

---

## Chapter 4

# Combinational Logic

---

### 4.1 INTRODUCTION

---

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs. A combinational circuit performs an operation that can be specified logically by a set of Boolean functions. In contrast, sequential circuits employ storage elements in addition to logic gates. Their outputs are a function of the inputs and the state of the storage elements. Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states. Sequential circuits are the building blocks of digital systems and are discussed in Chapters 5, 8, and 9.

### 4.2 COMBINATIONAL CIRCUITS

---

A combinational circuit consists of input variables, logic gates, and output variables. Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data. A block diagram of a combinational circuit is shown in Fig. 4.1. The  $n$  input binary variables come from an external source; the  $m$  output variables are produced by the internal combinational logic circuit and go to an external destination. Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0. (*Note:* Logic simulators show only 0's and 1's, not the actual analog signals.) In many applications, the source and destination are storage registers. If the registers are included with the combinational gates, then the total circuit must be considered to be a sequential circuit.



**FIGURE 4.1**  
Block diagram of combinational circuit

For  $n$  input variables, there are  $2^n$  possible binary input combinations. For each possible input combination, there is one possible output value. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables. A combinational circuit also can be described by  $m$  Boolean functions, one for each output variable. Each output function is expressed in terms of the  $n$  input variables.

In Chapter 1, we learned about binary numbers and binary codes that represent discrete quantities of information. The binary variables are represented physically by electric voltages or some other type of signal. The signals can be manipulated in digital logic gates to perform required functions. In Chapter 2, we introduced Boolean algebra as a way to express logic functions algebraically. In Chapter 3, we learned how to simplify Boolean functions to achieve economical (simpler) gate implementations. The purpose of the current chapter is to use the knowledge acquired in previous chapters to formulate systematic analysis and design procedures for combinational circuits. The solution of some typical examples will provide a useful catalog of elementary functions that are important for the understanding of digital systems. We'll address three tasks: (1) Analyze the behavior of a given logic circuit, (2) synthesize a circuit that will have a given behavior, and (3) write HDL models for some common circuits.

There are several combinational circuits that are employed extensively in the design of digital systems. These circuits are available in integrated circuits and are classified as standard components. They perform specific digital functions commonly needed in the design of digital systems. In this chapter, we introduce the most important standard combinational circuits, such as adders, subtractors, comparators, decoders, encoders, and multiplexers. These components are available in integrated circuits as medium-scale integration (MSI) circuits. They are also used as *standard cells* in complex very large-scale integrated (VLSI) circuits such as application-specific integrated circuits (ASICs). The standard cell functions are interconnected within the VLSI circuit in the same way that they are used in multiple-IC MSI design.

## 4.3 ANALYSIS PROCEDURE

The analysis of a combinational circuit requires that we determine the function that the circuit implements. This task starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or, possibly, an explanation of the circuit operation. If the logic diagram to be analyzed is accompanied by a function name or an explanation of what it is assumed to accomplish, then the analysis problem reduces to a verification of the stated function. The analysis can be performed manually by finding the Boolean functions or truth table or by using a computer simulation program.

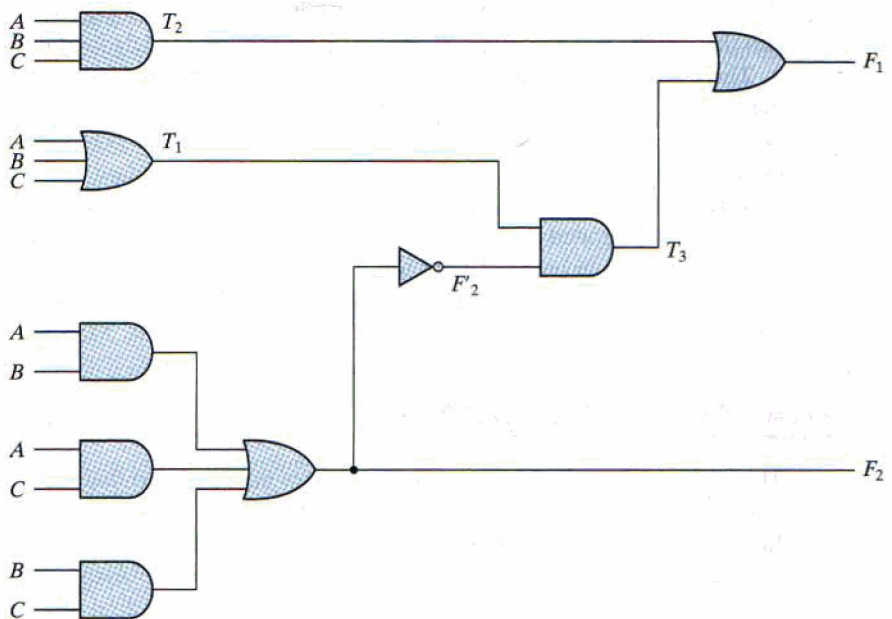
The first step in the analysis is to make sure that the given circuit is combinational and not sequential. The diagram of a combinational circuit has logic gates with no feedback paths or memory elements. A feedback path is a connection from the output of one gate to the input of a second gate that forms part of the input to the first gate. Feedback paths in a digital circuit define a sequential circuit and must be analyzed according to procedures outlined in Chapter 9.

Once the logic diagram is verified to be that of a combinational circuit, one can proceed to obtain the output Boolean functions or the truth table. If the function of the circuit is under investigation, then it is necessary to interpret the operation of the circuit from the derived Boolean functions or truth table. The success of such an investigation is enhanced if one has previous experience and familiarity with a wide variety of digital circuits.

To obtain the output Boolean functions from a logic diagram, we proceed as follows:

1. Label all gate outputs that are a function of input variables with arbitrary symbols—but with meaningful names. Determine the Boolean functions for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

The analysis of the combinational circuit of Fig. 4.2 illustrates the proposed procedure. We note that the circuit has three binary inputs— $A$ ,  $B$ , and  $C$ —and two binary outputs— $F_1$  and  $F_2$ .



**FIGURE 4.2**  
Logic diagram for analysis example

The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function only of input variables are  $T_1$  and  $T_2$ . Output  $F_2$  can easily be derived from the input variables. The Boolean functions for these three outputs are

$$F_2 = AB + AC + BC$$

$$T_1 = A + B + C$$

$$T_2 = ABC$$

Next, we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F_2' T_1$$

$$F_1 = T_3 + T_2$$

To obtain  $F_1$  as a function of  $A$ ,  $B$ , and  $C$ , we form a series of substitutions as follows:

$$\begin{aligned} F_1 &= T_3 + T_2 = F_2' T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

If we want to pursue the investigation and determine the information transformation task achieved by this circuit, we can draw the circuit from the derived Boolean expressions and try to recognize a familiar operation. The Boolean functions for  $F_1$  and  $F_2$  implement a circuit discussed in Section 4.5. Merely finding a Boolean representation of a circuit doesn't provide insight into its behavior, but in this example we will observe that the Boolean equations and truth table for  $F_1$  and  $F_2$  match those describing the functionality of what we call a full adder.

The derivation of the truth table for a circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, we proceed as follows:

1. Determine the number of input variables in the circuit. For  $n$  inputs, form the  $2^n$  possible input combinations and list the binary numbers from 0 to  $2^n - 1$  in a table.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

This process is illustrated with the circuit of Fig. 4.2. In Table 4.1, we form the eight possible combinations for the three input variables. The truth table for  $F_2$  is determined directly from the values of  $A$ ,  $B$ , and  $C$ , with  $F_2$  equal to 1 for any combination that has two or three inputs equal to 1. The truth table for  $F_2'$  is the complement of that of  $F_2$ . The truth tables for  $T_1$  and  $T_2$  are the OR and AND functions of the input variables, respectively. The values for  $T_3$  are derived from  $T_1$  and  $F_2'$ :  $T_3$  is equal to 1 when both  $T_1$  and  $F_2'$  are equal to 1, and  $T_3$  is equal to 0 otherwise. Finally,  $F_1$  is equal to 1 for those combinations in which either  $T_2$  or  $T_3$  or both are equal



**Table 4.1**  
*Truth Table for the Logic Diagram of Fig. 4.2*

A	B	C	$F_2$	$F_2'$	$T_1$	$T_2$	$T_3$	$F_1$
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

to 1. Inspection of the truth table combinations for  $A$ ,  $B$ ,  $C$ ,  $F_1$ , and  $F_2$  shows that it is identical to the truth table of the full adder given in Section 4.5 for  $x$ ,  $y$ ,  $z$ ,  $S$ , and  $C$ , respectively.

Another way of analyzing a combinational circuit is by means of logic simulation. This is not practical, however, because the number of input patterns that might be needed to generate meaningful outputs could be very large. But simulation has a very practical application in verifying that the functionality of a circuit actually matches its specification. In Section 4.12, we demonstrate the logic simulation and verification of the circuit of Fig. 4.2, using Verilog HDL.

## 4.4 DESIGN PROCEDURE

The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained. The procedure involves the following steps:

1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
2. Derive the truth table that defines the required relationship between inputs and outputs.
3. Obtain the simplified Boolean functions for each output as a function of the input variables.
4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

A truth table for a combinational circuit consists of input columns and output columns. The input columns are obtained from the  $2^n$  binary numbers for the  $n$  input variables. The binary values for the outputs are determined from the stated specifications. The output functions specified in the truth table give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly in the truth table, as they are often incomplete, and any wrong interpretation may result in an incorrect truth table.

The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program. Frequently, there is a variety of simplified expressions from which to choose. In a particular

application, certain criteria will serve as a guide in the process of choosing an implementation. A practical design must consider such constraints as the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits. Since the importance of each constraint is dictated by the particular application, it is difficult to make a general statement about what constitutes an acceptable implementation. In most cases, the simplification begins by satisfying an elementary objective, such as producing the simplified Boolean functions in a standard form. Then the simplification proceeds with further steps to meet other performance criteria.

### Code Conversion Example

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a code converter is a circuit that makes the two systems compatible even though each uses a different binary code.

To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates. The design procedure will be illustrated by an example that converts binary coded decimal (BCD) to the excess-3 code for the decimal digits.

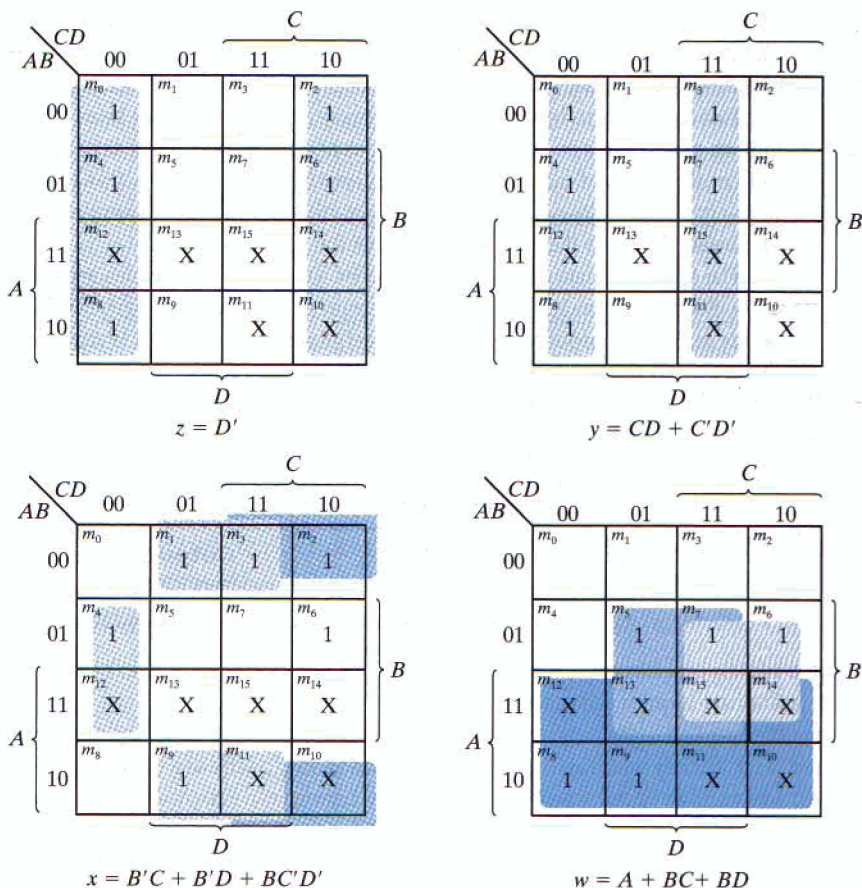
The bit combinations assigned to the BCD and excess-3 codes are listed in Table 1.5 (Section 1.7). Since each code uses four bits to represent a decimal digit, there must be four input variables and four output variables. We designate the four input binary variables by the symbols  $A$ ,  $B$ ,  $C$ , and  $D$ , and the four output variables by  $w$ ,  $x$ ,  $y$ , and  $z$ . The truth table relating the input and output variables is shown in Table 4.2. The bit combinations for the inputs and their

**Table 4.2**  
*Truth Table for Code-Conversion Example*

Input BCD				Output Excess-3 Code			
$A$	$B$	$C$	$D$	$w$	$x$	$y$	$z$
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

corresponding outputs are obtained directly from Section 1.7. Note that four binary variables may have 16 bit combinations, but only 10 are listed in the truth table. The six bit combinations not listed for the input variables are don't-care combinations. These values have no meaning in BCD and we assume that they will never occur. Therefore, we are at liberty to assign to the output variables either a 1 or a 0, whichever gives a simpler circuit.

The maps in Fig. 4.3 are plotted to obtain simplified Boolean functions for the outputs. Each one of the four maps represents one of the four outputs of the circuit as a function of the four input variables. The 1's marked inside the squares are obtained from the minterms that make the output equal to 1. The 1's are obtained from the truth table by going over the output columns one at a time. For example, the column under output  $z$  has five 1's; therefore, the map for  $z$  has five 1's, each being in a square corresponding to the minterm that makes  $z$  equal to 1. The six don't-care minterms 10 through 15 are marked with an X. One possible way to simplify the functions into sum-of-products form is listed under the map of each variable. (See Chapter 3.)



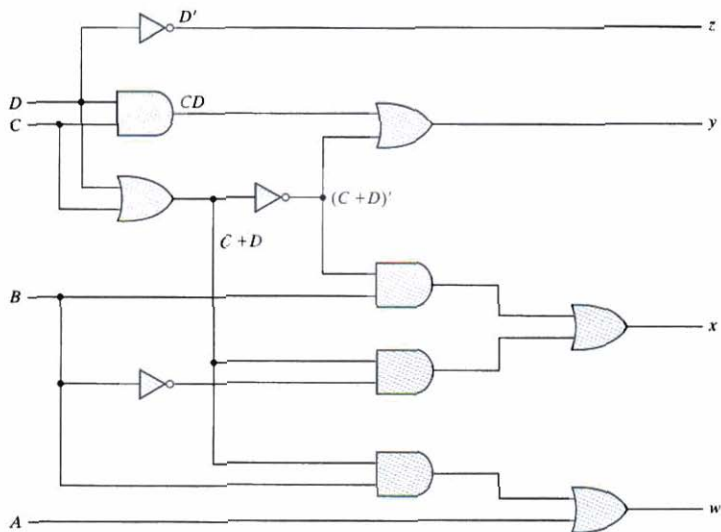
**FIGURE 4.3**  
Maps for BCD-to-excess-3 code converter

A two-level logic diagram may be obtained directly from the Boolean expressions derived from the maps. There are various other possibilities for a logic diagram that implements this circuit. The expressions obtained in Fig. 4.3 may be manipulated algebraically for the purpose of using common gates for two or more outputs. This manipulation, shown next, illustrates the flexibility obtained with multiple-output systems when implemented with three or more levels of gates:

$$\begin{aligned} z &= D' \\ y &= CD + C'D' = CD + (C + D)' \\ x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\ &= B'(C + D) + B(C + D)' \\ w &= A + BC + BD = A + B(C + D) \end{aligned}$$

The logic diagram that implements these expressions is shown in Fig. 4.4. Note that the OR gate whose output is  $C + D$  has been used to implement partially each of three outputs.

Not counting input inverters, the implementation in sum-of-products form requires seven AND gates and three OR gates. The implementation of Fig. 4.4 requires four AND gates, four OR gates, and one inverter. If only the normal inputs are available, the first implementation will require inverters for variables  $B$ ,  $C$ , and  $D$ , and the second implementation will require inverters for variables  $B$  and  $D$ . Thus, the three-level logic circuit requires fewer gates, all of which in turn require no more than two inputs.



**FIGURE 4.4**  
Logic diagram for BCD-to-excess-3 code converter



## 4.5 BINARY ADDER-SUBTRACTOR

Digital computers perform a variety of information-processing tasks. Among the functions encountered are the various arithmetic operations. The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations:  $0 + 0 = 0$ ,  $0 + 1 = 1$ ,  $1 + 0 = 1$ , and  $1 + 1 = 10$ . The first three operations produce a sum of one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a *carry*. When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half adder*. One that performs the addition of three bits (two significant bits and a previous carry) is a *full adder*. The names of the circuits stem from the fact that two half adders can be employed to implement a full adder.

A binary adder-subtractor is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers. We will develop this circuit by means of a hierarchical design. The half adder design is carried out first, from which we develop the full adder. Connecting  $n$  full adders in cascade produces a binary adder for two  $n$ -bit numbers. The subtraction circuit is included in a complementing circuit.

### Half Adder

From the verbal explanation of a half adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. We assign symbols  $x$  and  $y$  to the two inputs and  $S$  (for sum) and  $C$  (for carry) to the outputs. The truth table for the half adder is listed in Table 4.3. The  $C$  output is 1 only when both inputs are 1. The  $S$  output represents the least significant bit of the sum.

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

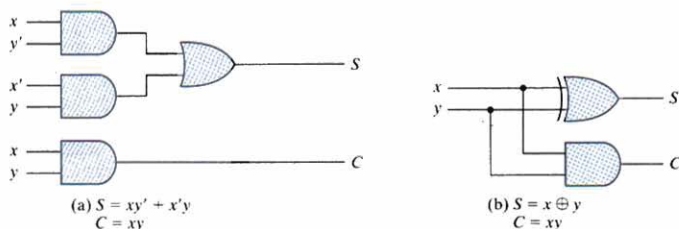
$$S = x'y + xy'$$

$$C = xy$$

The logic diagram of the half adder implemented in sum of products is shown in Fig. 4.5(a). It can be also implemented with an exclusive-OR and an AND gate as shown in Fig. 4.5(b). This form is used to show that two half adders can be used to construct a full adder.

**Table 4.3**  
*Half Adder*

$x$	$y$	$C$	$S$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



**FIGURE 4.5**  
Implementation of half adder

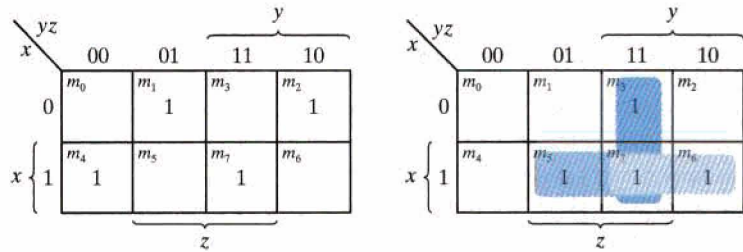
## Full Adder

A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs. Two of the input variables, denoted by  $x$  and  $y$ , represent the two significant bits to be added. The third input,  $z$ , represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols  $S$  for sum and  $C$  for carry. The binary variable  $S$  gives the value of the least significant bit of the sum. The binary variable  $C$  gives the output carry. The truth table of the full adder is listed in Table 4.4. The eight rows under the input variables designate all possible combinations of the three variables. The output variables are determined from the arithmetic sum of the input bits. When all input bits are 0, the output is 0. The  $S$  output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The  $C$  output has a carry of 1 if two or three inputs are equal to 1.

The input and output bits of the combinational circuit have different interpretations at various stages of the problem. On the one hand, physically, the binary signals of the inputs are considered binary digits to be added arithmetically to form a two-digit sum at the output. On the other hand, the same binary values are considered as variables of Boolean functions when expressed in the truth table or when the circuit is implemented with logic gates. The maps for the outputs of the full adder are shown in Fig. 4.6. The simplified expressions are

**Table 4.4**  
**Full Adder**

$x$	$y$	$z$	$C$	$S$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



**FIGURE 4.6**  
Maps for full adder

$$S = x'y'z + x'yz' + xy'z' + xyz$$

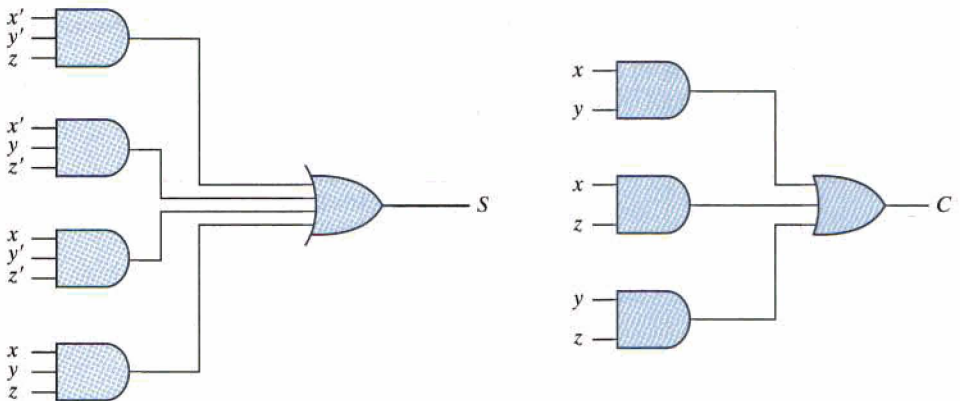
$$C = xy + xz + yz$$

The logic diagram for the full adder implemented in sum-of-products form is shown in Fig. 4.7. It can also be implemented with two half adders and one OR gate, as shown in Fig. 4.8. The  $S$  output from the second half adder is the exclusive-OR of  $z$  and the output of the first half adder, giving

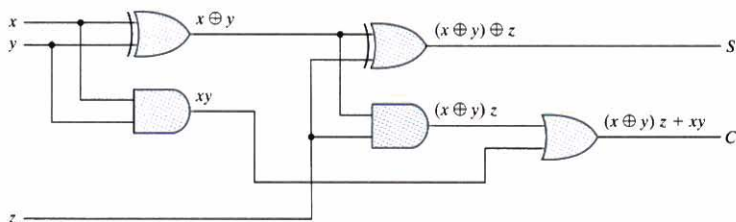
$$\begin{aligned} S &= z \oplus (x \oplus y) \\ &= z'(xy' + x'y) + z(xy' + x'y)' \\ &= z'(xy' + x'y) + z(xy + x'y') \\ &= xy'z' + x'yz' + xyz + x'y'z \end{aligned}$$

The carry output is

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$



**FIGURE 4.7**  
Implementation of full adder in sum-of-products form

**FIGURE 4.8**

Implementation of full adder with two half adders and an OR gate

## Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain. Figure 4.9 shows the interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder. The augend bits of  $A$  and the addend bits of  $B$  are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit. The carries are connected in a chain through the full adders. The input carry to the adder is  $C_0$ , and it ripples through the full adders to the output carry  $C_4$ . The  $S$  outputs generate the required sum bits. An  $n$ -bit adder requires  $n$  full adders, with each output carry connected to the input carry of the next higher order full adder.

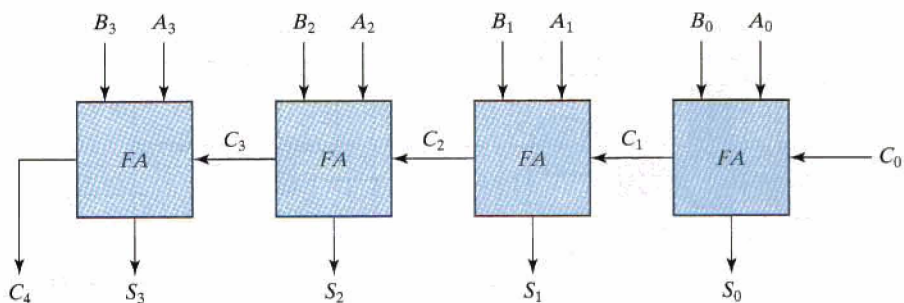
To demonstrate with a specific example, consider the two binary numbers  $A = 1011$  and  $B = 0011$ . Their sum  $S = 1110$  is formed with the four-bit adder as follows:

Subscript $i$ :	3	2	1	0	
Input carry	0	1	1	0	$C_i$
Augend	1	0	1	1	$A_i$
Addend	0	0	1	1	$B_i$
Sum	1	1	1	0	$S_i$
Output carry	0	0	1	1	$C_{i+1}$

The bits are added with full adders, starting from the least significant position (subscript 0), to form the sum bit and carry bit. The input carry  $C_0$  in the least significant position must be 0. The value of  $C_{i+1}$  in a given significant position is the output carry of the full adder. This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated. All the carries must be generated for the correct sum bits to appear at the outputs.

The four-bit adder is a typical example of a standard component. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the classical method would require a truth table with  $2^9 = 512$  entries, since there are nine inputs to





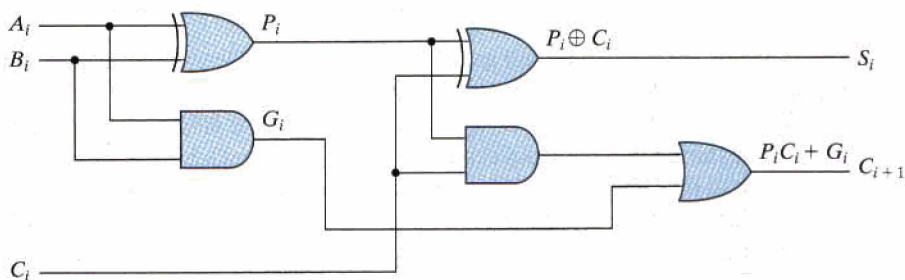
**FIGURE 4.9**  
Four-bit adder

the circuit. By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation.

## Carry Propagation

The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit. The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders. Since each bit of the sum output depends on the value of the input carry, the value of  $S_i$  at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. In this regard, consider output  $S_3$  in Fig. 4.9. Inputs  $A_3$  and  $B_3$  are available as soon as input signals are applied to the adder. However, input carry  $C_3$  does not settle to its final value until  $C_2$  is available from the previous stage. Similarly,  $C_2$  has to wait for  $C_1$  and so on down to  $C_0$ . Thus, only after the carry propagates and ripples through all stages will the last output  $S_3$  and carry  $C_4$  settle to their final correct value.

The number of gate levels for the carry propagation can be found from the circuit of the full adder. The circuit is redrawn with different labels in Fig. 4.10 for convenience. The input and



**FIGURE 4.10**  
Full adder with  $P$  and  $G$  shown

output variables use the subscript  $i$  to denote a typical stage of the adder. The signals at  $P_i$  and  $G_i$  settle to their steady-state values after they propagate through their respective gates. These two signals are common to all full adders and depend only on the input augend and addend bits. The signal from the input carry  $C_i$  to the output carry  $C_{i+1}$  propagates through an AND gate and an OR gate, which constitute two gate levels. If there are four full adders in the adder, the output carry  $C_4$  would have  $2 \times 4 = 8$  gate levels from  $C_0$  to  $C_4$ . For an  $n$ -bit adder, there are  $2n$  gate levels for the carry to propagate from input to output.

The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added. Although the adder—or, for that matter, any combinational circuit—will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is critical. An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. However, physical circuits have a limit to their capability. Another solution is to increase the complexity of the equipment in such a way that the carry delay time is reduced. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of *carry lookahead logic*.

Consider the circuit of the full adder shown in Fig. 4.10. If we define two new binary variables

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

$G_i$  is called a *carry generate*, and it produces a carry of 1 when both  $A_i$  and  $B_i$  are 1, regardless of the input carry  $C_i$ .  $P_i$  is called a *carry propagate*, because it determines whether a carry into stage  $i$  will propagate into stage  $i + 1$  (i.e., whether an assertion of  $C_i$  will propagate to an assertion of  $C_{i+1}$ ).

We now write the Boolean functions for the carry outputs of each stage and substitute the value of each  $C_i$  from the previous equations:

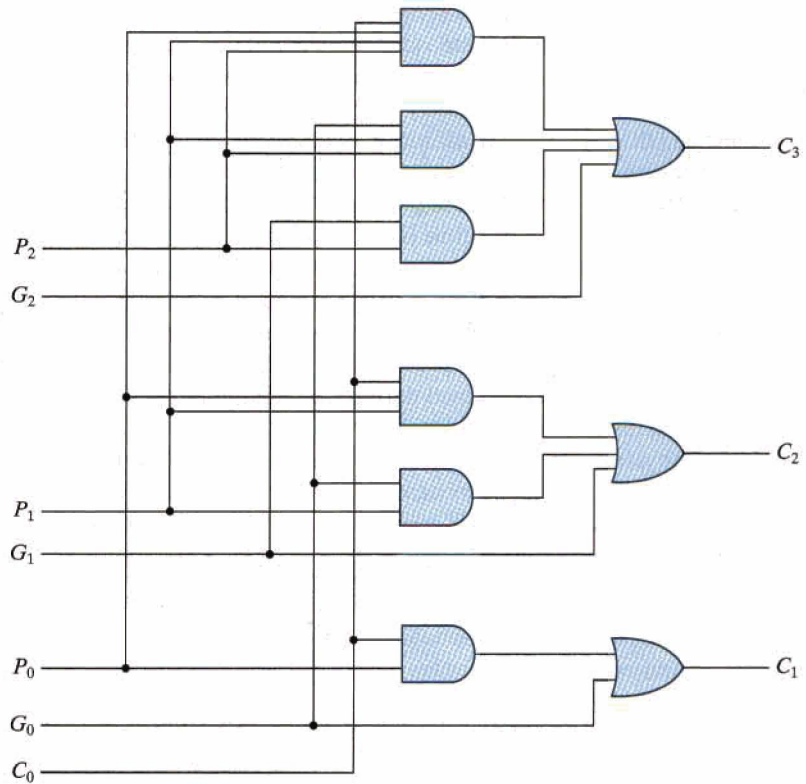
$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$

Since the Boolean function for each output carry is expressed in sum-of-products form, each function can be implemented with one level of AND gates followed by an OR gate (or by a two-level NAND). The three Boolean functions for  $C_1$ ,  $C_2$ , and  $C_3$  are implemented in the carry lookahead generator shown in Fig. 4.11. Note that this circuit can add in less time because  $C_3$  does not have to wait for  $C_2$  and  $C_1$  to propagate; in fact,  $C_3$  is propagated at the same time as  $C_1$  and  $C_2$ . This gain in speed of operation is achieved at the expense of additional complexity (hardware).

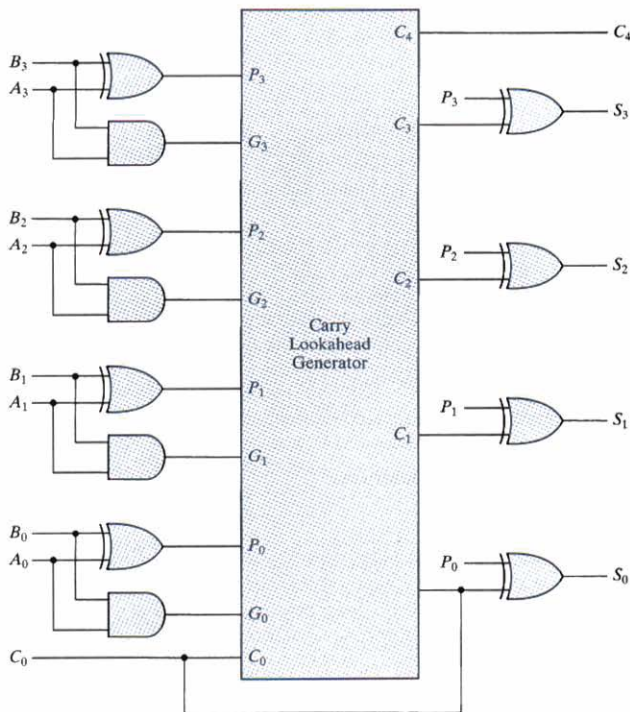


**FIGURE 4.11**  
Logic diagram of carry lookahead generator

The construction of a four-bit adder with a carry lookahead scheme is shown in Fig. 4.12. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the  $P_i$  variable, and the AND gate generates the  $G_i$  variable. The carries are propagated through the carry lookahead generator (similar to that in Fig. 4.11) and applied as inputs to the second exclusive-OR gate. All output carries are generated after a delay through two levels of gates. Thus, outputs  $S_1$  through  $S_3$  have equal propagation delay times. The two-level circuit for the output carry  $C_4$  is not shown. This circuit can easily be derived by the equation-substitution method.

## Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements, as discussed in Section 1.5. Remember that the subtraction  $A - B$  can be done by taking the 2's complement of  $B$  and adding it to  $A$ . The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

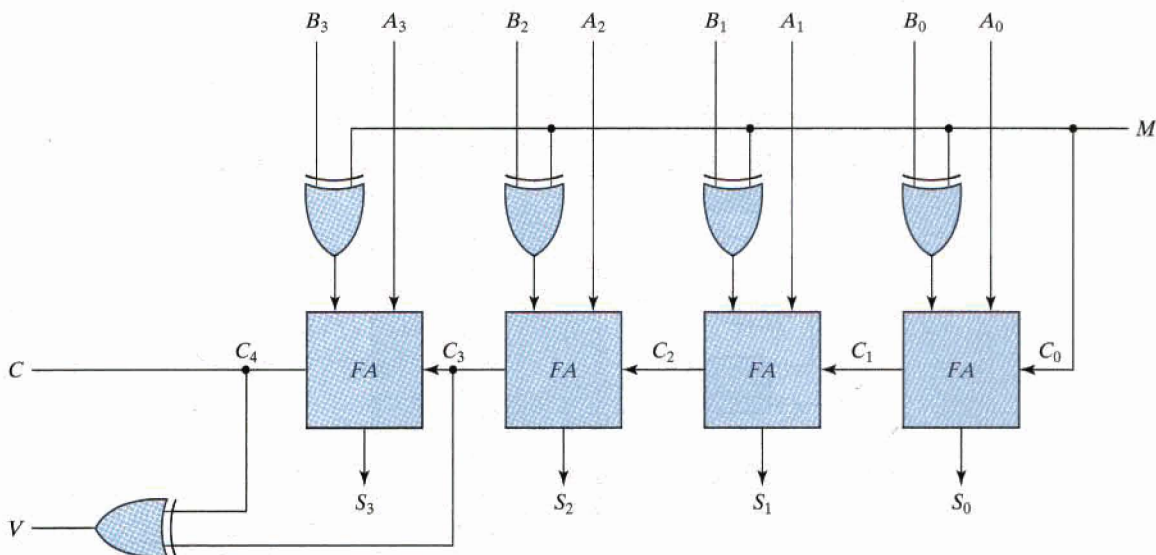


**FIGURE 4.12**  
Four-bit adder with carry lookahead

The circuit for subtracting  $A - B$  consists of an adder with inverters placed between each data input  $B$  and the corresponding input of the full adder. The input carry  $C_0$  must be equal to 1 when subtraction is performed. The operation thus performed becomes  $A$ , plus the 1's complement of  $B$ , plus 1. This is equal to  $A$  plus the 2's complement of  $B$ . For unsigned numbers, that gives  $A - B$  if  $A \geq B$  or the 2's complement of  $(B - A)$  if  $A < B$ . For signed numbers, the result is  $A - B$ , provided that there is no overflow. (See Section 1.6.)

The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder. A four-bit adder-subtractor circuit is shown in Fig. 4.13. The mode input  $M$  controls the operation. When  $M = 0$ , the circuit is an adder, and when  $M = 1$ , the circuit becomes a subtractor. Each exclusive-OR gate receives input  $M$  and one of the inputs of  $B$ . When  $M = 0$ , we have  $B \oplus 0 = B$ . The full adders receive the value of  $B$ , the input carry is 0, and the circuit performs  $A + B$ . When  $M = 1$ ,





**FIGURE 4.13**  
Four-bit adder-subtractor

we have  $B \oplus 1 = B'$  and  $C_0 = 1$ . The  $B$  inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation  $A$  plus the 2's complement of  $B$ . (The exclusive-OR with output  $V$  is for detecting an overflow.)

It is worth noting that binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as are unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently, depending on whether it is assumed that the numbers are signed or unsigned.

## Overflow

When two numbers with  $n$  digits each are added and the sum is a number occupying  $n + 1$  digits, we say that an overflow occurred. This is true for binary or decimal numbers, signed or unsigned. When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit by the width of the page to write down the sum. Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains  $n + 1$  bits cannot be accommodated by an  $n$ -bit word. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set that can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, two details are important: the leftmost bit always represents the sign, and negative

numbers are in 2's-complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example: Two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register. This is also true for -70 and -80. The two additions in binary are shown next, together with the last two carries:

carries:	0 1	carries:	1 0
+70	0 1000110	-70	1 0111010
+80	0 1010000	-80	1 0110000
+150	1 0010110	-150	0 1101010

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the 8-th bit) and the eight-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct. But since the answer cannot be accommodated within eight bits, we say that an overflow has occurred.

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred. This is indicated in the examples in which the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1. For this method to work correctly, the 2's complement of a negative number must be computed by taking the 1's complement and adding 1. This takes care of the condition when the maximum negative number is complemented.

The binary adder-subtractor circuit with outputs  $C$  and  $V$  is shown in Fig. 4.13. If the two binary numbers are considered to be unsigned, then the  $C$  bit detects a carry after addition or a borrow after subtraction. If the numbers are considered to be signed, then the  $V$  bit detects an overflow. If  $V = 0$  after an addition or subtraction, then no overflow occurred and the  $n$ -bit result is correct. If  $V = 1$ , then the result of the operation contains  $n + 1$  bits, but only the rightmost  $n$  bits of the number fit in the space available, so an overflow has occurred. The  $(n + 1)$ th bit is the actual sign and has been shifted out of position.

## 4.6 DECIMAL ADDER

Computers or calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in binary coded form. An adder for such a computer must employ arithmetic circuits that accept coded decimal numbers and present results in the same code. For binary addition, it is sufficient to consider a pair of significant bits together with a previous carry. A decimal adder requires a minimum of nine inputs and five outputs, since four bits are required to code each decimal digit and the circuit must have an input and output carry. There

is a wide variety of possible decimal adder circuits, depending upon the code used to represent the decimal digits. Here we examine a decimal adder for the BCD code. (See Section 1.7.)

### BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than  $9 + 9 + 1 = 19$ , the 1 in the sum being an input carry. Suppose we apply two BCD digits to a four-bit binary adder. The adder will form the sum in *binary* and produce a result that ranges from 0 through 19. These binary numbers are listed in Table 4.5 and are labeled by symbols  $K$ ,  $Z_8$ ,  $Z_4$ ,  $Z_2$ , and  $Z_1$ .  $K$  is the carry, and the subscripts under the letter  $Z$  represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The columns under the binary sum list the binary value that appears in the outputs of the four-bit binary adder. The output sum of two decimal digits must be represented in BCD and should appear in the form listed in the columns under "BCD Sum." The problem is to find a rule by which the binary sum is converted to the correct BCD digit representation of the number in the BCD sum.

In examining the contents of the table, it becomes apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain an invalid BCD representation.

**Table 4.5**  
*Derivation of BCD Adder*

Binary Sum					BCD Sum					Decimal
$K$	$Z_8$	$Z_4$	$Z_2$	$Z_1$	$C$	$S_8$	$S_4$	$S_2$	$S_1$	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

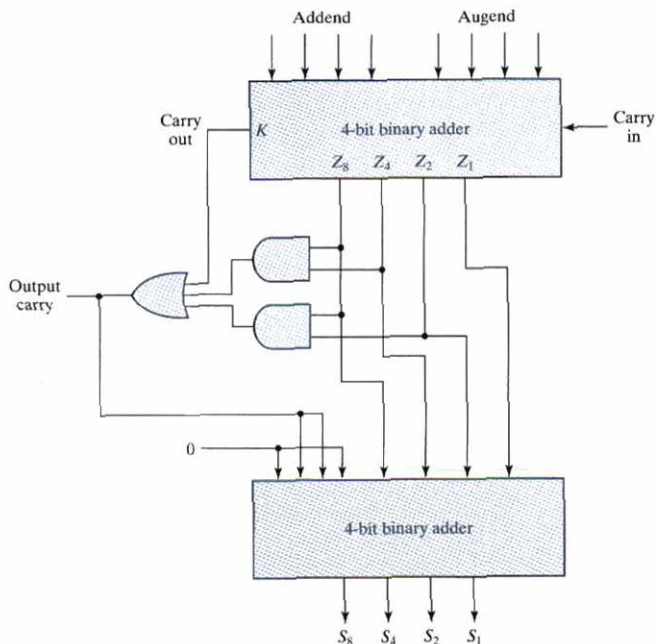
The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit that detects the necessary correction can be derived from the entries in the table. It is obvious that a correction is needed when the binary sum has an output carry  $K = 1$ . The other six combinations from 1010 through 1111 that need a correction have a 1 in position  $Z_8$ . To distinguish them from binary 1000 and 1001, which also have a 1 in position  $Z_8$ , we specify further that either  $Z_4$  or  $Z_2$  must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

When  $C = 1$ , it is necessary to add 0110 to the binary sum and provide an output carry for the next stage.

A BCD adder that adds two BCD digits and produces a sum digit in BCD is shown in Fig. 4.14. The two decimal digits, together with the input carry, are first added in the top four-bit adder to produce the binary sum. When the output carry is equal to 0, nothing is added to the binary sum.



**FIGURE 4.14**  
Block diagram of a BCD adder

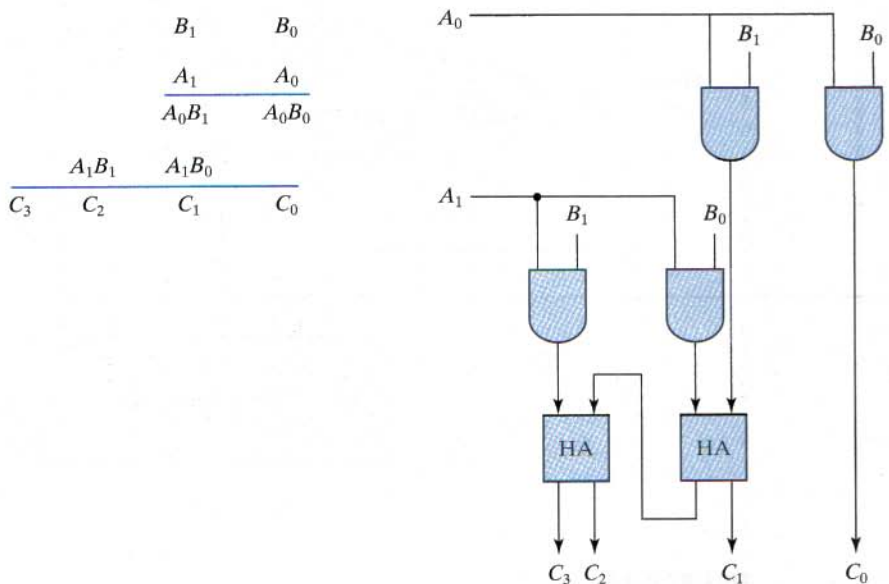


When it is equal to 1, binary 0110 is added to the binary sum through the bottom four-bit adder. The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. A decimal parallel adder that adds  $n$  decimal digits needs  $n$  BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher order stage.

## 4.7 BINARY MULTIPLIER

Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit. Each such multiplication forms a partial product. Successive partial products are shifted one position to the left. The final product is obtained from the sum of the partial products.

To see how a binary multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 4.15. The multiplicand bits are  $B_1$  and  $B_0$ , the multiplier bits are  $A_1$  and  $A_0$ , and the product is  $C_3C_2C_1C_0$ . The first partial product is formed by multiplying  $B_1B_0$  by  $A_0$ . The multiplication of two bits such as  $A_0$  and  $B_0$  produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation. Therefore, the partial product can be implemented with AND gates as shown in the diagram. The second partial product is formed by multiplying  $B_1B_0$  by  $A_1$  and shifting one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial

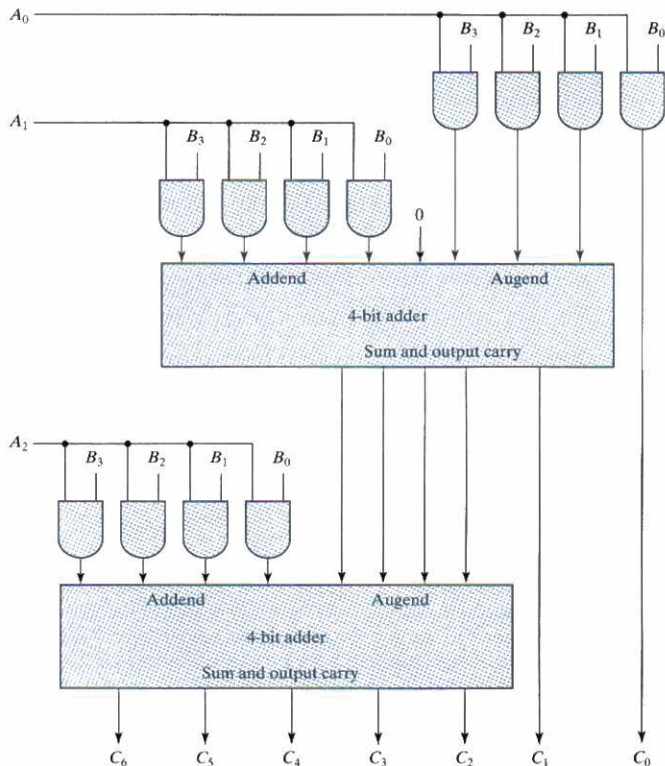


**FIGURE 4.15**  
Two-bit by two-bit binary multiplier

products. Note that the least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added with the partial product of the previous level to form a new partial product. The last level produces the product. For  $J$  multiplier bits and  $K$  multiplicand bits, we need  $(J \times K)$  AND gates and  $(J - 1) K$ -bit adders to produce a product of  $J + K$  bits.

As a second example, consider a multiplier circuit that multiplies a binary number represented by four bits by a number represented by three bits. Let the multiplicand be represented by  $B_3B_2B_1B_0$  and the multiplier by  $A_2A_1A_0$ . Since  $K = 4$  and  $J = 3$ , we need 12 AND gates and 2 four-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Fig. 4.16.



**FIGURE 4.16**  
Four-bit by three-bit binary multiplier

## 4.8 MAGNITUDE COMPARATOR

The comparison of two numbers is an operation that determines whether one number is greater than, less than, or equal to the other number. A *magnitude comparator* is a combinational circuit that compares two numbers  $A$  and  $B$  and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether  $A > B$ ,  $A = B$ , or  $A < B$ .

On the one hand, the circuit for comparing two  $n$ -bit numbers has  $2^{2n}$  entries in the truth table and becomes too cumbersome, even with  $n = 3$ . On the other hand, as one may suspect, a comparator circuit possesses a certain amount of regularity. Digital functions that possess an inherent well-defined regularity can usually be designed by means of an algorithm—a procedure which specifies a finite set of steps that, if followed, give the solution to a problem. We illustrate this method here by deriving an algorithm for the design of a four-bit magnitude comparator.

The algorithm is a direct application of the procedure a person uses to compare the relative magnitudes of two numbers. Consider two numbers,  $A$  and  $B$ , with four digits each. Write the coefficients of the numbers in descending order of significance:

$$A = A_3A_2A_1A_0$$

$$B = B_3B_2B_1B_0$$

Each subscripted letter represents one of the digits in the number. The two numbers are equal if all pairs of significant digits are equal:  $A_3 = B_3$ ,  $A_2 = B_2$ ,  $A_1 = B_1$ , and  $A_0 = B_0$ . When the numbers are binary, the digits are either 1 or 0, and the equality of each pair of bits can be expressed logically with an exclusive-NOR function as

$$x_i = A_iB_i + A_i'B_i' \quad \text{for } i = 0, 1, 2, 3$$

where  $x_i = 1$  only if the pair of bits in position  $i$  are equal (i.e., if both are 1 or both are 0).

The equality of the two numbers  $A$  and  $B$  is displayed in a combinational circuit by an output binary variable that we designate by the symbol  $(A = B)$ . This binary variable is equal to 1 if the input numbers,  $A$  and  $B$ , are equal, and is equal to 0 otherwise. For equality to exist, all  $x_i$  variables must be equal to 1, a condition that dictates an AND operation of all variables:

$$(A = B) = x_3x_2x_1x_0$$

The *binary* variable  $(A = B)$  is equal to 1 only if all pairs of digits of the two numbers are equal.

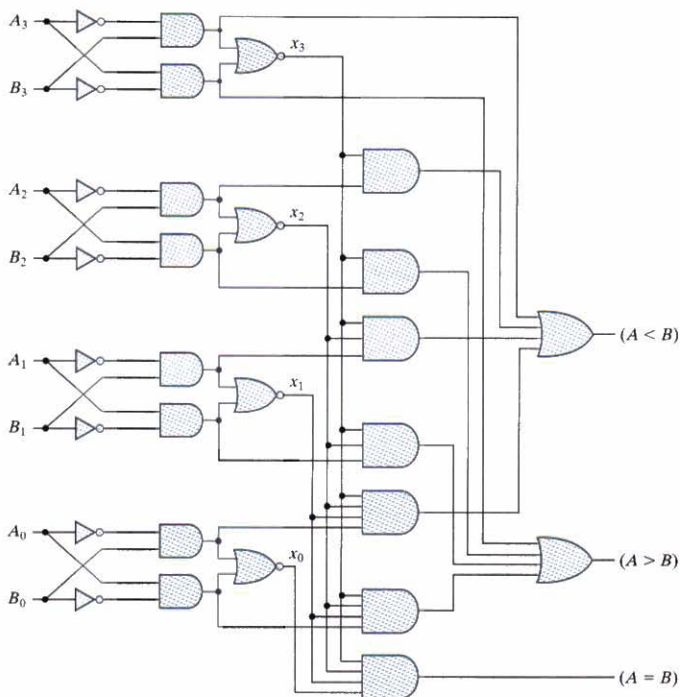
To determine whether  $A$  is greater or less than  $B$ , we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position. If the two digits of a pair are equal, we compare the next lower significant pair of digits. The comparison continues until a pair of unequal digits is reached. If the corresponding digit of  $A$  is 1 and that of  $B$  is 0, we conclude that  $A > B$ . If the corresponding digit of  $A$  is 0 and that of  $B$  is 1, we have  $A < B$ . The sequential comparison can be expressed logically by the two Boolean functions

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$

$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$$

The symbols  $(A > B)$  and  $(A < B)$  are *binary* output variables that are equal to 1 when  $A > B$  and  $A < B$ , respectively.

The gate implementation of the three output variables just derived is simpler than it seems because it involves a certain amount of repetition. The unequal outputs can use the same gates that are needed to generate the equal output. The logic diagram of the four-bit magnitude comparator is shown in Fig. 4.17. The four  $x$  outputs are generated 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.



**FIGURE 4.17**  
Four-bit magnitude comparator

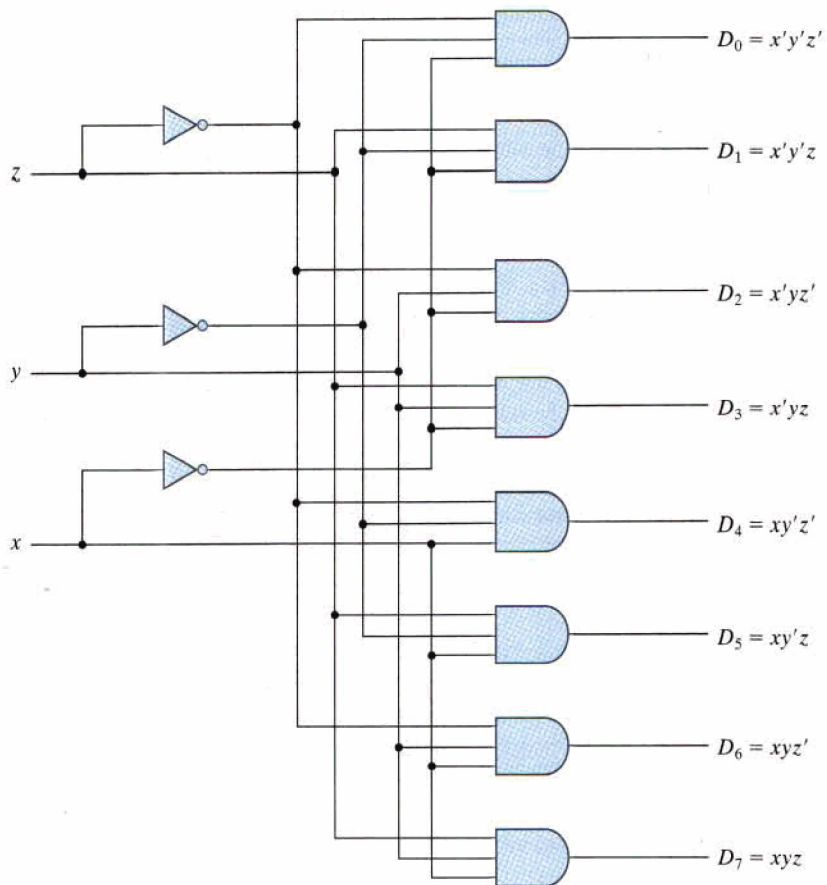


## 4.9 DECODERS

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. 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



**FIGURE 4.18**  
Three-to-eight-line decoder

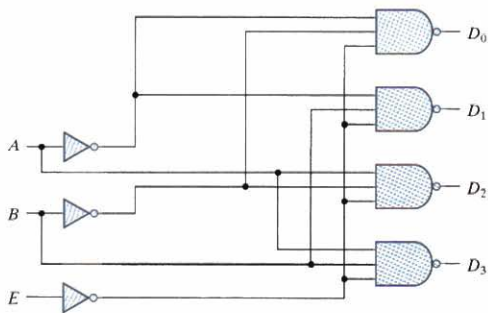
**Table 4.6**  
*Truth Table of a Three-to-Eight-Line Decoder*

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

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.

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



(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	0

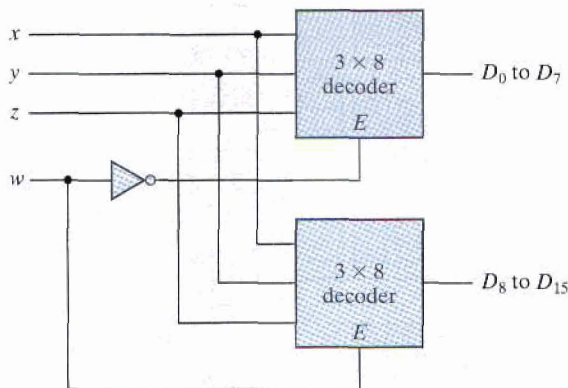
(b) Truth table

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

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 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 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.



**FIGURE 4.20**

4 × 16 decoder constructed with two 3 × 8 decoders

## 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.

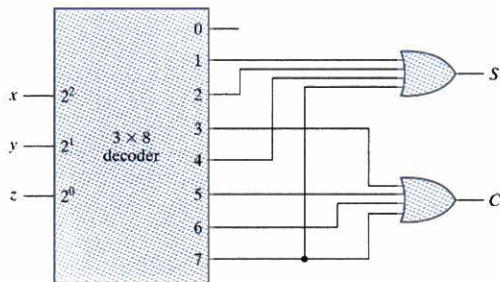
From the truth table of the full adder (see Table 4.4), we obtain the functions for the combinational circuit in sum-of-minterms form:

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

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

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.

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.



**FIGURE 4.21**  
Implementation of a full adder with a decoder



## 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.

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.

**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

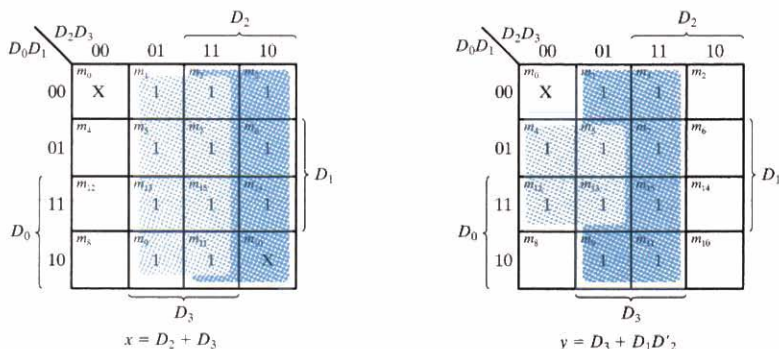
**Table 4.8**  
*Truth Table of a Priority Encoder*

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

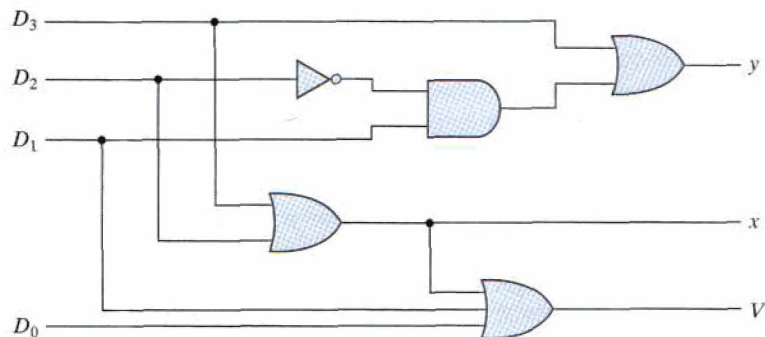
## Priority Encoder

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



**FIGURE 4.22**  
 Maps for a priority encoder



**FIGURE 4.23**  
Four-input priority encoder

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.

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:

$$x = D_2 + D_3$$

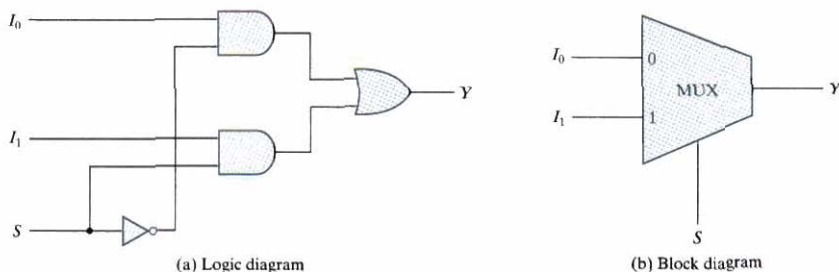
$$y = D_3 + D_1 D_2'$$

$$V = D_0 + D_1 + D_2 + D_3$$

## 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.

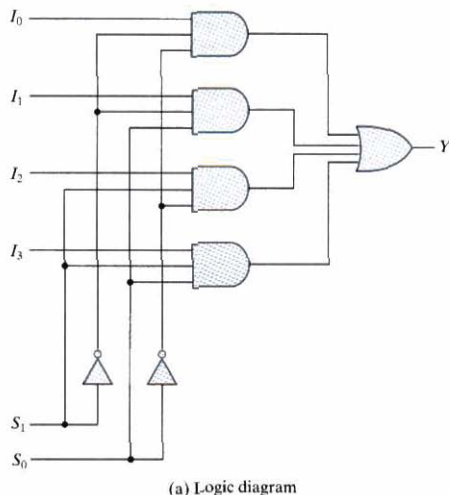
A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Fig. 4.24. The circuit has two data input lines, one output line, and one selection line  $S$ . When  $S = 0$ , the upper AND gate is enabled and  $I_0$  has a path to the output. When  $S = 1$ , the lower AND gate is enabled and  $I_1$  has a path to the output. The multiplexer acts like



**FIGURE 4.24**  
Two-to-one-line multiplexer

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



$S_1$	$S_0$	$Y$
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



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 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.

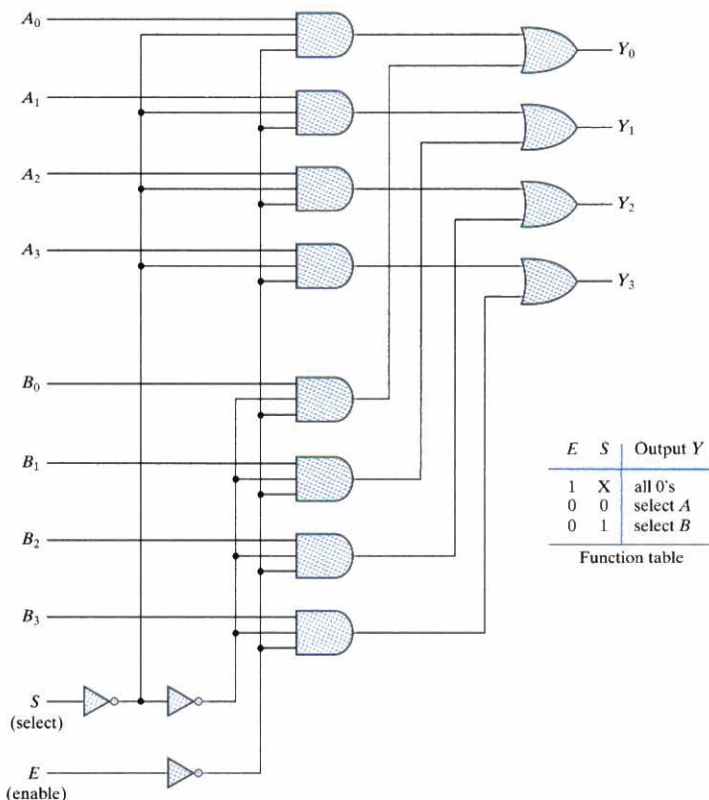
Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. As an illustration, a quadruple 2-to-1-line multiplexer is shown in Fig. 4.26. The circuit has four multiplexers, each capable of selecting one of two input lines. Output  $Y_0$  can be selected 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 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

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

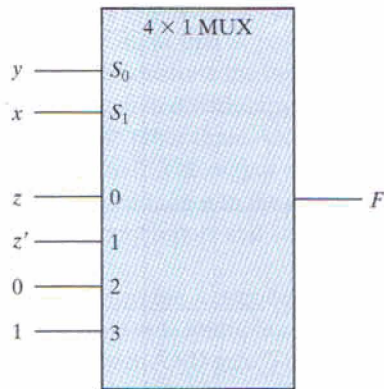


**FIGURE 4.26**  
Quadruple two-to-one-line multiplexer

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.

$x$	$y$	$z$	$F$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(a) Truth table



(b) Multiplexer implementation

**FIGURE 4.27**

Implementing a Boolean function with a multiplexer

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

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

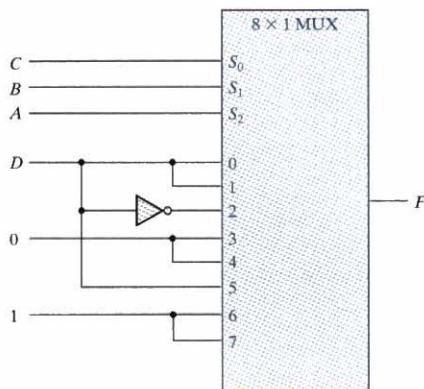
This function is implemented with a multiplexer with three selection inputs as shown in Fig. 4.28. Note that the first variable  $A$  must be connected to selection input  $S_2$  so that  $A$ ,  $B$ , and  $C$  correspond to selection inputs  $S_2$ ,  $S_1$ , and  $S_0$ , respectively. The values for the 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., 5 volts).

### 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



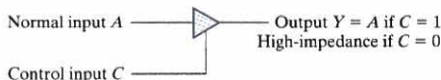
A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



**FIGURE 4.28**  
Implementing a four-input function with a multiplexer

(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 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.

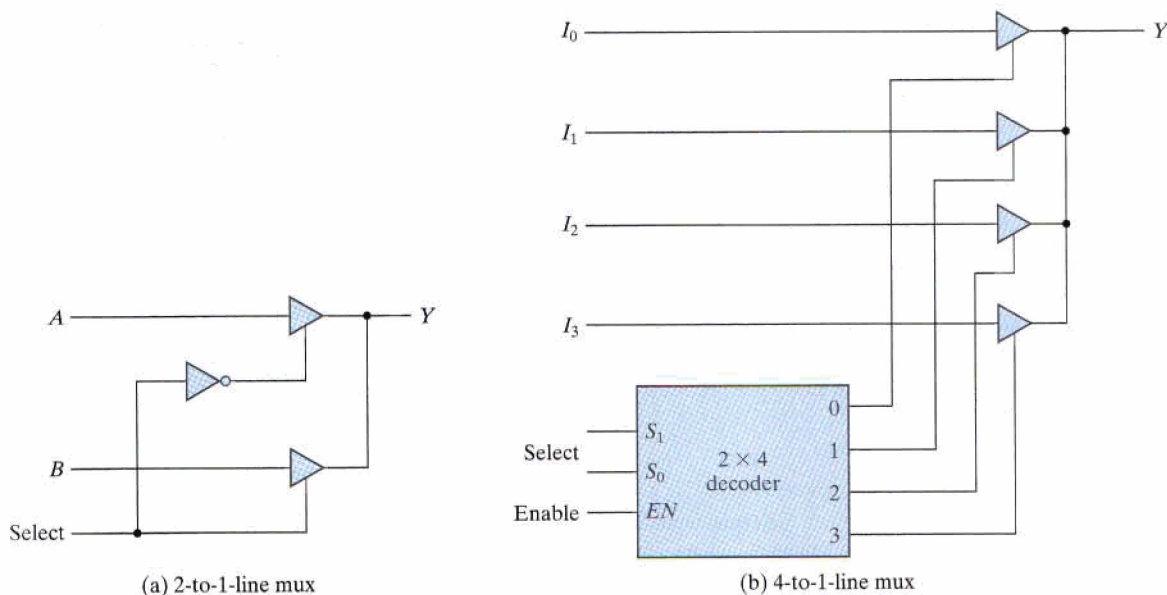


**FIGURE 4.29**  
Graphic symbol for a three-state buffer



The construction of multiplexers with three-state buffers is demonstrated in Fig. 4.30. Part (a) of the figure 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  $I_3$  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.



**FIGURE 4.30**  
Multiplexers with three-state gates

## 4.12 HDL MODELS OF COMBINATIONAL CIRCUITS

---

The Verilog hardware description language (HDL) was introduced in Section 3.10. In the current section, we present more elaborate examples and compare alternative descriptions of combinational circuits in Verilog. Sequential circuits are presented in the next chapter. 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. 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 consider switch-level modeling briefly in Section 10.10.

### 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 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

**Table 4.9**  
*Truth Table for Predefined Primitive Gates*

and	0	1	x	z	or	0	1	x	z
0	0	0	0	0	0	0	1	x	x
1	0	1	x	x	1	1	1	1	1
x	0	x	x	x	x	x	1	x	x
z	0	x	x	x	z	x	1	x	x

xor	0	1	x	z	not	input	output
0	0	1	x	x		0	1
1	1	0	x	x		1	0
x	x	x	x	x		x	x
z	x	x	x	x		z	x

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 *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 (leftmost) 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**

---

```
// 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 two half adders. The third module describes the four-bit adder by instantiating 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**


---

```

// 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

```

---

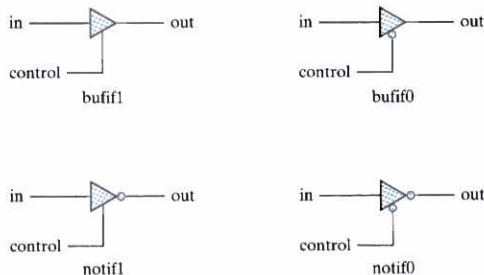
HDL Example 4.2 illustrates Verilog 2001, 2005 syntax, which eliminates extra typing of identifiers declaring the mode (e.g., **output**), type (**reg**), and declaration of a vector range (e.g., [3: 0]). The first version of the standard (1995) uses separate statements for these declarations.

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. 4.31. 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 4.31**  
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. 4.32.

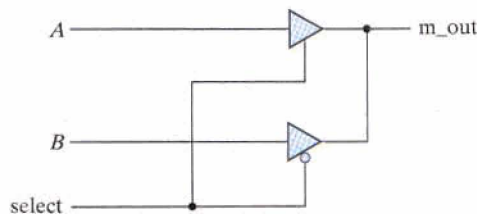
The HDL description must use a **tri** data type for the output:

```
// 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. 3.28). The nets **supply1** and **supply0** represent power supply and ground, respectively. They are used to hard-wire an input of a device to either 1 or 0.



**FIGURE 4.32**  
Two-to-one-line multiplexer with three-state buffers

## Dataflow Modeling

Dataflow modeling of combinational logic uses a number of operators that act on operands to produce desired results. 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. 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.

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 is declared explicitly by a net keyword (e.g., **wire**) or by declaring an identifier to be an output 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 an expression that uses operands and operators. As an example, assuming that the variables were declared, a two-to-one-line multiplexer with data inputs *A* and *B*, select input *S*, and output *Y* is described with the continuous assignment

$$\text{assign } Y = (A \& S) | (B \& \sim S);$$

**Table 4.10**  
*Some Verilog HDL Operators*

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



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 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 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 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!

#### HDL Example 4.5

---

```
// 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. 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 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 procedural assignment statements 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 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 = 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.

#### HDL Example 4.8

---

```
// 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'b01* 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. 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.

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:

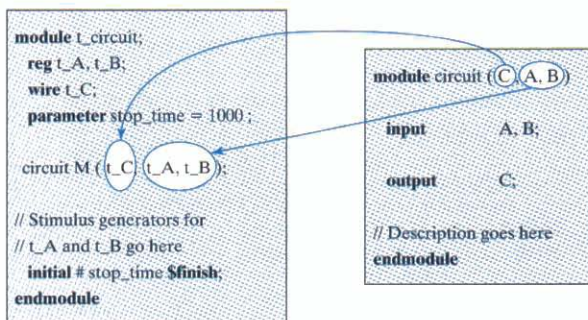
```
module test_module_name;
    // Declare local reg and wire identifiers.
    // Instantiate the design module under test.
    // Specify a stopwatch, using $finish to terminate the simulation.
```

```
// 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 4.33 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 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.



**FIGURE 4.33**  
Interaction between stimulus and design modules

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

*Task-name (format specification, argument list);*

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 B = %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 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, accurate method of analyzing combinational circuits to verify that they operate properly. 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*<sub>1</sub> and *F*<sub>2</sub> are declared as **wire**. The complement of *F*<sub>2</sub> 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. 4.33. 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 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 M_F4_32 (D[2], D[1], D[0], F1, F2);
    initial
        begin
            D = 3'b000;
            repeat (7) #10 D = D + 1'b1;
        end
    initial
        $monitor ("ABC = %b F1 = %b F2 = %b ", D, F1, F2);
endmodule

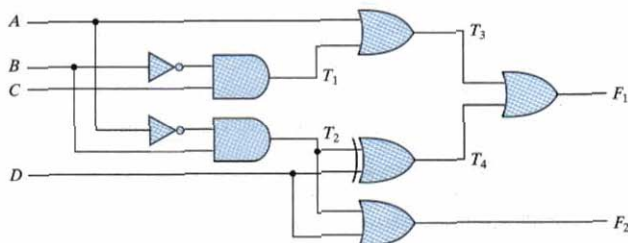
Simulation log: ABC = 000 F1 = 0 F2 = 0
ABC = 001 F1 = 1 F2 = 0 ABC = 010 F1 = 1 F2 = 0
ABC = 011 F1 = 0 F2 = 1 ABC = 100 F1 = 1 F2 = 0
ABC = 101 F1 = 0 F2 = 1 ABC = 110 F1 = 0 F2 = 1
ABC = 111 F1 = 1 F2 = 1
```

---

## PROBLEMS

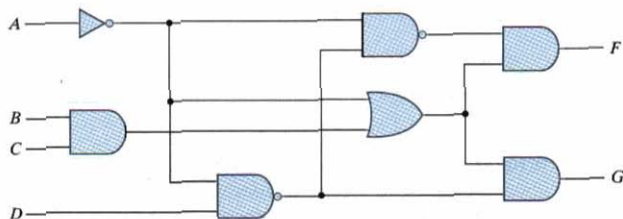
Answers to problems marked with \* appear at the end of the book. Where appropriate, a logic design and its related HDL modeling problem are cross referenced.

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



**FIGURE P4.1**

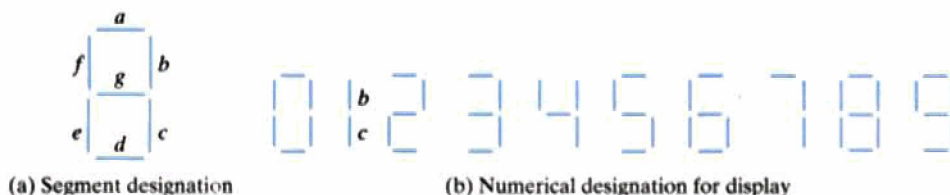
- Derive the Boolean expressions for  $T_1$  through  $T_4$ . Evaluate the outputs  $F_1$  and  $F_2$  as a function of the four inputs.
  - List the truth table with 16 binary combinations of the four input variables. Then list the binary values for  $T_1$  through  $T_4$  and outputs  $F_1$  and  $F_2$  in the table.
  - Plot the Boolean output 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 outputs  $F$  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),
- Write the Boolean functions for the four outputs in terms of the input variables.
  - If the circuit is listed 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.
- The output is 1 when the binary value of the inputs is less than 3. The output is 0 otherwise.
  - The output is 1 when the binary value of the inputs is an odd 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 two greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is three 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.
- Design a three-input majority circuit by finding the circuit's truth table, Boolean equation, and a logic diagram.
  - Write and verify a Verilog dataflow model of the circuit.
- 4.7** Design a combinational circuit that converts a four-bit Gray code (Table 1-6) to a four-bit binary number.
- Implement the circuit with exclusive-OR gates.
  - 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 the 8, 4, -2, -1 code to BCD (see Table 1.5). (HDL — see Problem 4.50.)
- 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.)

**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).
- Design a four-bit combinational circuit incrementer (a circuit that adds 1 to a four-bit binary number).
  - Design a four-bit combinational circuit decrements (a circuit that subtracts 1 from a four-bit binary number).
- 4.12** (a) 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  $Diff$  and the borrow in  $B_{out}$ .

- (b)\* 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

$$P_i = A_i + B_i$$

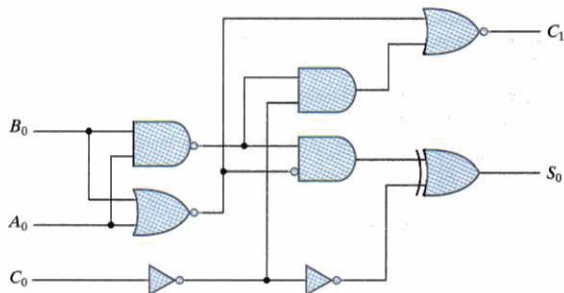
$$G_i = A_i B_i$$

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

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

$$S_i = (P_i G_i') \oplus C_i$$

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.



**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

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

IC type 74182 is a lookahead carry generator circuit that generates the carries with AND-OR-INVERT (AOI) gates (see Section 3-8.) The circuit assumes that the input terminals have the complements of the  $G$ 's, the  $P$ 's, and of  $C_i$ . 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 BCD digit. (HDL — see Problem 4.54.)
- 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** A binary multiplier 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 four-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 two-to-four-line decoder using (a) NOR gates only, and (b) NAND gates only. Include an enable input.
- 4.24** Design a BCD-to-decimal decoder using the unused combinations of the BCD code as don't-care conditions. (HDL — see Problem 4.60.)
- 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.
- 4.26** Construct a 4-to-16-line decoder with five 2-to-4-line decoders with enable.
- 4.27** A combinational circuit is specified by the following three Boolean functions:

$$F_1(A, B, C) = \Sigma(3, 5, 6)$$

$$F_2(A, B, C) = \Sigma(1, 4)$$

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

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 circuit defined by the following three Boolean functions:

$$(a) F_1 = x'y'z' + xz$$

$$F_2 = xy'z' + x'y$$

$$F_3 = x'y'z + xy$$

$$(b) F_1 = (y' + x)z$$

$$F_2 = y'z' + xy' + yz'$$

$$F_3 = (x' + y)z$$

- 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. (HDL—see Problem 4.57.)
- 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?
- 4.31** Construct a  $16 \times 1$  multiplexer with two  $8 \times 1$  and one  $2 \times 1$  multiplexers. Use block diagrams.
- 4.32** Implement the following Boolean function with a multiplexer (HDL—see Problem 4.46):
- $F(A, B, C, D) = \Sigma(0, 2, 5, 7, 11, 14)$
  - $F(A, B, C, D) = \Pi(3, 8, 12)$
- 4.33** Implement a full adder with two  $4 \times 1$  multiplexers.
- 4.34** An  $8 \times 1$  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:
- $I_1 = I_2 = I_7 = 0$ ;  $I_3 = I_5 = 1$ ;  $I_0 = I_4 = D$ ; and  $I_6 = D'$ .
  - $I_1 = I_2 = 0$ ;  $I_3 = I_7 = 1$ ;  $I_4 = I_5 = D$ ; and  $I_0 = I_6 = D'$ .
- Determine the Boolean function that the multiplexer implements.
- 4.35** Implement the following Boolean function with a  $4 \times 1$  multiplexer and external gates.
- $F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$
  - $F(A, B, C, D) = \Sigma(1, 2, 4, 7, 8, 9, 10, 11, 13, 15)$
- 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  $11$ . The functions may have to be implemented with external gates and with connections to power and ground.
- 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. (See Problems 4.13 and 4.40.)
- 4.38** Write the HDL dataflow description of a quadruple two-to-one-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 is for “not equal to,” bit 3 is for “greater than,” bit 2 is for “less than,” bit 1 for “greater than or equal to,” 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 a cyclic behavior.
- 4.42**
- Write an HDL gate-level description of the BCD-to-excess-3 converter circuit shown in Fig. 4.4 (see Problem 4.22).
  - 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*), eight-bit input datapaths (*A*[7: 0] and *B*[7: 0]), an eight-bit output datapath (*y*[7: 0]), and performs the arithmetic and logical operations listed below

Sel	Operation	Description
000	$y = 8'b0$	
001	$y = A \& B$	Bitwise and
010	$y = A   B$	Bitwise or
011	$y = A \wedge B$	Bitwise exclusive or
100	$y = A + B$	Add (Assume A and B are unsigned)
101	$y = A - B$	Subtract
110	$y = \sim A$	Bitwise complement
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** Repeat Problem 4.32, using a dataflow description.
- 4.47** Repeat Problem 4.37, using a dataflow description.
- 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 in Problem 4.1.
- 4.50\*** Using a case statement, develop and simulate a behavioral model of the 84-2-1 to BCD code converter described in Problem 4.8.
- 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 decrements 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 BCD digit (see Problem 4.18).

- 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 four-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 vacated positions with the bit that was in the MSB before the shift occurred (shift arithmetic right).
- 4.59 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 vacated positions with 0s (shift logical left).
- 4.60 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).
- 4.61 Using the port syntax of the IEEE 1364-2001 standard, write and verify a gate-level model of the 4-bit even parity checker shown in Fig. 3.36.
- 4.62 Using continuous assignment statements and the port syntax of the IEEE 1364-2001 standard, write and verify an HDL model of the 4-bit even parity checker shown in Fig. 3.36.

## 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*, 3d ed. Boston: Allyn Bacon.
5. GAUSKI, 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. 2000. *Logic and Computer Design Fundamentals*, 2d 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. 1992. *Fundamentals of Logic Design*, 4th ed. St. Paul, MN: West.
12. THOMAS, D. E., and P. R. MOORBY. 1998. *The Verilog Hardware Description Language*, 4th ed. Boston: Kluwer Academic Publishers.
13. WAKERLY, J. F. 2000. *Digital Design: Principles and Practices*, 3d ed. Upper Saddle River, NJ: Prentice Hall.