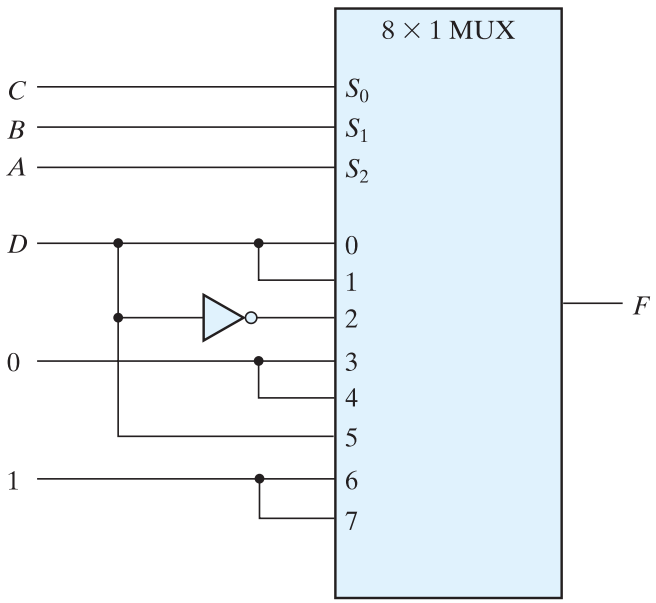


FIGURE 4.28
Implementing a four-input function with a multiplexer

data inputs are determined from the truth table listed in the figure. The corresponding data line number is determined from the binary combination of ABC. For example, the table shows that when $ABC = 101$, $F = D$, so the input variable D is applied to data input 5. The binary constants 0 and 1 correspond to two fixed signal values. When integrated circuits are used, logic 0 corresponds to signal ground and logic 1 is equivalent to the power signal, depending on the technology (e.g., 3 V).

Three-State Gates

A multiplexer can be constructed with three-state gates—digital circuits that exhibit three states. Two of the states are signals equivalent to logic 1 and logic 0 as in a conventional gate. The third state is a *high-impedance* state in which (1) the logic behaves like an open circuit, which means that the output appears to be disconnected, (2) the circuit has no logic significance, and (3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used is the buffer gate.

The graphic symbol for a three-state buffer gate is shown in Fig. 4.29. It is distinguished from a normal buffer by an input control line entering the bottom of the symbol. The buffer has a normal input, an output, and a control input that determines the state of the output. When the control input is equal to 1, the output is enabled and the gate behaves like a conventional buffer, with the output equal to the normal input. When the control

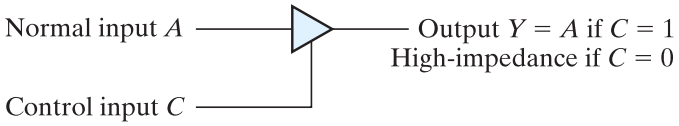
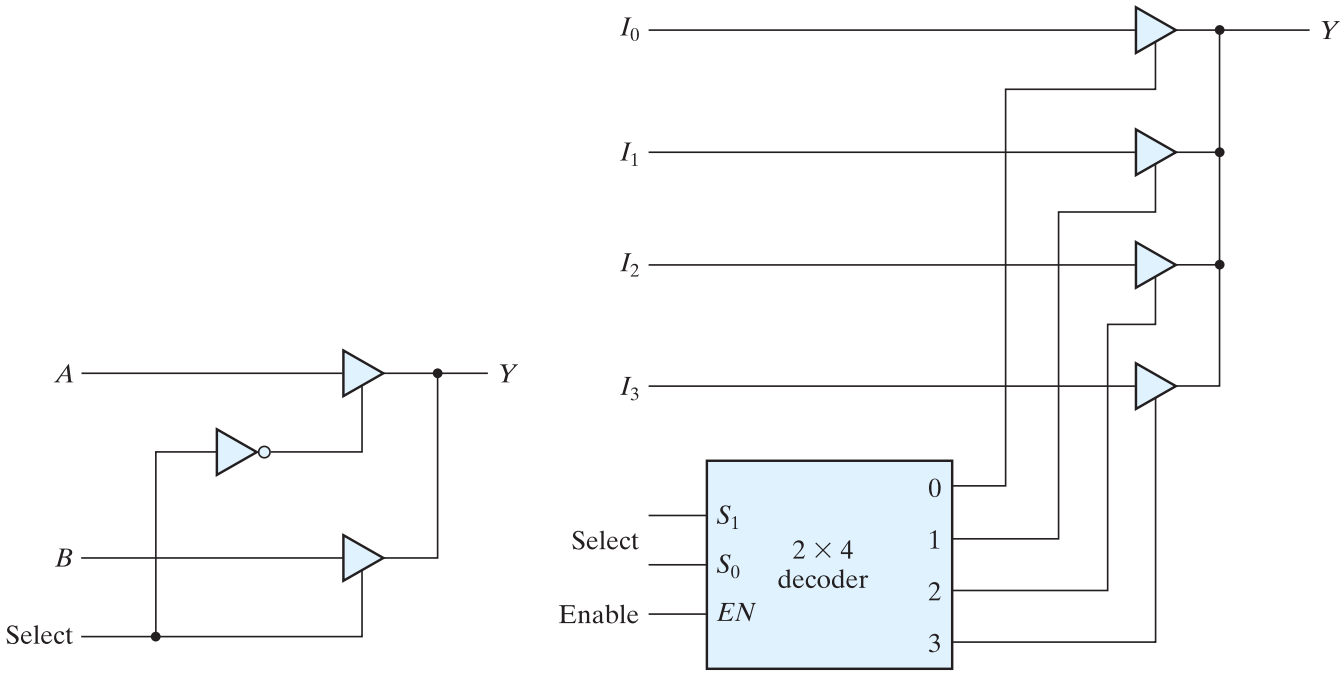


FIGURE 4.29
Graphic symbol for a three-state buffer

input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common line without endangering loading effects.

The construction of multiplexers with three-state buffers is demonstrated in Fig. 4.30. Figure 4.30(a) shows the construction of a two-to-one-line multiplexer with 2 three-state buffers and an inverter. The two outputs are connected together to form a single output line. (Note that this type of connection cannot be made with gates that do not have three-state outputs.) When the select input is 0, the upper buffer is enabled by its control input and the lower buffer is disabled. Output Y is then equal to input A. When the select input is 1, the lower buffer is enabled and Y is equal to B.

The construction of a four-to-one-line multiplexer is shown in Fig. 4.30(b). The outputs of 4 three-state buffers are connected together to form a single output line. The control inputs to the buffers determine which one of the four normal inputs I_0 through



(a) 2-to-1-line mux

(b) 4-to-1-line mux

FIGURE 4.30
Multiplexers with three-state gates

I_3 will be connected to the output line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only 1 three-state buffer has access to the output while all other buffers are maintained in a high-impedance state. One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0 and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation reveals that this circuit is another way of constructing a four-to-one-line multiplexer.

4.12 HDL MODELS OF COMBINATIONAL CIRCUITS

The Verilog HDL was introduced in Section 3.10. In the current section, we introduce additional features of Verilog, present more elaborate examples, and compare alternative descriptions of combinational circuits in Verilog. Sequential circuits are presented in Chapter 5. As mentioned previously, the module is the basic building block for modeling hardware with the Verilog HDL. The logic of a module can be described in any one (or a combination) of the following modeling styles:

- Gate-level modeling using instantiations of predefined and user-defined primitive gates.
- Dataflow modeling using continuous assignment statements with the keyword **assign**.
- Behavioral modeling using procedural assignment statements with the keyword **always**.

Gate-level (structural) modeling describes a circuit by specifying its gates and how they are connected with each other. Dataflow modeling is used mostly for describing the Boolean equations of combinational logic. We'll also consider here behavioral modeling that is used to describe combinational and sequential circuits at a higher level of abstraction. Combinational logic can be designed with truth tables, Boolean equations, and schematics; Verilog has a construct corresponding to each of these "classical" approaches to design: user-defined primitives, continuous assignments, and primitives, as shown in Fig. 4.31. There is one other modeling style, called switch-level modeling. It is sometimes used in the simulation of MOS transistor circuit models, but not in logic synthesis. We will not consider switch-level modeling.

Gate-Level Modeling

Gate-level modeling was introduced in Section 3.10 with a simple example. In this type of representation, a circuit is specified by its logic gates and their interconnections. Gate-level modeling provides a textual description of a schematic diagram. The Verilog HDL

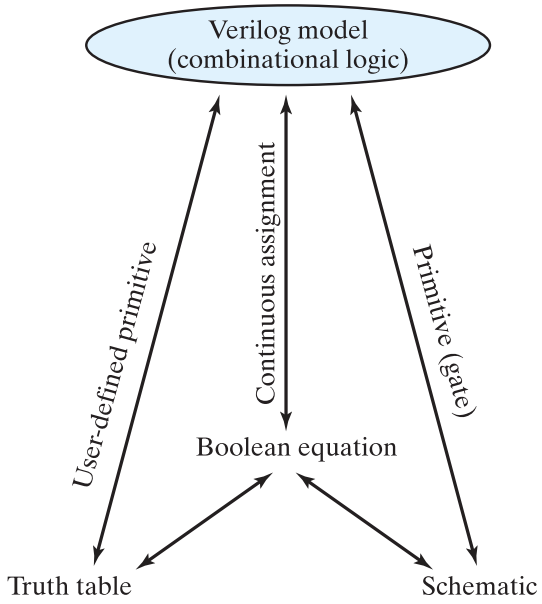


FIGURE 4.31
Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type. The other eight are the same as the ones listed in Section 2.8. They are all declared with the lowercase keywords **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, and **buf**. Primitives such as **and** are *n*-input primitives. They can have any number of scalar inputs (e.g., a three-input **and** primitive). The **buf** and **not** primitives are *n*-output primitives. A single input can drive multiple output lines distinguished by their identifiers.

The Verilog language includes a functional description of each type of gate, too. The logic of each gate is based on a four-valued system. When the gates are simulated, the simulator assigns one value to the output of each gate at any instant. In addition to the two logic values of 0 and 1, there are two other values: *unknown* and *high impedance*. An unknown value is denoted by **x** and a high impedance by **z**. An unknown value is assigned during simulation when the logic value of a signal is ambiguous—for instance, if it cannot be determined whether its value is 0 or 1 (e.g., a flip-flop without a reset condition). A high-impedance condition occurs at the output of three-state gates that are not enabled or if a wire is inadvertently left unconnected. The four-valued logic truth tables for the **and**, **or**, **xor**, and **not** primitives are shown in Table 4.9. The truth table for the other four gates is the same, except that the outputs are complemented. Note that for the **and** gate, the output is 1 only when both inputs are 1 and the output is 0 if any input is 0. Otherwise, if one input is **x** or **z**, the output is **x**. The output of the **or** gate is 0 if both inputs are 0, is 1 if any input is 1, and is **x** otherwise.

When a primitive gate is listed in a module, we say that it is *instantiated* in the module. In general, component instantiations are statements that reference lower level components in the design, essentially creating unique copies (or *instances*) of those components in the higher level module. Thus, a module that uses a gate in its description is said to

Table 4.9
Truth Table for Predefined Primitive Gates

and	0	1	**x**	**z**
0	0	0	0	0
1	0	1	**x**	**x**
x	0	**x**	**x**	**x**
z	0	**x**	**x**	**x**

or	0	1	**x**	**z**
0	0	1	**x**	**x**
1	1	1	1	1
x	**x**	1	**x**	**x**
z	**x**	1	**x**	**x**

xor	0	1	**x**	**z**
0	0	1	**x**	**x**
1	1	0	**x**	**x**
x	**x**	**x**	**x**	**x**
z	**x**	**x**	**x**	**x**

not	input	output
	0	1
	1	0
	x	**x**
	z	**x**

instantiate the gate. Think of instantiation as the HDL counterpart of placing and connecting parts on a circuit board.

We now present two examples of gate-level modeling. Both examples use identifiers having multiple bit widths, called *vectors*. The syntax specifying a vector includes within square brackets two numbers separated with a colon. The following Verilog statements specify two vectors:

```
output [0: 3] D;
wire   [7: 0] SUM;
```

The first statement declares an output vector D with four bits, 0 through 3. The second declares a wire vector *SUM* with eight bits numbered 7 through 0. (*Note*: The first (left-most) number (array index) listed is always the most significant bit of the vector.) The individual bits are specified within square brackets, so *D[2]* specifies bit 2 of D. It is also possible to address parts (contiguous bits) of vectors. For example, *SUM[2: 0]* specifies the three least significant bits of vector *SUM*.

HDL Example 4.1 shows the gate-level description of a two-to-four-line decoder. (See Fig. 4.19.) This decoder has two data inputs A and B and an enable input E. The four outputs are specified with the vector D. The **wire** declaration is for internal connections. Three **not** gates produce the complement of the inputs, and four **nand** gates provide the outputs for D. Remember that *the output is always listed first in the port list of a primitive*, followed by the inputs. This example describes the decoder of Fig. 4.19 and follows the procedures established in Section 3.10. Note that the keywords **not** and **nand** are written only once and do not have to be repeated for each gate, but commas must be inserted at the end of each of the gates in the series, except for the last statement, which must be terminated with a semicolon.

HDL Example 4.1 (Two-to-Four-Line Decoder)

```
// Gate-level description of two-to-four-line decoder
// Refer to Fig. 4.19 with symbol E replaced by enable, for clarity.

module decoder_2x4_gates (D, A, B, enable);
 output      [0: 3]          D;
 input                       A, B;
 input                       enable;
 wire                        A_not,B_not, enable_not;

 not
   G1  (A_not, A),
   G2  (B_not, B),
   G3  (enable_not, enable);
 nand
   G4  (D[0], A_not, B_not, enable_not),
   G5  (D[1], A_not, B, enable_not),
   G6  (D[2], A, B_not, enable_not),
   G7  (D[3], A, B, enable_not);
endmodule
```

Two or more modules can be combined to build a hierarchical description of a design. There are two basic types of design methodologies: top down and bottom up. In a *top-down* design, the top-level block is defined and then the subblocks necessary to build the top-level block are identified. In a *bottom-up* design, the building blocks are first identified and then combined to build the top-level block. Take, for example, the binary adder of Fig. 4.9. It can be considered as a top-block component built with four full-adder blocks, while each full adder is built with two half-adder blocks. In a top-down design, the four-bit adder is defined first, and then the two adders are described. In a bottom-up design, the half adder is defined, then each full adder is constructed, and then the four-bit adder is built from the full adders.

A bottom-up hierarchical description of a four-bit adder is shown in HDL Example 4.2. The half adder is defined by instantiating primitive gates. The next module describes the full adder by instantiating and connecting two half adders. The third module describes the four-bit adder by instantiating and connecting four full adders. Note that the first character of an identifier cannot be a number, but can be an underscore, so the module name *_4bitadder* is valid. An alternative name that is meaningful, but does not require a leading underscore, is *adder_4_bit*. The instantiation is done by using the name of the module that is instantiated together with a new (or the same) set of port names. For example, the half adder *HA1* inside the full adder module is instantiated with ports *S1*, *C1*, *x,* and *y*. This produces a half adder with outputs *S1* and *C1* and inputs *x* and *y*.

HDL Example 4.2 (Ripple-Carry Adder)

```
// Gate-level description of four-bit ripple carry adder
// Description of half adder (Fig. 4.5b)

// module half_adder (S, C, x, y);                    // Verilog 1995 syntax
// output  S, C;
// input   x, y;

module half_adder (output S, C, input x, y);          // Verilog 2001, 2005 syntax
// Instantiate primitive gates
  xor (S, x, y);
  and (C, x, y);
endmodule

// Description of full adder (Fig. 4.8)               // Verilog 1995 syntax
// module full_adder (S, C, x, y, z);
// output          S, C;
// input           x, y, z;

module full_adder (output S, C, input x, y, z);       // Verilog 2001, 2005 syntax
  wire S1, C1, C2;

// Instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule

// Description of four-bit adder (Fig. 4.9)       // Verilog 1995 syntax
// module ripple_carry_4_bit_adder (Sum, C4, A, B, C0);
// output [3: 0]   Sum;
// output          C4;
// input  [3: 0]   A, B;
// input           C0;
// Alternative Verilog 2001, 2005 syntax:

module ripple_carry_4_bit_adder ( output [3: 0] Sum, output C4,
  input [3: 0] A, B, input C0);
  wire          C1, C2, C3;          // Intermediate carries
// Instantiate chain of full adders
full_adder      FA0 (Sum[0], C1, A[0], B[0], C0),
                FA1 (Sum[1], C2, A[1], B[1], C1),
                FA2 (Sum[2], C3, A[2], B[2], C2),
                FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```

HDL Example 4.2 illustrates Verilog 2001, 2005 syntax, which eliminates extra typing of identifiers declaring the mode (e.g., **output**), type (**reg**), and declaration of a vector range (e.g., [3: 0]) of a port. The first version of the standard (1995) uses separate statements for these declarations.

Note that modules can be instantiated (nested) within other modules, but module declarations cannot be nested; that is, a module definition (declaration) cannot be placed within another module declaration. In other words, a module definition cannot be inserted into the text between the **module** and **endmodule** keywords of another module. The only way one module definition can be incorporated into another module is by instantiating it. Instantiating modules within other modules creates a hierarchical decomposition of a design. A description of a module is said to be a *structural* description if it is composed of instantiations of other modules. Note also that *instance names* must be specified when defined modules are instantiated (such as *FA0* for the first full adder in the third module), but using a name is optional when instantiating primitive gates. Module *ripple_carry_4_bit_adder* is composed of instantiated and interconnected full adders, each of which is itself composed of half adders and some *glue logic*. The top level, or parent module, of the design hierarchy is the module *ripple_carry_4_bit_adder*. Four copies of *full_adder* are its child modules, etc. *C0* is an input of the cell forming the least significant bit of the chain, and *C4* is the output of the cell forming the most significant bit.

Three-State Gates

As mentioned in Section 4.11, a three-state gate has a control input that can place the gate into a high-impedance state. The high-impedance state is symbolized by **z** in Verilog. There are four types of three-state gates, as shown in Fig. 4.32. The **bufif1** gate behaves like a normal buffer if *control* = 1. The output goes to a high-impedance state **z** when *control* = 0. The **bufif0** gate behaves in a similar fashion, except that the high-impedance state occurs when *control* = 1. The two **notif** gates operate in a similar manner, except that the output is the complement of the input when the gate is not in a high-impedance state. The gates are instantiated with the statement

$$gate\,name\,(output, input, control);$$

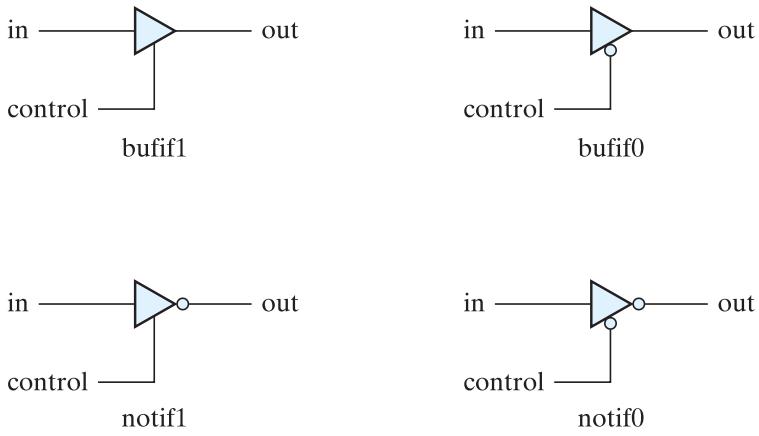


FIGURE 4.32
Three-state gates

The gate name can be that of any 1 of the 4 three-state gates. In simulation, the output can result in 0, 1, **x,** or **z.** Two examples of gate instantiation are

```
bufif1   (OUT, A, control);
notif0   (Y, B, enable);
```

In the first example, input A is transferred to OUT when $control = 1$. OUT goes to **z** when $control = 0$. In the second example, output $Y =$ **z** when $enable = 1$ and output $Y = B'$ when $enable = 0$.

The outputs of three-state gates can be connected together to form a common output line. To identify such a connection, Verilog HDL uses the keyword **tri** (for tristate) to indicate that the output has multiple drivers. As an example, consider the two-to-one-line multiplexer with three-state gates shown in Fig. 4.33.

The HDL description must use a **tri** data type for the output:

```
// Mux with three-state output

module mux_tri (m_out, A, B, select);
  output  m_out;
  input   A, B, select;
  tri     m_out;

  bufif1 (m_out, A, select);
  bufif0 (m_out, B, select);
endmodule
```

The 2 three-state buffers have the same output. In order to show that they have a common connection, it is necessary to declare *m_out* with the keyword **tri.**

Keywords **wire** and **tri** are examples of a set of data types called *nets*, which represent connections between hardware elements. In simulation, their value is determined by a continuous assignment statement or by the device whose output they represent. The word *net* is not a keyword, but represents a class of data types, such as **wire, wor, wand, tri, supply1,** and **supply0.** The **wire** declaration is used most frequently. In fact, if an identifier is used, but not declared, the language specifies that it will be interpreted (by default) as a **wire**. The net **wor** models the hardware implementation of the wired-OR configuration (emitter-coupled logic). The **wand** models the wired-AND configuration (open-collector technology; see Fig. 3.26). The nets **supply1** and **supply0** represent power supply and ground, respectively. They are used to hardwire an input of a device to either 1 or 0.

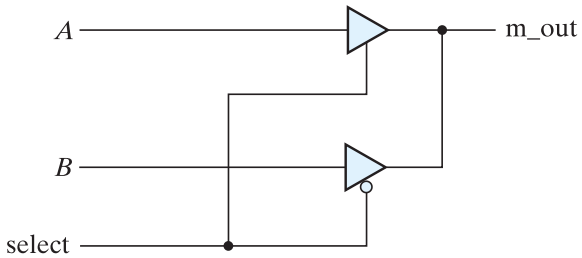


FIGURE 4.33
Two-to-one-line multiplexer with three-state buffers

Dataflow Modeling

Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result. Verilog HDL provides about 30 different operators. Table 4.10 lists some of these operators, their symbols, and the operation that they perform. (A complete list of operators supported by Verilog 2001, 2005 can be found in Table 8.1 in Section 8.2.) It is necessary to distinguish between arithmetic and logic operations, so different symbols are used for each. The plus symbol (+) indicates the arithmetic operation of addition; the bitwise logic AND operation (conjunction) uses the symbol &. There are special symbols for bitwise logical OR (disjunction), NOT, and XOR. The equality symbol uses two equals signs (without spaces between them) to distinguish it from the equals sign used with the **assign** statement. The bitwise operators operate bit by bit on a pair of vector operands to produce a vector result. The concatenation operator provides a mechanism for appending multiple operands. For example, two operands with two bits each can be concatenated to form an operand with four bits. The conditional operator acts like a multiplexer and is explained later, in conjunction with HDL Example 4.6.

It should be noted that a bitwise operator (e.g., ~) and its corresponding logical operator (e.g., !) may produce different results, depending on their operand. If the operands are scalar the results will be identical; if the operands are vectors the result will not necessarily match. For example, ~(1010) is (0101), and !(1010) is 0. A binary value is considered to be logically true if it is not 0. In general, use the bitwise operators to describe arithmetic operations and the logical operators to describe logical operations.

Dataflow modeling uses continuous assignments and the keyword **assign.** A continuous assignment is a statement that assigns a value to a net. The data type family *net* is used in Verilog HDL to represent a physical connection between circuit elements. A net

Table 4.10
Some Verilog HDL Operators

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

is declared explicitly by a net keyword (e.g., **wire**) or by declaring an identifier to be an input port. The logic value associated with a net is determined by what the net is connected to. If the net is connected to an output of a gate, the net is said to be *driven* by the gate, and the logic value of the net is determined by the logic values of the inputs to the gate and the truth table of the gate. If the identifier of a net is the left-hand side of a continuous assignment statement or a procedural assignment statement, the value assigned to the net is specified by a Boolean expression that uses operands and operators. As an example, assuming that the variables were declared, a two-to-one-line multiplexer with scalar data inputs A and B, select input S, and output Y is described with the continuous assignment

assign Y = (A && S) || (B && S)

The relationship between Y, A, B, and S is declared by the keyword **assign,** followed by the target output Y and an equals sign. Following the equals sign is a Boolean expression. In hardware terms, this assignment would be equivalent to connecting the output of the OR gate to wire Y.

The next two examples show the dataflow models of the two previous gate-level examples. The dataflow description of a two-to-four-line decoder with active-low output enable and inverted output is shown in HDL Example 4.3. The circuit is defined with four continuous assignment statements using Boolean expressions, one for each output. The dataflow description of the four-bit adder is shown in HDL Example 4.4. The addition logic is described by a single statement using the operators of addition and concatenation. The plus symbol (+) specifies the binary addition of the four bits of A with the four bits of B and the one bit of *C_in*. The target output is the *concatenation* of the output carry *C_out* and the four bits of *Sum*. Concatenation of operands is expressed within braces and a comma separating the operands. Thus, *{C_out, Sum}* represents the five-bit result of the addition operation.

HDL Example 4.3 (Dataflow: Two-to-Four Line Decoder)

```
// Dataflow description of two-to-four-line decoder

// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

module decoder_2x4_df (                    // Verilog 2001, 2005 syntax
  output    [0: 3]      D,
  input                 A, B,
                        enable
);
  assign    D[0] = !((!A) && (!B) && (!enable)),
            D[1] = !(*!A) && B && (!enable)),
            D[2] = !(A && B && (!enable)
            D[3] = !(A && B && (!enable))
endmodule
```

HDL Example 4.4 (Dataflow: Four-Bit Adder)

```
// Dataflow description of four-bit adder

// Verilog 2001, 2005 module port syntax

module binary_adder (
  output [3: 0]          Sum,
  output                 C_out,
  input [3: 0]           A, B,
  input                  C_in
);

  assign {C_out, Sum} = A + B + C_in;
endmodule
```

Dataflow HDL models describe combinational circuits by their *function* rather than by their gate structure. To show how dataflow descriptions facilitate digital design, consider the 4-bit magnitude comparator described in HDL Example 4.5. The module specifies two 4-bit inputs A and B and three outputs. One output (A_lt_B) is logic 1 if A is less than B, a second output (A_gt_B) is logic 1 if A is greater than B, and a third output (A_eq_B) is logic 1 if A is equal to B. Note that equality (identity) is symbolized with two equals signs (= =) to distinguish the operation from that of the assignment operator (=). A Verilog HDL synthesis compiler can accept this module description as input, execute synthesis algorithms, and provide an output netlist and a schematic of a circuit equivalent to the one in Fig. 4.17, all without manual intervention! The designer need not draw the schematic.

HDL Example 4.5 (Dataflow: Four-Bit Comparator)

```
// Dataflow description of a four-bit comparator //V2001, 2005 syntax

module mag_compare
( output                 A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]           A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A = = B);
endmodule
```

The next example uses the conditional operator (? :). This operator takes three operands:

condition ? true-expression : false-expression;

The condition is evaluated. If the result is logic 1, the true expression is evaluated and used to assign a value to the left-hand side of an assignment statement. If the result is

logic 0, the false expression is evaluated. The two conditions together are equivalent to an if–else condition. HDL Example 4.6 describes a two-to-one-line multiplexer using the conditional operator. The continuous assignment

```
assign OUT = select ? A : B;
```

specifies the condition that $OUT = A$ if $select = 1$, else $OUT = B$ if $select = 0$.

HDL Example 4.6 (Dataflow: Two-to-One Multiplexer)

```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output    m_out;
  input     A, B;
  input     select;

  assign m_out = (select)? A : B;
endmodule
```

Behavioral Modeling

Behavioral modeling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits. Here, we give two simple combinational circuit examples to introduce the subject. Behavioral modeling is presented in more detail in Section 5.6, after the study of sequential circuits.

Behavioral descriptions use the keyword **always**, followed by an optional event control expression and a list of procedural assignment statements. The event control expression specifies when the statements will execute. The target output of a procedural assignment statement must be of the **reg** data type. Contrary to the **wire** data type, whereby the target output of an assignment may be continuously updated, a **reg** data type retains its value until a new value is assigned.

HDL Example 4.7 shows the behavioral description of a two-to-one-line multiplexer. (Compare it with HDL Example 4.6.) Since variable *m_out* is a target output, it must be declared as **reg** data (in addition to the **output** declaration). The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variables listed after the @ symbol. (Note that there is no semicolon (;) at the end of the **always** statement.) In this case, these variables are the input variables *A*, *B*, and *select*. The statements execute if *A*, *B*, or *select* changes value. Note that the keyword **or**, instead of the bitwise logical OR operator "|", is used between variables. The conditional statement **if–else** provides a decision based upon the value of the *select* input. The **if** statement can be written without the equality symbol:

```
if (select) OUT = A;
```

The statement implies that *select* is checked for logic 1.

HDL Example 4.7 (Behavioral: Two-to-One Line Multiplexer)

```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);
  output      m_out;
  input       A, B, select;
  reg         m_out;

  always      @(A or B or select)
   if (select == 1) m_out = A;
   else m_out 5 B;
endmodule
```

HDL Example 4.8 describes the function of a four-to-one-line multiplexer. The *select* input is defined as a two-bit vector, and output *y* is declared to have type **reg**. The **always** statement, in this example, has a sequential block enclosed between the keywords **case** and **endcase**. The block is executed whenever any of the inputs listed after the **@** symbol changes in value. The **case** statement is a multiway conditional branch construct. Whenever *in_0, in_1, in_2, in_3* or *select* change, the case expression (*select*) is evaluated and its value compared, from top to bottom, with the values in the list of statements that follow, the so-called **case** items. The statement associated with the first **case** item that matches the **case** expression is executed. In the absence of a match, no statement is executed. Since *select* is a two-bit number, it can be equal to 00, 01, 10, or 11. The **case** items have an implied priority because the list is evaluated from top to bottom.

The list is called a *sensitivity list* (Verilog 2001, 2005) and is equivalent to the *event control expression* (Verilog 1995) formed by "ORing" the signals. Combinational logic is reactive—when an input changes an output may change.

HDL Example 4.8 (Behavioral: Four-to-One Line Multiplexer)

```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg m_out,
  input       in_0, in_1, in_2, in_3,
  input [1: 0] select
);
always @ (in_0, in_1, in_2, in_3, select)      // Verilog 2001, 2005 syntax
  case (select)
    2'b00:          m_out = in_0;
    2'b01:          m_out = in_1;
    2'b10:          m_out = in_2;
    2'b11:          m_out = in_3;
  endcase
endmodule
```

Binary numbers in Verilog are specified and interpreted with the letter **b** preceded by a prime. The size of the number is written first and then its value. Thus, 2′*b*01 specifies a two-bit binary number whose value is 01. Numbers are stored as a bit pattern in memory, but they can be referenced in decimal, octal, or hexadecimal formats with the letters ***d′ o′,*** and ***h′,*** respectively. For example, 4′HA = 4′d10 = 4′b1010 and have the same internal representation in a simulator. If the base of the number is not specified, its interpretation defaults to decimal. If the size of the number is not specified, the system assumes that the size of the number is at least 32 bits; if a host simulator has a larger word length—say, 64 bits—the language will use that value to store unsized numbers. The integer data type (keyword **integer**) is stored in a 32-bit representation. The underscore (_) may be inserted in a number to improve readability of the code (e.g., 16′b0101_1110_0101_0011). It has no other effect.

The **case** construct has two important variations: **casex** and **casez**. The first will treat as don't-cares any bits of the **case** expression or the **case** item that have logic value **x** or **z**. The **casez** construct treats as don't-cares only the logic value **z**, for the purpose of detecting a match between the **case** expression and a **case** item.

The list of case items need not be complete. If the list of **case** items does not include all possible bit patterns of the **case** expression, no match can be detected. Unlisted **case** items, i.e., bit patterns that are not explicitly decoded can be treated by using the **default** keyword as the last item in the list of **case** items. The associated statement will execute when no other match is found. This feature is useful, for example, when there are more possible state codes in a sequential machine than are actually used. Having a **default** case item lets the designer map all of the unused states to a desired next state without having to elaborate each individual state, rather than allowing the synthesis tool to arbitrarily assign the next state.

The examples of behavioral descriptions of combinational circuits shown here are simple ones. Behavioral modeling and procedural assignment statements require knowledge of sequential circuits and are covered in more detail in Section 5.6.

Writing a Simple Test Bench

A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit in order to test it and observe its response during simulation. Test benches can be quite complex and lengthy and may take longer to develop than the design that is tested. The results of a test are only as good as the test bench that is used to test a circuit. Care must be taken to write stimuli that will test a circuit thoroughly, exercising all of the operating features that are specified. However, the test benches considered here are relatively simple, since the circuits we want to test implement only combinational logic. The examples are presented to demonstrate some basic features of HDL stimulus modules. Chapter 8 considers test benches in greater depth.

In addition to employing the **always** statement, test benches use the **initial** statement to provide a stimulus to the circuit being tested. We use the term "**always** statement" loosely. Actually, **always** is a Verilog language construct specifying *how* the associated statement is to execute (subject to the event control expression). The **always** statement

executes repeatedly in a loop. The **initial** statement executes only once, starting from simulation time 0, and may continue with any operations that are delayed by a given number of time units, as specified by the symbol #. For example, consider the **initial** block

```
initial
  begin
        A = 0; B = 0;
    #10 A = 1;
    #20 A = 0; B = 1;
  end
```

The block is enclosed between the keywords **begin** and **end.** At time 0, *A* and *B* are set to 0. Ten time units later, *A* is changed to 1. Twenty time units after that (at $t = 30$), *A* is changed to 0 and *B* to 1. Inputs specified by a three-bit truth table can be generated with the **initial** block:

```
initial
  begin
    D = 3'b000;
    repeat (7)
    #10 D = D + 3'b001;
  end
```

When the simulator runs, the three-bit vector *D* is initialized to 000 at time = 0. The keyword **repeat** specifies a looping statement: *D* is incremented by 1 seven times, once every 10 time units. The result is a sequence of binary numbers from 000 to 111.

A stimulus module has the following form:

```
module test_module_name;
  // Declare local reg and wire identifiers.
  // Instantiate the design module under test.
  // Specify a stopwatch, using $finish to terminate the simulation.
  // Generate stimulus, using initial and always statements.
  // Display the output response (text or graphics (or both)).
endmodule
```

A test module is written like any other module, but it typically has no inputs or outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local **reg** data type. The outputs of the design module that are displayed for testing are declared in the stimulus module as local **wire** data type. The module under test is then instantiated, using the local identifiers in its port list. Figure 4.34 clarifies this relationship. The stimulus module generates inputs for the design module by declaring local identifiers *t_A* and *t_B* as **reg** type and checks the output of the design unit with the **wire** identifier *t_C*. The local identifiers are then used to instantiate the design module being tested. The simulator associates the (actual) local identifiers within the test bench, *t_A*, *t_B*, and *t_C*, with the formal identifiers of the

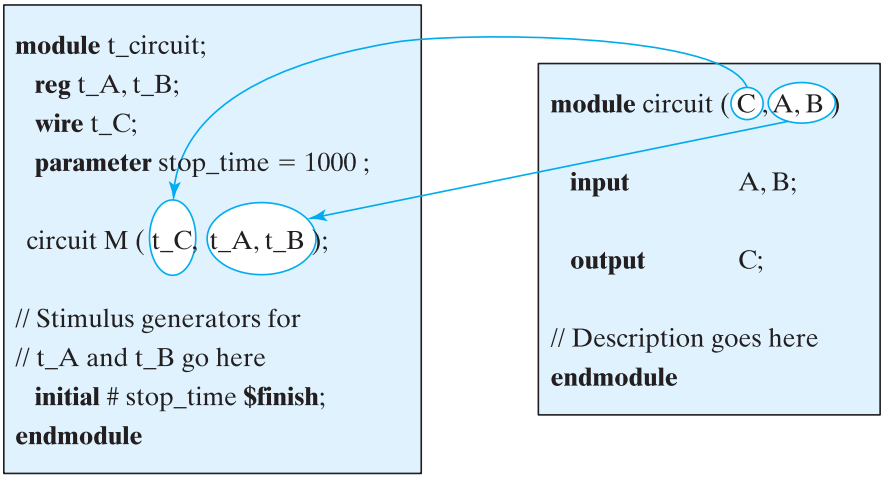


FIGURE 4.34
Interaction between stimulus and design modules

module (*A, B, C*). The association shown here is based on *position* in the port list, which is adequate for the examples that we will consider. The reader should note, however, that Verilog provides a more flexible *name association* mechanism for connecting ports in larger circuits.

The response to the stimulus generated by the **initial** and **always** blocks will appear in text format as standard output and as waveforms (timing diagrams) in simulators having graphical output capability. Numerical outputs are displayed by using Verilog *system tasks*. These are built-in system functions that are recognized by keywords that begin with the symbol **$.** Some of the system tasks that are useful for display are

$display—display a one-time value of variables or strings with an end-of-line return,

$write**—same as **$display, but without going to next line,

$monitor—display variables whenever a value changes during a simulation run,

$time—display the simulation time,

$finish—terminate the simulation.

The syntax for **$display, $write,** and **$monitor** is of the form

Task-name (*format specification, argumentlist*);

The format specification uses the symbol % to specify the radix of the numbers that are displayed and may have a string enclosed in quotes (″). The base may be binary, decimal, hexadecimal, or octal, identified with the symbols %b, %d, %h, and %o, respectively (%B, %D, %H, and %O are valid too). For example, the statement

```
$display ("%d %b %b", C, A, B);
```

specifies the display of *C* in decimal and of *A* and *B* in binary. Note that there are no commas in the format specification, that the format specification and argument list

are separated by a comma, and that the argument list has commas between the variables. An example that specifies a string enclosed in quotes may look like the statement

$display ("time = %0d A = %b", **$time,** A, B);

and will produce the display

time = 3 A = 10 B = 1

where (*time* =), (*A* =), and (*B* =) are part of the string to be displayed. The format specifiers %0d, %b, and %b specify the base for **$time,** *A*, and *B*, respectively. In displaying time values, it is better to use the format %0d instead of %d. This provides a display of the significant digits without the leading spaces that %d will include. (%d will display about 10 leading spaces because time is calculated as a 32-bit number.)

An example of a stimulus module is shown in HDL Example 4.9. The circuit to be tested is the two-to-one-line multiplexer described in Example 4.6. The module *t_mux_2x1_df* has no ports. The inputs for the mux are declared with a **reg** keyword and the outputs with a **wire** keyword. The mux is instantiated with the local variables. The **initial** block specifies a sequence of binary values to be applied during the simulation. The output response is checked with the **$monitor** system task. Every time a variable in its argument changes value, the simulator displays the inputs, output, and time. The result of the simulation is listed under the simulation log in the example. It shows that *m_out* = *A* when *select* = 1 and *m_out* = *B* when *select* = 0 verifying the operation of the multiplexer.

HDL Example 4.9 (Test Bench)

```
// Test bench with stimulus for mux_2x1_df

module t_mux_2x1_df;
  wire          t_mux_out;
  reg           t_A, t_B;
  reg           t_select;
  parameter stop_time = 50;

mux_2x1_df M1 (t_mux_out, t_A, t_B, t_select);        // Instantiation of circuit to be tested

initial # stop_time $finish;

  initial begin                                       // Stimulus generator
          t_select = 1; t_A = 0; t_B = 1;
    #10 t_A = 1; t_B = 0;
    #10 t_select = 0;
    #10 t_A = 0; t_B = 1;
  end

  initial begin                                       // Response monitor
   // $display (″  time  Select  A  B  m_out″);
   // $monitor ($time,, ″ %b  %b  %b  %b ″, t_select, t_A, t_B, t_m_out);
```

```
    $monitor ("time =", $time,, "select = %b A = %b B = %b OUT = %b",
    t_select, t_A, t_B, t_mux_out);
  end
endmodule

// Dataflow description of two-to-one-line multiplexer

// from Example 4.6
module mux_2x1_df (m_out, A, B, select);
  output      m_out;
  input       A, B;
  input       select;

  assign m_out = (select)? A : B;
endmodule
```

Simulation log:

```
select = 1 A = 0 B = 1 OUT = 0 time = 0
select = 1 A = 1 B = 0 OUT = 1 time = 10
select = 0 A = 1 B = 0 OUT = 0 time = 20
select = 0 A = 0 B = 1 OUT = 1 time = 30
```

Logic simulation is a fast and accurate method of verifying that a model of a combinational circuit is correct. There are two types of verification: functional and timing. In *functional* verification, we study the circuit logical operation independently of timing considerations. This can be done by deriving the truth table of the combinational circuit. In *timing* verification, we study the circuit's operation by including the effect of delays through the gates. This can be done by observing the waveforms at the outputs of the gates when they respond to a given input. An example of a circuit with gate delays was presented in Section 3.10 in HDL Example 3.3. We next show an HDL example that produces the truth table of a combinational circuit. A **$monitor** system task displays the output caused by the given stimulus. A commented alternative statement having a **$display** task would create a header that could be used with a **$monitor** statement to eliminate the repetition of names on each line of output.

The analysis of combinational circuits was covered in Section 4.3. A multilevel circuit of a full adder was analyzed, and its truth table was derived by inspection. The gate-level description of this circuit is shown in HDL Example 4.10. The circuit has three inputs, two outputs, and nine gates. The description of the circuit follows the interconnections between the gates according to the schematic diagram of Fig. 4.2. The stimulus for the circuit is listed in the second module. The inputs for simulating the circuit are specified with a three-bit **reg** vector *D*. *D[2]* is equivalent to input *A*, *D[1]* to input *B*, and *D[0]* to input *C*. The outputs of the circuit F_1 and F_2 are declared as **wire.** The complement of *F2* is named *F2_b* to illustrate a common industry practice for designating the complement of a signal (instead of appending *_not*). This procedure

follows the steps outlined in Fig. 4.34. The **repeat** loop provides the seven binary numbers after 000 for the truth table. The result of the simulation generates the output truth table displayed with the example. The truth table listed shows that the circuit is a full adder.

HDL Example 4.10 (Gate-Level Circuit)

```
// Gate-level description of circuit of Fig. 4.2

module Circuit_of_Fig_4_2 (A, B, C, F1, F2);
  input  A, B, C;
  output F1, F2;
  wire   T1, T2, T3, F2_b, E1, E2, E3;
  or   g1 (T1, A, B, C);
  and g2 (T2, A, B, C);
  and g3 (E1, A, B);
  and g4 (E2, A, C);
  and g5 (E3, B, C);
  or   g6 (F2, E1, E2, E3);
  not  g7 (F2_b, F2);
  and g8 (T3, T1, F2_b);
  or   g9 (F1, T2, T3);
endmodule

// Stimulus to analyze the circuit

module test_circuit;
  reg [2: 0] D;
  wire F1, F2;
  Circuit_of_Fig_4_2 (D[2], D[1], D[0], F1, F2);
  initial
   begin
     D = 3'b000;
     repeat (7) #10 D = D 1 1'b1;
   end
  initial
  $monitor ("ABC = %b F1 = %b F2 =%b", D, F1, F2);

endmodule

Simulation log:  ABC = 000 F1 = 0 F2 =0
ABC = 001 F1 = 1 F2 =0 ABC = 010 F1 = 1 F2 =0
ABC = 011 F1 = 0 F2 =1 ABC = 100 F1 = 1 F2 =0
ABC = 101 F1 = 0 F2 =1 ABC = 110 F1 = 0 F2 =1
ABC = 111 F1 = 1 F2 =1
```

PROBLEMS

(Answers to problems marked with * appear at the end of the text. Where appropriate, a logic design and its related HDL modeling problem are cross-referenced.)

4.1 Consider the combinational circuit shown in Fig. P4.1. (HDL—see Problem 4.49.)

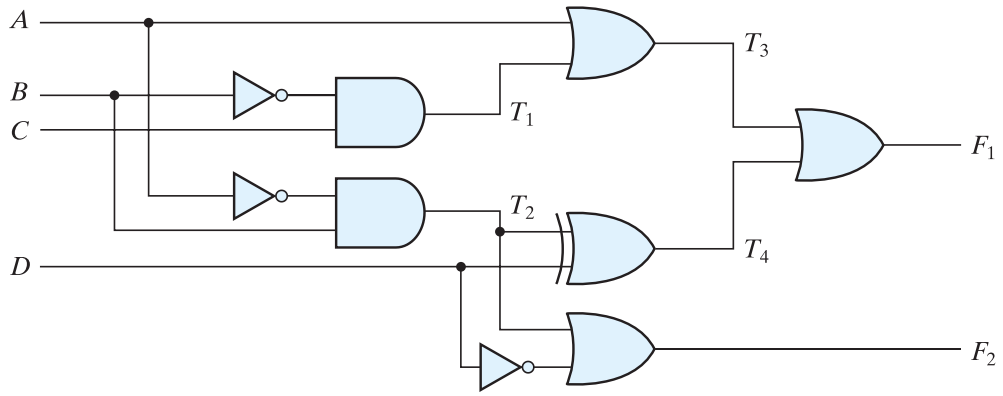


FIGURE P4.1

(a)* Derive the Boolean expressions for T_1 through T_4. Evaluate the outputs F_1 and F_2 as a function of the four inputs.
(b) List the truth table with 16 binary combinations of the four input variables. Then list the binary values for T_1 through T_4 and outputs F_1 and F_2 in the table.
(c) Plot the output Boolean functions obtained in part (b) on maps and show that the simplified Boolean expressions are equivalent to the ones obtained in part (a).

4.2* Obtain the simplified Boolean expressions for output F and G in terms of the input variables in the circuit of Fig. P4.2.

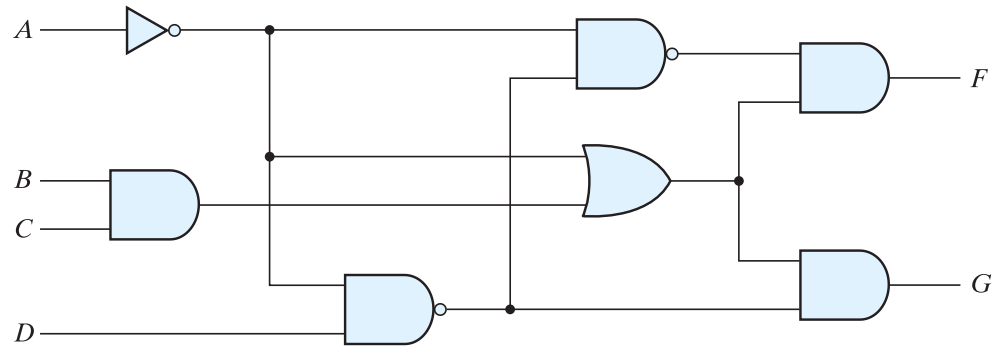


FIGURE P4.2

4.3 For the circuit shown in Fig. 4.26 (Section 4.11),
(a) Write the Boolean functions for the four outputs in terms of the input variables.
(b)* If the circuit is described in a truth table, how many rows and columns would there be in the table?

4.4 Design a combinational circuit with three inputs and one output.
(a)* The output is 1 when the binary value of the inputs is less than 3. The output is 0 otherwise.
(b) The output is 1 when the binary value of the inputs is an even number.

4.5 Design a combinational circuit with three inputs, x, y, and z, and three outputs, A, B, and C. When the binary input is 0, 1, 2, or 3, the binary output is one greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is two less than the input.

4.6 A majority circuit is a combinational circuit whose output is equal to 1 if the input variables have more 1's than 0's. The output is 0 otherwise.

(a)* Design a 3-input majority circuit by finding the circuit's truth table, Boolean equation, and a logic diagram.

(b) Write and verify a Verilog gate-level model of the circuit.

4.7 Design a combinational circuit that converts a four-bit Gray code (Table 1.6) to a bit four-binary number.

(a)* Implement the circuit with exclusive-OR gates.

(b) Using a case statement, write and verify a Verilog model of the circuit.

4.8 Design a code converter that converts a decimal digit from

(a)* The 8, 4, –2, –1 code to BCD (see Table 1.5). (HDL—see Problem 4.50.)

(b) The 8, 4, –2, –1 code to Gray code.

4.9 An ABCD-to-seven-segment decoder is a combinational circuit that converts a decimal digit in BCD to an appropriate code for the selection of segments in an indicator used to display the decimal digit in a familiar form. The seven outputs of the decoder *(a, b, c, d, e, f, g)* select the corresponding segments in the display, as shown in Fig. P4.9(a). The numeric display chosen to represent the decimal digit is shown in Fig. P4.9(b). Using a truth table and Karnaugh maps, design the BCD-to-seven-segment decoder using a minimum number of gates. The six invalid combinations should result in a blank display. (HDL—see Problem 4.51.)

(a) Segment designation (b) Numerical designation for display

FIGURE P4.9

4.10* Design a four-bit combinational circuit 2's complementer. (The output generates the 2's complement of the input binary number.) Show that the circuit can be constructed with exclusive-OR gates. Can you predict what the output functions are for a five-bit 2's complementer?

4.11 Using four half-adders (HDL—see Problem 4.52),

(a) Design a full-subtractor circuit incrementer. (A circuit that adds one to a four-bit binary number.)

(b)* Design a four-bit combinational decrementer (a circuit that subtracts 1 from a four-bit binary number).

4.12 Design a half-subtractor circuit with inputs x and y and outputs *Diff* and B_{out}. The circuit subtracts the bits $x - y$ and places the difference in D and the borrow in B_{out}.

(a) Design a full-subtractor circuit with three inputs x, y, B_{in} and two outputs *Diff* and B_{out}. The circuit subtracts $x - y - B_{in}$, where B_{in} is the input borrow, B_{out} is the output borrow, and *Diff* is the difference.

4.13* The adder–subtractor circuit of Fig. 4.13 has the following values for mode input M and data inputs A and B.

	M	A	B
(a)	0	0111	0110
(b)	0	1000	1001
(c)	1	1100	1000
(d)	1	0101	1010
(e)	1	0000	0001

In each case, determine the values of the four *SUM* outputs, the carry C, and overflow V. (HDL—see Problems 4.37 and 4.40.)

4.14* Assume that the exclusive-OR gate has a propagation delay of 10 ns and that the AND or OR gates have a propagation delay of 5 ns. What is the total propagation delay time in the four-bit adder of Fig. 4.12?

4.15 Derive the two-level Boolean expression for the output carry C_4 shown in the lookahead carry generator of Fig. 4.12.

4.16 Define the carry propagate and carry generate as

$$P_i = A_i + B_i$$
$$G_i = A_iB_i$$

respectively. Show that the output carry and output sum of a full adder becomes

$$C_{i+1} = (C_i' G_i' + P_i')'$$
$$S_i = (P_iG_i') \oplus C_i$$

The logic diagram of the first stage of a four-bit parallel adder as implemented in IC type 74283 is shown in Fig. P4.16. Identify the P_i' and G_i' terminals and show that the circuit implements a full-adder circuit.

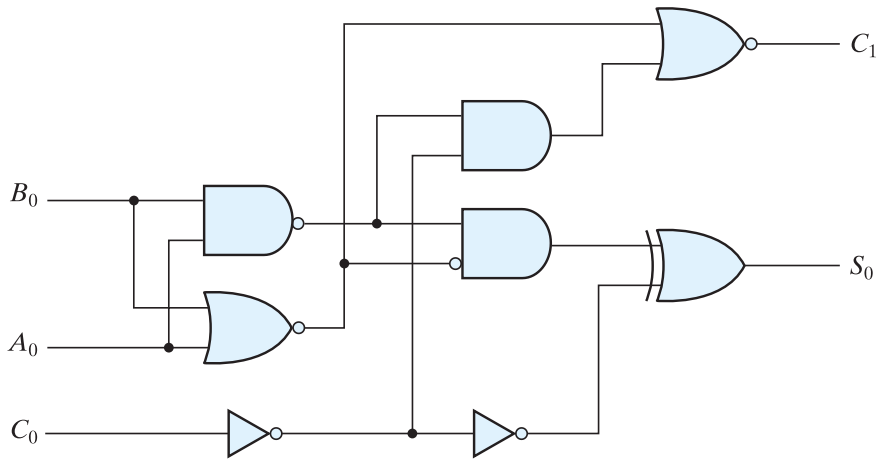


FIGURE P4.16
First stage of a parallel adder

4.17 Show that the output carry in a full adder circuit can be expressed in the AND-OR-INVERT form

$$C_{i+1} = G_i + P_iC_i = (G_i'P_i' + G_i'C_i')'$$

IC type 74182 is a lookahead carry generator circuit that generates the carries with AND-OR-INVERT gates (see Section 3.8). The circuit assumes that the input terminals have the complements of the G's, the P's, and of C_1. Derive the Boolean functions for the lookahead carries C_2, C_3, and C_4 in this IC. (*Hint:* Use the equation-substitution method to derive the carries in terms of C_i')

4.18 Design a combinational circuit that generates the 9's complement of a
(a)* BCD digit. (HDL—see Problem 4.54(a).)
(b) Gray-code digit. (HDL—see Problem 4.54(b).)

4.19 Construct a BCD adder–subtractor circuit. Use the BCD adder of Fig. 4.14 and the 9's complementer of problem 4.18. Use block diagrams for the components. (HDL—see Problem 4.55.)

4.20 For a binary multiplier that multiplies two unsigned four-bit numbers,
(a) Using AND gates and binary adders (see Fig. 4.16), design the circuit.
(b) Write and verify a Verilog dataflow model of the circuit.

4.21 Design a combinational circuit that compares two 4-bit numbers to check if they are equal. The circuit output is equal to 1 if the two numbers are equal and 0 otherwise.

4.22* Design an excess-3-to-binary decoder using the unused combinations of the code as don't-care conditions. (HDL—see Problem 4.42.)

4.23 Draw the logic diagram of a 2-to-4-line decoder using (a) NOR gates only and (b) NAND gates only. Include an enable input. (HDL—see Problems 4.36, 4.45.)

4.24 Design a BCD-to-decimal decoder using the unused combinations of the BCD code as don't-care conditions.

4.25 Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and a 2-to-4-line decoder. Use block diagrams for the components. (HDL—see Problem 4.63.)

4.26 Construct a 4-to-16-line decoder with five 2-to-4-line decoders with enable. (HDL—see Problem 4.64.)

4.27 A combinational circuit is specified by the following three Boolean functions:

$$F_1(A, B, C) = \Sigma(1, 4, 6)$$
$$F_2(A, B, C) = \Sigma(3, 5)$$
$$F_3(A, B, C) = \Sigma(2, 4, 6, 7)$$

Implement the circuit with a decoder constructed with NAND gates (similar to Fig. 4.19) and NAND or AND gates connected to the decoder outputs. Use a block diagram for the decoder. Minimize the number of inputs in the external gates.

4.28 Using a decoder and external gates, design the combinational circui defined by the following three Boolean functions:

(a) $F_1 = x'yz' + xz$
$F_2 = xy'z' + x'y$
$F_3 = x'y'z' + xy$

(b) $F_1 = (y' + x)z$
$F_2 = y'z' + x'y + yz'$
$F_3 = (x + y)z$

4.29* Design a four-input priority encoder with inputs as in Table 4.8, but with input D_0 having the highest priority and input D_3 the lowest priority.

4.30 Specify the truth table of an octal-to-binary priority encoder. Provide an output V to indicate that at least one of the inputs is present. The input with the highest subscript number has the highest priority. What will be the value of the four outputs if inputs D_2 and D_6 are 1 at the same time? (HDL—see Problem 4.65.)

4.31 Construct a 16×1 multiplexer with two 8×1 and one 2×1 multiplexers. Use block diagrams. (HDL—see Problem 4.67.)

4.32 Implement the following Boolean function with a multiplexer (HDL—see Problem 4.46):

(a) $F(A, B, C, D) = \Sigma(0, 2, 5, 8, 10, 14)$

(b) $F(A, B, C, D) = \Pi(2, 6, 11)$

4.33 Implement a full adder with two 4×1 multiplexers.

4.34 An 8×1 multiplexer has inputs A, B, and C connected to the selection inputs S_2, S_1, and S_0, respectively. The data inputs I_0 through I_7 are as follows:

(a)* $I_1 = I_2 = I_7 = 0$; $I_3 = I_5 = 1$; $I_0 = I_4 = D$; and $I_6 = D'$.

(b) $I_1 = I_2 = 0$; $I_3 = I_7 = 1$; $I_4 = I_5 = D$; and $I_0 = I_6 = D'$.

Determine the Boolean function that the multiplexer implements.

4.35 Implement the following Boolean function with a 4×1 multiplexer and external gates.

(a)* $F_1(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$

(b) $F_2(A, B, C, D) = \Sigma(1, 2, 5, 7, 8, 10, 11, 13, 15)$

Connect inputs A and B to the selection lines. The input requirements for the four data lines will be a function of variables C and D. These values are obtained by expressing F as a function of C and D for each of the four cases when $AB = 00, 01, 10$, and 11. These functions may have to be implemented with external gates. (HDL—see Problem 4.47.)

4.36 Write the HDL gate-level description of the priority encoder circuit shown in Fig. 4.23. (HDL—see Problem 4.45.)

4.37 Write the HDL gate-level hierarchical description of a four-bit adder–subtractor for unsigned binary numbers. The circuit is similar to Fig. 4.13 but without output V. You can instantiate the four-bit full adder described in HDL Example 4.2. (HDL—see Problems 4.13 and 4.40.)

4.38 Write the HDL dataflow description of a quadruple 2-to-1-line multiplexer with enable (see Fig. 4.26).

4.39* Write an HDL behavioral description of a four-bit comparator with a six-bit output $Y[5:0]$. Bit 5 of Y is for "equals," bit 4 for "not equal to," bit 3 for "greater than," bit 2 for "less than," bit 1 for "greater than or equal," and bit 0 for "less than or equal to."

4.40 Using the conditional operator (?:), write an HDL dataflow description of a four-bit adder–subtractor of unsigned numbers. (See Problems 4.13 and 4.37.)

4.41 Repeat problem 4.40 using an always statement.

4.42 (a) Write an HDL gate-level description of the BCD-to-excess-3 converter circuit shown in Fig. 4.4 (see Problem 4.22).
(b) Write a dataflow description of the BCD-to-excess-3 converter using the Boolean expressions listed in Fig. 4.3.
(c)* Write an HDL behavioral description of a BCD-to-excess-3 converter.
(d) Write a test bench to simulate and test the BCD-to-excess-3 converter circuit in order to verify the truth table. Check all three circuits.

4.43 Explain the function of the circuit specified by the following HDL description:

```
module Prob4_43 (A, B, S, E, Q);
 input  [1:0] A, B;
 input        S, E;
 output [1:0] Q;
 assign Q = E ? (S ? A : B) : 'bz;
endmodule
```

4.44 Using a case statement, write an HDL behavioral description of a eight-bit arithmetic-logic unit (ALU). The circuit has a three-bit select bus (Sel), sixteen-bit input datapaths (A[15:0] and B[15:0]), an eight-bit output datapath (y[15:0]), and performs the arithmetic and logic operations listed below.

Sel	Operation	Description
000	y = 8′b0	
001	y = A & B	Bitwise AND
010	y = A | B	Bitwise OR
011	y = A ^ B	Bitwise exclusive OR
100	y = ~A	Bitwise complement
101	y = A − B	Subtract
110	y = A + B	Add (Assume A and B are unsigned)
111	y = 8′hFF	

4.45 Write an HDL behavioral description of a four-input priority encoder. Use a four-bit vector for the *D* inputs and an **always** block with if–else statements. Assume that input *D*[3] has the highest priority (see Problem 4.36).

4.46 Write a Verilog dataflow description of the logic circuit described by the Boolean function in Problem 4.32.

4.47 Write a Verilog dataflow description of the logic circuit described by the Boolean function in Problem 4.35.

4.48 Develop and modify the eight-bit ALU specified in Problem 4.44 so that it has three-state output controlled by an enable input, *En*. Write a test bench and simulate the circuit.

4.49 For the circuit shown in Fig. P4.1,
(a) Write and verify a gate-level HDL model of the circuit.
(b) Compare your results with those obtained for Problem 4.1.

4.50 Using a case statement, develop and simulate a behavioral model of
(a)* The 8, 4, –2, –1 to BCD code converter described in Problem 4.8(a).
(b) The 8, 4, –2, –1 to Gray code converter described in Problem 4.8(b).

4.51 Develop and simulate a behavioral model of the ABCD-to-seven-segment decoder described in Problem 4.9.

4.52 Using a continuous assignment, develop and simulate a dataflow model of
(a) The four-bit incrementer described in Problem 4.11(a).
(b) The four-bit decrementer described in Problem 4.11(b).

4.53 Develop and simulate a structural model of the decimal adder shown in Fig. 4.14.

4.54 Develop and simulate a behavioral model of a circuit that generates the 9's complement of
(a) a BCD digit (see Problem 4.18(a)).
(b) a Gray-code digit (see Problem 4.18(b).)

4.55 Construct a hierarchical model of the BCD adder–subtractor described in Problem 4.19. The BCD adder and the 9's complementer are to be described as behavioral models in separate modules, and they are to be instantiated in a top-level module.

4.56* Write a continuous assignment statement that compares two 4-bit numbers to check if their bit patterns match. The variable to which the assignment is made is to equal 1 if the numbers match and 0 otherwise.

4.57* Develop and verify a behavioral model of the four-bit priority encoder described in Problem 4.29.

4.58 Write a Verilog model of a circuit whose 32-bit output is formed by shifting its 32-bit input three positions to the right and filling the vacant positions with the bit that was in the MSN before the shift occurred (shift arithmetic right).Write a Verilog model of a circuit whose 32-bit output is formed by shifting its 32-bit input three positions to the left and filling the vacant positions with 0 (shift logical left).

4.59 Write a Verilog model of a BCD-to-decimal decoder using the unused combinations of the BCD code as don't-care conditions (see Problem 4.24).

4.60 Using the port syntax of the IEEE 1364-2001 standard, write and verify a gate-level model of the four-bit even parity checker shown in Fig. 3.34.

4.61 Using continuous assignment statements and the port syntax of the IEEE 1364-2001 standard, write and verify a gate-level model of the four-bit even parity checker shown in Fig. 3.34.

4.62 Write and verify a gate-level hierarchical model of the circuit described in Problem 4.25.

4.63 Write and verify a gate-level hierarchical model of the circuit described in Problem 4.26.

4.64 Write and verify a Verilog model of the octal-to-binary circuit described in Problem 4.30.

4.65 Write a hierarchical gate-level model of the multiplexer described in Problem 4.31.

REFERENCES

1. Bhasker, J. 1997. *A Verilog HDL Primer*. Allentown, PA: Star Galaxy Press.
2. Bhasker, J. 1998. *Verilog HDL Synthesis*. Allentown, PA: Star Galaxy Press.
3. Ciletti, M. D. 1999. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL*. Upper Saddle River, NJ: Prentice Hall.
4. Dietmeyer, D. L. 1988. *Logic Design of Digital Systems*, 3rd ed. Boston: Allyn Bacon.

5. Gajski, D. D. 1997. *Principles of Digital Design*. Upper Saddle River, NJ: Prentice Hall.
6. Hayes, J. P. 1993. *Introduction to Digital Logic Design*. Reading, MA: Addison-Wesley.
7. Katz, R. H. 2005. *Contemporary Logic Design*. Upper Saddle River, NJ: Pearson Prentice Hall.
8. Mano, M. M. and C. R. Kime. 2007. *Logic and Computer Design Fundamentals*, 4th ed. Upper Saddle River, NJ: Prentice Hall.
9. Nelson, V. P., H. T. Nagle, J. D. Irwin, and B. D. Carroll. 1995. *Digital Logic Circuit Analysis and Design*. Englewood Cliffs, NJ: Prentice Hall.
10. Palnitkar, S. 1996. *Verilog HDL: A Guide to Digital Design and Synthesis*. Mountain View, CA: SunSoft Press (a Prentice Hall title).
11. Roth, C. H. 2009. *Fundamentals of Logic Design*, 6th ed. St. Paul, MN: West.
12. Thomas, D. E. and P. R. Moorby. 2002. *The Verilog Hardware Description Language*, 5th ed. Boston: Kluwer Academic Publishers.
13. Wakerly, J. F. 2005. *Digital Design: Principles and Practices*, 4th ed. Upper Saddle River, NJ: Prentice Hall.

WEB SEARCH TOPICS

Boolean equation
Combinational logic
Truth table
Exclusive–OR
Comparator
Multiplexer
Decoder
Priority encoder
Three-state inverter
Three-state buffer