

# **Chapter 4 Combinational Logic**

# CHAPTER OBJECTIVES

1. Given its logic diagram, know how to analyze a combinational logic circuit.
2. Understand the functionality of a half adder and a full-adder.
3. Understand the concepts of overflow and underflow.
4. Understand the implementation of a binary adder.
5. Understand the implementation of a binary coded decimal (BCD) adder.
6. Understand the implementation of a binary multiplier.
7. Understand fundamental combinational logic circuits: decoder, encoder, priority encoder, multiplexer, and three-state gate.
8. Know how to implement a Boolean function with a multiplexer.
9. Understand the distinction between gate-level, dataflow, and behavioral modeling with HDLs.
10. Be able to write a gate-level Verilog or VHDL model of a fundamental logic circuit.
11. Be able to write a hierarchical hardware description language (HDL) model of a combinational logic circuit.
12. Be able to write a dataflow model of a fundamental combinational logic circuit.
13. Be able to write a Verilog continuous assignment statement, or a VHDL signal assignment statement.
14. Know how to declare a Verilog procedural block, or a VHDL process.

15. Be able to write a simple testbench.

## 4.1 INTRODUCTION

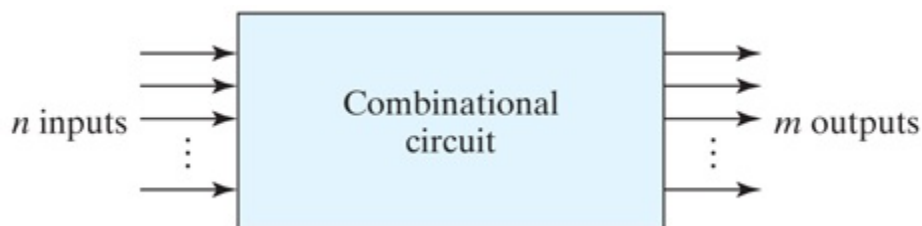
Logic circuits for digital systems may be combinational or sequential. A logic circuit is *combinational* if its outputs at any time are a function of only the present inputs [1–5]. A combinational circuit performs an operation that can be specified logically by a set of Boolean functions. In contrast, 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 to the circuit, the outputs of a *sequential* circuit depend at any time on not only the 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 more detail in [Chapters 5](#) and [8](#).

## 4.2 COMBINATIONAL CIRCUITS

A combinational circuit consists of an interconnection of logic gates. 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 input signals acting on the internal combinational logic circuit, and go to an external destination. Each input and output variable exists physically as an analog signal [1](#) whose values are interpreted to be a binary signal that represents logic 0 and logic 1. (Note: Logic simulators display only 0's and 1's, not the actual analog signals.) In many applications, the source and destination of the signals are storage registers. [2](#) If the circuit includes storage registers with the combinational gates, then the total circuit must be considered to be a sequential circuit.

[1](#) Typically a voltage.

[2](#) See Section 1.8.



**FIGURE 4.1**

Block diagram of combinational circuit

For  $n$  input variables, there are  $2^n$  possible combinations of the binary inputs. For each possible input combination, there is one possible value for each output variable. Thus, a combinational circuit can be specified with a truth table [3](#) that lists the output values for each combination of input

variables. A combinational circuit can also be described by  $m$  Boolean functions, one for each output variable. Each output function is expressed in terms of the  $n$  input variables.

[3](#) See Section 1.9.

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 the 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. This chapter uses the knowledge acquired in previous chapters to formulate systematic analysis and design procedures for combinational circuits. Knowing how to work systematically will make efficient use of your time. 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 synthesizable 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 OF COMBINATIONAL CIRCUITS

Analysis of a combinational circuit determines its *functionality*, that is, *the logic 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 of a circuit is to make sure that it is combinational and not sequential. **The logic 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 whose output forms part of the input to the first gate. Feedback paths in a digital circuit define a sequential circuit and must be analyzed by special methods and will not be considered here.

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 of a combinational circuit from its logic diagram, we proceed as follows:

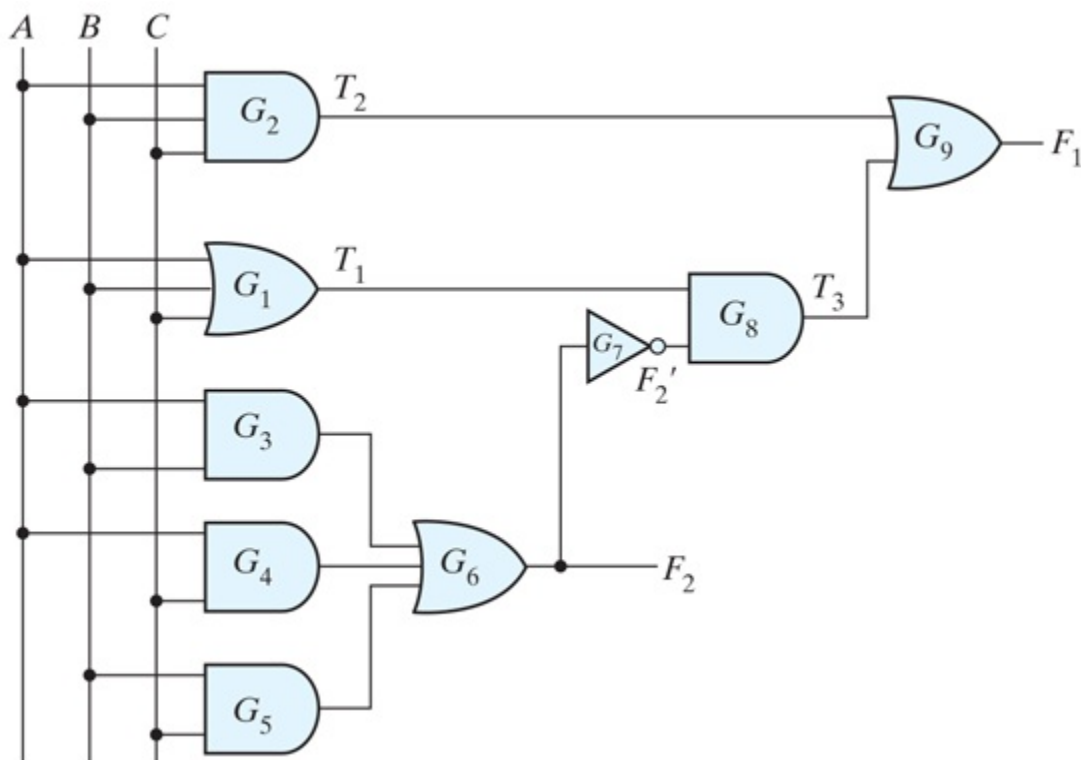
1. With arbitrary, but meaningful, symbols, label the outputs of all gates whose inputs include at least one input of the circuit. Determine the Boolean functions for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. 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.

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

$$F_2 = A B + A C + B C \quad T_1 = A + B + C \quad T_2 = A B C$$



**FIGURE 4.2**

Logic diagram for analysis example

[Description](#)



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 inputs  $A$ ,  $B$ , and  $C$ , we form a series of substitutions as follows:

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

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 whose set of inputs consists of only inputs to the circuit.
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  $F_2$ . 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 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.

**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

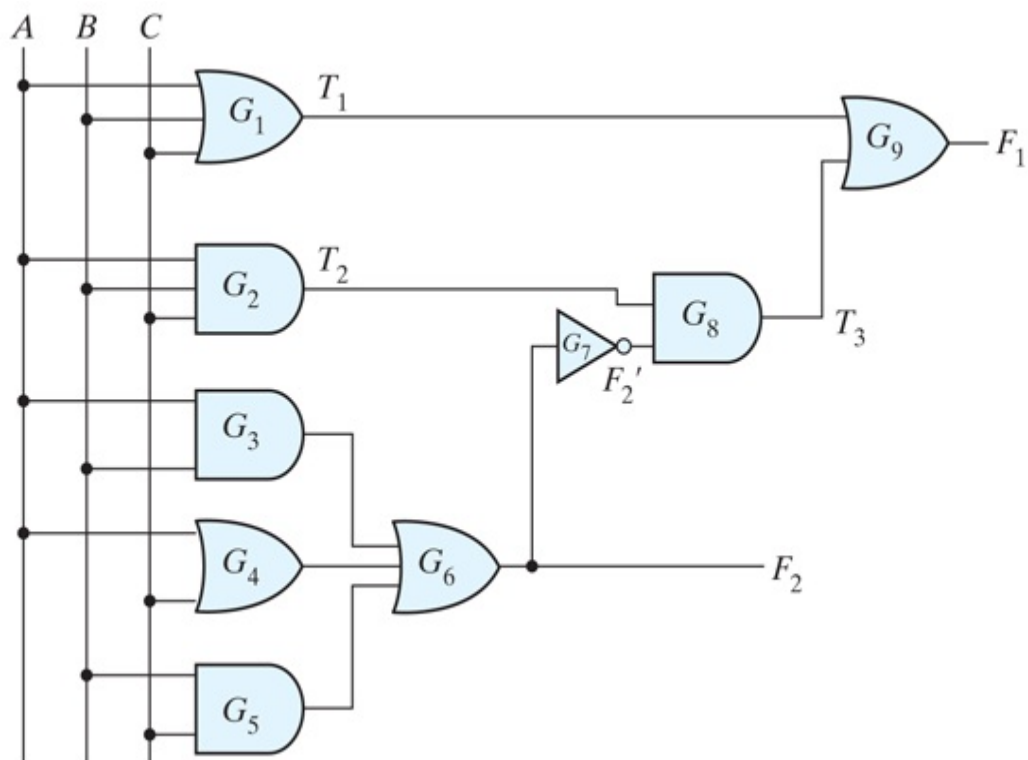
1 1 0 1 0 1 0 0 0

1 1 1 1 0 1 1 0 1

Another way of analyzing a combinational circuit is by means of logic simulation. This is not always 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. Simulation will require developing an HDL model of a circuit.

## Practice Exercise 4.1

1. Analyze the logic diagram in [Fig. PE4.1](#) and find the Boolean expressions for  $F_1$  and  $F_2$ .



**FIGURE PE4.1**

[Description](#)

**Answer:**  $T_1 = A + B + C$

$$T_2 = A B C$$

$$F_2 = A B + A + B + B C = A + B + B C = A + B$$

$$F_2' = A' B'$$

$$T_3 = (A B C)(A' B') = 0$$

$$F_1 = T_1 = A + B + C$$

## 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 [4–7]. The procedure involves the following steps [4–7]:

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 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, the converter must have 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 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 in actual operation of the circuit. Therefore, we are at liberty to assign to the output variables either a 1 or a 0, whichever gives a simpler circuit.

4 An excess-3 code is obtained from the corresponding binary value plus 3. For example, the excess-3 code for 2 10 is the binary code for 5 10 that is, 0101.

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

**Input BCD Output Excess-3 Code**

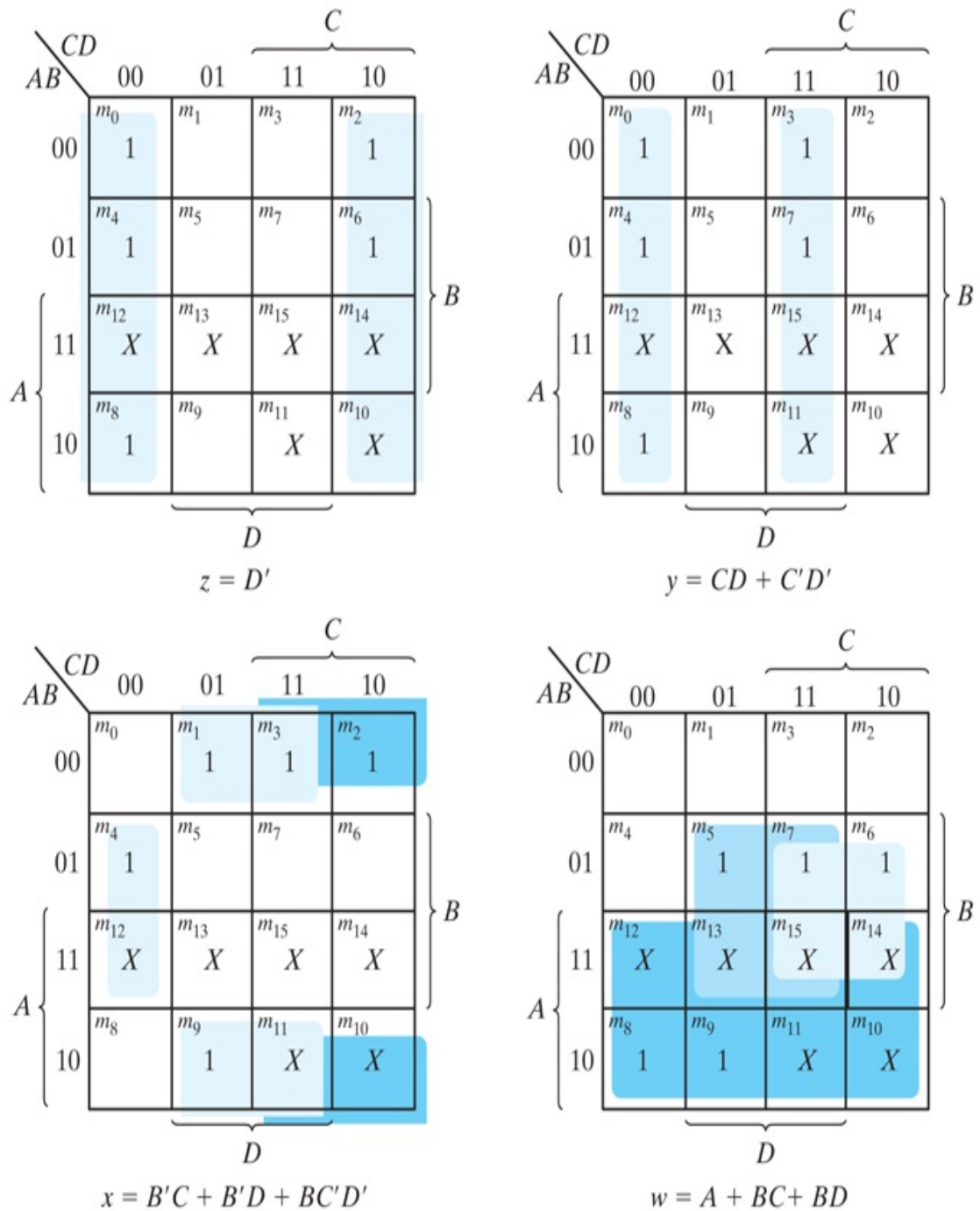
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0

1 0 0 0    1    0    1    1

1 0 0 1    1    1    0    0

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

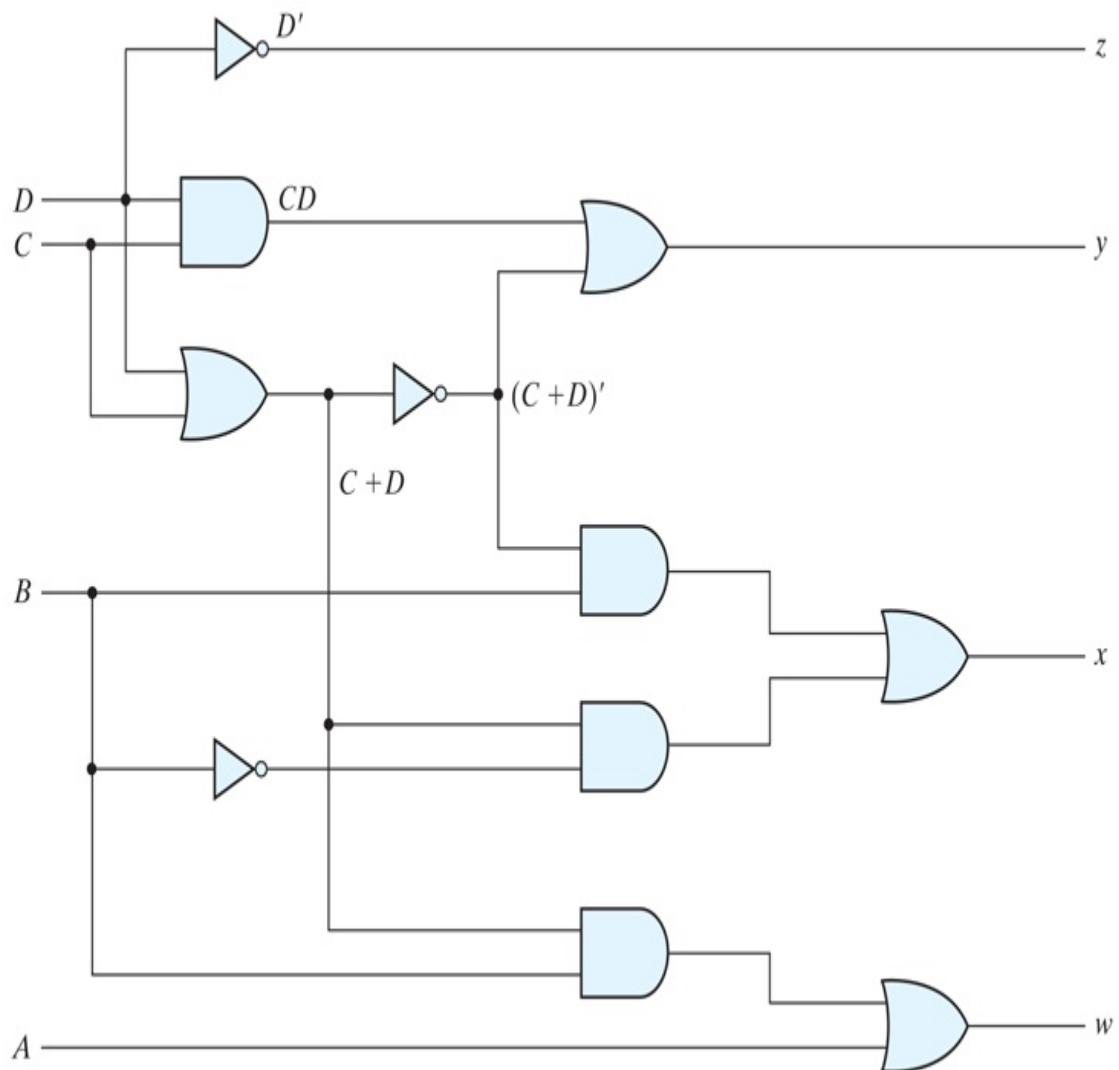
#### [Description](#)

A two-level logic diagram for each output 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:

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

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.



**FIGURE 4.4**

## Logic diagram for BCD-to-excess-3 code converter

### [Description](#)

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.

In general, multilevel logic circuits exploit subcircuits that can be used to form more than one output. Here,  $(C + D)$  is used in forming  $x$ ,  $y$ , and  $w$ . The result is a circuit with fewer gates. Logic synthesis tools automatically find and exploit subcircuits that are used by multiple outputs.

## 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. [5](#) 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.

5 The carry ( $C$ ) bit is the most significant bit: the sum ( $S$ ) bit is the least significant bit.

## Table 4.3 *Half Adder*

$x \ Y \ C \ S$

0 0 0 0

0 1 0 1

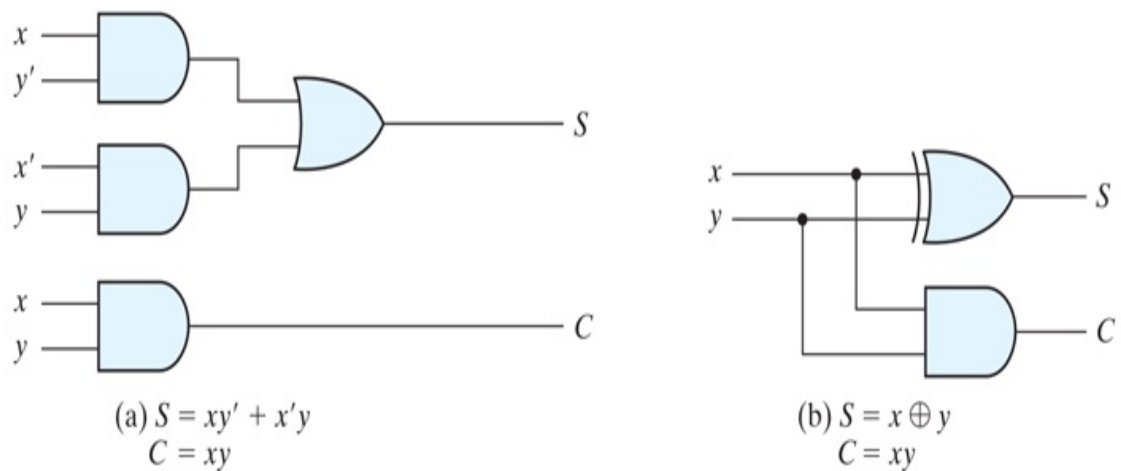
1 0 0 1

1 1 1 0

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

$$S = x' y + x y' \quad C = x y$$

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



**FIGURE 4.5**

Implementation of half adder

[Description](#)

## Full Adder

Addition of  $n$ -bit binary numbers requires the use of a full adder, and the process of addition proceeds on a bit-by-bit basis, right to left, beginning with the least significant bit. After the least significant bit, addition at each position not only adds the respective bits of the words, but must also consider a possible carry bit from addition at the previous position.

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 decimal value from 0 to 3, and binary representation of the decimal digits 2 or 3 needs two bits. 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 formed by adding the input carry and the bits of the words. 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.

## 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

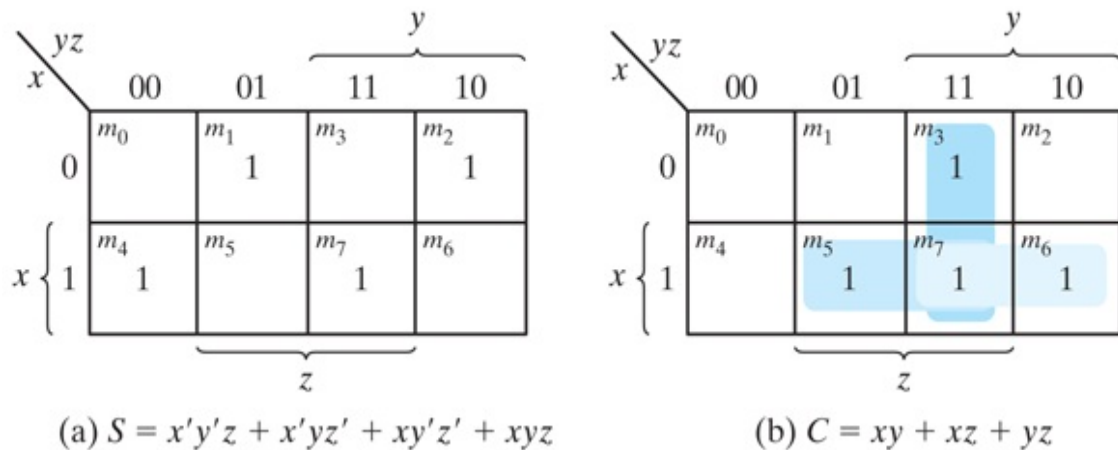
1 1 0 1 0

1 1 1 1 1

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 K-maps for the outputs of the full adder are shown in [Fig. 4.6](#). The simplified expressions are

$$S = x' y' z + x' y z' + x y' z' + x y z \quad C = x y + x z + y z$$



# FIGURE 4.6

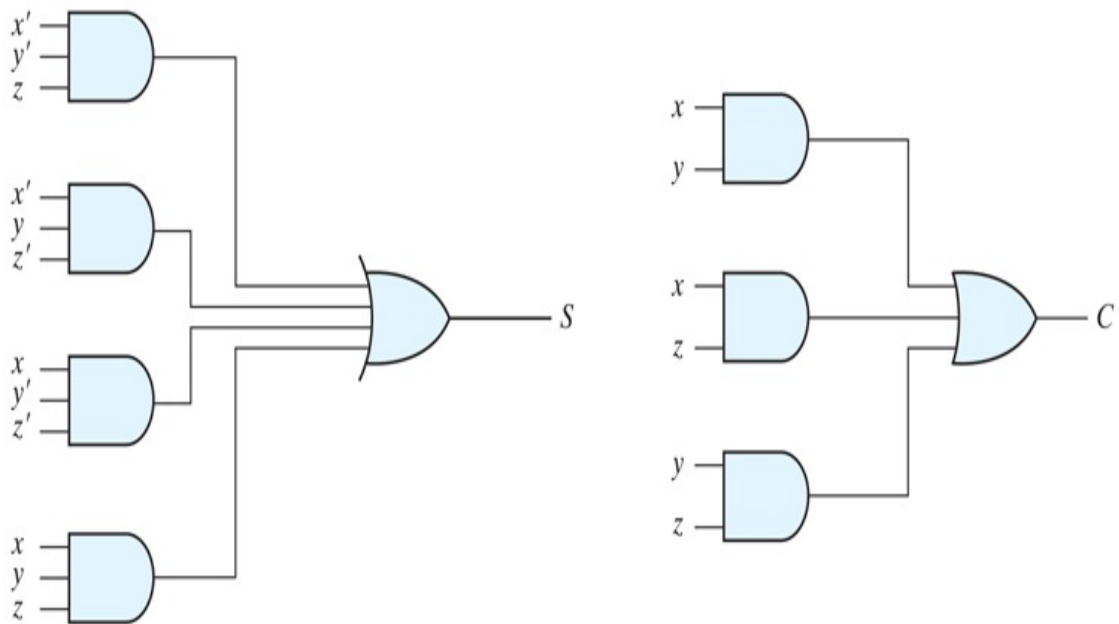
K-Maps for full adder

## Description

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

$$S = z \oplus (x \oplus y) = z' (x y' + x' y) + z (x y' + x' y)' = z' (x y' + x' y) + z (x y + x' y') = x y' z' + x' y z' + x y z + x' y' z$$

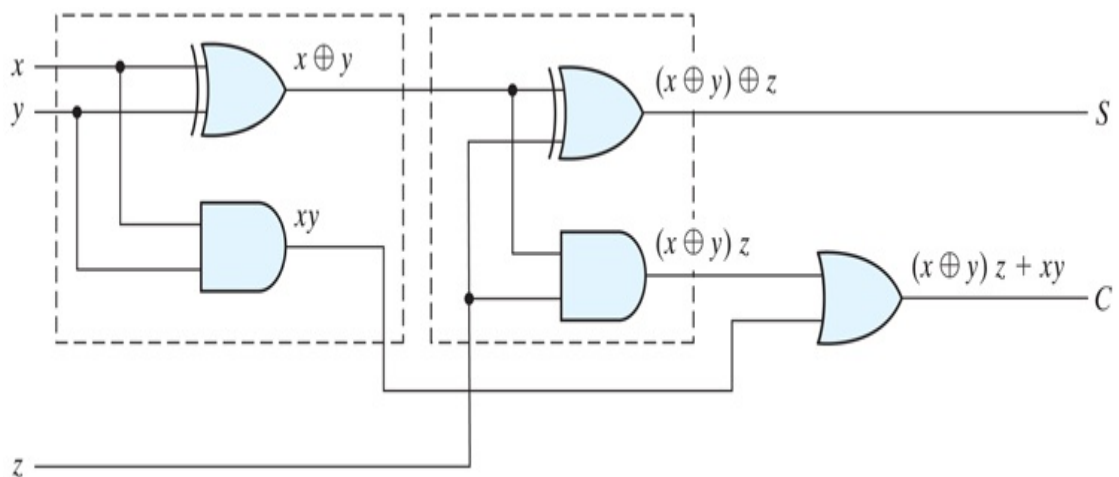




**FIGURE 4.7**

Implementation of full adder in sum-of-products form

[Description](#)



**FIGURE 4.8**

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

### Description

The carry output is

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

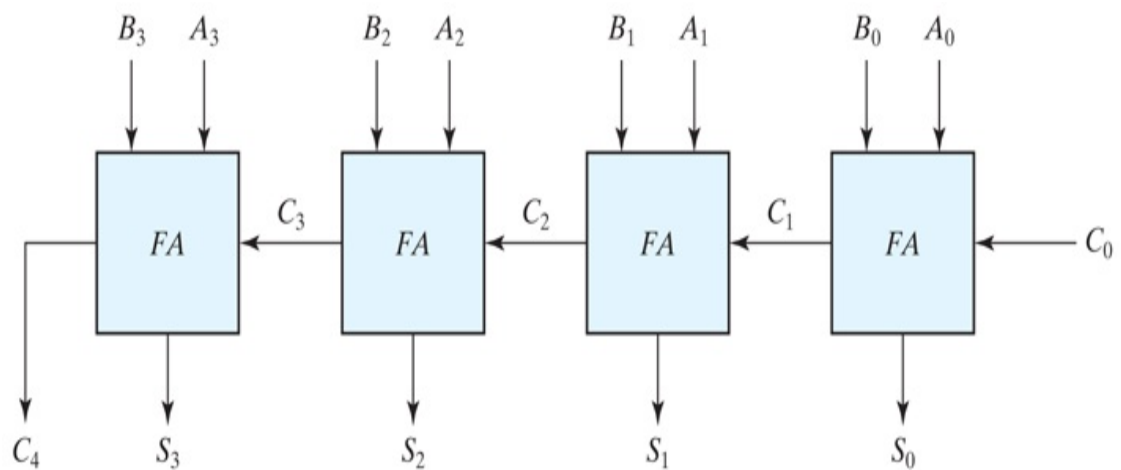
## Practice Exercise 4.2

1. Explain how a half adder and a full adder differ in their functionality.

**Answer:** A half adder adds only two (data) bits to produce a sum and carry-out bit. A full adder adds three input bits (two data bits and a carry-in bit) to produce a sum and carry-out bit.

## 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. Addition of  $n$ -bit numbers requires a chain of  $n$  full adders or a chain of one half adder and  $n - 1$  full adders. In the former case, the input carry to the least significant position is fixed at 0. [Figure 4.9](#) shows the connection 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.



**FIGURE 4.9**

Four-bit adder

### Description

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<sub>*i*</sub>

Augend        1 0 1 1    A<sub>*i*</sub>

Addend       0 0 1 1    B<sub>*i*</sub>

Sum            1 1 1 0    S<sub>*i*</sub>

Output carry 0 0 1 1 C<sub>*i* + 1</sub>

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.

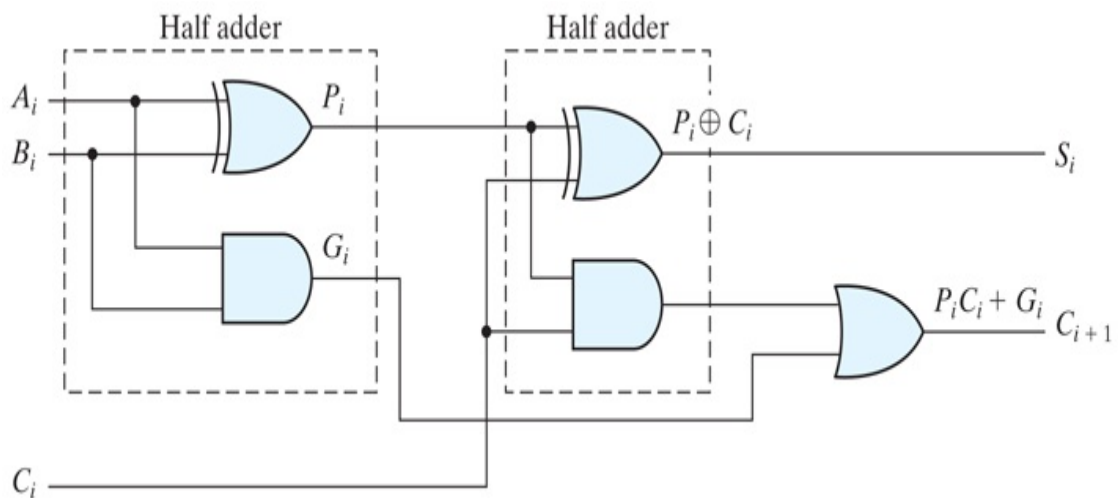
The four-bit adder is a typical example of a standard component. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the classical method would require a truth table with  $2^9 = 512$  entries, since there are nine inputs to the circuit. By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation.

## Carry Propagation

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 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 half adders and depend on only 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.



**FIGURE 4.10**

Full adder with  $P$  and  $G$  shown

### [Description](#)

The carry propagation time is an important characteristic 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

$$P_i = A_i \oplus B_i \quad G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

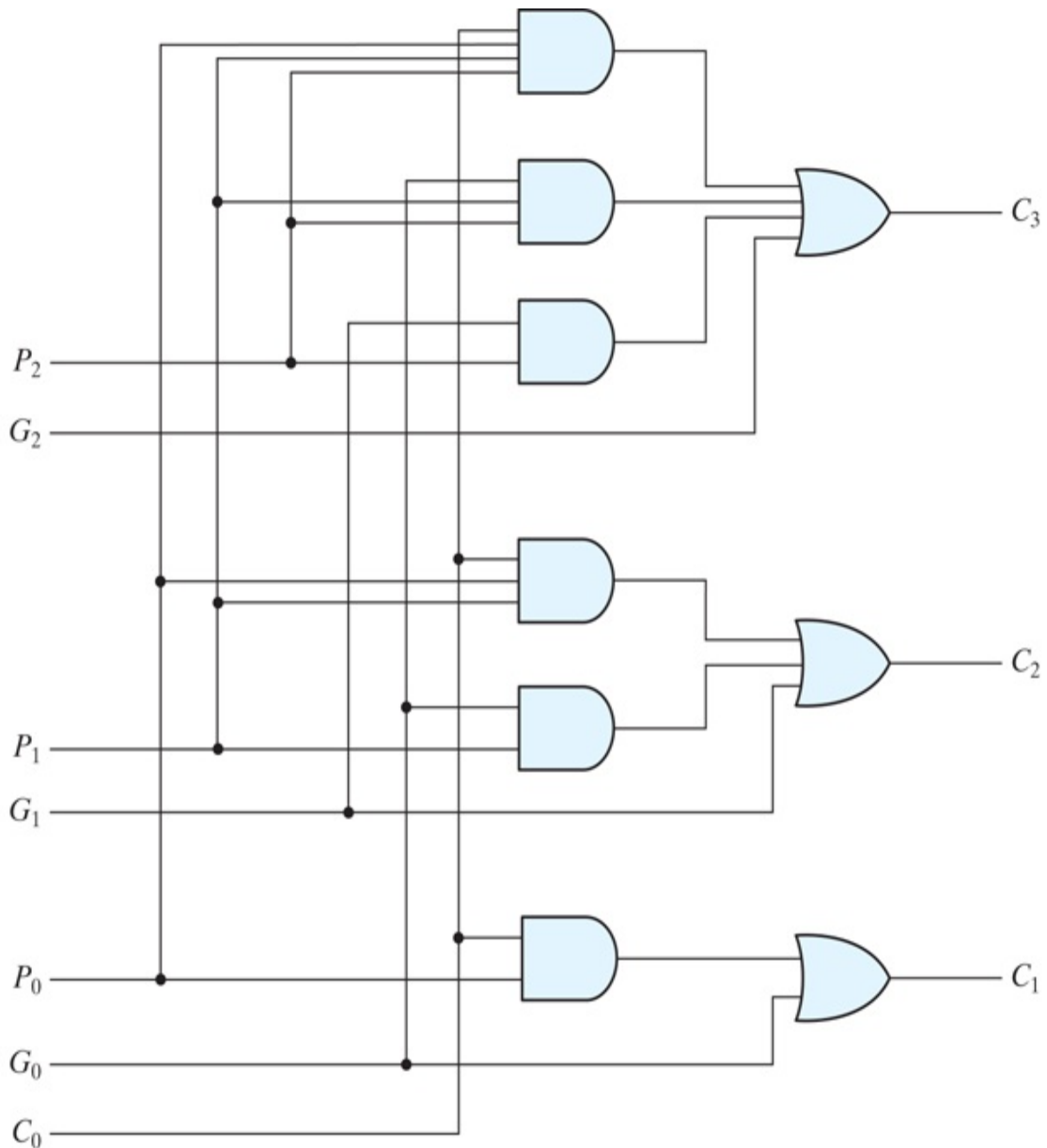
$$S_i = P_i \oplus C_i \quad C_{i+1} = G_i + P_i C_i$$

$G_i$  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  $C_i$ .  $G_i$  indicates that the data into stage  $i$  generates a carry into stage  $i + 1$ .  $P_i$  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:

$$\begin{aligned} C_0 &= \text{input carry} \\ C_1 &= G_0 + P_0 C_0 \\ C_2 &= G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0 \\ C_3 &= G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \end{aligned}$$

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).



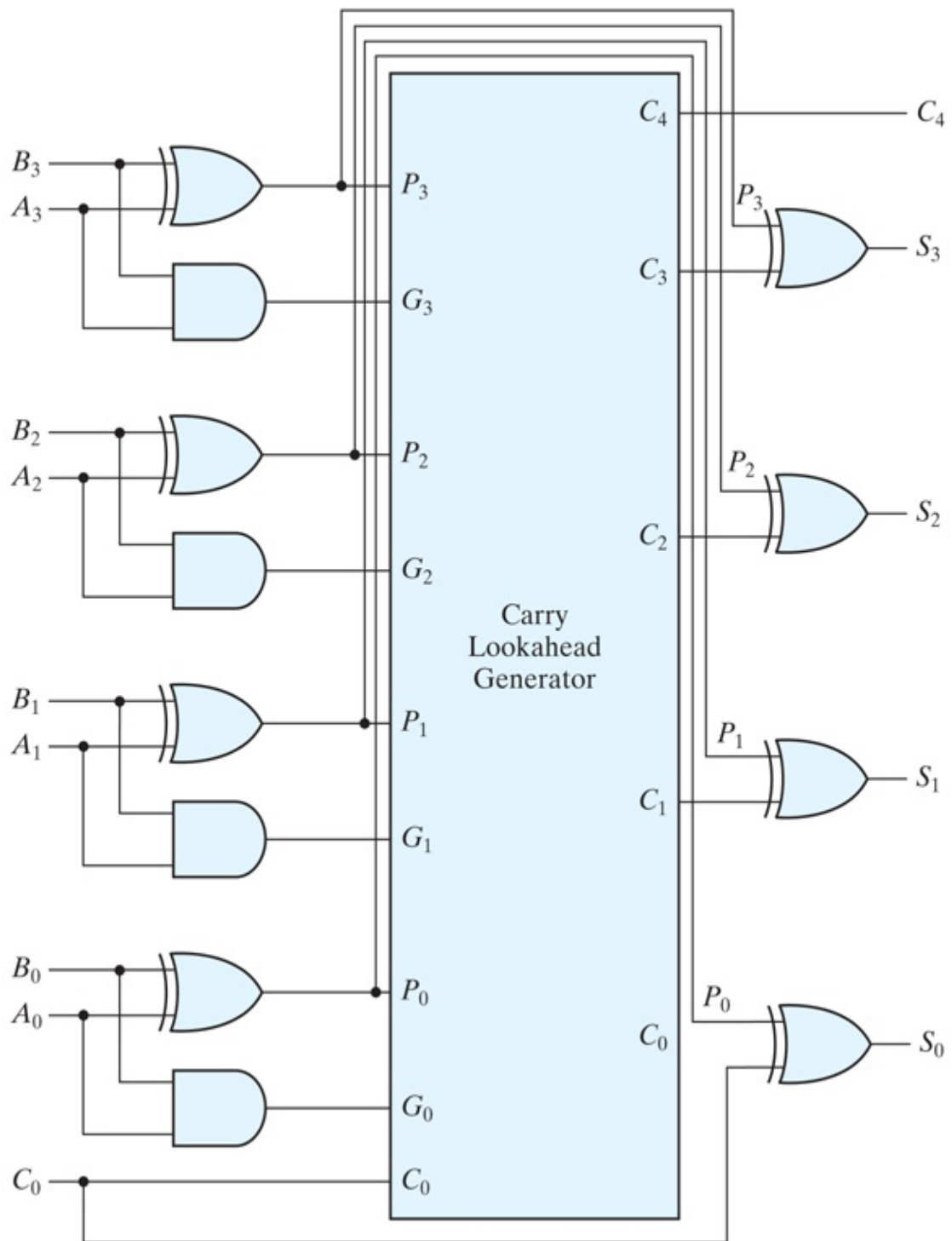
**FIGURE 4.11**

Logic diagram of carry lookahead generator

#### [Description](#)

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 only 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.





# FIGURE 4.12

Four-bit adder with carry lookahead

[Description](#)

## Practice Exercise 4.3

1. What is the main drawback of a ripple adder?

**Answer:** The time required to add long data words may be prohibitive, because the carry has to propagate from the least significant bit to the most significant bit.

## Practice Exercise 4.4

1. What is the main drawback of a carry lookahead adder?

**Answer:** Its hardware is more complex than the hardware for a ripple carry adder.

## Practice Exercise 4.5

1. Add the following two binary words and find the sum and carry bit:  $A = 1100\_0101$ ,  $B = 1010\_1010$ .

**Answer:** Sum =  $0110\_1111$ , Carry = 1

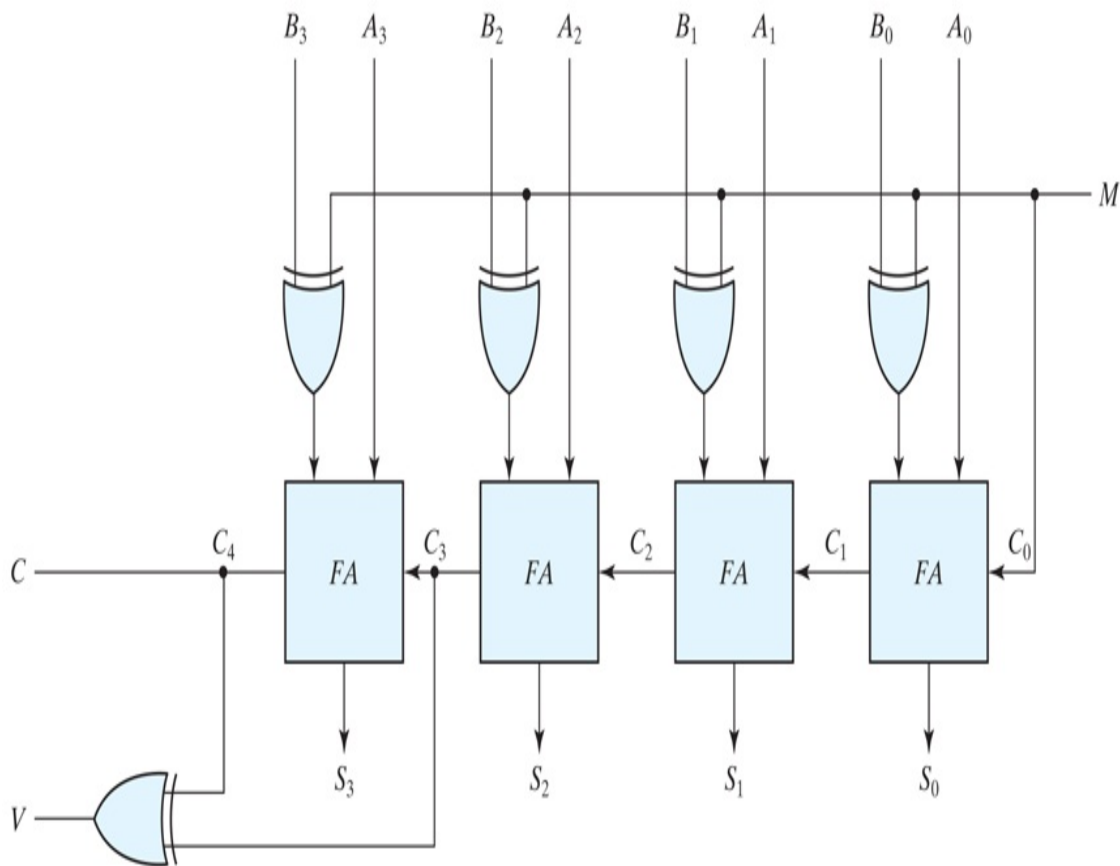
## Binary Subtractor

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 of a full adder.

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  $A$  plus  $B$ . When  $M = 1$ , 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.)



**FIGURE 4.13**

Four-bit adder–subtractor (with overflow detection)

### Description

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.

## Practice Exercise 4.6

1. Find  $A - B$  with  $A = 1001_2$  and  $B = 0110_2$  ;

**Answer:**  $A - B = 1\_0011\ 2$

## 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:**      **0 1**

$$+ 70 \quad 0 \ 1000110 \quad - 70 \quad 1 \ 0111010$$

$$+ 80 \quad \underline{0 \ 1010000} \quad - 80 \quad \underline{1 \ 0110000}$$

$$150 \quad 1 \ 0010110 \quad - 150 \quad 0 \ 1101010$$

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the eighth 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 an 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.

### Table 4.5 *Derivation of BCD Adder*

Binary Sum					BCD Sum					Decimal							
<i>K</i>	<i>Z</i>	<i>8</i>	<i>Z</i>	<i>4</i>	<i>Z</i>	<i>2</i>	<i>Z</i>	<i>1</i>	<i>C</i>	<i>S</i>	<i>8</i>	<i>S</i>	<i>4</i>	<i>S</i>	<i>2</i>	<i>S</i>	<i>1</i>
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	2
0	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	3
0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	4
0	0	1	0	1	0	0	0	0	0	0	1	0	0	1	0	0	5
0	0	1	1	0	0	0	0	0	0	0	1	1	0	0	0	0	6
0	0	1	1	1	0	0	0	0	0	0	1	1	1	0	0	0	7
0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	8
0	1	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	9
0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	10
0	1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	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

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. 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.

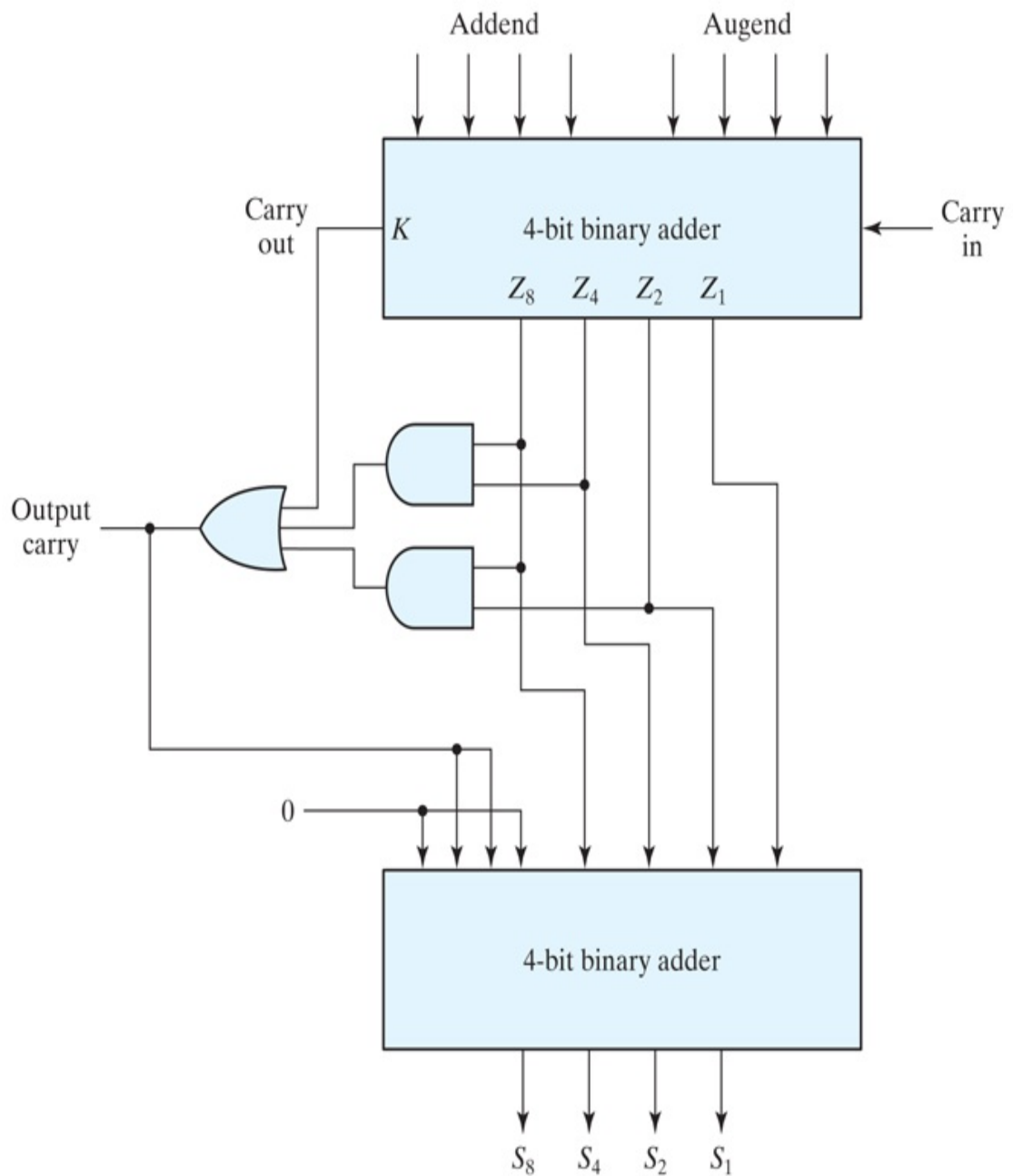
The logic circuit that detects the necessary correction can be derived from the entries in the table. It is obvious that a correction is needed when the binary sum has an output carry  $K = 1$ . The other six combinations from 1010 through 1111 that need a correction have a 1 in position Z 8 . To distinguish 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



$$C = K + Z_8 Z_4 + Z_8 Z_2$$

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. 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.



**FIGURE 4.14**

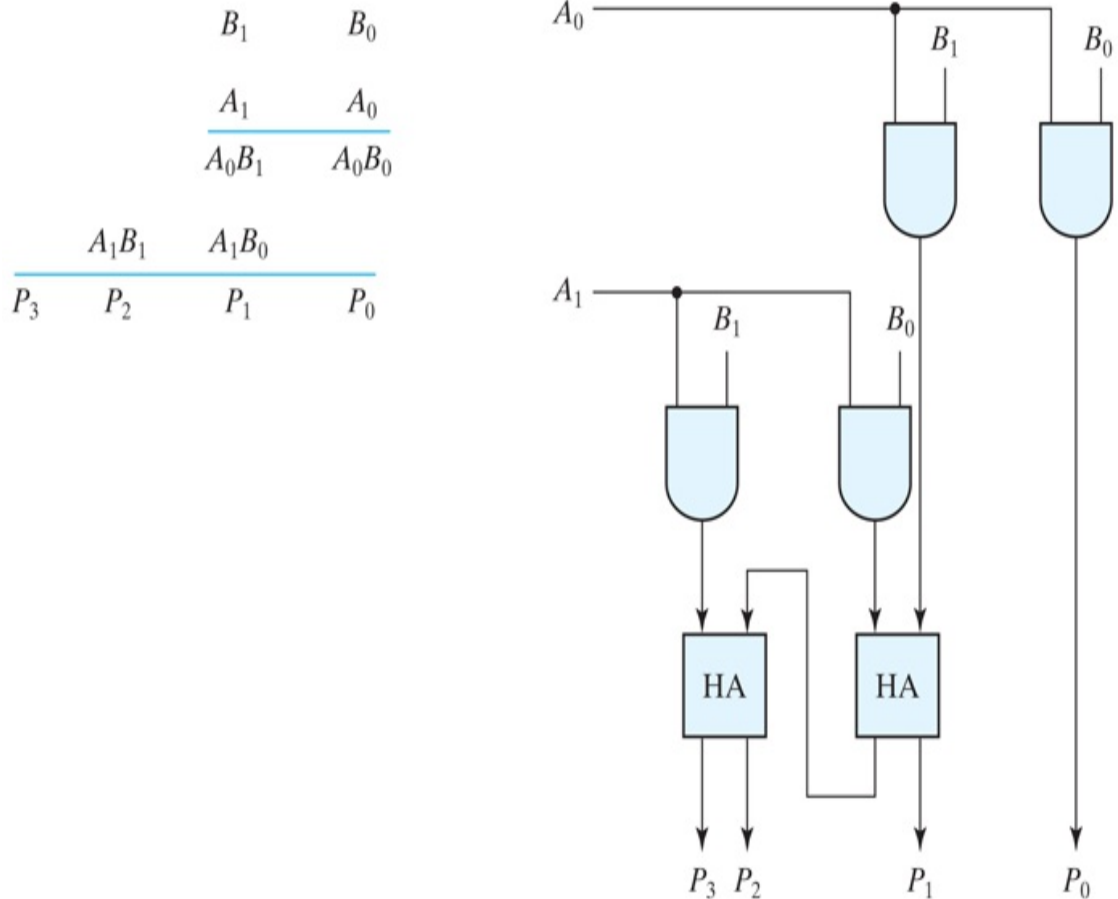
Block diagram of a BCD adder

[Description](#)

## 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  $P_3 P_2 P_1 P_0$ . The first partial product is formed by multiplying  $B_1 B_0$  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  $B_1 B_0$  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 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.



## FIGURE 4.15

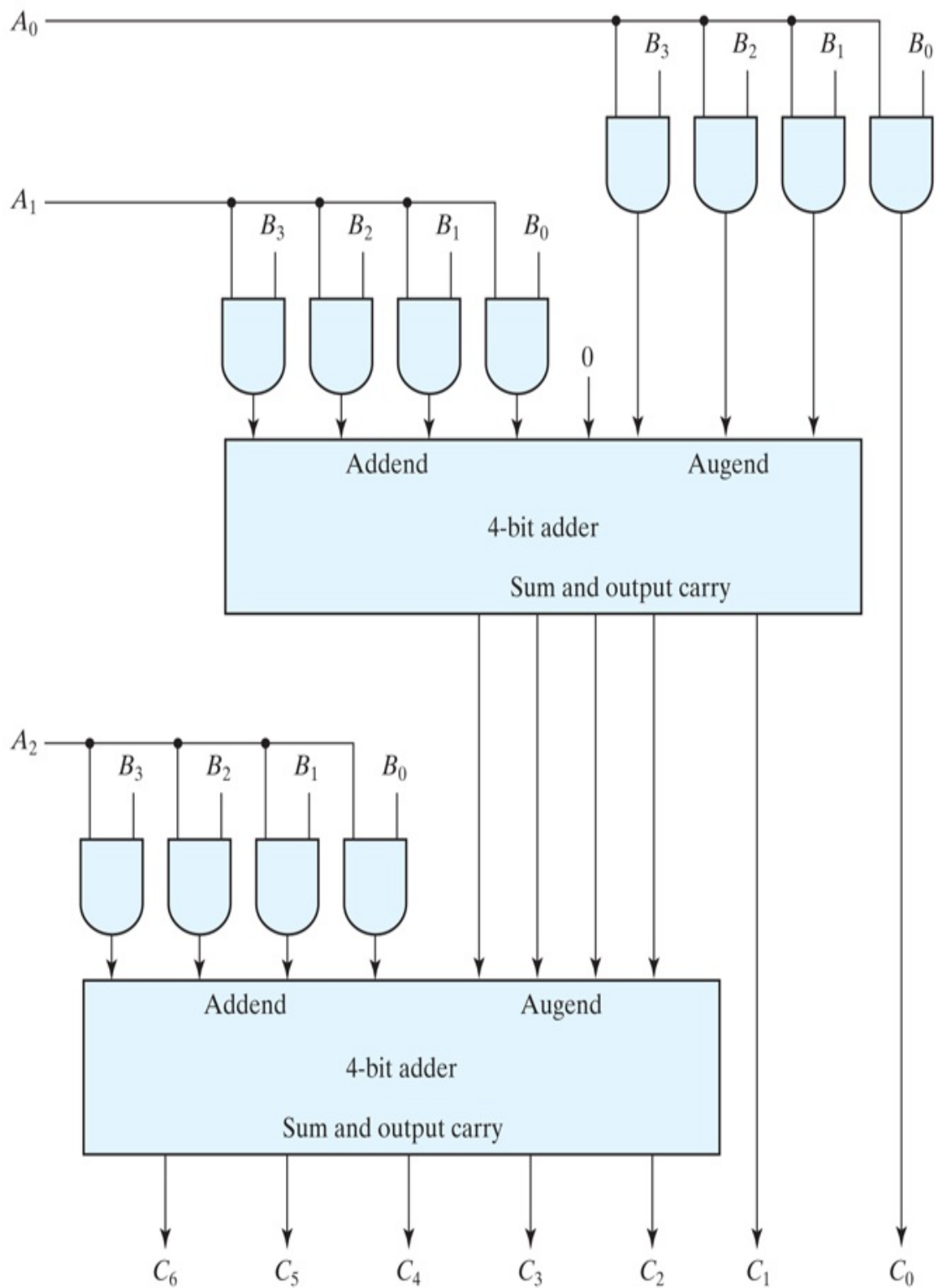
Two-bit by two-bit binary multiplier

### Description

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  $B_3 B_2 B_1 B_0$  and the multiplier by  $A$

2 A 1 A 0 . Since  $K = 4$  and  $J = 3$  , we need 12 AND gates and two 4-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

[Description](#)

## 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:

$$A = A_3 A_2 A_1 A_0 \quad B = B_3 B_2 B_1 B_0$$

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

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

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:

$$(A = B) = x_3 x_2 x_1 x_0$$

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

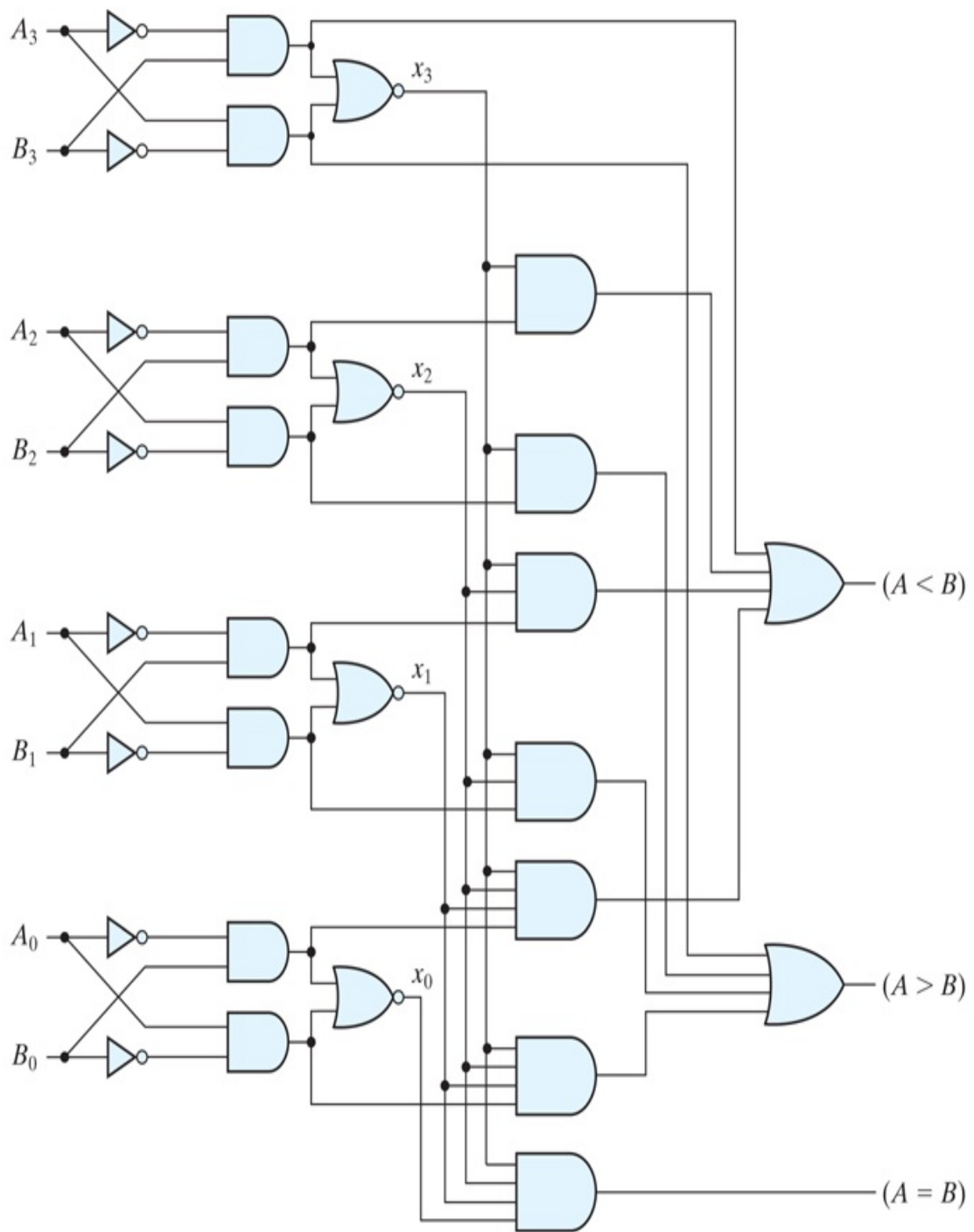
To determine whether  $A$  is greater or less than  $B$ , we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position. If the two digits of a pair are equal, we compare the next lower significant pair of digits. The comparison continues until a pair 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

$$(A > B) = A_3 B_3' + x_3 A_2 B_2' + x_3 x_2 A_1 B_1' + x_3 x_2 x_1 A_0 B_0' \\ (A < B) = A_3' B_3 + x_3 A_2' B_2 + x_3 x_2 A_1' B_1 + x_3 x_2 x_1 A_0' B_0$$

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

[Description](#)

## Practice Exercise 4.7

1. Find the product  $(0101)_2 \times (1001)_2$ .

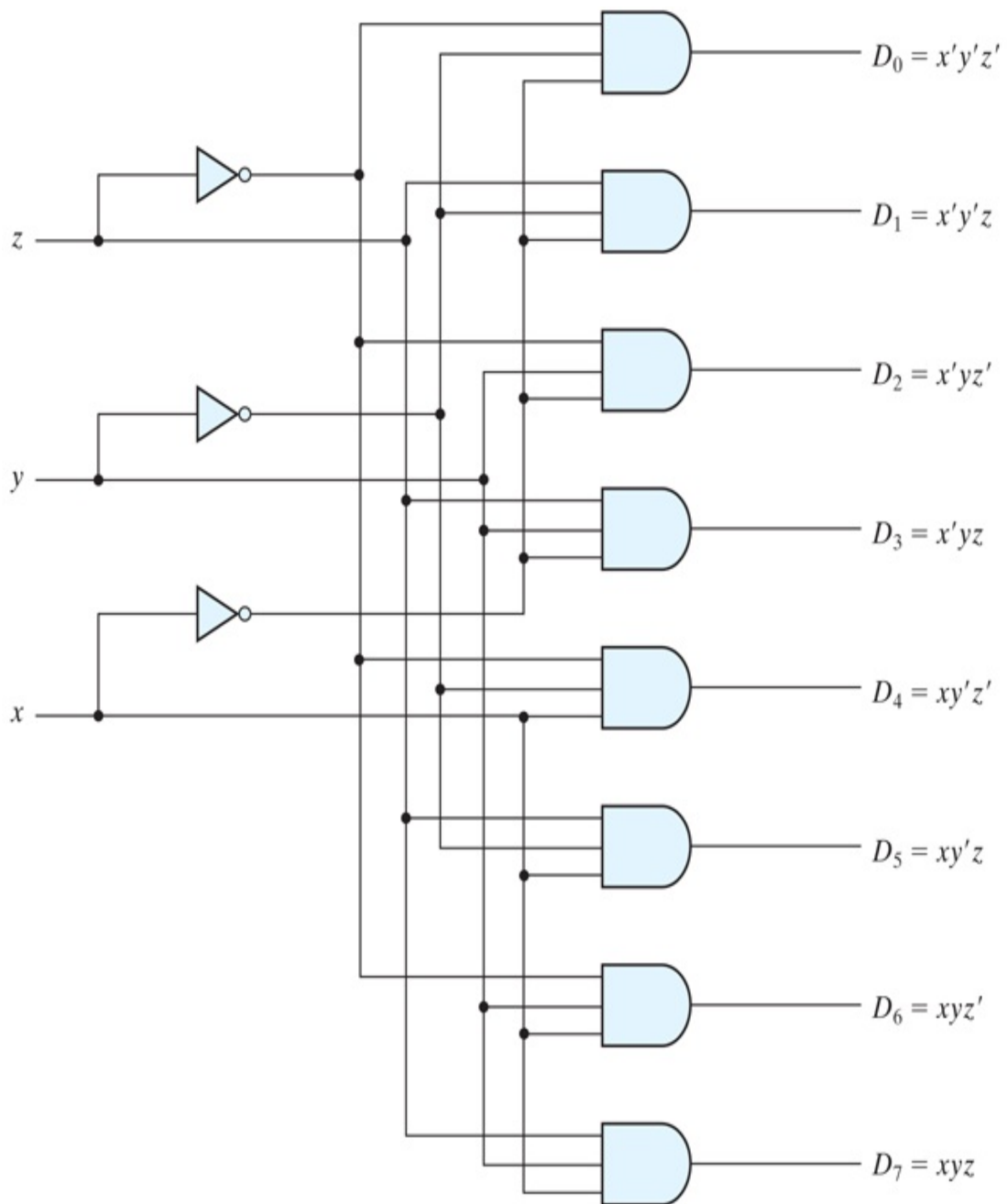
**Answer:**  $0101101_2$

## 4.9 DECODERS

Discrete quantities of information are represented in digital systems by binary codes. A binary code of  $n$  bits is capable of representing up to  $2^n$  distinct elements of coded information. A *decoder* is a combinational circuit that converts binary information from  $n$  input lines to a maximum of  $2^n$  unique output lines. If the  $n$ -bit coded information has unused combinations, the decoder may have fewer than  $2^n$  outputs.

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

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



**FIGURE 4.18**

Three-to-eight-line decoder

### [Description](#)

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

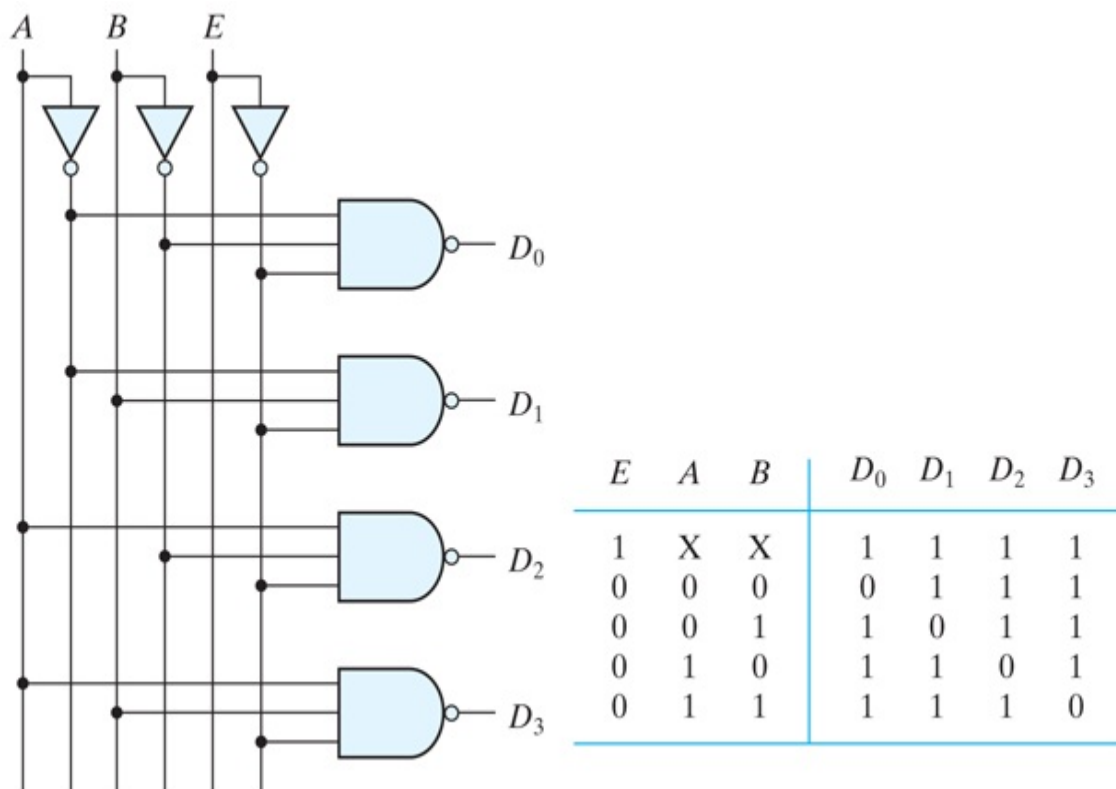
is equal to 1 represents the minterm equivalent of the binary number currently available in the input lines.

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

Inputs			Outputs							
<i>x</i>	<i>y</i>	<i>z</i>	<b>D 0</b>	<b>D 1</b>	<b>D 2</b>	<b>D 3</b>	<b>D 4</b>	<b>D 5</b>	<b>D 6</b>	<b>D 7</b>
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

Some decoders are constructed with NAND gates. Since a NAND gate

produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Furthermore, decoders include one or more *enable* inputs to control the circuit operation. A two-to-four-line decoder with an enable input constructed with NAND gates is shown in [Fig. 4.19](#). The circuit operates with complemented outputs and a complement enable input. The outputs of the decoder are 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.



**FIGURE 4.19**

## Two-to-four-line decoder with enable input

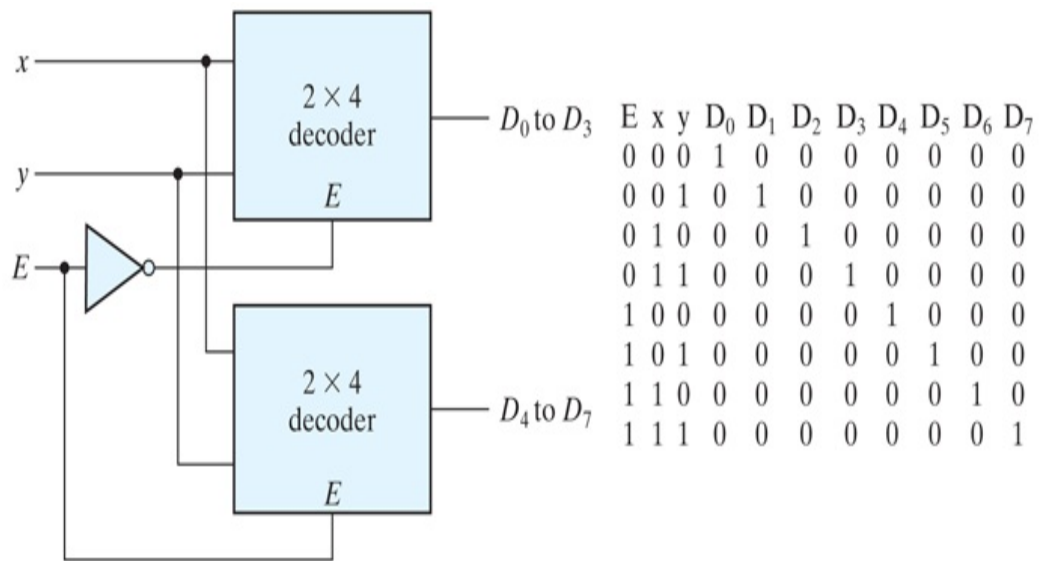
### Description

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 PE4.8**

[Description](#)

## Combinational Logic Implementation

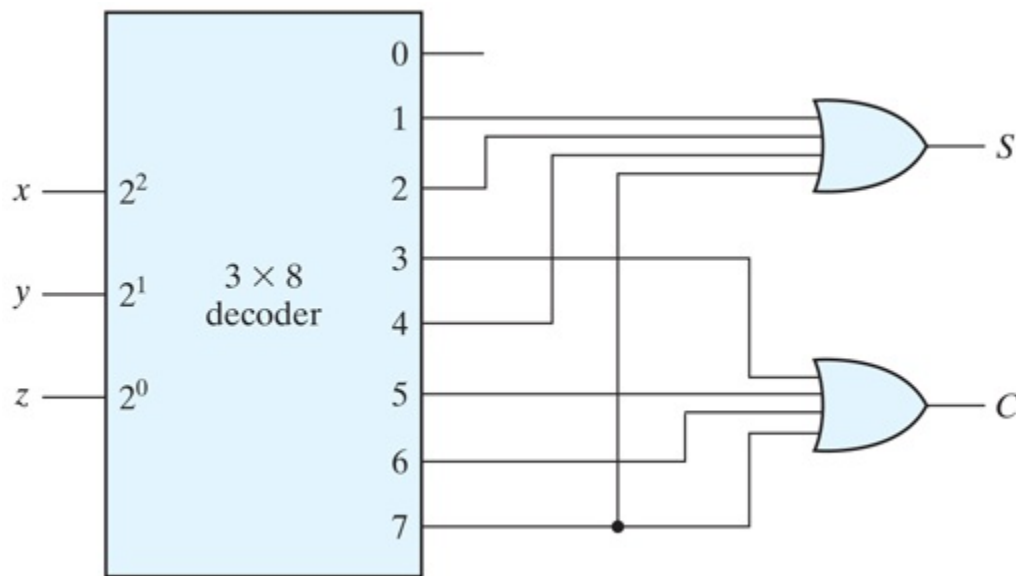
A decoder provides the  $2^n$  minterms of  $n$  input variables. Each asserted output of the decoder is associated with a unique pattern of input bits. Since any Boolean function can be expressed in sum-of-minterms form, a decoder that generates the minterms of the function, together with an external 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:

$$S(x, y, z) = \Sigma(1, 2, 4, 7) \quad C(x, y, z) = \Sigma(3, 5, 6, 7)$$

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



## FIGURE 4.21

Implementation of a full adder with a decoder

### [Description](#)

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

implements a sum-of-minterms function and is equivalent to a two-level AND-OR circuit.

## 4.10 ENCODERS

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has  $2^n$  (or fewer) input lines and  $n$  output lines. The output lines, as an aggregate, generate the binary code corresponding to each 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.

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

Inputs								Outputs		
D 0	D 1	D 2	D 3	D 4	D 5	D 6	D 7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0

0 0 0 0 0 1 0 0 1 0 1

0 0 0 0 0 0 1 0 1 1 0

0 0 0 0 0 0 0 1 1 1 1

The encoder 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:

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

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.

## Priority Encoder

A priority encoder is an encoder circuit that includes the priority function,

and handles the possibility that inputs might be in contention. 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.

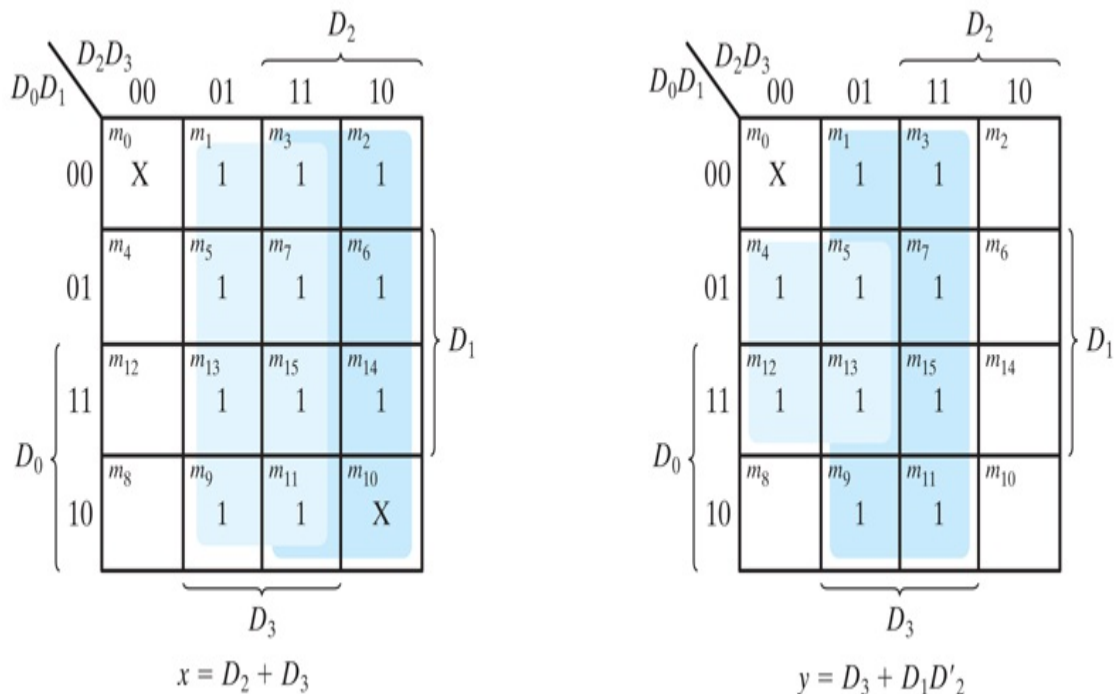
**Table 4.8 *Truth Table of a Priority Encoder***

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

According to [Table 4.8](#), the higher the subscript number, the higher the priority of the input is. Input D 3 has the highest priority, so, regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3). D 2 has the next priority level. The output is 10 if D 2 = 1 , provided that D 3 = 0 , regardless of the values of the other two lower priority inputs. The output for D 1 is generated only if higher priority inputs are 0, and so on down the priority levels.

The K-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:

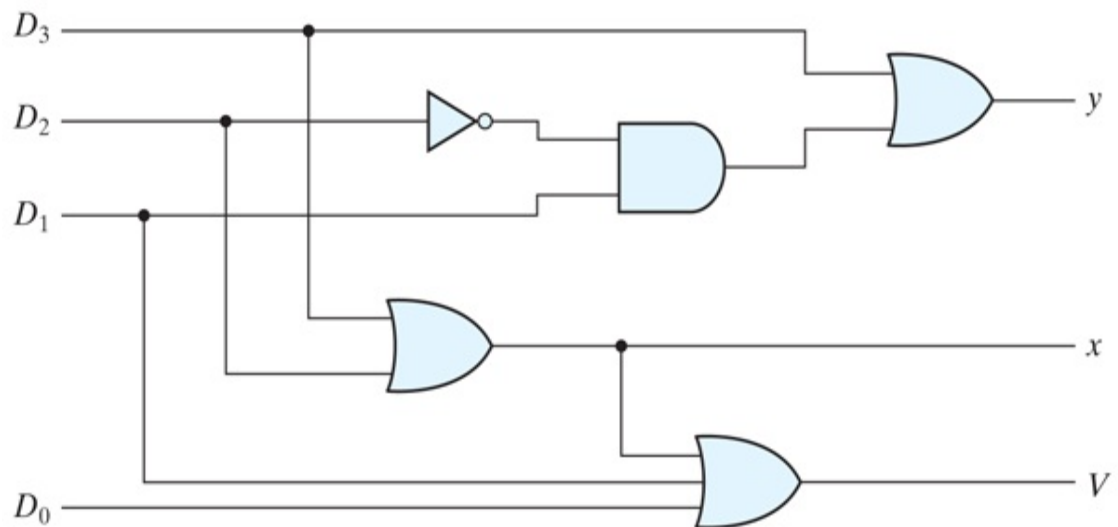
$$x = D_2 + D_3 \quad y = D_3 + D_1 D_2' \quad V = D_0 + D_1 + D_2 + D_3$$



**FIGURE 4.22**

Maps for a priority encoder

[Description](#)



**FIGURE 4.23**

Four-input priority encoder

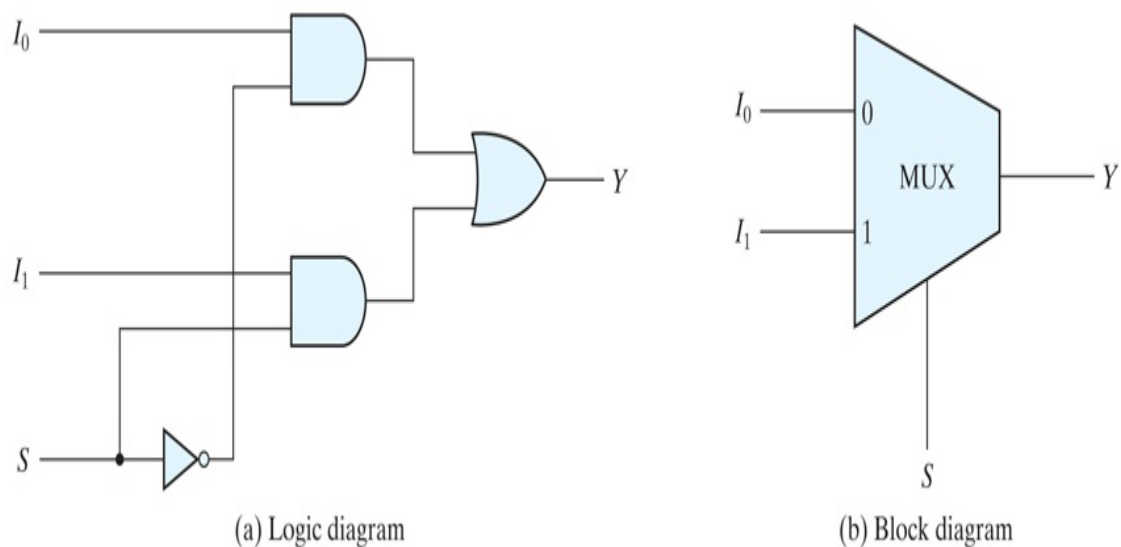
[Description](#)



## 4.11 MULTIPLEXERS

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

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

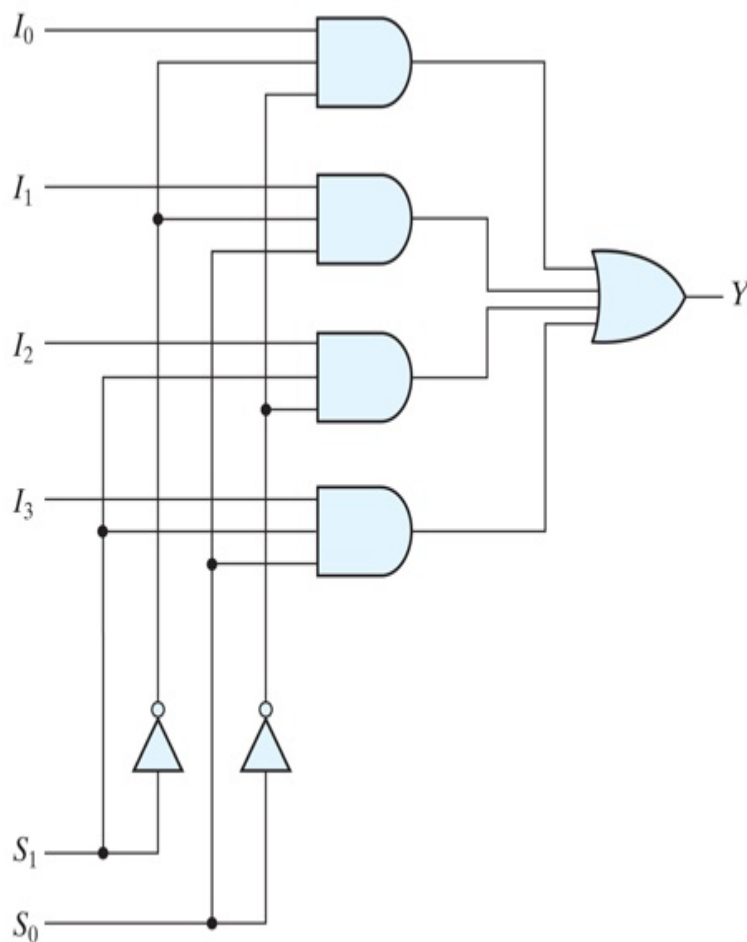


**FIGURE 4.24**

Two-to-one-line multiplexer

[Description](#)

A four-to-one-line multiplexer is shown in Fig. 4.25. Each of the four inputs,  $I_0$  through  $I_3$ , is applied to one input of an AND gate. Selection lines  $S_1$  and  $S_0$  are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate that provides the one-line output. The function table lists the input that is passed to the output for each combination of the binary selection values. To demonstrate the operation of the circuit, consider the case when  $S_1 S_0 = 10$ . 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.



(a) Logic diagram

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

(b) Function table

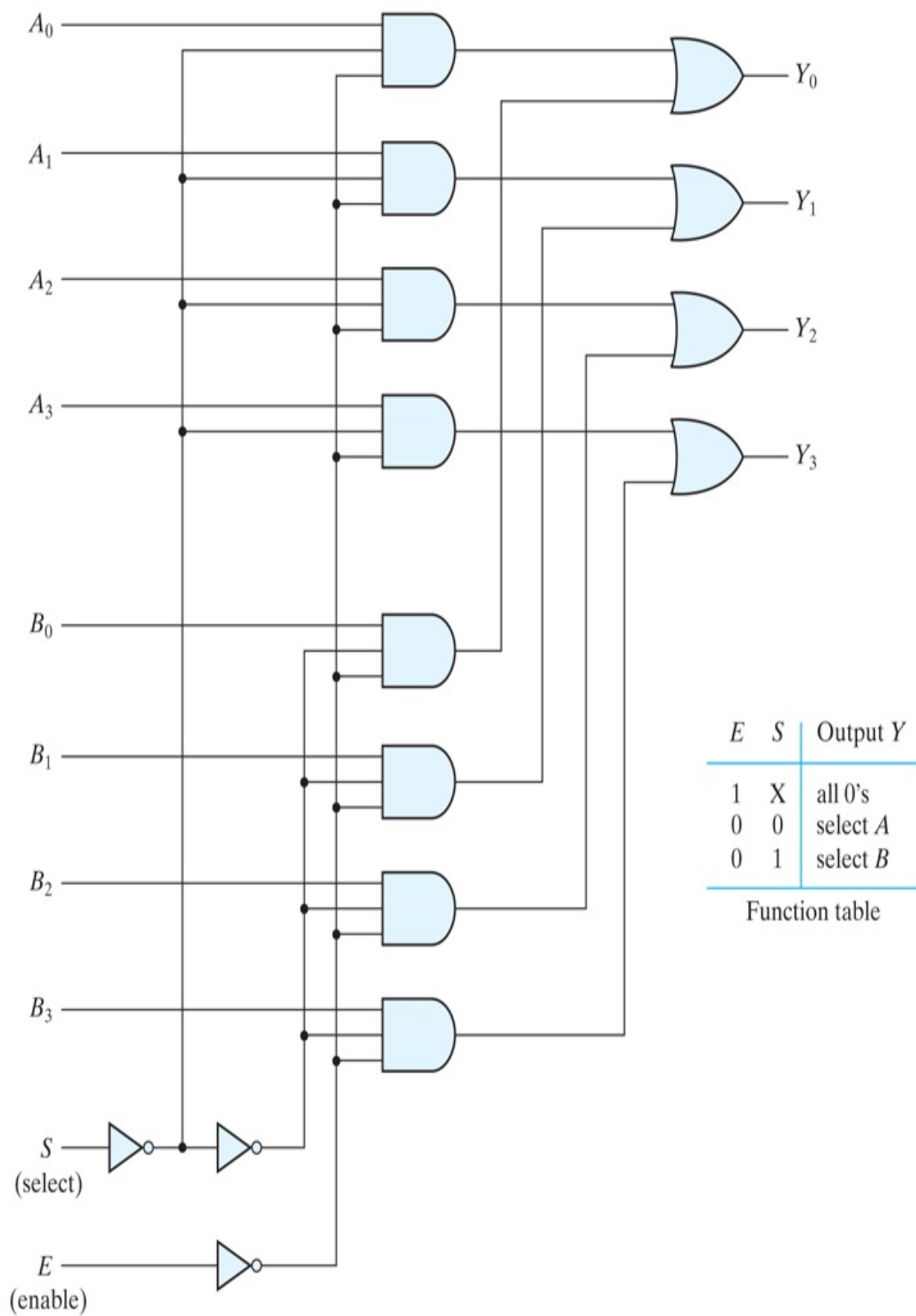
**FIGURE 4.25**

## Four-to-one-line multiplexer

### Description

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  $S$ .



**FIGURE 4.26**

Quadruple two-to-one-line multiplexer

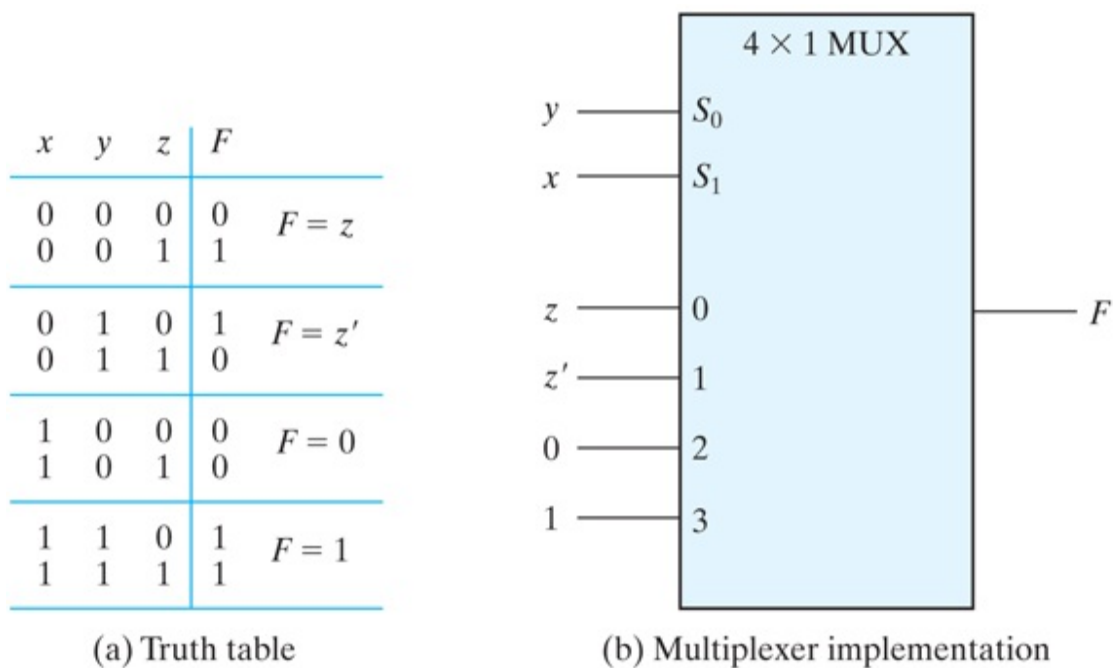
# Boolean Function Implementation with Multiplexers

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, instead of  $n$  selection inputs. The first  $n - 1$  variables of the function are connected to the selection inputs of the multiplexer. The remaining single variable of the function is used for the data inputs. If the single variable is denoted by  $z$ , each data input of the multiplexer will be  $z$ ,  $z'$ , 1, or 0. To demonstrate this procedure, consider the Boolean function

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

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



## FIGURE 4.27

Implementing a Boolean function with a multiplexer

### Description

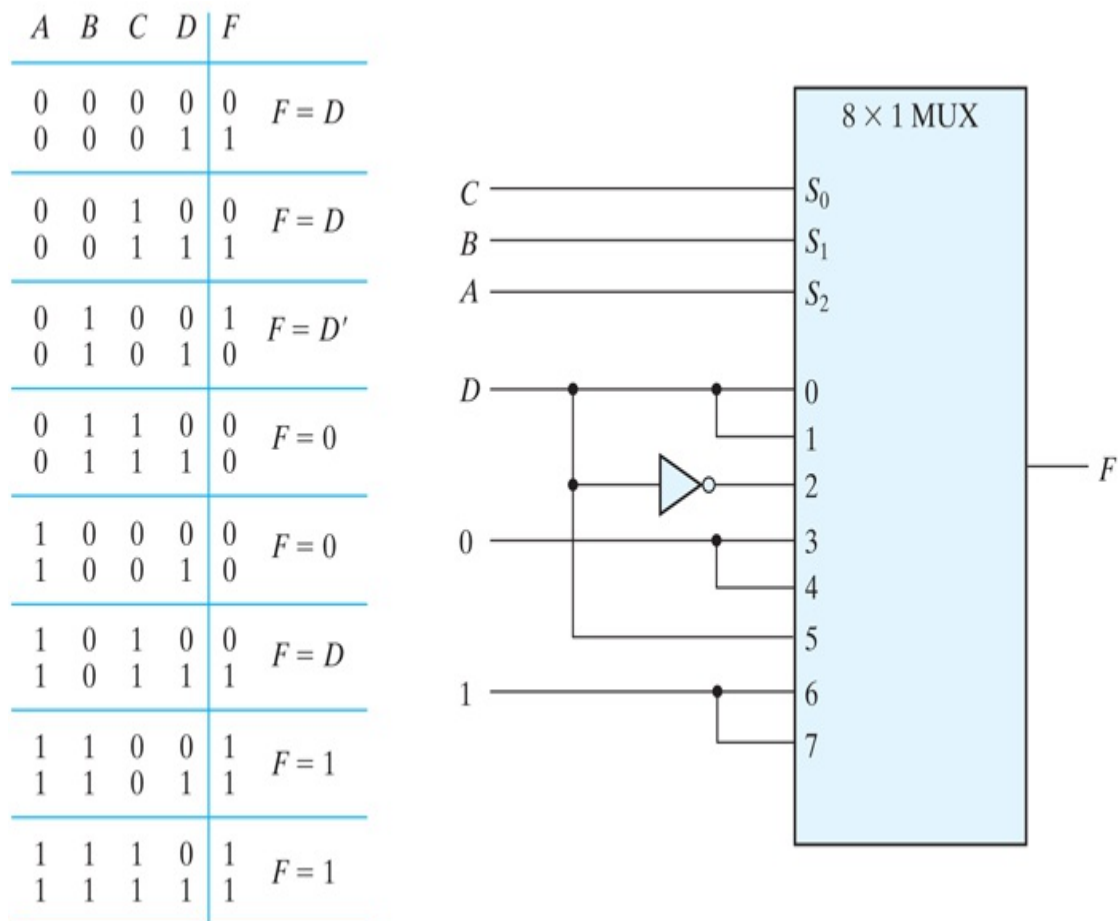
The general procedure for implementing any Boolean function of  $n$  variables with a multiplexer with  $n - 1$  selection inputs and  $2^{n-1}$  data inputs follows from the previous example. To begin with, Boolean function is listed in a truth table. Then first  $n - 1$  variables in the table are 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  $A B C = 101$ ,  $F = D$ , so the input variable  $D$  is applied to data input 5. The binary constants 0 and 1 correspond to two fixed signal values. When integrated circuits are used, logic 0 corresponds to signal ground and logic 1 is equivalent to the power signal, depending on the technology (e.g., 3 V).



**FIGURE 4.28**

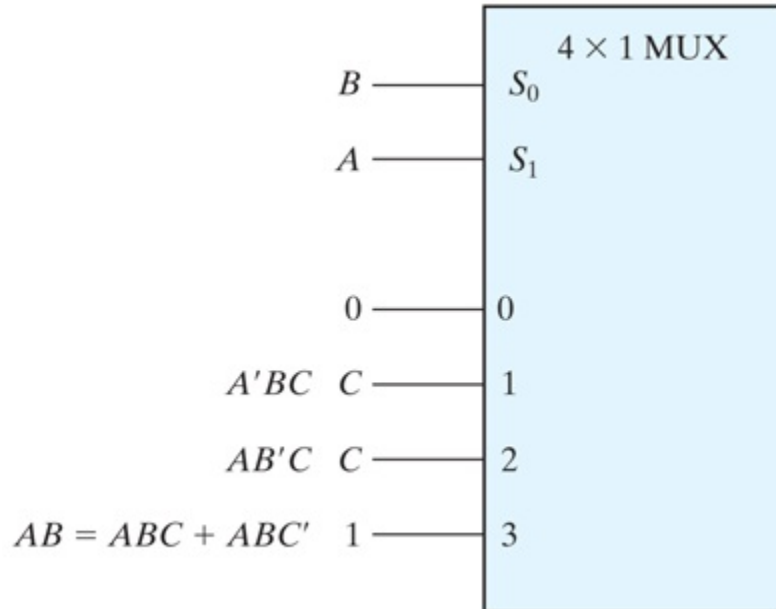
Implementing a four-input function with a multiplexer

[Description](#)

## Practice Exercise 4.9

1. Implement the Boolean function  $F(A, B, C) = \Sigma(3, 5, 6, 7)$  with a multiplexer.

**Answer:**



**FIGURE PE4.9**

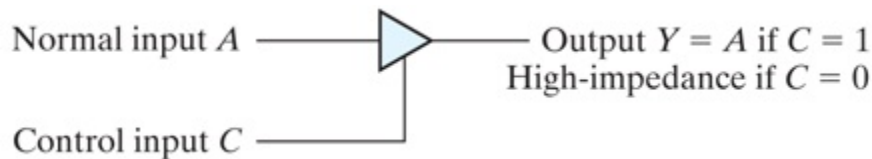
## Three-State Gates

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

The graphic symbol for a three-state buffer gate is shown in [Fig. 4.29](#). It is distinguished from a normal buffer by an input control line entering the bottom of the symbol. The buffer has a normal input, an output, and a control input that determines the state of the output. When the control



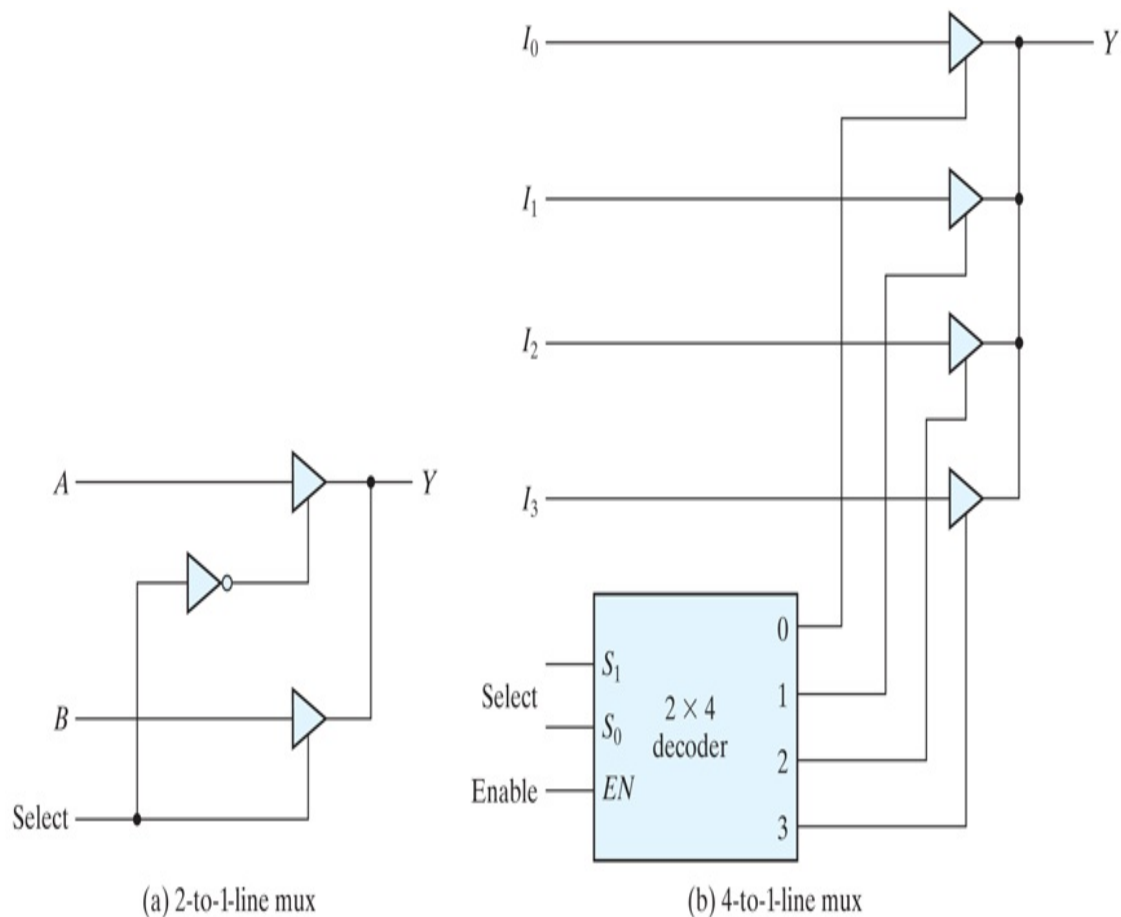
input is equal to 1, the output is enabled and the gate behaves like a conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state 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](#). [Figure 4.30\(a\)](#) shows the construction of a two-to-one-line multiplexer with 2 three-state buffers and an inverter. The two outputs are connected together to form a single output line. (Note that this type of connection cannot be made with gates that do not have three-state outputs.) When the selected 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  $B$ .



**FIGURE 4.30**

Multiplexers with three-state gates

### Description

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.

## 4.12 HDL MODELS OF COMBINATIONAL CIRCUITS

Basic features of Verilog and VHDL were introduced in [Chapter 3](#). This section introduces additional features of those languages, presents more elaborate examples, and compares alternative descriptions of combinational circuits.[6](#)

[6](#) Sequential circuits and their models are presented in Chapter 5.

Verilog and VHDL support three common styles of modeling combinational circuits:

- Gate-level modeling, also called *structural* modeling, instantiates and interconnects basic logic circuits to form a more complex circuit having a desired functionality. Gate-level modeling describes a circuit by specifying its gates and how they are connected with each other.[7](#)

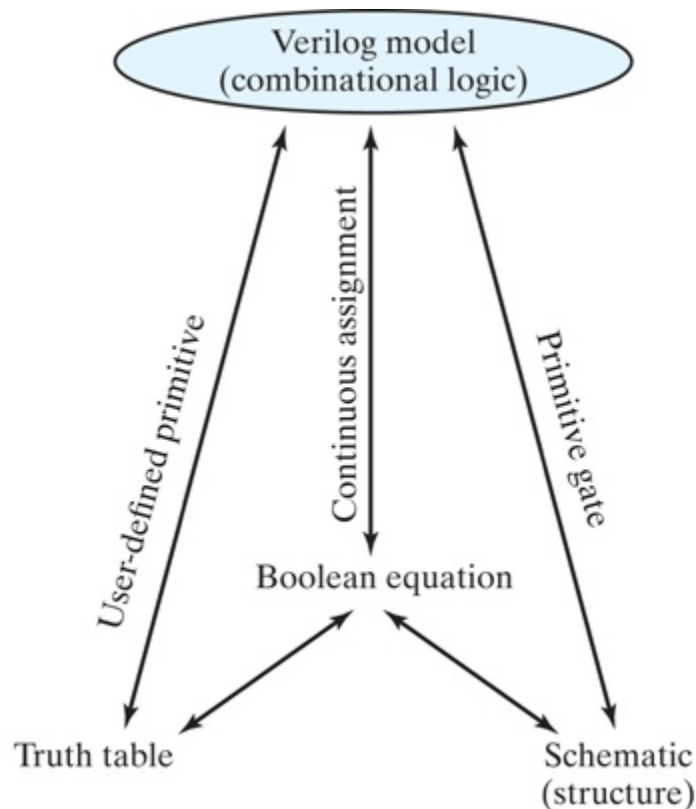
[7](#) Verilog also supports switch-level modeling for directly representing MOS transistor circuits. This style is sometimes used in modeling and simulation, but not in synthesis. We will not use switch-level modeling in this text, but we provide a brief introduction in Appendix A.3. For additional information see the Verilog language reference manual.

- Dataflow modeling uses HDL operators and *assignment* statements to describe the functionality represented by Boolean equations.
- Behavioral modeling uses language-specific procedural statements to form an abstract model of a circuit. Behavioral modeling describes combinational and sequential circuits at a higher level of abstraction than gate-level modeling or dataflow modeling [6–9].

In general, combinational logic can be described with Boolean equations, logic diagrams, and truth tables. The ways that these three options are supported by a HDL depends on the language [1–3].

# Verilog

Verilog has a construct corresponding to each of three “classical” approaches to designing combinational logic: continuous assignments (Boolean equations), built-in primitives (logic diagrams), and user-defined primitives (truth tables), as depicted in [Fig. 4.31](#).



**FIGURE 4.31**

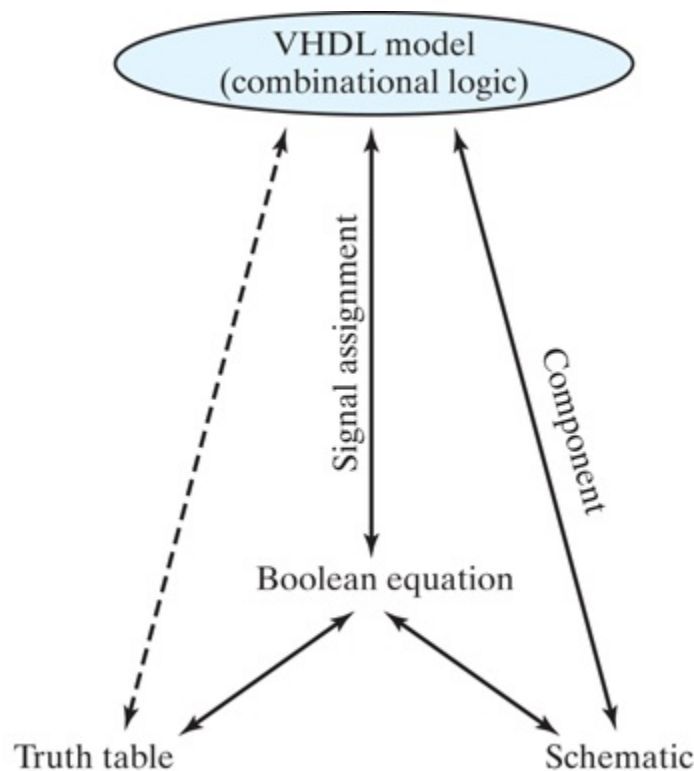
Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

[Description](#)

## VHDL

VHDL has constructs for describing combinational logic using Boolean equations and logic diagrams (schematics), as depicted in [Fig. 4.32](#) [10,

11]. Concurrent signal assignment statements implement Boolean equations. There are no built-in gates, but user-defined components can be used to implement a circuit described by a logic diagram or a truth table. If a combinational circuit is specified by a truth table, its output functions must be derived and used to create Boolean functions whose expressions can be described with concurrent signal assignment statements.



**FIGURE 4.32**

Relationship of VHDL constructs to truth tables, Boolean equations, and schematics three-state gates

## Gate-Level Modeling

Gate-level modeling, which was introduced in [Chapter 3](#) by a simple example, specifies a logic circuit by its gates and their interconnections. Gate-level modeling provides a textual description of a logic diagram (schematic) [12-13].

# Verilog (Primitives)

Verilog includes 12 basic logic 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 declared with the lowercase keywords **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, and **buf**.

Primitives such as **and** are *n*-input primitives, because they can have any number of scalar inputs (e.g., a three-input **and** primitive). The **buf** and **not** primitives are *n*-output primitives because a single input to a **buf** or **not** gate can drive multiple outputs.

The Verilog language includes a functional description of each type of gate, with the logic of each gate based on a four-valued system. [8](#) The functional descriptions specify the output of each primitive for every combination of its inputs. 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 left unconnected.

[8](#) The logic system for switch-level models includes 4 values and 8 strengths. Switch-level models are discussed in Appendix A.3.

The four-valued logic truth tables for the **and**, **or**, **xor**, and **not** primitives are shown in [Table 4.9](#). The table is organized for two inputs, with a row-column format in which the possible values of one input occupy a row corresponding to a value of the other input. The truth table for the other four gates are organized in the same way. Note that the output of the **and** gate is 1 only when both of its inputs are 1, and the output is 0 if any input is 0. Otherwise, if one input is **x** or **z**, the output is **x**. The output of the **or** gate is 0 if both inputs are 0, is 1 if any input is 1, and is **x** otherwise. The logic table for a two-input gate can be used for an *n*-input gate by combining pairwise the result for the first two inputs with the third input, etc.

## 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 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

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.

## Verilog (Vectors)

In many designs it is helpful to use identifiers having multiple bit widths, called *vectors*. The syntax specifying a vector includes within square brackets two whole 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, labeled 0 through 3. The second declares a wire vector, *SUM*, with eight bits numbered and descending from 7 to and including 0. (*Note:* The first (leftmost) number (array index) listed is always interpreted as 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, the sub-vector *SUM*[2:0] specifies the three least significant bits of vector *SUM*.

## VHDL (User-Defined Components)

VHDL does not have predefined gate-level primitive elements. Gate-level (structural) models in VHDL are created by (1) defining entity-architecture pairs having specified functionality, and (2) instantiating them as components within the structural model (i.e., architecture) of an entity. If the functionality of a logic circuit is specified by a truth table, it is necessary to declare a component, which can be instantiated in an entity.

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

The gate-level description of a two-to-four-line decoder (see [Fig. 4.19](#)) has two data inputs *A* and *B* and an enable input *E*. The four outputs are specified with the vector *D*.

## Verilog

In the Verilog model three **not** gates produce the complement of the inputs, and four **nand** gates provide the outputs for the bits of *D*. Remember that *the output is always listed first in the port list of a primitive*, followed by the inputs. Note that the keywords **not** and **nand** are written only once and do not have to be repeated for each instance of the **nand** gate, but commas must be inserted at the end of each instantiation of the gates in the series, except for the last statement, which must be terminated by a semicolon. The **wire** declaration is for internal connections.

```
// 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),           // Comma-separated list of primitives
        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
```

## Practice Exercise 4.10 (Verilog)

1. Write a continuous assignment statement that is equivalent to the logic of *G4* in *decoder\_2x4\_gates*.

**Answer:** `assign D[0]=!(A) && (!B) && (!enable);`

# Practice Exercise 4.10 (VHDL)

1.

```
library ieee;
use ieee.std_logic_1164.all;
-- Declare entity-architecture pairs that will be component

-- Model for inverter component

entity inv_gate is
  port (B: out std_logic; A: in std_logic);
end inv_gate;

architecture Boolean_Equation of inv_gate is
begin
  B <= not A;
end Boolean_Equation;

entity nand3_gate is
  port (D: out std_logic; A, B, C: in std_logic);
end nand3_gate;

architecture Boolean_Eq of nand2_gate
begin
  C <= not (A and B and C);
end Boolean_Eq;

-- Gate-level description of two-to-four line decoder
entity decoder_2x4_gates_vhdl is
  port (A, B, enable: in std_logic; D: out std_logic_vector
end decoder_2x4_gates_vhdl;

architecture Structure of decoder_2x4_gates_vhdl is
  -- Identify components and ports
  component inv_gate
    port (B: out std_logic; A: in std_logic);
  end component;

  component nand3_gate
    port (D: out std_logic; A, B, C: in std_logic);
  end component;

  signal A_not, B_not, enable_not;  -- Internal signal:
begin  -- Instantiate components and connect ports via por
  G1: inv_gate port map (A_not, A);
  G2: inv_gate port map (B_not, B);
  G3: inv_gate port map (enable_not, enable);

  G4: nand3_gate port map (D(0), A_not, B_not, enable_not);
```

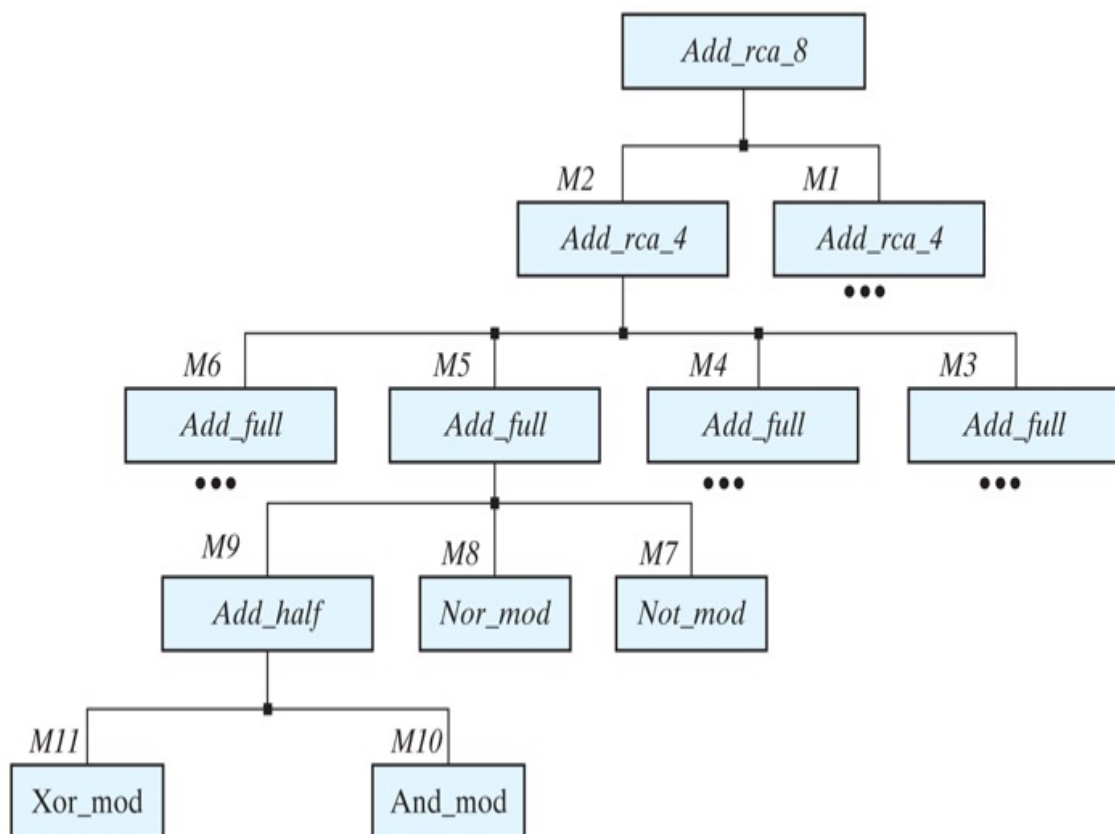
```

G5: nand3_gate port map (D(1), A_not, B, enable_not);
G6: nand3_gate port map (D(2), A, B_not, enable_not);
G7: nand3_gate port map (D(3), A, B, enable_not);
end Structure

```

## Hierarchical Modeling

A hierarchical system can be composed of multiple design objects organized in a hierarchical structure. The hierarchy is formed by instantiating subcircuits within circuits [8–11]. For example, an 8-bit adder can be formed by instantiating and connecting two identical 4-bit adders. A 4-bit adder can be formed by instantiating and interconnecting four full adders. The full adder is declared once, but it is instantiated (used) repeatedly. [Figure 4.33](#) shows the hierarchical structure of an 8-bit ripple-carry adder, and [Fig. 4.34](#) shows the functional blocks of the hierarchy and their interfaces.



**FIGURE 4.33**

Design hierarchy of an 8-bit ripple-carry adder. For simplicity, some blocks are omitted where they replicate what is already shown

[Description](#)



## FIGURE 4.34

Decomposition of an 8-bit ripple carry adder into a chain of two 4-bit adders; [9](#) each 4-bit adder consists of a chain of four full adders. The full adders are composed of half adders and one OR gate; the half adders are composed of logic gates.

### [Description](#)

[9](#) Note: In Verilog vectors are written as `a[7:0]`, etc, as shown here; in VHDL vectors are written as `a(7 downto 0)`, etc.

The design object at the top of the design hierarchy is the *parent module* (Verilog) or *parent design entity* (VHDL). The underlying objects are referred to as *children*. Instantiating, or nesting, objects within objects creates a parent–child relationship and gives an explicit representation of the structure.

Two basic types of design methodologies can create a hierarchy: 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 4-bit binary adder of [Fig. 4.9](#). It can be considered as a top-block component built with four full adder blocks; each full adder is built with two half-adders. In a top–down design, the four-bit adder is defined first, and then full adders are defined and interconnected. In a bottom-up design, the half adder is defined, then the full adder is constructed; the four-bit adder is built by instantiating and interconnecting the full-adders. [10](#)

[10](#) Note that the first character of an identifier cannot be a number, but can be an underscore. Thus, the eight-bit adder could be named `_8bit_adder`. An alternative name that is meaningful and does not present the possibility of neglecting the leading underscore character is `Add_rca_8`.

## HDL Example 4.2 (Hierarchical

# Modeling—Eight-Bit Adder)

## Verilog

At the bottom of the design hierarchy shown in [Fig. 4.33](#) a half adder is composed of primitive gates. At the next level of the hierarchy, a full adder is formed by instantiating and connecting a pair of half adders. The third module describes the eight-bit adder by instantiating and linking together two four-bit adders. This example illustrates optional Verilog 2001, 2005 syntax, which eliminates extra typing of identifiers declaring the mode (e.g., **output**), type (**reg**), and declaration of a vector range (e.g., [3: 0]) of a port. The first version of the standard (1995) uses separate statements for these declarations; the revised standard includes the declarations within the port.

```
module Add_half (input a, b, output c_out, sum),
  xor G1(sum, a, b);          // Gate instance names are option
  and G2(c_out, a, b);
endmodule
```

```
module Add_full (input a, b, c_in, output c_out, sum);    // see
Fig.
4.8
```

```
  wire w1, w2, w3;          // w1 is c_out; w2 is sum
  Add_half M1 (a, b, w1, w2);
  Add_half M0 (w2, c_in, w3, sum);
  or (c_out, w1, w3);
endmodule
```

```
module Add_rca_4 (input [3:0] a, b, input c_in output c_out, ou
  wire c_in1, c_in3, c_in4;          // Intermediate carries
  Add_full M0 (a[0], b[0], c_in, c_in1, sum[0]);
  Add_full M1 (a[1], b[1], c_in1, c_in2, sum[1]);
  Add_full M2 (a[2], b[2], c_in2, c_in3, sum[2]);
  Add_full M3 (a[3], b[3], c_in3, c_out, sum[3]);
endmodule
```



```

module Add_rca_8 (input [7:0] a, b, input c_in, output c_out, c
  wire c_in4;
  Add_rca_4 M0 (a[3:0], b[3:0], c_in, c_in4, sum[3:0]);
  Add_rca_4 M1 (a[7:4], b[7:4], c_in4, c_out, sum[7:4]);
endmodule

```

Verilog modules can be instantiated within other modules, but module declarations cannot be nested; that is, a module declaration cannot be inserted into the text between the **module** and **endmodule** keywords of another module. Also, instance names (e.g., M0) must be specified when a module is instantiated within another module.

## VHDL

A VHDL hierarchical model of *Add\_rca\_8\_vhdl*, an 8-bit adder, constructs components for the logic gates in [Fig. 4.34](#), and uses them in the half adders and full adders. Once *Add\_full\_vhdl* and *Add\_half\_vhdl* are written they can be used to create *Add\_rca\_4\_vhdl* and *Add\_rca\_8\_vhdl*.

```

library ieee;
use ieee.std_logic_1164.all;

-- Model for 2-input AND component
entity and2_gate is
  port (A, B: in Std_Logic; C: out Std_Logic);
end and2_gate;

architecture Boolean_Equation of and2_gate is
begin
  C <= A and B;      -- Logic operator
end Boolean_Equation;

-- Model for 2-input OR component
entity or2_gate is
  port (A, B: in Std_Logic; C: out Std_Logic);
end or2_gate;

architecture Boolean_Equation of or2_gate is
begin
  C <= A or B;      -- Logic operator
end Boolean_Equation;

```

```
-- Model for exclusive-or component
entity xor_2_gate is
  port (A, B: in Std_Logic; C: out Std_Logic);
end xor_2_gate;

architecture Boolean_Equation of xor_2_gate is
begin
  C <= A xor B;
end Boolean_Equation;
```

The components *and2\_gate* and *xor2\_gate* are then used in models for *Add\_half\_vhdl* and *Add\_full\_vhdl*.

```
entity Add_half_vhdl is
  port (a, b: in std_logic; c_out, sum: out std_logic);
end Add_half;

architecture Structure of Add_half is
  component and2_gate          -- Identify component being used
  port (a, b: in std_logic; c: out std_logic);  -- Identify port
end component;

  component xor2_gate          -- Component declaration
  port (a, b: in std_logic; c: out std_logic);
and component;

begin          -- Instantiate components and connect ports
  G1: xor2_gate  port map (a, b, sum);
  G2: and2_gate  port map (a, b, c_out,);
end Structure;

entity Add_full_vhdl is
  port (a, b, c_in: in std_logic; c_out, sum: out std_logic);
end Add_full_vhdl

architecture Structure of Add_full_vhdl is
  component or2_gate
  port (a, b: in std_logic; c: out std_logic);
end component;
  component Add_half_vhdl
  port (a, b: in std_logic; c_out, sum: out std_logic);
end component;
  signal w1, w2, w3: std_logic;
```

```

begin
  M0: Add_half_vhdl port map (b, c_in, c_out, sum);
  M1 Add_half port map (a, b, w1, w2);
  G1 or2_gate port map (w1, w3, c_out);
end Structure;

entity Add_rca_4_vhdl is
  port (A, B: in bit_vector (3 downto 0); c_in: in Std_Logic;
        c_out: out Std_Logic; sum: out bit_vector (3 downto 0)
end Add_rca_4_vhdl;

architecture Structure of Add_rca_4_vhdl is
  component Add_full_rca_vhdl
    port (a, b: in Std_Logic_Vector (3 downto 0); c_in: in Std_Logic;
          sum: out Std_Logic_Vector (3 downto 0);
    end component;
  signal c_in1, c_in2, c_in3;
begin
  M0: Add_full_vhdl port map (a(0), b(0), c_in, c_in1, sum(0));
  M1: Add_full_vhdl port map (a(1), b(1), c_in1, c_in2, sum(1))
  M2: Add_full_vhdl port map (a(2), b(2), c_in2, c_in3, sum(2))
  M3: Add_full_vhdl port map (a(3), b(3), c_in3, c_out, sum(3))
end Structure;

entity Add_rca_8_vhdl is
  port (a, b: in Std_Logic_Vector (7 downto 0); c_in: in Std_Logic;
        c_out: out Std_Logic, sum: Std_Logic_Vector (7 downto 0)
end Add_rca_8_vhdl;

architecture Structure of Add_rca_8_vhdl is
  component Add_rca_4_vhdl;
    port (a, b: in Std_Logic_Vector (3 downto 0); c_in: in Std_Logic;
          c_out: out Std_Logic; sum: Std_Logic_Vector (3 downto 0)
    end component;
  signal c_in4 -- Connects 4-bit adders
  M0 Add_rca_4_vhdl port map (a(3 downto 0), b(3 downto 0), c_in,
    sum(3 downto 0 ));
  M1 Add_rca_4_vhdl port map (a(7 downto 4), b(7 downto 4), c_in,
    sum(7 downto 4 ));
end Structure

```

The code for *Add\_rca\_8* illustrates how gate-level design in VHDL becomes bulky with declarations of components. Hierarchical design can be made simple if component declarations exploit dataflow models at the lower levels of the hierarchy. For example, a half adder can be designed

and used as a component in the design of a full-adder.

```
entity half_adder_vhdl is  
  port (S, C: out Std_Logic; x, y: in Std_Logic);  
end half_adder_vhdl;
```

```
architecture Dataflow of half_adder_vhdl is  
  S <= x xor y;  
  C <= x and y;  
end Dataflow;
```

```
entity full_adder_vhdl is  
  port (S, C: out Std_Logic; x, y, z: in Std_Logic);  
end half_adder_vhdl
```

```
architecture Structural of full_adder_vhdl is  
  signal S1, C1, C2: Std_Logic;  
  component half_adder_vhdl port (S, C: out Std_Logic; x, y, z:  
begin  
  HA1: half_adder_vhdl port map (S => S1, C => C1, x => x, y =>  
  HA2: half_adder_vhdl port map (S => S, C => C2, x => S1, y =>  
  C <= C2 or C1;  
end Structural;
```

```
entity ripple_carry_4_bit_adder_vhdl is  
  port (Sum: out Std_Logic_Vector (3 downto 0)); C4: out Std_Log  
  Std_Logic_Vector (3 downto 0); C0: in Std_Logic);  
end ripple_carry_4_bit_adder_vhdl;
```

```
architecture Structural of ripple_carry_4_bit_adder_vhdl is  
  component full_adder_vhdl port Sum: out Std_Logic_Vector (3 d  
  Std_Logic; A, B: in Std_Logic_Vector (3 downto 0); C0: in S1  
  signal C1, C2, C3: Std_Logic;
```

```
begin  
  FA0: full_adder_vhdl port map (S => Sum(0), C => C1, x => A(0)  
  FA1: full_adder_vhdl port map (S => Sum(1), C => C2, x => A(1)  
  FA2: full_adder_vhdl port map (S => Sum(2), C => C3, x => A(2)  
  FA3: full_adder_vhdl port map (S => Sum(3), C => C4, x => A(3)  
end ripple_carry_4_bit_adder_vhdl;
```

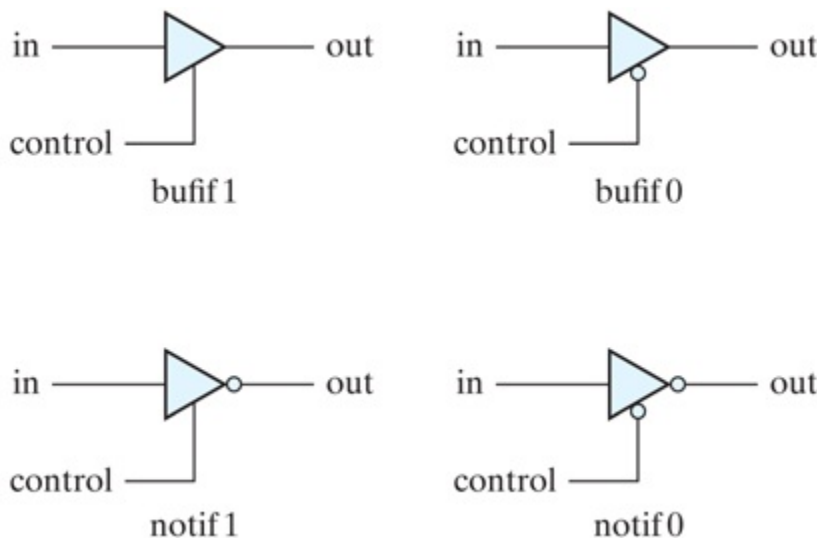
## HDL Models of Three-State Gates

A three-state gate has a data signal input, a data signal output, and a control input. The control input determines whether the gate is in its normal operating state or in its high-impedance state.

## Verilog (Predefined Buffers and Inverters)

Verilog has four types of predefined three-state gates, as shown in [Fig. 4.35](#). 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, but 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 4.35**

Three-state gates

### [Description](#)

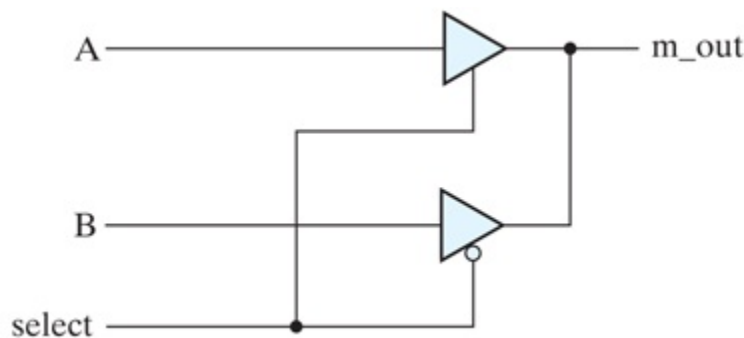
The gate name can be that of any 1 of the 4 three-state gates. In simulation,

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, *OUT* has the same value as *A* 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 explicitly identify such a connection, Verilog uses the net-type keyword **tri** (for tristate) to indicate that the identifier has multiple drivers. As an example, consider the two-to-one-line multiplexer with three-state gates shown in [Fig. 4.36](#).



**FIGURE 4.36**

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

The description must use a **tri** data type for the output, because *m\_out* has two drivers:

```
// 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 outputs of the three-state buffers are identical (*m\_out*). 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**, **triand**, **trior**, **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), for example, as a **wire**. The net **wor** models the hardware implementation of the wired-OR configuration (emitter-coupled logic). The **wand** models the wired-AND configuration (open-collector technology; see [Fig. 3.26](#)). The nets **supply1** and **supply0** represent power supply and ground, respectively. They are used to fix an input of a device to either logical 1 or logical 0.

## VHDL (User-Defined Buffers and Inverters)

VHDL does not have predefined buffers or inverters. Instead, they must be declared as entity-architecture pairs having the functionality of a three-state device, and then instantiated as components. The model of a three-state gate in VHDL has a control input which determines whether the output is enabled. If enabled, the output of a buffer is equal to its input. If not, the output has a logic value of z. Similarly, the output of an enabled inverter will be the complement of its input; if not enabled, the output will have a value of z. The models of a buffer and an inverter that are enabled when the control input is 1 are given below:

```
entity bufif1_vhdl is  
  port (buf_in, control: in Std_Logic; buf_out: out Std_Logic);  
end bufif1_vhdl;
```

```
architecture Dataflow of bufif1_vhdl is  
begin  
  buf_out <= buf_in when control = '1'; else 'z';
```

```
end Dataflow;
```

```
entity notif1 is  
  port (not_in, control: in Std_Logic; not_out: out Std_Logic);  
end notif1;
```

```
architecture Dataflow of notif1 is  
begin  
  not_out <= not (not_in) when control = '1'; else z;  
end Dataflow;
```

## Practice Exercise 4.11

1. Describe the functionality of a three-state inverter.

**Answer:** The output signal of a three-state inverter is the complement of the input signal if the inverter is enabled. If a three-state inverter is not enabled, the output is the high impedance state.

## Practice Exercise 4.12—(VHDL)

1. Write a signal assignment statement for use in the architecture of *notif0\_vhdl*, a three-state inverter component having output signal *y\_out*, input signal *x\_in*, and active-low control signal *enable\_b*.

**Answer:** *y\_out* <= not (*x\_in*) when *enable\_b* = '0'; else z

## Number Representation

Numbers in HDLs are represented in formats that enable interpretation and specify their size and base. The size of a number indicates, in bits, the length of its corresponding binary word. The value expresses the number in the indicated base.

## Verilog



Numbers in Verilog are represented by the format `n ' Bv`, where `n` is the number of bits used to store the value, `B` is the base for interpreting the value, and `v` is the value to be interpreted and stored. If the base is not specified, the number is, by default, to be interpreted as a decimal value. If the specified size exceeds the number of bits needed to represent the interpreted value, 0s are used to pad the number to full size. If the size is not specified, the number assumes the size implied by the expression in which it is used. A variable that is assigned `' 0` gets all 0s.

Binary numbers in Verilog are specified and interpreted with the letter **b** preceded by a prime. The size of the number is written first and then its value. Thus, `2 ' b 01` specifies a two-bit binary number whose value is 01. Numbers are stored as a bit pattern in memory, but they can be written and referenced in decimal, octal, or hexadecimal formats with the letters `' d`, `' o`, and `' h` respectively. For example, `4 ' hA = 4 ' d 10 = 4 ' b 1010` and have the same 4-bit internal representation in a simulator. If the base of the number is not specified, its interpretation defaults to decimal. If the size of the number is not specified, the system assumes that the size of the number is at least 32 bits; if a host simulator has a larger word length—say, 64 bits—the language will use that value to store unsized numbers. The integer data type (keyword **integer**) is stored in a 32-bit representation. The underscore (`_`) may be inserted in a number to improve readability of the code (e.g., `16 ' b0101_1110_0101_0011`). It has no other effect.

## VHDL

VHDL is a strongly typed language. The type of assignments to variables must generally match the type of the variable. Most of the variables in the examples in this text have type `Std_Logic`. Numbers in `Std_Logic` are written as binary values, and VHDL requires that they be enclosed in single quotes. For example, the text `'0` and `'1` indicate binary values of 0 and 1 respectively. `Std_Logic_Vector` constants are written in the format `NumberBase"Value"`, where `Number` indicates the number of bits used to represent and/or store the value, `Base` indicates the base of the number, and `value` is the number to be interpreted in the indicated base. The bases are indicated by a single letter as B (Binary), O (octal), D (decimal), and X (hexadecimal). A number that is not specified in this manner defaults to a binary value. If no size is given the number of bits in the value is used.

# HDL Example 4.3 (Number Representation)

## Verilog

1. `3'b110` stores in 3 bits the binary equivalent of decimal 6.
2. `8'hA5` stores in 8 bits the binary equivalent of hexadecimal A5 H =  $1010\_0101_2 = 165_{10}$ .
3. `8'b101` stores in 8 bits the binary value `0000_0101`. Note the padding with 0s.

## VHDL

1. `3b"110"` stores in 3 bits the binary equivalent of decimal 6.
2. `8X"A5"` stores in 8 bits the binary equivalent of hexadecimal A5 H =  $1010\_0101_2 = 165_{10}$ .
3. `8b"101"` stores in 8 bits the binary value `0000_0101`.
4. `B"010"` is stored with three bits as `010`.
5. `X"BC"` is stored as `10111100`.

## Prctice Exercise 4.13

1. What is the binary word that will be stored for `A = B5H`?

**Answer:** `10110101`

## Dataflow Modeling

Dataflow models describe combinational circuits by their *function* rather than by their gate structure. A common form of dataflow modeling of combinational logic uses concurrent signal assignment statements and built-in language operators to express how signals are assigned values.

## Verilog (Predefined Data Types)

Verilog has two families of predefined data types: nets and variables (also referred to as registers). [11](#) The net family includes the data type **wire**, which corresponds to signals associated with structural connections between design elements, and with implicit combinational logic represented by continuous assignment statements. [12](#) The *variable* family of data types is distinguished by its members being assigned value by procedural statements, and by their retaining an assigned value until a new value is assigned. The keywords of some of the types in this family are **reg**, **integer**, and **time**. A **reg** may be a scalar or a vector quantity; an **integer** is sized to the word length of the host machine, and is at least 32 bits wide; a variable having type **time** is represented by an unsigned 64-bit quantity.

[11](#) Note: The words *net* and *register* are not Verilog keywords.

[12](#) An undeclared identifier is, by default, interpreted to be a wire. The default nettype can be reassigned to be any of the predefined net types.

## Verilog (Predefined Operators)

Verilog 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 8.3](#).) The operators supported by Verilog 1995, 2005 are supported by SystemVerilog too. [13](#) However, SystemVerilog also supports the assignment and increment operators listed in [Table 4.11](#), which are *not* supported by the above-cited versions of Verilog.

[13](#) Other operators supported exclusively by SystemVerilog will not be discussed here, but can be found in SystemVerilog for Design, S. Sutherland, S. Davidmann, and P. Flake, Kluwer Academic Publishers, -

Norwell, Mass., 2004.

## **Table 4.10 *Some Verilog Operators***

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

? : conditional

## **Table 4.11a *SystemVerilog*** ***Assignment Operators*** [15](#)

<b>Operator</b>	<b>Description</b>
<b>+=</b>	Add RHS to LHS and assign
<b>-=</b>	Subtract RHS from LHS and assign
<b>*=</b>	Multiply LHS by RHS and assign
<b>/=</b>	Divide LHS by RHS and assign
<b>%=</b>	Divide LHS by RHS and assign remainder
<b>&amp;=</b>	Bitwise AND RHS with LHS and assign
<b> =</b>	Bitwise OR RHS with LHS and assign
<b>^=</b>	Bitwise exclusive OR RHS with LHS and assign
<b>&lt;&lt;=</b>	Bitwise left-shift the LHS by the number of times indicated by the RHS and assign

- >>= Bitwise right-shift the LHS by the number of times indicated by the RHS and assign
- <<<= Arithmetic-shift the LHS by the number of times indicated by the RHS and assign
- >>>= Arithmetic-shift the LHS by the number of times indicated by the RHS and assign

[15](#) LHS denotes left-hand side; RHS denotes right-hand side.

## Table 4.11b *SystemVerilog Increment/Decrement Operators*

Usage	Operation	Description
$j = i++;$	Postincrement	$j$ gets $i$ , then $i$ is incremented by 1
$j = i--;$	Postdecrement	$j$ gets $i$ , then $i$ is decremented by 1
$j = ++i;$	Preincrement	$i$ is incremented by 1, then $j$ gets $i$
$j = --i;$	Predecrement	$i$ is decremented by 1, then $j$ gets $i$

It is necessary to distinguish between arithmetic and logic operations, so different symbols are used for each. The plus symbol ( + ) indicates a sign and the arithmetic operation of addition; the bitwise logic AND operation

uses the symbol `&`. Arithmetic operators treat their operands as unsigned integers. Synthesis tools are able to synthesize hardware to implement `+`, `-`, and `*`, but `/` is restricted to divisors that are powers of 2. [14](#) There are special symbols for bitwise logical AND, OR, NOT, and XOR. The equality (identity) symbol uses two equals signs (without spaces between them) to distinguish it from the equals sign used with the **assign** statement. The bitwise operators operate bit-by-bit on a pair of vector operands to produce a vector result. The concatenation operator provides a mechanism for appending multiple operands. For example, two operands with two bits each can be concatenated to form an operand with four bits. The conditional operator acts like a multiplexer and is explained later in [HDL Example 4.6](#).

[14](#) Division by a power of 2 is equivalent to shifting the dividend to the right by the appropriate positions, producing a result which can be synthesized.

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

A common form of dataflow modeling in Verilog uses continuous assignments and the keyword **assign**. A continuous assignment assigns a value to a net. The data type family *net* is used in Verilog HDL to represent a signal corresponding to 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 input port of a module. The logic value associated with a net is determined by what the net is connected to. If the net is connected to the 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 a net is external to a module and attached to one of its outputs, the value of the net is determined by logic within the module. If the identifier of a net is the left-hand side of a continuous assignment statement, the value assigned to the net is specified by a Boolean expression that uses operands and operators.

As an example, assuming that the variables were declared, a two-to-one-line multiplexer with scalar data inputs *A* and *B*, select input *S*, and output *Y* is described with the continuous assignment

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

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

The next two examples show the dataflow models of the two previous gate-level examples. The dataflow description of a two-to-four-line decoder with active-low output-enable and inverted output is shown in [Example 4.3](#). The circuit is defined with four continuous assignment statements using Boolean expressions, one for each output. The dataflow description of a four-bit adder is shown in [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 separates the operands with a comma. Thus, { *C\_out* , *Sum* } represents the five-bit result of the addition operation.

## VHDL (Predefined Data Types)

[Table 4.12](#) lists the predefined data types of VHDL. String literals require that their characters be enclosed in double quotes. There are two ways to write a *bit\_vector* literal. One way is to write it as a comma-separated string of bits. For example, ('1', '1', '0', '0'). A second way is to write it as a string literal: "1100".

### Table 4.12 *Predefined VHDL Data Types*



VHDL Data Type	Value
bit	'0' or '1'
boolean	FALSE or TRUE
integer	$-(2^{31} - 1) \leq \text{INTEGER VALUE} \leq (2^{31} - 1)$
positive	$1 \leq \text{INTEGER VALUE} \leq (2^{31} - 1)$
natural	$0 \leq \text{INTEGER VALUE} \leq (2^{31} - 1)$
real	$-1.0 \text{ e } 38 \leq \text{FLOATING POINT VALUE} \leq 1.0 \text{ E } 38$
character	Alphabetical characters (a . . . z, A . . . Z), digits (0, . . . 9), special characters (e.g., %) each enclosed in single quotes
time	integer with units fs, ps, ns, us, ms, sec, min, or hr

## VHDL (Vectors, Arrays)

A VHDL identifier having multiple bits is a one-dimensional [16](#) array, also called a *vector*. An array is an ordered set of elements of identical type, uniquely identified by their index. The *bit range* of the indices of a vector determines the number of bits. For example, *A(7 **downto** 0)* and *B(0 **to** 7)* each hold eight bits. The indices of an array are integers. An array must be declared as a named object of a named array type. For example,

[16](#) VHDL also supports multi-dimensional arrays; the examples in this text do not make use of that feature.

```
type Opcode is array (7 downto 0) of bit;  
signal Arith: Opcode := "10000110";  
constant code_2: Opcode := "01011010";
```

Here, *Opcode* is a declared type of 8-bit vectors. *Arith* has type *Opcode* and is initialized to 10000110. A vector that is not initialized in its declaration is initialized by default to all '0' bits. The elements of a vector can be initialized individually by including them in a parentheses-enclosed, comma-separated list of values, each value enclosed by '. For example, *C* := ('1', '0', '0', '1') defines a vector *C* having value 1001<sub>2</sub>. It is optional to specify elements of a vector by explicitly indicating index-pairs of elements. For example, *D* := (0 => '1', 1 => '1', 2 => '0', 3 => '1') specifies *D* having value 0101<sub>2</sub>, given that *D* was declared to have a bit range of 0 to 3. In this notation, the keyword **others** assigns values to elements that have not been assigned by their index. For example, *D* := (0, 2 => '0', **others** => '1') creates *D* having value 0101<sub>2</sub>. If desired, all of the bits of a vector can be initialized to '1' as follows:

```
signal Arith: Opcode := (others => '1');
```

The elements of a vector can be referenced by a parentheses-enclosed index. For example, *Arith*(2) is the third bit from the right. A contiguous range of elements, called a *slice*, can be addressed too: *Arith* (6 **downto** 4) is a three-bit wide sub-array of *Arith*.

The syntax template for declaring arrays is as follows:

```
type array_type_name is array (start_index to end_index) of arr  
type array_type_name is array (start_index downto end_index) of  
type array_type_name is array (range_type range range_start to  
type array_type_name is array (range_type range range_star
```

Some examples are

```
type Nibble is array (3 downto 0) of bit;  
signal Nib_1: Nibble;
```

```
type Data_word is array (15 downto 0) of bit;  
signal word_1: Data_word := "0011001111001100";
```

The assignment `Nib_1 <= Data_word(15: 12)` gives `Nib_1 = " 0011 "`.



## VHDL (Predefined Operators, Concurrent Signal Assignment)

Dataflow models in VHDL are composed of *concurrent signal assignment* statements. The simplest form of a concurrent signal assignment statement has the syntax template:

```
signal_name <= expression [after delay];
```

An expression in a signal assignment is composed of Boolean operators and variables. VHDL has the set of predefined operators shown in [Table 4.13](#). The table is organized with operators having the lowest priority occupying the first row, and those having highest priority in the bottom row, that is, priorities increase from top to bottom in the table.

### Table 4.13 *VHDL Operators*

Operator Type	Symbol	Operand(s)	Result	Precedence
Binary Logical	<b>and or nand nor xor xnor</b>	Bit, boolean, boolean_vector, bit_vector,	<b>Same as operands</b>	
Relational	<b>= / = &lt; &lt; = &gt; &gt; =</b>	Two expression matched in type and size	<b>FALSE, TRUE</b>	
Shift Operators	<b>sll srl sla sra rol ror</b>	bit_vector	bit_vector	
Addition Operators	<b>+ -</b>	Integer Real number	Integer Real number	
Concatenation Operator	<b>&amp;</b>	Vectors	Vectors	
Unary Sign Operator	<b>+ -</b>			
Multiplication Operators	<b>./mod rem</b>			
Miscellaneous Operators	<b>not abs ..</b>	Numerical Numerical Integer, Floating Point	Exponentiated by integer	

## Description

The signal assignment statements within an architecture are continuously active and execute concurrently. By continuously active we mean that the simulator continuously monitors the signals in the RHS expression of a concurrent signal assignment and evaluates it when a change occurs in one or more of them. In simulation, the *signal assignment operator* ( **<=** ) determines the value of the left-side named signal by evaluating the *expression* on the RHS. The value is assigned after an optional time delay. [17](#) If a delay is specified, the assignment of value is after the execution and evaluation of the *expression*, at a time determined by *delay*. When is the expression evaluated? The event scheduling mechanism of a logic simulator is triggered by events in the signals in the RHS expression.

[17](#) The square brackets in the syntax template of a signal assignment statement denote an optional part of the statement. The content enclosed by the brackets, but not the brackets, are part of the statement.

*An event is a change in the value of a signal.* When an event occurs in the RHS of a signal assignment statement, the simulator (1) suspends

execution, (2) evaluates the expression using the current value of any signals that are referenced in the expression, (3) assigns value to the named signal at the left side of the statement, and then (4) resumes execution. This mechanism mimics a physical circuit, where a change in an input triggers a causal chain of events as the effects of the change propagate through the gates of a circuit, that is, a relationship exists between an event and another event that triggered it. Subsequently triggered events can be ordered according to when they are triggered relative to other events. That ordering is sometimes described as having events *scheduled* and separated by an infinitesimal “delta” delay, which establishes an ordering in the underlying data structures of the simulation. Those structures can be viewed as a doubly linked list of structures consisting of ordered values of time and lists of events that occur at a given time.

The delay in a signal assignment statement is called an *inertial delay* because successive changes in the value of the RHS expression will not cause changes in the LHS signal if the interval of time between successive changes in the RHS expression is too small. The (optional) *delay* given in a signal assignment statement determines the minimum interval between successive changes in the RHS expression that will cause successive changes in the LHS signal. Inertial delay models the physical behavior of gates whose outputs do not change if the duration of an input transition is brief. The input transition must persist sufficiently long for it to have an effect.

Another kind of delay mechanism, called *transport delay*, [18](#) causes an event to be scheduled for the LHS signal regardless of the duration of the interval between successive changes in the value of the RHS expression. [19](#) To express transport delay, a signal assignment statement is modified by the keyword **transport** to have the following form:

[18](#) Sometimes referred to as a *pipeline* delay.

[19](#) Inertial delay is the default mechanism for propagation delay.

```
signal_name <= transport expression after delay;
```

Delay modeling can be useful in simulation, but synthesis tools ignore the “after” clause of a signal assignment because they implement only functionality, not an implied physical characteristic that is technology-

dependent. A synthesized device inherits whatever delay the technology dictates.

The port of an entity defines the signals by which the entity interacts with the external world. The logic within an architecture may use the input signals of an entity and may declare additional signals that are used in composing a description of the functionality of the circuit. The simplest form of a signal declaration statement in VHDL uses the keyword **signal** and has the syntax template:

```
signal list_of_signal_identifiers: type_name [constraint] [:= i
```

The optional constraint is used to denote the index range of a vector (e.g., 7 **downto** 0), or a range of values (e.g., **range** 0 **to** 3). The optional initial value provides a value for the simulator to use when the simulation begins. [20](#) A signal that is declared in an architecture may not be listed in the port of an entity that is paired with the architecture. Moreover, a signal may be referenced in only the architecture in which it is declared. Here are some examples of signal declarations:

[20](#) The default initial value of an **integer** is 0.

```
-- 16-bit vector initialized to 0:  
signal A_Bus: bit (15 downto 0) := '0000000000000000';  
-- An integer whose value is between 0 and 63:  
integer C, D: integer range 0 to 63;
```

When the value of a declared signal is outside its specified range a VHDL compiler will cite an error condition.

VHDL *constants* may be declared at the start of the code of an architecture, and may be referenced anywhere within the architecture. The simplest form of a constant declaration statement uses the keyword **constant**: and has the syntax template

```
constant constant_identifier: type_name [constraint] := constan
```

Constants are used to simplify and clarify VHDL code. They may not be reassigned a value. Here are some examples of constant declarations:

```
constant word_length : integer := 64;
constant prop_delay: time := 2.5 ns;
```

## HDL Example 4.4 (Dataflow: Two-to-Four Line Decoder)

### Verilog

```
// 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
```

### VHDL

```
-- Dataflow description of two-to-four-line decoder—See
Fig. 4.19
. Note: The figure uses
-- symbol E, but the VHDL model uses enable to clearly indicate
```

```
entity decoder_2x4_df_vhdl is
    port (D: out Std_Logic_Vector (3 downto 0); A, B, enable: in S
end decoder_2x4_df_vhdl;
```

```
Architecture Dataflow of decoder_2x4_df_vhdl is
```

```

begin
  D(0) <= not ((not A) and (not B)      and (not enable));
  D(1) <= not (not A) and B              and not (enable);
  D(2) <= not (A and (not B)            and (not enable));
  D(3) <= not (A and B                  and (not enable));
end Dataflow;

```

## HDL Example 4.5 (Dataflow: Four-Bit Adder)

### Verilog

```

// Dataflow description of four-bit adder
// Verilog 2001, 2005 module port syntax
module binary_adder (
  output C_out,
  output [3: 0] Sum,
  input [3: 0] A, B,
  input C_in
);
  assign {C_out, Sum} = A + B + C_in    // Continuous assignmen
endmodule

```

In *binary\_adder*, Verilog automatically accommodates the addition of the words, even though they have different sizes and are, strictly speaking, of different types.

### VHDL

```

-- Dataflow description of four-bit adder
entity binary_adder is
  port (Sum: out Std_Logic_Vector (3 downto 0); C_out: out Std_L
        A, B: in Std_Logic_Vector (3 downto 0); C_in: in Std_L
  end binary_adder;

```

```

architecture Dataflow of binary_adder is
  begin
    C_out & Sum <= A + B + ('000' & C_in);    -- Compatible wor

```



```
end Dataflow;
```

## HDL Example 4.6 (Dataflow: Four-Bit Comparator)

A 4-bit magnitude comparator has 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  $B$ .

### Verilog

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

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

### VHDL

```
-- Dataflow description of four-bit comparator

entity mag_compare is
  port (A_lt_B, A_eq_B, A_gt_B: out Std_Logic; A, B: in Std_Logic)
end mag_compare;

architecture Dataflow of mag_compare is
begin
  A_lt_B <= (A < B);
  A_gt_B <= (A > B);
  A_eq_B <= (A = B);
end Dataflow;
```

A synthesis compiler can accept these dataflow descriptions 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, and with assurance that the schematic is correct.

## Verilog (Conditional Operator)

A Verilog conditional operator takes three operands [21](#):

[21](#) The conditional operator is a ternary operator, requiring three operands.

```
condition ? true_expression : false_expression  
;
```

The condition is evaluated. If the result is logic 1, *true\_expression* is evaluated and used to assign a value to the LHS of an assignment statement. If the result is logic 0, *false\_expression* is evaluated, and the result is assigned to the LHS. The two conditions together are equivalent to an if-else condition.

## VHDL (Conditional Signal Assignment)

The VHDL *conditional* signal assignment selects between two possible assignments, depending on the evaluation of a condition.

## HDL Example 4.7 (Dataflow: Two-to-One Multiplexer)

### Verilog

```
// 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;           // Conditional operator
endmodule

```

## VHDL

```

-- Dataflow description of two-to-one multiplexer
entity mux_2x1_df_vhdl is
  port (m_out: out Std_Logic; A, B, select: in Std_Logic);
end mux_2x1_df_vhdl;

architecture Dataflow of mux_2x1_df_vhdl is
begin
  m_out <= A when select = '1'; else B;      // Conditional signal assignment
end Dataflow;

```

## 4.13 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. Behavioral models execute one or more *procedural statements* when launched by a sensitivity mechanism, commonly called a *sensitivity list*, which monitors signals and launches execution of the behavioral description. Procedural statements are like those found in other programming languages, for example, assignments and statements which control the sequence of execution, for example, **for**, **loop**, **case**, and **if** statements. This section considers behavioral modeling of combinational logic. Behavioral modeling of sequential logic will be considered in later chapters.

### Verilog (Procedural Assignment Statements)

Verilog behavioral descriptions of hardware are declared with the keyword **always**, followed by an optional *event control expression* (sensitivity list) and a **begin . . . end** block of procedural assignment statements. [22](#) Verilog has two types of assignment statements: *continuous* and *procedural*. We have seen that continuous assignments use the keyword **assign** and the = operator. Procedural assignments are those made within the scope of an **always** or **initial** procedural statement. Procedural assignments may use the blocking assignment operator =, or the nonblocking assignment operator <=, depending on whether the assignment represents sequential behavior or concurrent behavior. The event control expression in a procedural statement effectively specifies *when* the associated statements will begin to execute, because it suspends execution of the procedural statement until one or more of the signals in the expression has an event (qualified or otherwise). In its absence, the associated statements begin execution immediately at the beginning of simulation.

[22](#) The keyword **initial** is used to write behaviors for a testbench, but not

to model hardware. The term *procedural assignment* distinguishes assignments made within an **always** or **initial** block from those made by *continuous assignment* statements.

## VHDL (Process Statements, Variables)

In addition to concurrent signal assignment statements and instantiation of components, a VHDL *process* provides a third mechanism for describing concurrent behavior. A **process** is formed by the keyword **process**, accompanied by an optional *sensitivity list*, and followed by declarations, definitions, and a **begin . . . end process** block of statements. The statements within a process are referred to as *procedural* statements and as *sequential* statements—they are like (procedural) statements in other programming languages, and they execute (sequentially) in the order in which they are listed. Behavioral models of combinational circuits can be implemented in VHDL with a *process* statement. In this section we consider only combinational logic; later chapters will consider synchronous sequential logic in the context of finite state machines.

VHDL processes execute *concurrently* with other (1) process statements, (2) concurrent signal assignment statements, and (3) instantiated components. The assignment statements within a process execute *sequentially* in the order in which they are listed with other statements in the process. The syntax template for a process is given below:

```
process (signal_name, signal_name, . . . , signal_name)
  type_declarations
  variable declarations
  constant_declarations
  function_declarations
  procedure_declarations
begin
  sequential_assignment statements
end process
```

In simulation a process executes once immediately, at  $t = 0$ , and then pauses until one or more of the signals in its sensitivity list changes. When

that occurs the process becomes active again.

There are two types of sequential assignment statements: variable assignments and signal assignments. A *variable* is a storage container similar to a signal, but not having a physical connotation of connecting the structural elements of a circuit or dynamically holding a logic value that is determined by the circuit. It merely holds data, like a variable in other program languages. By implication, the value of a variable can change. A declaration of a variable has the same syntax as the declaration of a signal, but with the keyword **variable**:

```
variable list_of_names_of_variables: type_of_variable;
```

For example, **variable** *A, B, C*: **bit** declares three variables having type **bit**. Note: signals may not be declared in a process, but a variable may be declared.

A *variable assignment* has the same syntax as a signal assignment, but uses a different assignment operator (**:** **=** ). For example, *c o u n t* **:=** ' 5 ' .

The variable assignment statements in a process execute when they are encountered in the ordered list of statements; the effect of their execution is immediate—that is, memory is updated. In contrast, signal assignment statements in a process are evaluated immediately, when they are encountered, but their effect is not assigned until the process terminates. This distinction will be discussed in more detail later.

A process can model combinational (i.e., level-sensitive) logic, and sequential logic (e.g., edge-sensitive), such as the logic describing a flip-flop in a synchronous state machine. Remember, a process executes once at the beginning of simulation; thereafter, its sensitivity list determines when the associated **begin** . . . **end** block statement will execute—the process executes when a signal in its sensitivity list changes. For example, the statements associated with the sensitivity list **@** (*clock*) will start executing when *clock* has an event.

Next, HDL [Examples 4.8](#) and [4.9](#) present behavioral models of combinational logic. Behavioral modeling is presented in more detail in [Section 5.6](#), after sequential circuits. [HDL Example 4.8](#), alternative dataflow description of a two-to-four-line decoder, uses a level-sensitive

procedural statement instead of continuous assignments (see [HDL Example 4.4](#)).

## HDL Example 4.8 (Behavioral: Alternative Two-to-Four Line Decoder)

### Verilog

```
module decoder_2x4_df_beh (    // Verilog 2001, 2005 syntax
    output  [0: 3] D,
    input    A, B,
            enable
);
always @ (A, B, enable) begin

    D[0] <= (!((!A) && (!B) && (!enable))),
    D[1] <= (!((!A) && B && (!enable))),
    D[2] <= !(A && (!B) && (!enable)),
    D[3] <= !(A && B && (!enable));
end;
endmodule
```

With nonblocking ( `<=` ) assignments, the order in which the statements assigning value to the bits of *D* are listed does not affect the outcome.

### VHDL

```
entity decoder_2x4_df_beh_vhdl is
    port (D: out Std_Logic_Vector (3 downto 0); A, B, enable: in
end decoder_2x4_df_vhdl;
```

```
Architecture Behavioral of decoder_2x4_df_beh_vhdl is
```

```

begin
  process (A, B, enable) begin
    D(0) <= not ((not A) and (not B)      and (not enable));
    D(1) <= not (not A) and B      and not (enable);
    D(2) <= not (A and (not B)      and (not enable));
    D(3) <= not (A and B      and (not enable));
  end Behavioral;

```

## HDL Example 4.9 (Behavioral: Two-to-One Line Multiplexer)

### Verilog (Procedural Statement)

```

// 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)      // Alternative: always @ (A
    if (select == 1) m_out = A;
    else m_out = B;
endmodule

```

The signal *m\_out* in *mux\_2x1\_beh* must be of the **reg** data type, because it is assigned value by a Verilog procedural assignment statement. Contrary to the **wire** data type, whereby the target of an assignment may be continuously monitored and updated, *a **reg** data type is not necessarily monitored, [23](#) and retains its value until a new value (in simulation memory) is assigned.* Historically, the type-name **reg** has been a source of confusion to designers because it suggests that a **reg**-type variable corresponds to a hardware register. It may, but not necessarily so. This confusion is also due to the family of variables being referred to as a *register* family, which conveys the semantic of data storage. Our later discussion of synthesis will relate synthesis outcomes to coding. [24](#)

[23](#) A variable having type **reg** will be monitored if it appears in an event control expression.



[24](#) SystemVerilog circumvents this issue by defining a new data type, **logic**, which has no reference to hardware and has no implication for memory.

The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variables listed in the sensitivity list after the **@** symbol. (Note that there is no semicolon (;) at the end of the **always** statement.) In this example, 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 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) m_out = A;
```

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

## VHDL (process, if Statement)

The combinational logic of a two-channel multiplexer can be modeled by a VHDL process statement. The process below executes when a change occurs in the value of *A*, *B*, or *select*. A value assigned to *m\_out* by the process is retained in memory until a subsequent execution of the process changes it. [25](#)

[25](#) A concurrent signal assignment in the body of an architecture gets a value whenever the RHS changes; a signal assignment in the body of a process gets its value when a signal assignment statement executes, and it retains that value until a subsequent signal assignment executes and changes the stored value.

```
-- VHDL behavioral description of two-channel multiplexer
entity mux_2x1_beh_vhdl is
  port (m_out: out Std_Logic; A, B: in Std_Logic;
        select: in Std_Logic);
end mux_2x1_beh_vhdl;
```

```
Architecture Behavioral of mux_2x1_beh_vhdl is
begin
```

```

process (A, B, select) begin
  if select = '1' then m_out <= A; else m_out <= B; end_if;
end process;
end Behavioral;

```

The syntax template for the **if** statement in VHDL has several forms:

(1) *if boolean\_expression then sequential\_statements*  
*end if;*

(2) *if boolean\_expression then sequential\_statements*  
*else sequential\_statements*  
*end if;*

(3) *if boolean\_expression then sequential\_statements*  
*elsif boolean\_expression then sequential\_statements*  
*· · ·*  
*elsif boolean\_expression then sequential\_statements*  
*end if;*

(4) *if boolean\_expression then sequential\_statements*  
*elsif boolean\_expression then sequential\_statements*  
*· · ·*  
*elsif boolean\_expression then sequential\_statements*  
*else sequential\_statements*  
*end if;*

## HDL Example 4.10 (Behavioral: Four-to-One Line Multiplexer)

This example provides behavioral descriptions of a four-to-one-line multiplexer. A two-bit vector input, *select*, determines which of the four input channels provides value to the output.

### Verilog

```

// 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,
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

```

## VHDL

```

-- VHDL behavioral description of four-channel multiplexer
entity mux_4x1_beh_vhdl is
  port (m_out: out Std_Logic; in_0, in_1, in_2, in_3: in Std_Log
    select: in Std_Logic_Vector (1 downto 0));
end mux_4x1_beh_vhdl;

```

```

Architecture Behavioral of mux_4x1_beh_vhdl is
begin
  process (in_0, in_1, in_2, in_3, select) begin
    case select is
      when 0 => m_out = '0';
      when 1 => m_out = '1';
      when 2 => m_out = '2';
      when 3 => m_out = '3';
      when others => m_out = '0';
    endcase;
  end process;
end Behavioral;

```

## VHDL (Conditional and Selected Signal Assignments)

The process in *mux\_4x1\_beh\_vhdl* in [HDL Example 4.10](#) is equivalent to the following conditional signal assignments:

```

m_out <= in_0 when select = '00'; else
m_out <= in_1 when select = '01'; else
m_out <= in_2 when select = '10'; else
m_out <= in_3 when select = '11'; end if;

```

Another alternative process using a *selected signal assignment* is given below. [26](#)

[26](#) A single identifier *m\_out* receives value; in general, an expression can be assigned to the LHS in a selected signal assignment statement.

channel select <= A & B; -- a previously declared channel selector signal

```
process (in_0, in_1, in_2, in_3, channel_select) begin
with channel_select select
m_out <=      in_0 when channel_select = '00',
              in_1 when channel_select = '01',
              in_2 when channel_select = '10',
              in_3 when channel_select = '11',
              '1' when others;    // Use if channel_select is not
end process;
```

The syntax template for a *selected signal assignment* is given below:

```
with expression select
signal_name <=  value when choices,
                value when choices,
                value when choices;
```

## Verilog (case, casex, casez Statements)

Signal *m\_out* in *mux\_4x1\_beh* is declared to have type **reg** because it is assigned value by a procedural statement. It will retain its value until it is explicitly changed by a procedural statement. 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. (Alternatively, a **default** case item and an associated case expression can be included in the list to ensure that a statement will always be executed.) Since *select* is a two-bit number, it can be equal to 00, 01, 10, or 11. Note: the **case** items have an implied priority because the list is evaluated from top to bottom.

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

The list of case items need not be complete. If the list of **case** items does not include all possible bit patterns of the **case** expression, no match can be detected. Unlisted **case** items, that is, 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.

Industry practice has concluded that it is ill-advised to use the case x or case z constructs in RTL code that is intended to be synthesized. These constructs consider don't-care bits in both the case expression and the case item. Synthesis tools do not treat the case expression as having don't cares, that is, each bit is either a specified 0 or 1. Consequently, code that uses case x or case z might have mismatches between the results produced by a synthesized circuit and the results produced by simulation. Such mismatches are difficult and costly to detect. SystemVerilog addresses this issue.

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](#).

The event control expression is also called a *sensitivity list* (Verilog 2001, 2005) when it is expressed as a comma-separated list that is equivalent to an event-OR expression. Both forms express the fact that combinational

logic is reactive—it senses a change in an input signal, and when an input changes an output may change.

## VHDL (case Statement)

The sensitivity list of the process in *mux\_4x1\_beh\_vhdl*, the model of the four-channel multiplexer, is sensitive to a change in any of the data channels, and a change in the bits of *select*. When a change is detected, a **case** statement tests the bits of *select*, in sequence, to check whether they match the select bus of the multiplexer. If so, the data into that channel is steered to the output.

*Choices* represents a single value or a list of values separated by vertical bars, that is, the expression may be tested against several possible choices. For example, the statement `signal_name <= '1' when A & B = '00' or A & B = '10' ;` [28](#) considers two values of the concatenation A&B. The effect of the statement is to compare the expression to a listed choice. At the first match the value is assigned to the named signal. A restriction of the selected signal assignment statement is that the *choices* must be mutually exclusive and must exhaust all possibilities for the result of evaluating the expression. The keyword *others* can be used in the last *when* clause to cover values of expression that are not explicitly cited.

[28](#) Remember, **&** is the VHDL operator for concatenation.

The syntax template for the case statement is

```
case expression is
  when case_choice_1 => sequential_statement1

  when case_choice_2 => sequential_statement2

  when case_choice_3 => sequential_statement3

  . . .
[when others b sequential_statement1]
end case;
```

The case statement requires that the case choice explicitly include all possible values of the case expression. If they are not listed, the “**others**”

clause is required (shown here in square brackets as an option). If the choices associated with “**others**” do not require action, the null statement should be used, that is, `w h e n o t h e r s => n u l l ;`

## 4.14 WRITING A SIMPLE TESTBENCH

A testbench is an HDL program that describes and applies a stimulus to an HDL model of a circuit to test it and to observe its response during simulation. Testbenches 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 testbench that is used to test a circuit, so care must be taken to write stimuli that will test a circuit thoroughly, exercising all of the operating features that are specified. The examples presented here demonstrate some basic features of HDL stimulus models. [Chapter 8](#) considers testbenches in more depth.

### Verilog

In addition to employing the **always** statement, Verilog testbenches 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, as a loop. *The **initial** statement executes only once*, starting from simulation time 0, and may continue executing with any assignments that are delayed by a given number of time units, as specified by the symbol #. The statement expires when the last statement in its block executes, which may or may not coincide with the end of simulation. 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**. The blocking



assignment statements within the block are processed sequentially, subject to the delay control operator `#`. This operator has the effect of suspending the simulator until the associated time has elapsed. Then the simulator resumes operation. In reality, nothing is suspended or turned off; the delay control operator affects the scheduling of the assignment created by the next assignment statement as though the simulator was suspended. 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. As another example, 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
```

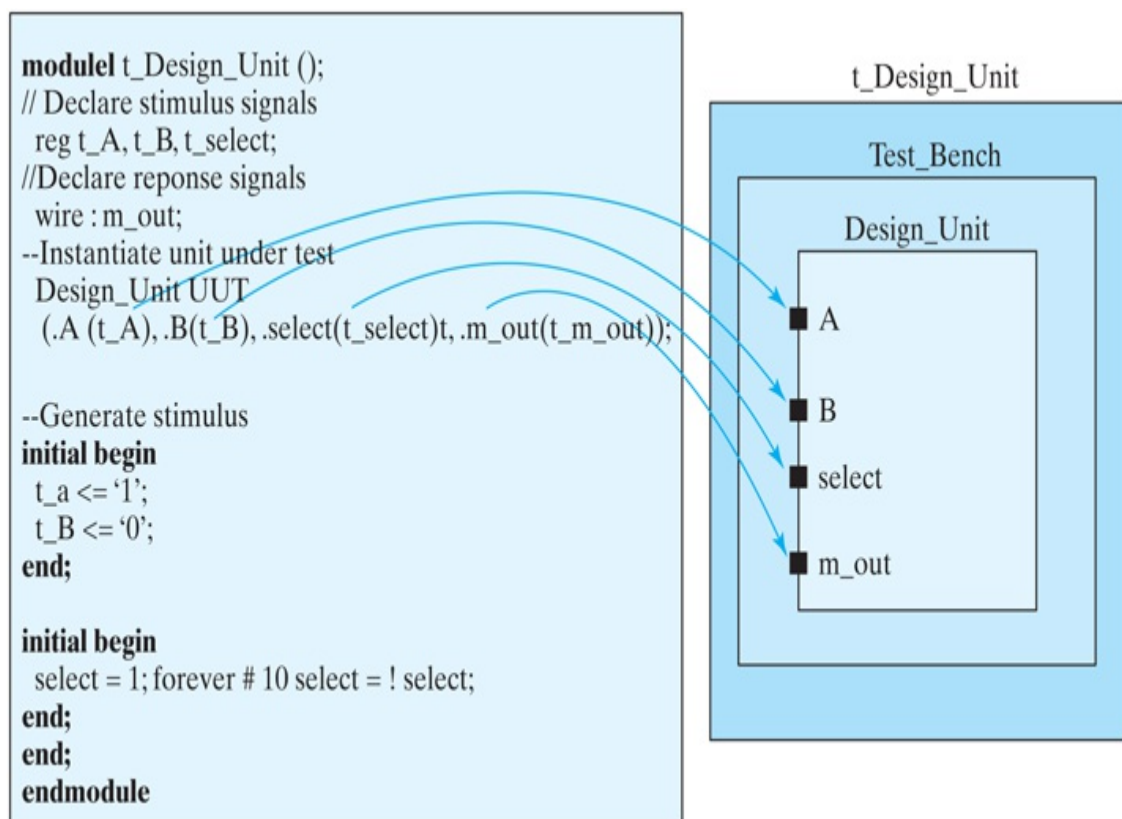
When the simulator runs, the three-bit vector *D* is initialized to 000 at time = 0. The keyword **repeat** specifies a looping statement: *D* is incremented by 1 seven times, once every 10 time units. The result is a sequence of binary numbers from 000 to 111.

A simple 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 simul
  // 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 unit under test (UUT) for simulation are declared in the stimulus module as local **reg** data type. Each output of the design module that is displayed for testing is 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 4.37](#) clarifies this relationship between the formal signals of the unit being

tested and the actual signals declared locally in the testbench. 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 stimulate the design module being tested. The simulator associates the (actual) local identifiers of the inputs within the testbench,  $t\_A$ ,  $t\_B$ , and  $t\_C$ , with the formal identifiers of the module ( $A$ ,  $B$ , and  $C$ ). 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 also provides a more flexible *name association* mechanism for connecting ports in larger circuits. It will be demonstrated in later examples.



## FIGURE 4.37

Interaction between testbench and Verilog design unit

### [Description](#)

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, which 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  
**\$write**—same as **\$display**, but without going to next line,  
**\$monitor**—display variables whenever a value changes during a si  
**\$time**—display the simulation time, and  
**\$finish**—terminate the simulation.

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

```
T a s k - n a m e ( f o r m a t s p e c i f i c a t i o n , a r g u m e n t l i s t )  
;
```

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 ( *t i m e* = ), ( *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.)

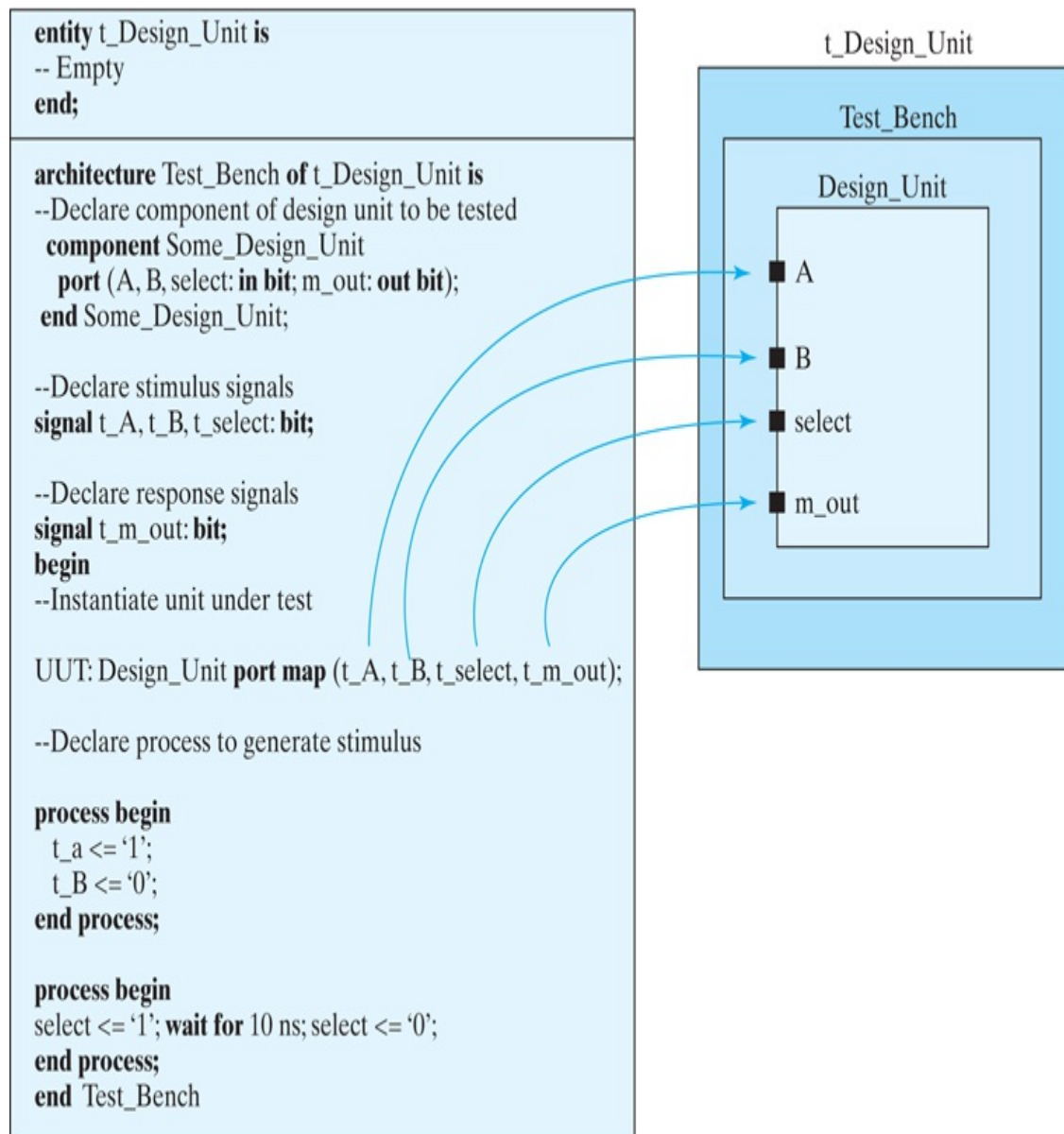
An example of a stimulus module is shown in [HDL Example 4.9](#). The circuit to be tested is the two-to-one-line multiplexer described in [Example 4.6](#). The module `t_mux_2x1_df` has no ports. The inputs for the mux are declared with a **reg** keyword and the outputs with a **wire** keyword. The 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, outputs, and time. The result of the simulation is listed under the simulation log in the example. It shows that `m _ o u t = A` when `s e l e c t = 1` and `m _ o u t = B` when `s e l e c t = 0` verifying the operation of the multiplexer.

The fine print of the specification for Verilog 1995 indicates that the order in which multiple **initial** or **always** behaviors execute is not determined by the language itself, but depends on the implementation of the simulator. This means that the designer cannot depend on the listing of procedural blocks to determine the order in which they will execute by a simulator, so having initialization of variables depend implicitly on such an ordering is not advisable and may lead to unexpected results in simulation. Verilog 2001 allowed variables to be initialized when they are declared. For example, **integer** `k = 5` ; declares an integer, `k`, and specifies its initial value. However, the order in which such declarations will be executed relative to initial procedural blocks is not specified, and so the initial value of such variables is not deterministic. SystemVerilog eliminates this issue by specifying that all variables that are initialized in their declarations will be evaluated prior to the execution of any events at the start of simulation time zero.

## VHDL

A VHDL testbench is an entity-architecture pair written specifically to apply stimulus signals to verify the functionality of a design. The entity of a testbench is self-contained—it does not have inputs or outputs. The architecture of a testbench includes an instance of the design *unit under test* (UUT), and VHDL process statements that generate signals to test the design. A simulator applies the input signals to the UUT, and presents text

or graphical data describing the response of the UUT to the stimulus. Logic simulators having graphical output can display the signals at the level of the testbench and at levels of the hierarchy within the UUT. [Figure 4.38](#) shows the relationship between a VHDL testbench and the UUT, and the association of local signals with the formal names of the signals in the port of the UUT.



## FIGURE 4.38

Interaction between testbench and VHDL design unit

[Description](#)

In [Fig. 4.38](#) the UUT is instantiated as a component in the architecture of the testbench. The signals that are applied to the UUT and the signals that are outputs of the UUT are declared within the architecture of the testbench. A process asserts values for the stimulus signals (i.e., the data channels); a second process generates *select*, which is specified to assert a value of ‘1’ when simulation begins, and to switch to a value of ‘0’ after 10 ns have elapsed.

The stimulus signals are local to the testbench. For clarity, they can be named by adding the prefix *t\_* to the signals in the port of the UUT. Either concurrent signal assignments or process statements can provide the values of the inputs to the UUT.

## HDL Example 4.11 (Testbench)

### Verilog

```
// Testbench 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
    // Alternative association of ports by name:
    // mux_2x1_df M1 (.mux_out (t_mux_out), .A(t_A), .B(t_B), .select(t_select));

    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_mux_out);
        $monitor (" time = ", $time,, " t_select = %b t_A = %b t_B = %b t_mux_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:

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

Note that 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.

#### VHDL

```
-- Testbench with stimulus for mux_2x1_df_vhdl
entity t_mux_2x1_df_vhdl is
    port ();
end t_mux_2x1_df_vhdl;

architecture Dataflow of t_mux_2x1_df_vhdl is
    signal t_A, t_B, t_C: Std_Logic;
    signal select: Std_Logic_Vector (1 downto 0);
    signal t_mux_out: Std_Logic;
    component mux_2x1_df_vhdl
        port (A, B: in Std_Logic; C: out Std_Logic; select: in Std_L
    begin

    -- Stimulus signal assignments
    t_select <= 1; t_A <= 0; t_B <= 1;
    wait 10 ns;
    t_A <= 1; t_B <= 0;
    wait 10 ns;
    t_select <= 0;
    wait 10 ns;
    t_A <= 0; t_B <= 1;
end Dataflow;

    -- Instantiate UUT
    M0: mux_2x1_df_vhdl port map (A => t_A, B => t_B, C => t_C,
end Dataflow;
```

## 4.15 LOGIC SIMULATION

Logic simulation provides a fast and accurate method of verifying that a model of a combinational circuit is correct. It creates a visual representation of the behavior of a digital circuit by computing and displaying logic values corresponding to electrical waveforms in physical hardware.

There are two types of verification: functional and timing. In *functional* verification, we study the logical operation of the circuit independently of physical timing delays of gates, using so-called zero-delay models, which ignore the propagation delay of physical gates. *Timing* verification studies a circuit's operation by including the effect of delays through gates. The process determines whether the specification for the operating speed of the circuit can be met. For example, it must determine that the clock frequency of a sequential circuit is not compromised by the propagation delay of signals from a source register passing through combinational logic before reaching a destination register. Timing verification is beyond the scope of this text.

Logic simulation is usually accomplished with event-driven simulators. At any instant of time most signals (gate outputs) in digital hardware are quiescent, that is, they do not change value. Since relatively few gates change at any time, logic simulators exploit this topological latency by using an “event-driven” scheme in which computational effort is expended only at those times at which one or more signals change their value. Event-driven simulation is the main reason why it is feasible to simulate the logical behavior of circuits containing millions of logic gates.

An event is said to occur in a sequential circuit when a signal undergoes a change in value. A simulation of a digital circuit is said to be “event-driven” when the activity of the simulator is initiated only at those times when the signals in the model experience a change. Rather than recomputing the values of all signals at prescribed time steps, as in analog simulation, event-driven digital simulation computes new values of only those signals that are affected by the events that have already occurred, and only at those times when changes actually occur. For example, a change on one or both of the inputs to the **and** gate in [Fig. 4.39](#) might



cause its output to change value (according to the input/output truth table for the **and** gate in the simulator's logic system). Subsequently, this change causes the output of the **not** gate to change. The simulator monitors signals *A* and *B*, and when they change it determines whether to *schedule* a change for signal *C*. When the scheduled change in signal *C* occurs, the simulator schedules an event for signal *D*, and so on. It is characteristic of event-driven simulation that events on the circuit's input signals propagate through the circuit, and possibly to its outputs. At a given time step of the simulator, events are propagated and scheduled until no events remain to be scheduled at the present time or a future time. The action at the present time of evaluating and scheduling future events is referred to as a *simulation cycle*.



## FIGURE 4.39

Circuit for event-driven simulation

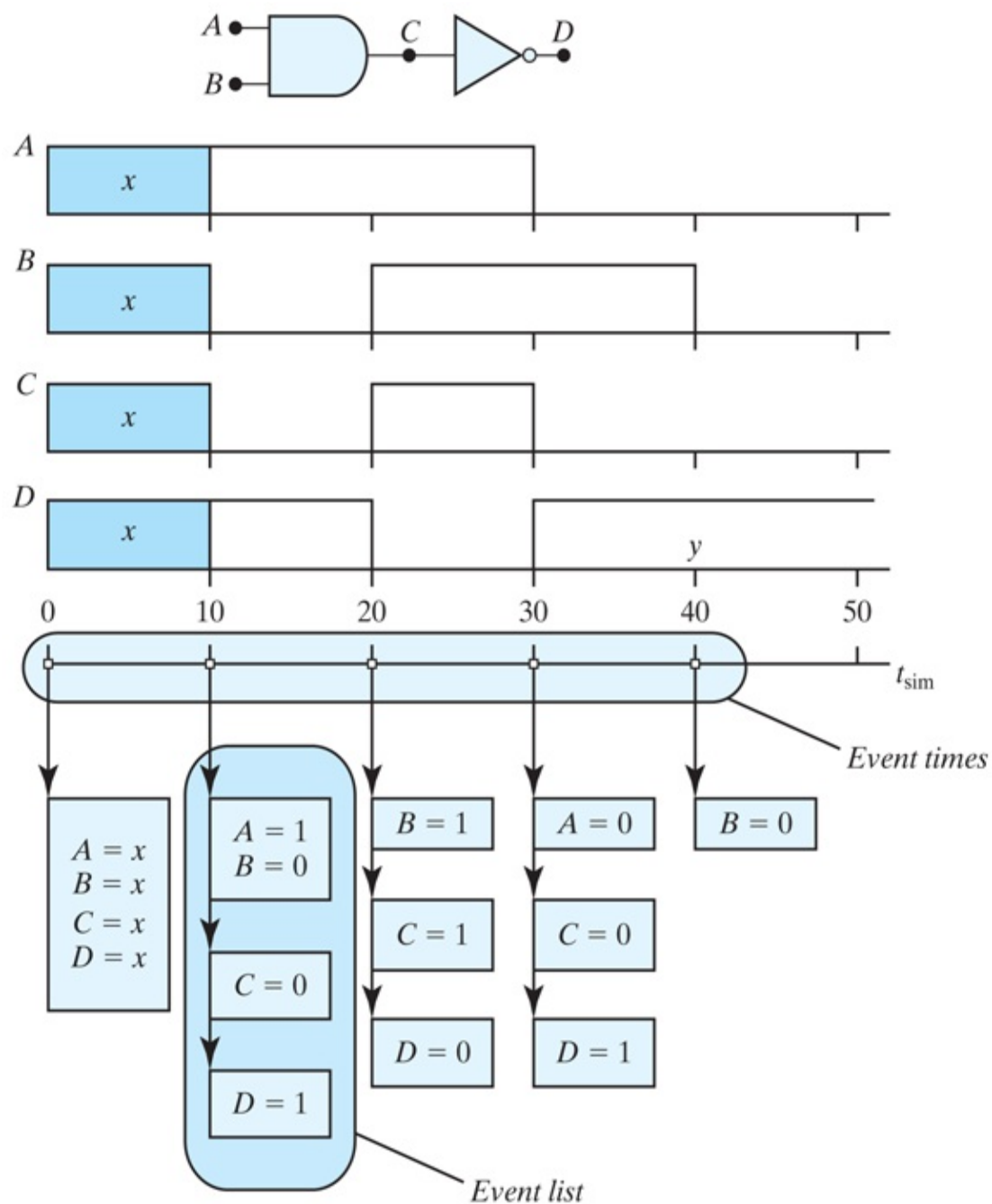
When a signal in the circuit being simulated changes value an elaborate set of data structures enables the simulator to consider updating only those signals that could be affected by the event. The remaining signals are ignored because there is no need to recompute their values. A logic simulator creates and manages an ordered list of “event-times,” that is, those discrete times at which events have been scheduled to occur. An “event queue,” (i.e., “signal-change” list, *sig\_ch(t)*) is associated with each of the event times. It consists of the names and new values of those signals that are to change at that time. Events at a given time step may cause additional events to be scheduled at the present time, but later in the queue. When the queue is empty and there are no more events to be scheduled, the simulator advances time to the next time at which an event exists in the queue of events at that time.

At the beginning of a simulation, a simulator automatically creates an initial event-time list at time  $t_{sim} = 0$ . All variables are assigned their initial value (default or specified explicitly), say ‘x,’ which indicates that the physical logic value is initially unknown. When simulation begins the

simulator expands the event-list to include entries for value changes of the circuit's input signals (e.g.,  $A$ ,  $B$ ) at appropriate times. It then considers the next event-time and updates the values of signals that are in the corresponding signal-change list. Then it updates the event-time list to include new entries for signals whose values were affected by the changes that were just effected (e.g.,  $sig\_ch(10)$  is augmented by the event  $C = 0$ ). As simulation time advances, data structures are removed from a signal-change list as the associated variables are evaluated and possibly assigned their values. When  $sig\_ch(t)$  becomes empty, the engine proceeds to the next event-time and repeats the process. When the event-time list is empty the event activity is idle until the simulation is terminated.

## HDL Example 4.10

[Figure 4.40](#) shows the output waveforms that are produced by a and-invert circuit having zero propagation delays when its input waveforms are as shown (a shaded area denotes the 'x' value of a signal). The "event-time" list and its associated data structures show which signals have an event. It is convenient to display this relationship on a simulator time axis as depicted in the figure. At a given event-time, the signal-change list has been *ordered* to illustrate the causal relationships between the scheduled changes. For example, at time  $t_{sim} = 20$  the change of signal  $B$  causes the change in signal  $C$ , which causes the change in signal  $D$ . The simulator suspends  $t_{sim}$  while it updates memory to assign value to  $B$ , detects the need to schedule  $C$ , schedules  $C$ , and changes  $C$ . When  $C$  changes, the simulator notes that  $D$  must change, schedules the change in  $D$ , and then changes  $D$ . All of these actions occur at the same instant of *simulator time*,  $t_{sim} = 20$ , but they occur sequentially w.r.t. to a single thread of activity on the host processor. When the activity at  $t_{sim} = 20$  ceases, the simulator advances to the next time at which there is a nonempty event list, and then digests those events. This continues until there are no more event lists to digest.



**FIGURE 4.40**

Representation of event-driven simulation (with zero delay)

[Description](#)

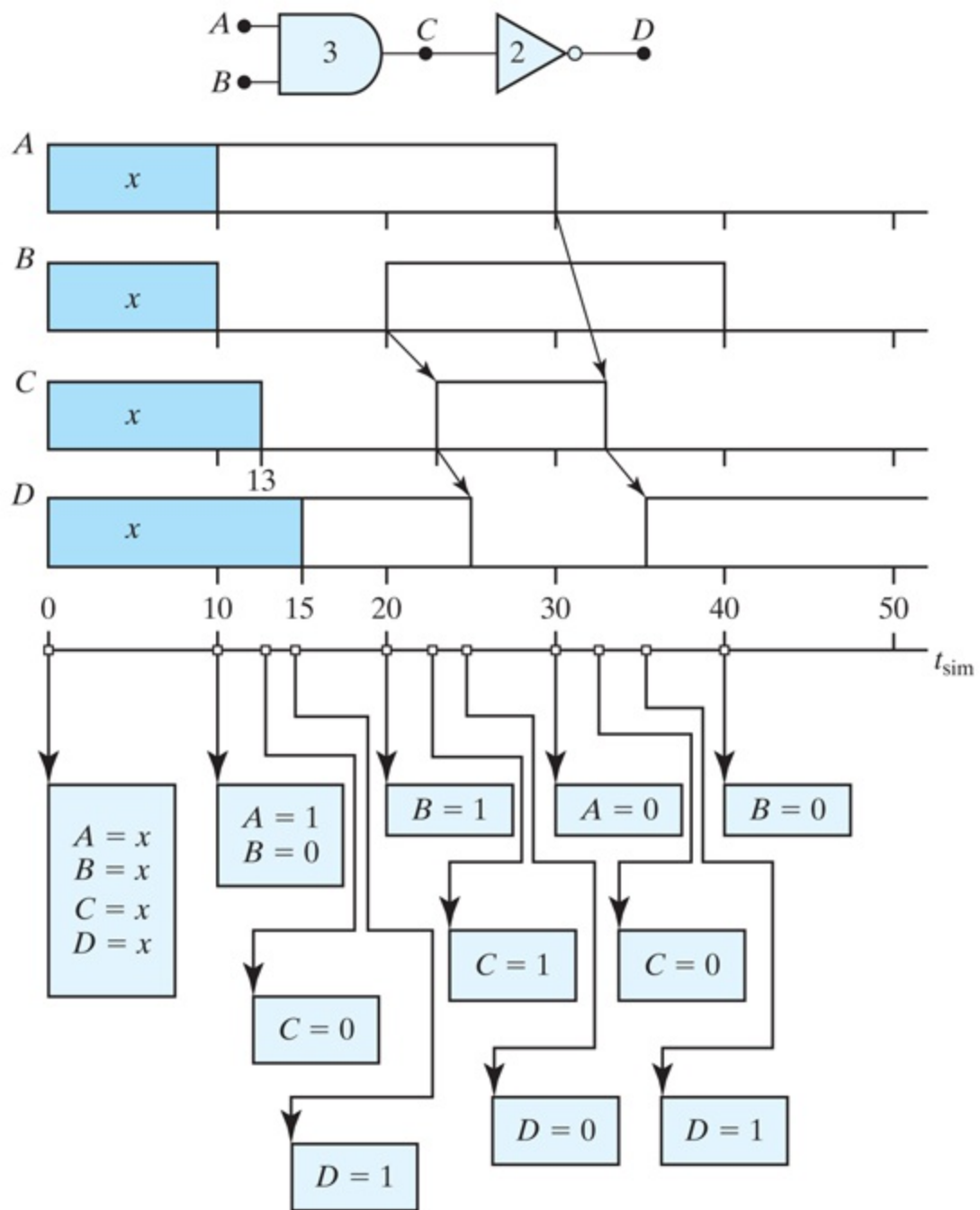
## Effect of Propagation Delay

A logic simulator must manage the scheduling of events for all of the signals in the circuit that is being simulated. A realistic simulation takes into account the actual propagation delays of the physical circuit elements. Each logic device may have a propagation delay associated with its behavior. When propagation delays are included in the models, signal changes do not propagate instantaneously through the circuit. The simulator uses these delay times to schedule the placement of events in the event lists.

## HDL Example 4.12 (Propagation Delay)

The logic gates in the circuit in [Figure 4.41](#) have the indicated propagation delays between the time when their input signals change and when their output is affected by the change. The logic waveforms and event lists [29](#) are depicted below the waveforms. Notice that changes to *C* occur three time units after changes to *A* and *B*, and signal *D* changes two time units after *C*. Thus, propagation delays affect the location of event lists on the simulator's time axis.

[29](#) Event lists are typically implemented as linked list data structures in the simulator engine.



**FIGURE 4.41**

Representation of event-driven simulation (with propagation delay)

### Description

An example of a circuit with gate delays was presented in [Section 3.9](#) in

[HDL Example 3.3](#). We next show an HDL example that produces the truth table of a combinational circuit. 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 has three inputs, two outputs, and nine gates. The model follows the interconnections between the gates according to the schematic diagram of [Fig. 4.2](#).

## HDL Example 4.13 (Logic Simulation)

### Verilog

The stimulus for the circuit is listed in the second module. The inputs are specified with a three-bit **reg** vector *D*. *D*[2] is equivalent to input *A*, *D*[1] to input *B*, and *D*[0] to input *C*. The outputs of the circuit *F* 1 and *F* 2 are declared as type **wire**. The complement of *F*2 is named *F2\_b* to illustrate a common practice for designating the complement of a signal (instead of appending *\_not*). The procedure follows the steps represented by [Fig. 4.37](#). 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.

```
// 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 UUT (D[2], D[1], D[0], F1, F2);    // Inst
    initial

        begin    // Apply stimulus
            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

```

## VHDL

Logic simulation of the full-adder circuit in [Fig. 4.2](#) first declares the components that will compose the circuit:

```

entity or2_gate is
    port (w: out Std_Logic; x, y: in Std_Logic);
end or2_gate;

architecture Dataflow of or2_gate is
begin
    w <= x or y;
end Dataflow;

entity or3_gate is
    port (w: out Std_Logic; x, y, z: in Std_Logic);
end or3_gate;

```

```

architecture Dataflow of or3_gate is
begin
    w <= x or y or z;
end Dataflow;

```

```

entity and2_gate is
    port (w: out Std_Logic; x, y: in Std_Logic);
end and2_gate;

```

```

architecture Dataflow of and2_gate is
begin
    w <= x and y;
end Dataflow;

```

```

entity and3_gate is
    port (w: out Std_Logic; x, y, z: in Std_Logic);
end and 3_gate;

```

```

architecture Dataflow of and3_gate is
begin
    w <= x and y and z;
end Dataflow;

```

```

entity not_gate is
    port (x: in Std_Logic; y: out Std_Logic);
end not_gate;

```

```

architecture Dataflow of not_gate is
begin
    y <= not x;
end Dataflow;

```

```

entity Circuit_of_Fig_4_2 is
    port (A, B, C: in Std_Logic; F1, F2: out Std_Logic);
end Circuit_of_ Fig. 4.2

```

The components are instantiated and connected (by name) to form the circuit:

```

architecture Structural of Circuit_of Fig_4_2 is
    signal: T1, T2, T3, F2_b, E1, E2, E3: Std_Logic;
    component or2_gate      port (w: out Std_Logic; x, y: in Std_Lo
    component or3_gate      port (w: out Std_Logic; x, y, z: in Std

```



```

component and2_gate      port (w: out Std_Logic; x, y: in Std_Lo
component and3_gate      port (w: out Std_Logic; x, y, z: in Std
component not_gate       port (x: in Std_Logic; y: out Std_Logic
begin
G1: or3_gate      port map (w => T1, x => A, y => B, z => C);
G2: and3_gate     port map (w => T2, x => A, y => B, z => C);
G3: and2_gate     port map (w => E1, x => A, y => B);
G4: and2_gate     port map (w => E2, x => A, y => C);
G5: and2_gate     port map (w => E3, x => B, y => C);
G6: or3_gate      port map (w => F2, x => E1, y => E2, z => E3)'
G7: not_gate      port map (x => F2, y => F2_b);
G8: and2_gate     port map (w => T3, x => T1, y => F2_b);
G9: or2_gate      port map (w => F1, x => T2, y => T3);
end Structural;

entity t_ Circuit_of_Fig_4_2 is
  port ();
end t_ Circuit_of_Fig_4_2;

```

Finally, *Test\_Bench*, the architecture of *t\_Circuit\_of\_Fig\_4\_2*, is declared. Within it, *Circuit\_of\_Fig\_4\_2* is declared as a component and instantiated. The signals of its port are connected by name to the stimulus and outputs that were declared locally within *Test\_Bench*.

```

architecture Test_Bench of t_Circuit_of_Fig_4_2 is
  signal t_A, t_B, t_C: Std_Logic;
  signal t_F1, t_F2: Std_Logic;

  integer k range 0 to 7: 0;
component Circuit_of_Fig_4_2 port (A, B, C: in Std_Logic; F1, F
-- UUT is a component

begin

-- Instantiate (by name) the UUT
UUT: Circuit_of_Fig_4_2 port map (F1 => t_F1, F2 => t_F2, A =>

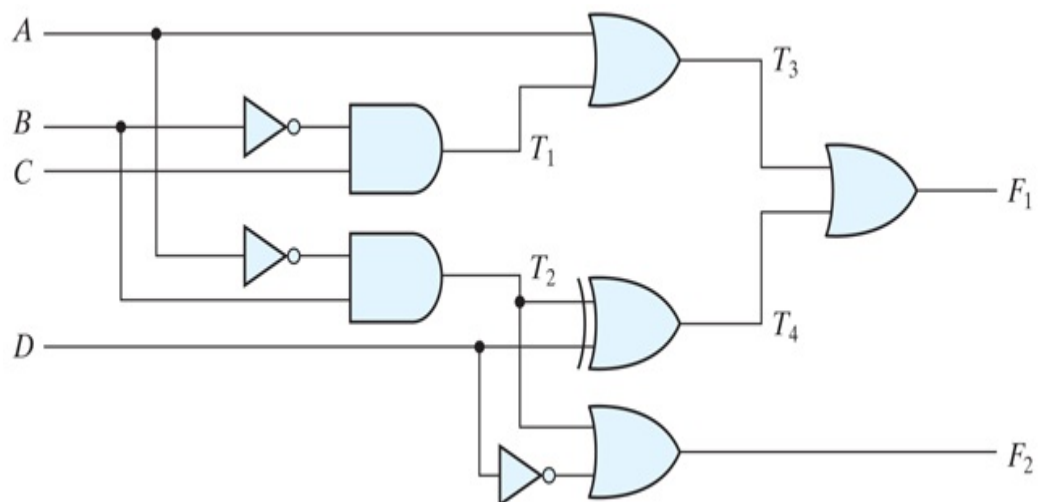
-- Apply stimulus signals
t_A & t_B & t_C <= '000';
while k <= 7 loop
  t_A & t_B & t_C <= t_A & t_B & t_C + '001';
  k := k + 1;
end loop;
end Test_Bench;

```

# 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.) Unless SystemVerilog is explicitly named, the HDL compiler for solving a problem may be Verilog, SystemVerilog, or VHDL. Note: For each problem that requires writing and verifying an HDL model, a basic test plan should be written to identify which functional features are to be tested during the simulation and how they will be tested. For example, a reset on-the-fly could be tested by asserting the reset signal while the simulated machine is in a state other than the reset state. The test plan is to guide development of a testbench that will implement the plan. Simulate the model, using the testbench, and verify that the behavior is correct.

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

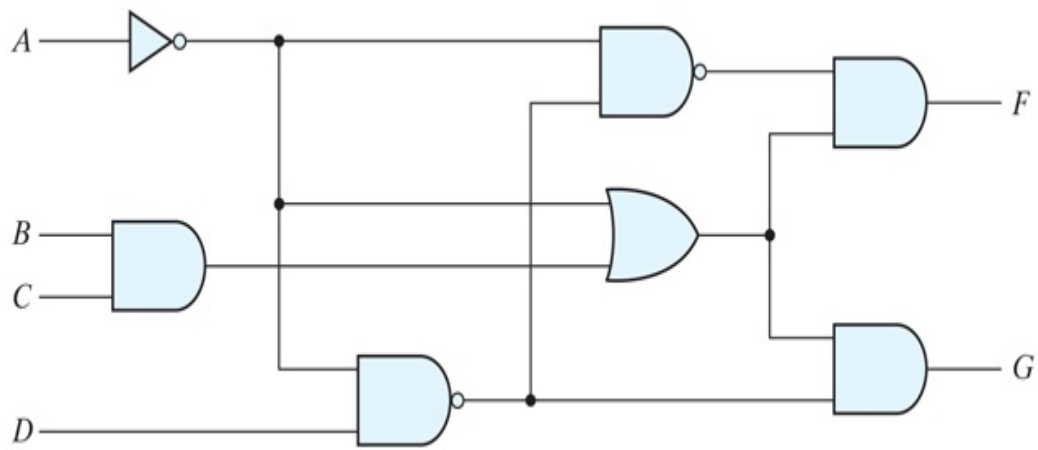


## FIGURE P4.1

### [Description](#)

1. (a)\* Derive the Boolean expressions for T 1 through T 4 .  
Evaluate the outputs F 1 and F 2 as a function of the four inputs.

2. (b) List the truth table with 16 binary combinations of the four input variables. Then list the binary values for T 1 through T 4 and outputs F 1 and F 2 in the table.
  3. (c) Plot the output Boolean functions obtained in part (b) on maps and show that the simplified Boolean expressions are equivalent to the ones obtained in part (a).
2. 4.2\* Obtain the simplified Boolean expressions for output  $F$  and  $G$  in terms of the input variables in the circuit of [Fig. P4.2](#).



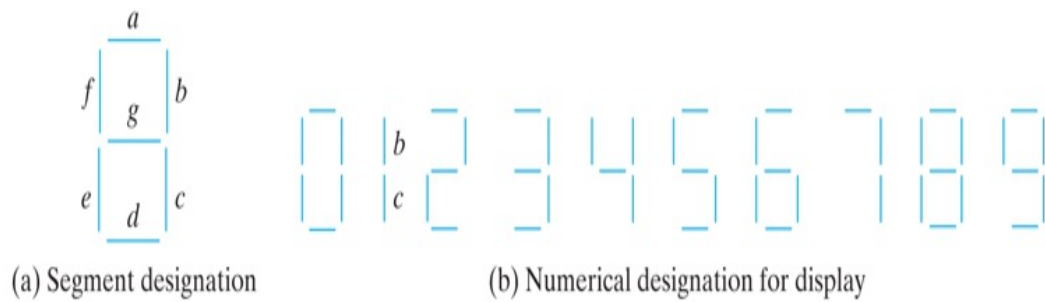
**FIGURE P4.2**

[Description](#)

3. 4.3 For the circuit shown in [Fig. 4.26](#) ([Section 4.11](#)),
  1. (a) Write the Boolean functions for the four outputs in terms of the input variables.
  2. (b)\* If the circuit is described in a truth table, how many rows and columns would there be in the table?
4. 4.4 Design a combinational circuit with three inputs and one output.
  1. (a)\* The output is 1 when the binary value of the inputs is less than 3. The output is 0 otherwise.
  2. (b) The output is 1 when the binary value of the inputs is an even

number.

5. 4.5 Design a combinational circuit with three inputs  $x$ ,  $y$ , and  $z$  and three outputs  $A$ ,  $B$ , and  $C$ . When the binary input is 0, 1, 2, or 3, the binary output is one greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is two less than the input.
6. 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.
  1. (a)\* Design a three-input majority circuit by finding the circuit's truth table, Boolean equation, and a logic diagram.
  2. (b) Write and verify a HDL gate-level model of the circuit.
7. 4.7 Design a combinational circuit that converts a four-bit Gray code ([Table 1.6](#)) to a four-bit binary number.
  1. (a)\* Implement the circuit with exclusive-OR gates.
  2. (b) Using a case statement, write and verify a HDL model of the circuit.
8. 4.8 Design a code converter that converts a decimal digit from
  1. (a)\* The 8, 4,  $-2$ ,  $-1$  code to BCD (see [Table 1.5](#)). (HDL—see [Problem 4.50](#).)
  2. (b) The 8, 4,  $-2$ ,  $-1$  code to Gray code.
9. 4.9 A BCD-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

### Description

10. 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?
11. 4.11 Using four half adders (HDL—see [Problem 4.52](#)),
  1. (a) Design a full-subtractor circuit incrementer. (A circuit that adds one to a four-bit binary number.)
  2. (b) Design a four-bit combinational decrementer. (A circuit that subtracts 1 from a four-bit binary number.)
12. 4.12
  1. (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  $D$  and the borrow in  $B\ out$ .
  2. (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.
13. 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](#).)

14. 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](#)?
15. 4.15 Derive the two-level Boolean expression for the output carry  $C_4$  shown in the lookahead carry generator of [Fig. 4.12](#).
16. 4.16 Define the carry propagate and carry generate for a lookahead carry generator as

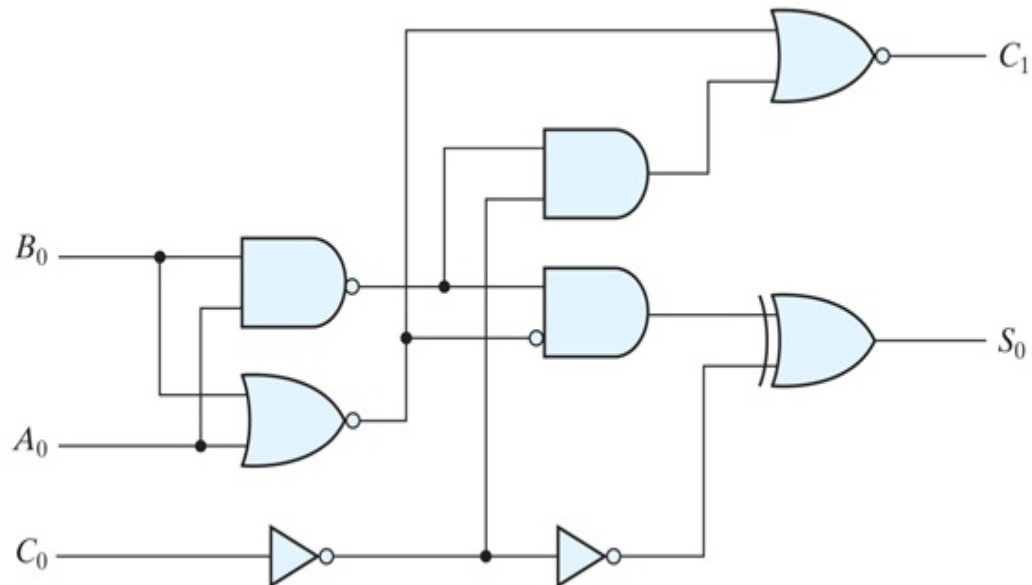
$$P_i = A_i + B_i \quad G_i = A_i B_i$$

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

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

The logic diagram of the first stage of a four-bit parallel adder implemented in IC type 74283 is shown in [Fig. P4.16](#). Identify the P

$i'$  and  $G_i'$  terminals and show that the circuit implements a full-adder circuit.



**FIGURE P4.16**

### Description

17. 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 gates (see [Section 3.8](#)). The circuit assumes that the input terminals have the complements of the  $G$ 's, the  $P$ 's, and of  $C_1$ . Derive the Boolean functions for the lookahead carries  $C_2$ ,  $C_3$ , and  $C_4$  in this IC. (*Hint*: Use the equation-substitution method to derive the carries in terms of  $C_i'$ .)

18. 4.18 Design a combinational circuit that generates the 9's complement of a
1. (a)\* BCD digit. (HDL—see [Problem 4.54\(a\)](#).)
  2. (b) Gray-code digit. (HDL—see [Problem 4.54\(b\)](#).)

19. 4.19 Construct a BCD adder D-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](#).)
20. 4.20 For a binary multiplier that multiplies two unsigned four-bit numbers,
  1. (a) Using AND gates and binary adders (see [Fig. 4.16](#)), design the circuit.
  2. (b) Write and verify a HDL dataflow model of the circuit.
21. 4.21 Design a combinational circuit that compares two 4-bit numbers to check if they are equal. The circuit output is equal to 1 if the two numbers are equal and 0 otherwise.
22. 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](#).)
23. 4.23 Draw the logic diagram of a 2-to-4-line decoder using (a) NOR gates only and (b) NAND gates only. Include an enable input. (HDL—see [Problems 4.36](#) and [4.45](#).)
24. 4.24 Design a BCD-to-decimal decoder using the unused combinations of the BCD code as don't-care conditions.
25. 4.25 Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and a 2-to-4-line decoder. Use block diagrams for the components. (HDL—see [Problem 4.62](#).)
26. 4.26 Construct a 4-to-16-line decoder with five 2-to-4-line decoders with enable. (HDL—see [Problem 4.63](#).)
27. 4.27 A combinational circuit is specified by the following three Boolean functions:

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

Implement the circuit with a decoder constructed with NAND gates (similar to [Fig. 4.19](#)) and NAND or AND gates connected to the



decoder outputs. Use a block diagram for the decoder. Minimize the number of inputs in the external gates.

28. 4.28 Using a decoder and external gates, design the combinational circuit defined by the following three Boolean functions:

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

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

29. 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](#).)

30. 4.30 Specify the truth table of an octal-to-binary priority encoder. Provide an output  $V$  to indicate that at least one of the inputs is present. The input with the highest subscript number has the highest priority. What will be the value of the four outputs if inputs  $D_2$  and  $D_6$  are 1 at the same time? (HDL—see [Problem 4.64](#).)

31. 4.31 Construct a  $16 \times 1$  multiplexer with two  $8 \times 1$  and one  $2 \times 1$  multiplexers. Use block diagrams. (HDL—see [Problem 4.65](#).)

32. 4.32 Implement the following Boolean function with a multiplexer (HDL—see [Problem 4.46](#)):

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

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

33. 4.33 Implement a full adder with two  $4 \times 1$  multiplexers.

34. 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:

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

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

Determine the Boolean function that the multiplexer implements.

35. 4.35 Implement the following Boolean function with a  $4 \times 1$  multiplexer and external gates.

1. (a)\*  $F_1(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$
2. (b)  $F_2(A, B, C, D) = \Sigma(1, 2, 5, 7, 8, 10, 11, 13, 15)$

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

36. 4.36 Write the HDL gate-level description of the priority encoder circuit shown in [Fig. 4.23](#). (HDL—see [Problem 4.45](#).)
37. 4.37 Write the HDL gate-level hierarchical description of a four-bit adder–subtractor for unsigned binary numbers. The circuit is similar to [Fig. 4.13](#) but without output  $V$ . You can instantiate the four-bit full adder described in [HDL Example 4.2](#). (HDL—see [Problems 4.13](#) and [4.40](#).)
38. 4.38 Write the HDL dataflow description of a quadruple 2-to-1-line multiplexer with enable (see [Fig. 4.26](#)).
39. 4.39\* Write an HDL behavioral description of a four-bit comparator with a six-bit output  $Y[5 : 0]$ . Bit index 5 of  $Y$  is for “equals,” bit 4 for “not equal to,” bit 3 for “greater than,” bit 2 for “less than,” bit 1 for “greater than or equal,” and bit 0 for “less than or equal to.”
40. 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](#).)
41. 4.41 Repeat [Problem 4.40](#) using a Verilog **always** statement or a VHDL **process**.

42. 4.42

1. (a) Write an HDL gate-level description of the BCD-to-excess-3 converter circuit shown in [Fig. 4.4](#) (see [Problem 4.22](#)).
2. (c) Write a dataflow description of the BCD-to-excess-3 converter using the Boolean expressions listed in [Fig. 4.3](#).
3. (d)\* Write an HDL behavioral description of a BCD-to-excess-3 converter.
4. (e) Write a testbench to simulate and test the BCD-to-excess-3 converter circuit in order to verify the truth table. Check all three circuits.

43. 4.43 Explain the function of the circuit specified by the following HDL description:

**Verilog**

```
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
```

**VHDL**

```
architecture
begin
```

```
Q <= A when S = '1' and E = '1'; else '0' when S = '0' and
```

44. 4.44 Using a case statement, write an HDL behavioral description of an eight-bit arithmetic-logic unit (ALU). The circuit has a three-bit select bus (Sel), 16-bit input datapaths (A and B), an eight-bit output datapath (y), and performs the arithmetic and logic operations listed below.

Sel	Description
-----	-------------

000 Reset  $y$  to all 0's

001 Bitwise AND

010 Bitwise OR

011 Bitwise exclusive-OR

100 Bitwise complement

101 Subtract

110 Add (Assume  $A$  and  $B$  are unsigned)

111 Set  $y$  to all 1's

45. 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](#)).
46. 4.46 Write an HDL dataflow description of the logic circuit described by the Boolean function in [Problem 4.32](#).
47. 4.47 Write an HDL dataflow description of the logic circuit described by the Boolean function in [Problem 4.35](#).
48. 4.48 Modify the eight-bit ALU specified in [Problem 4.44](#) and develop an HDL description so that it has three-state output controlled by an enable input,  $En$ . Write a testbench and simulate the circuit.
49. 4.49 For the circuit shown in [Fig. P4.1](#),

1. (a) Write and verify a gate-level HDL model of the circuit.
  2. (b) Compare your results with those obtained for [Problem 4.1](#).
50. 4.50 Using a case statement, develop and simulate an HDL behavioral model of
1. (a)\* The 8, 4, - 2, - 1 to BCD code converter described in [Problem 4.8\(a\)](#).
  2. (b) The 8, 4, - 2, - 1 to Gray code converter described in [Problem 4.8\(b\)](#).
51. 4.51 Develop and simulate an HDL behavioral model of the ABCD-to-seven-segment decoder—described in [Problem 4.9](#).
52. 4.52 Using a Verilog continuous assignment or VHDL signal assignment, develop and simulate an HDL dataflow model of
1. (a) The four-bit incrementer described in [Problem 4.11\(a\)](#).
  2. (b) The four-bit decrementer described in [Problem 4.11\(b\)](#).
53. 4.53 Develop and simulate an HDL structural model of the decimal adder shown in [Fig. 4.14](#).
54. 4.54 Develop and simulate a HDL behavioral model of a circuit that generates the 9's complement of
1. (a) a BCD digit (see [Problem 4.18\(a\)](#)).
  2. (b) a Gray-code digit (see [Problem 4.18\(b\)](#)).
55. 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.
56. 4.56\* Write a Verilog continuous assignment statement or a VHDL signal assignment statement that compares two 4-bit numbers to check if their bit patterns match. The variable to which the assignment is made is equal to 1 if the numbers match and 0

otherwise.

57. 4.57\* Develop and verify an HDL behavioral model of the four-bit priority encoder described in [Problem 4.29](#).
58. 4.58 Write an HDL model of a circuit whose 32-bit output is formed by shifting its 32-bit input three positions to the right and filling the vacant positions with the bit that was in the MSB before the shift occurred (shift arithmetic right). Write an HDL model of a circuit whose 32-bit output is formed by shifting its 32-bit input three positions to the left and filling the vacant positions with 0 (shift logical left).
59. 4.59 Write an HDL model of a BCD-to-decimal decoder using the unused combinations of the BCD code as don't-care conditions (see [Problem 4.24](#)).
60. 4.60 Using the port syntax of the IEEE 1364-2001 standard, write and verify a gate-level model of the four-bit even parity checker shown in [Fig. 3.34](#).
61. 4.61 Using Verilog continuous assignment statements or a VHDL signal assignment statement, write and verify a gate-level model of the four-bit even parity checker shown in [Fig. 3.34](#).
62. 4.62 Write and verify a gate-level hierarchical HDL model of the circuit described in [Problem 4.25](#).
63. 4.63 Write and verify a gate-level hierarchical HDL model of the circuit described in [Problem 4.26](#).
64. 4.64 Write and verify a HDL model of the octal-to-binary circuit described in [Problem 4.30](#).
65. 4.65 Write a hierarchical gate-level HDL model of the multiplexer described in [Problem 4.31](#).

# REFERENCES

- 1.Dietmeyer, D. L. 1988. *Logic Design of Digital Systems*, 3rd ed., Boston: Allyn Bacon.
- 2.Gajski, D. D. 1997. *Principles of Digital Design*. Upper Saddle River, NJ: Prentice Hall.
- 3.Hayes, J. P. 1993. *Introduction to Digital Logic Design*. Reading, MA: Addison-Wesley.
- 4.Katz, R. H. 2005. *Contemporary Logic Design*. Upper Saddle River, NJ: Prentice Hall.
- 5.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.
- 6.Bhasker, J. 1997. *A Verilog HDL Primer*. Allentown, PA: Star Galaxy Press.
- 7.Bhasker, J. 1998. *Verilog HDL Synthesis*. Allentown, PA: Star Galaxy Press.
- 8.Ciletti, M. D. 1999. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL*. Upper Saddle River, NJ: Prentice Hall.
- 9.Mano, M. M. and C. R. Kime. 2007. *Logic and Computer Design Fundamentals*, 4th ed., Upper Saddle River, NJ: Prentice Hall.
- 10.Roth, C. H. and L. L., Kinney. 2014. *Fundamentals of Logic Design*, 7th ed., St. Paul, MN: Cengage Learning.
- 11.Wakerly, J. F. 2005. *Digital Design: Principles and Practices*, 4th ed., Upper Saddle River, NJ: Prentice Hall.
- 12.Palnitkar, S. 1996. *Verilog HDL: A Guide to Digital Design and Synthesis*. Mountain View, CA: SunSoft Press (a Prentice Hall title).

- 13. Thomas, D. E. and P. R. Moorby. 2002. *The Verilog Hardware Description Language*, 5th ed., Boston: Kluwer Academic Publishers.



# WEB SEARCH TOPICS

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