

---

# Chapter 3

## Gate-Level Minimization

---

### 3.1 INTRODUCTION

---

*Gate-level minimization* is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit. This task is well understood, but is difficult to execute by manual methods when the logic has more than a few inputs. Fortunately, computer-based logic synthesis tools can minimize a large set of Boolean equations efficiently and quickly. Nevertheless, it is important that a designer understand the underlying mathematical description and solution of the problem. This chapter serves as a foundation for your understanding of that important topic and will enable you to execute a manual design of simple circuits, preparing you for skilled use of modern design tools. The chapter will also introduce a hardware description language that is used by modern design tools.

### 3.2 THE MAP METHOD

---

The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, when it is expressed algebraically it can appear in many different, but equivalent, forms. Boolean expressions may be simplified by algebraic means as discussed in Section 2.4. However, this procedure of minimization is awkward because it lacks specific rules to predict each succeeding step in the manipulative process. The map method presented here provides a simple, straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method is also known as the *Karnaugh map* or *K-map*.

A K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function. In fact, the map presents a visual diagram of all possible ways a function may be expressed in standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which the simplest can be selected.

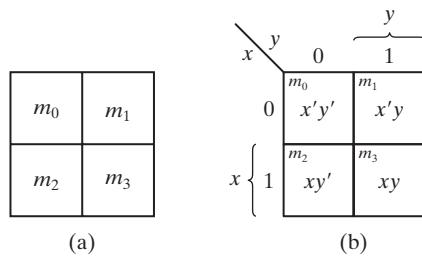
The simplified expressions produced by the map are always in one of the two standard forms: sum of products or product of sums. It will be assumed that the simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate. We will see subsequently that the simplest expression is not unique: It is sometimes possible to find two or more expressions that satisfy the minimization criteria. In that case, either solution is satisfactory.

### Two-Variable K-Map

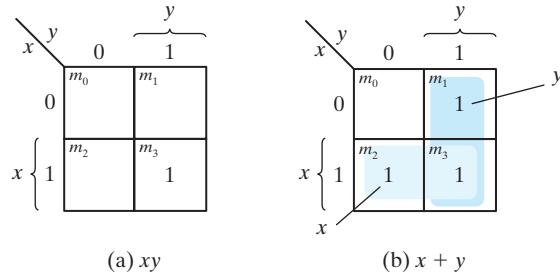
The two-variable map is shown in Fig. 3.1(a). There are four minterms for two variables; hence, the map consists of four squares, one for each minterm. The map is redrawn in (b) to show the relationship between the squares and the two variables  $x$  and  $y$ . The 0 and 1 marked in each row and column designate the values of variables. Variable  $x$  appears primed in row 0 and unprimed in row 1. Similarly,  $y$  appears primed in column 0 and unprimed in column 1.

If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables. As an example, the function  $xy$  is shown in Fig. 3.2(a). Since  $xy$  is equal to  $m_3$ , a 1 is placed inside the square that belongs to  $m_3$ . Similarly, the function  $x + y$  is represented in the map of Fig. 3.2(b) by three squares marked with 1's. These squares are found from the minterms of the function:

$$m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$$



**FIGURE 3.1**  
Two-variable K-map



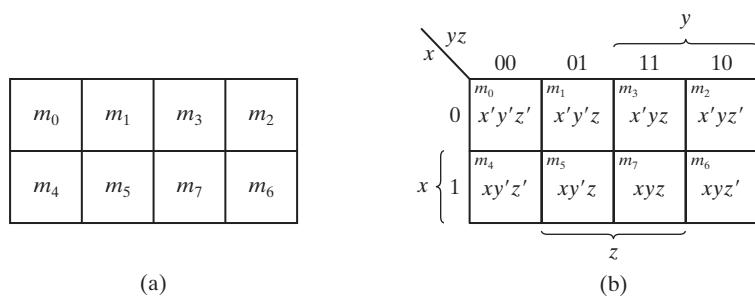
## **FIGURE 3.2**

### Representation of functions in the map

The three squares could also have been determined from the intersection of variable  $x$  in the second row and variable  $y$  in the second column, which encloses the area belonging to  $x$  or  $y$ . In each example, the minterms at which the function is asserted are marked with a 1.

## Three-Variable K-Map

A three-variable K-map is shown in Fig. 3.3. There are eight minterms for three binary variables; therefore, the map consists of eight squares. Note that the minterms are arranged, not in a binary sequence, but in a sequence similar to the Gray code (Table 1.6). The characteristic of this sequence is that **only one bit changes in value from one adjacent column to the next**. The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables. For example, the square assigned to  $m_5$  corresponds to row 1 and column 01. When these two numbers are concatenated, they give the binary number 101, whose decimal equivalent is 5. Each cell of the map corresponds to a unique minterm, so another way of looking at square  $m_5 = xy'z$  is to consider it to be in the row marked  $x$  and the column belonging to  $y'z$  (column 01). Note that there are four squares in which each variable is equal to 1 and four in which each is equal to 0. The variable appears unprimed in the former four



## **FIGURE 3.3**

### Three-variable K-map

squares and primed in the latter. For convenience, we write the variable with its letter symbol under the four squares in which it is unprimed.

To understand the usefulness of the map in simplifying Boolean functions, we must recognize the basic property possessed by adjacent squares: **Any two adjacent squares in the map differ by only one variable**, which is primed in one square and unprimed in the other. For example,  $m_5$  and  $m_7$  lie in two adjacent squares. Variable  $y$  is primed in  $m_5$  and unprimed in  $m_7$ , whereas the other two variables are the same in both squares. From the postulates of Boolean algebra, it follows that the sum of two minterms in adjacent squares can be simplified to a single product term consisting of only two literals. To clarify this concept, consider the sum of two adjacent squares such as  $m_5$  and  $m_7$ :

$$m_5 + m_7 = xy'z + xyz = xz(y' + y) = xz$$

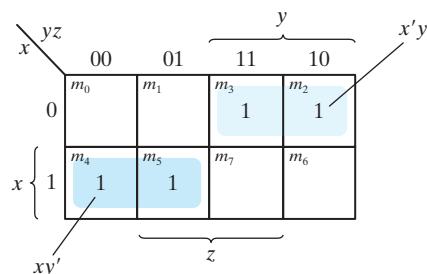
Here, the two squares differ by the variable  $y$ , which can be removed when the sum of the two minterms is formed. Thus, any two minterms in adjacent squares (vertically or horizontally, but not diagonally, adjacent) that are ORed together will cause a removal of the dissimilar variable. The next four examples explain the procedure for minimizing a Boolean function with a K-map.

### EXAMPLE 3.1

Simplify the Boolean function

$$F(x, y, z) = \Sigma(2, 3, 4, 5)$$

First, a 1 is marked in each minterm square that represents the function. This is shown in Fig. 3.4, in which the squares for minterms 010, 011, 100, and 101 are marked with 1's. The next step is to find possible adjacent squares. These are indicated in the map by two shaded rectangles, each enclosing two 1's. The upper right rectangle represents the area enclosed by  $x'y$ . This area is determined by observing that the two-square area is in row 0, corresponding to  $x'$ , and the last two columns, corresponding to  $y$ . Similarly, the lower left rectangle represents the product term  $xy'$ . (The second row represents  $x$  and the two left columns represent  $y'$ .) The sum of four minterms can be replaced by a sum of



**FIGURE 3.4**

Map for Example 3.1,  $F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$

only two product terms. The logical sum of these two product terms gives the simplified expression

$$F = x'y + xy'$$

In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other. In Fig. 3.3(b),  $m_0$  is adjacent to  $m_2$  and  $m_4$  is adjacent to  $m_6$  because their minterms differ by one variable. This difference can be readily verified algebraically:

$$m_0 + m_2 = x'y'z' + x'yz' = x'z'(y' + y) = x'z'$$

Consequently, we must modify the definition of adjacent squares to include this and other similar cases. We do so by considering the map as being drawn on a surface in which the right and left edges touch each other to form adjacent squares.

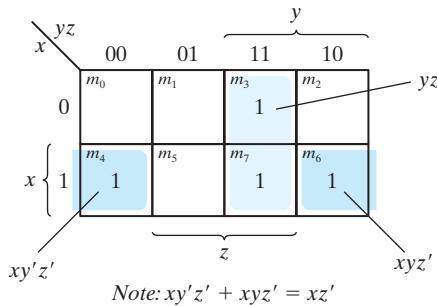
## EXAMPLE 3.2

## Simplify the Boolean function

$$F(x, y, z) = \Sigma(3, 4, 6, 7)$$

The map for this function is shown in Fig. 3.5. There are four squares marked with 1's, one for each minterm of the function. Two adjacent squares are combined in the third column to give a two-literal term  $yz$ . The remaining two squares with 1's are also adjacent by the new definition. These two squares, when combined, give the two-literal term  $xz'$ . The simplified function then becomes

$$F = yz + xz'$$



## FIGURE 3.5

**Map for Example 3.2,**  $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

Consider now any combination of four adjacent squares in the three-variable map. Any such combination represents the logical sum of four minterms and results in an expression with only one literal. As an example, the logical sum of the four adjacent minterms 0, 2, 4, and 6 reduces to the single literal term  $z'$ :

$$\begin{aligned}m_0 + m_2 + m_4 + m_6 &= x'y'z' + x'yz' + xy'z' + xyz' \\&= x'z'(y' + y) + xz'(y' + y) \\&= x'z' + xz' = z'(x' + x) = z'\end{aligned}$$

The number of adjacent squares that may be combined must always represent a number that is a power of two, such as 1, 2, 4, and 8. As more adjacent squares are combined, we obtain a product term with fewer literals.

One square represents one minterm, giving a term with three literals.

Two adjacent squares represent a term with two literals.

Four adjacent squares represent a term with one literal.

Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.

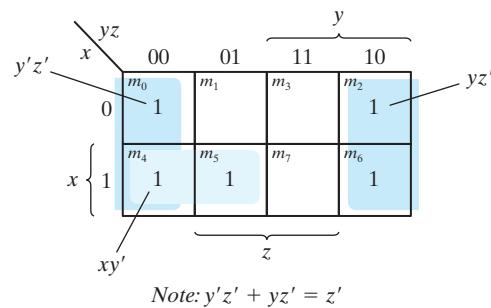
### EXAMPLE 3.3

Simplify the Boolean function

$$F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$$

The map for  $F$  is shown in Fig. 3.6. First, we combine the four adjacent squares in the first and last columns to give the single literal term  $z'$ . The remaining single square, representing minterm 5, is combined with an adjacent square that has already been used once. This is not only permissible, but rather desirable, because the two adjacent squares give the two-literal term  $xy'$  and the single square represents the three-literal minterm  $xy'z$ . The simplified function is

$$F = z' + xy'$$



**FIGURE 3.6**

Map for Example 3.3,  $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$

If a function is not expressed in sum-of-minterms form, it is possible to use the map to obtain the minterms of the function and then simplify the function to an expression with a minimum number of terms. It is necessary, however, to make sure that the algebraic expression is in sum-of-products form. Each product term can be plotted in the map in one, two, or more squares. The minterms of the function are then read directly from the map.

### EXAMPLE 3.4

For the Boolean function

$$F = A'C + A'B + AB'C + BC$$

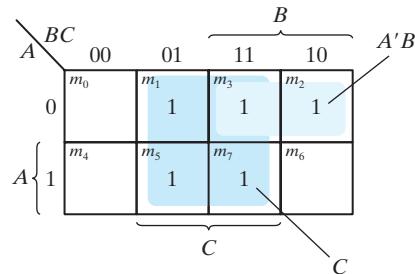
- (a) Express this function as a sum of minterms.
- (b) Find the minimal sum-of-products expression.

Note that  $F$  is a sum of products. Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term,  $A'C$ , are found in Fig. 3.7 from the coincidence of  $A'$  (first row) and  $C$  (two middle columns) to give squares 001 and 011. Note that, in marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term. This happens with the second term,  $A'B$ , which has 1's in squares 011 and 010. Square 011 is common with the first term,  $A'C$ , though, so only one 1 is marked in it. Continuing in this fashion, we determine that the term  $AB'C$  belongs in square 101, corresponding to minterm 5, and the term  $BC$  has two 1's in squares 011 and 111. The function has a total of five minterms, as indicated by the five 1's in the map of Fig. 3.7. The minterms are read directly from the map to be 1, 2, 3, 5, and 7. The function can be expressed in sum-of-minterms form as

$$F(A, B, C) = \Sigma(1, 2, 3, 5, 7)$$

The sum-of-products expression, as originally given, has too many terms. It can be simplified, as shown in the map, to an expression with only two terms:

$$F = C + A'B$$



**FIGURE 3.7**

Map of Example 3.4,  $A'C + A'B + AB'C + BC = C + A'B$

that cover minterms  $m_3$ ,  $m_9$ , and  $m_{11}$ . There are four possible ways that the function can be expressed with four product terms of two literals each:

$$\begin{aligned} F &= BD + B'D' + CD + AD \\ &= BD + B'D' + CD + AB' \\ &= BD + B'D' + B'C + AD \\ &= BD + B'D' + B'C + AB' \end{aligned}$$

The previous example has demonstrated that the identification of the prime implicants in the map helps in determining the alternatives that are available for obtaining a simplified expression.

The procedure for finding the simplified expression from the map requires that we first determine all the essential prime implicants. The simplified expression is obtained from the logical sum of all the essential prime implicants, plus other prime implicants that may be needed to cover any remaining minterms not covered by the essential prime implicants. Occasionally, there may be more than one way of combining squares, and each combination may produce an equally simplified expression.

### Five-Variable Map

Maps for more than four variables are not as simple to use as maps for four or fewer variables. A five-variable map needs 32 squares and a six-variable map needs 64 squares. When the number of variables becomes large, the number of squares becomes excessive and the geometry for combining adjacent squares becomes more involved.

Maps for more than four variables are difficult to use and will not be considered here.

## 3.4 PRODUCT-OF-SUMS SIMPLIFICATION

---

The minimized Boolean functions derived from the map in all previous examples were expressed in sum-of-products form. With a minor modification, the product-of-sums form can be obtained.

The procedure for obtaining a minimized function in product-of-sums form follows from the basic properties of Boolean functions. The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the standard sum-of-products form of a function denote the complement of the function. From this observation, we see that the complement of a function is represented in the map by the squares not marked by 1's. If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified sum-of-products expression of the complement of the function (i.e., of  $F'$ ). The complement of  $F'$  gives us back the function  $F$  in product-of-sums form (a consequence of DeMorgan's theorem). Because of the generalized DeMorgan's theorem, the function so obtained is automatically in product-of-sums form. The best way to show this is by example.

**EXAMPLE 3.7**

Simplify the following Boolean function into (a) sum-of-products form and (b) product-of-sums form:

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

The 1's marked in the map of Fig. 3.12 represent all the minterms of the function. The squares marked with 0's represent the minterms not included in  $F$  and therefore denote the complement of  $F$ . Combining the squares with 1's gives the simplified function in sum-of-products form:

$$(a) F = B'D' + B'C' + A'C'D$$

If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:

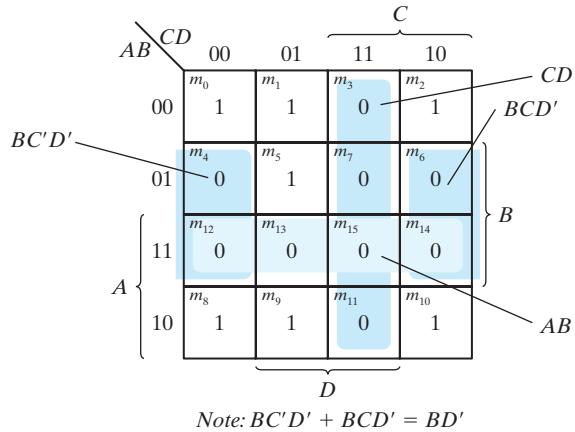
$$F' = AB + CD + BD'$$

Applying DeMorgan's theorem (by taking the dual and complementing each literal as described in Section 2.4), we obtain the simplified function in product-of-sums form:

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



The gate-level implementation of the simplified expressions obtained in Example 3.7 is shown in Fig. 3.13. The sum-of-products expression is implemented in (a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in (b) in its product-of-sums



**FIGURE 3.12**

Map for Example 3.7,  $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10) = B'D' + B'C' + A'C'D = (A' + B')(C' + D')(B' + D)$

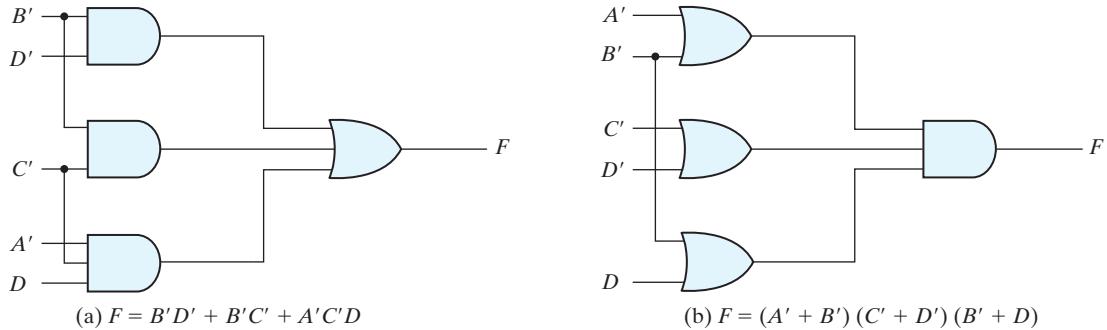


FIGURE 3.13

Gate implementations of the function of Example 3.7

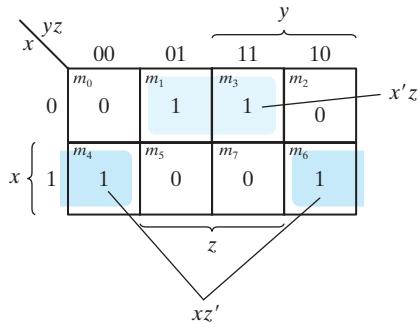
**Table 3.1**  
*Truth Table of Function F*

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

form with a group of OR gates, one for each OR term. The outputs of the OR gates are connected to the inputs of a single AND gate. In each case, it is assumed that the input variables are directly available in their complement, so inverters are not needed. The configuration pattern established in Fig. 3.13 is the general form by which any Boolean function is implemented when expressed in one of the standard forms. AND gates are connected to a single OR gate when in sum-of-products form; OR gates are connected to a single AND gate when in product-of-sums form. Either configuration forms two levels of gates. Thus, the implementation of a function in a standard form is said to be a two-level implementation. The two-level implementation may not be practical, depending on the number of inputs to the gates.

Example 3.7 showed the procedure for obtaining the product-of-sums simplification when the function is originally expressed in the sum-of-minterms canonical form. The procedure is also valid when the function is originally expressed in the product-of-maxterms canonical form. Consider, for example, the truth table that defines the function  $F$  in Table 3.1. In sum-of-minterms form, this function is expressed as

$$F(x, y, z) = \Sigma(1, 3, 4, 6)$$



**FIGURE 3.14**  
Map for the function of Table 3.1

In product-of-maxterms form, it is expressed as

$$F(x, y, z) = \Pi(0, 2, 5, 7)$$

In other words, the 1's of the function represent the minterms and the 0's represent the maxterms. The map for this function is shown in Fig. 3.14. One can start simplifying the function by first marking the 1's for each minterm that the function is a 1. The remaining squares are marked by 0's. If, instead, the product of maxterms is initially given, one can start marking 0's in those squares listed in the function; the remaining squares are then marked by 1's. Once the 1's and 0's are marked, the function can be simplified in either one of the standard forms. For the sum of products, we combine the 1's to obtain

$$F = x'z + xz'$$

For the product of sums, we combine the 0's to obtain the simplified complemented function

$$F' = xz + x'z'$$

which shows that the exclusive-OR function is the complement of the equivalence function (Section 2.6). Taking the complement of  $F'$ , we obtain the simplified function in product-of-sums form:

$$F = (x' + z')(x + z)$$

To enter a function expressed in product-of-sums form into the map, use the complement of the function to find the squares that are to be marked by 0's. For example, the function

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

can be entered into the map by first taking its complement, namely,

$$F' = ABC + B'D'$$

---

## Chapter 4

# Combinational Logic

---

### 4.1 INTRODUCTION

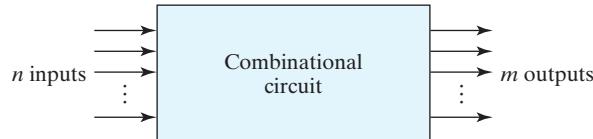
---

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs. A combinational circuit performs an operation that can be specified logically by a set of Boolean functions. In contrast, sequential circuits employ storage elements in addition to logic gates. Their outputs are a function of the inputs and the state of the storage elements. Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states. Sequential circuits are the building blocks of digital systems and are discussed in Chapters 5 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 internal combinational logic circuit and go to an external destination. Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0. (*Note:* Logic simulators show only 0's and 1's, not the actual analog signals.) In many applications, the source and



**FIGURE 4.1**  
Block diagram of combinational circuit

destination are storage registers. If the registers are included with the combinational gates, then the total circuit must be considered to be a sequential circuit.

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

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

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

### 4.3 ANALYSIS PROCEDURE

The analysis of a combinational circuit requires that we determine the function that the circuit implements. This task starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or, possibly, an explanation of the circuit operation.

If the logic diagram to be analyzed is accompanied by a function name or an explanation of what it is assumed to accomplish, then the analysis problem reduces to a verification of the stated function. The analysis can be performed manually by finding the Boolean functions or truth table or by using a computer simulation program.

The first step in the analysis is to make sure that the given circuit is combinational and not sequential. **The diagram of a combinational circuit has logic gates with no feedback paths or memory elements.** A feedback path is a connection from the output of one gate to the input of a second gate 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 from a logic diagram, we proceed as follows:

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

The analysis of the combinational circuit of Fig. 4.2 illustrates the proposed procedure. We note that the circuit has three binary inputs— $A$ ,  $B$ , and  $C$ —and two binary outputs— $F_1$  and  $F_2$ . The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function only of input variables are  $T_1$  and  $T_2$ . Output  $F_2$  can easily be derived from the input variables. The Boolean functions for these three outputs are

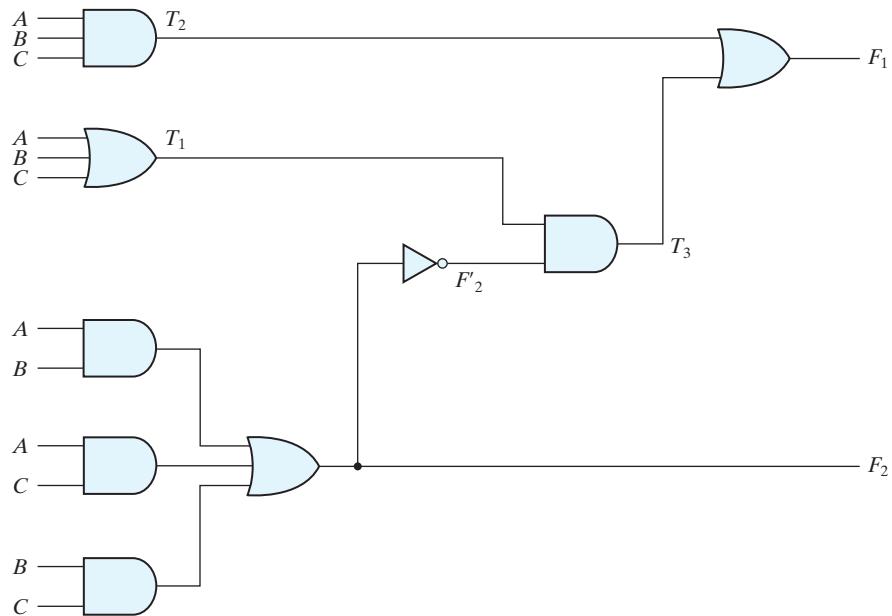
$$\begin{aligned}F_2 &= AB + AC + BC \\T_1 &= A + B + C \\T_2 &= ABC\end{aligned}$$

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

$$\begin{aligned}T_3 &= F'_2 T_1 \\F_1 &= T_3 + T_2\end{aligned}$$

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

$$\begin{aligned}F_1 &= T_3 + T_2 = F'_2 T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC \\&= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\&= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\&= A'BC' + A'B'C + AB'C' + ABC\end{aligned}$$



**FIGURE 4.2**  
Logic diagram for analysis example

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

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

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

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

A	B	C	F <sub>2</sub>	F' <sub>2</sub>	T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	F <sub>1</sub>
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

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.

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

## 4.4 DESIGN PROCEDURE

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

1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
2. Derive the truth table that defines the required relationship between inputs and outputs.

3. Obtain the simplified Boolean functions for each output as a function of the input variables.
4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

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

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

### Code Conversion Example

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

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

The bit combinations assigned to the BCD and excess-3 codes are listed in Table 1.5 (Section 1.7). Since each code uses four bits to represent a decimal digit, there must

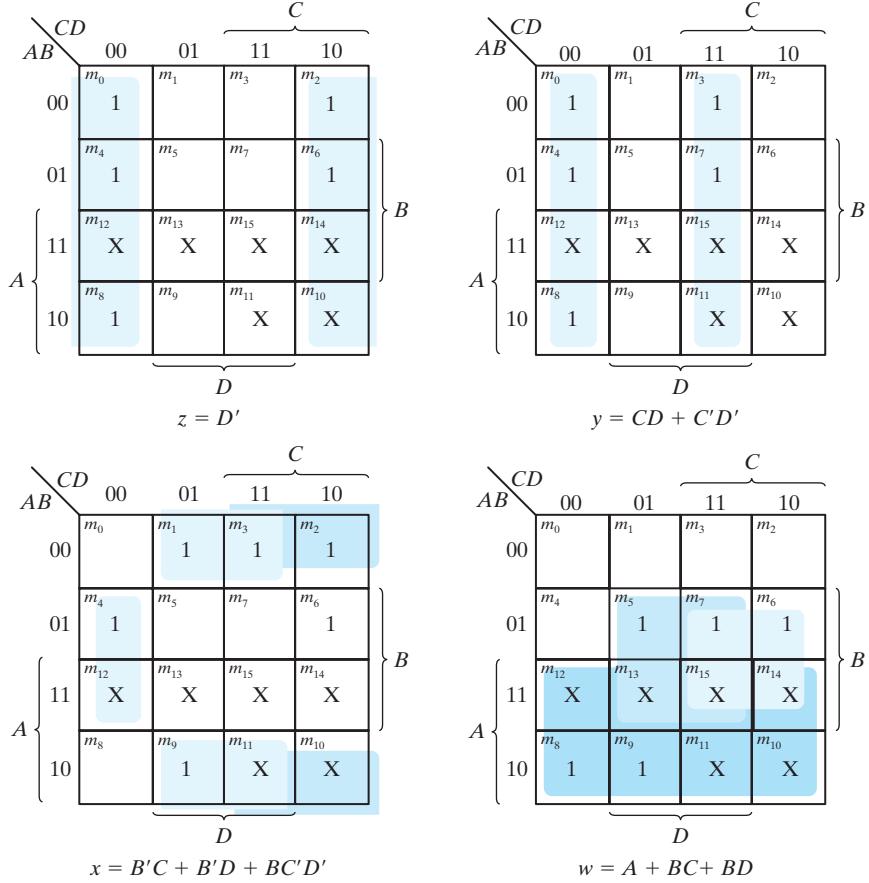
**Table 4.2**  
*Truth Table for Code Conversion Example*

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

be four input variables and four output variables. We designate the four input binary variables by the symbols  $A$ ,  $B$ ,  $C$ , and  $D$ , and the four output variables by  $w$ ,  $x$ ,  $y$ , and  $z$ . The truth table relating the input and output variables is shown in Table 4.2. The bit combinations for the inputs and their 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.

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.)

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



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

implemented with three or more levels of gates:

$$z = D'$$

$$y = CD + C'D' = CD + (C + D)'$$

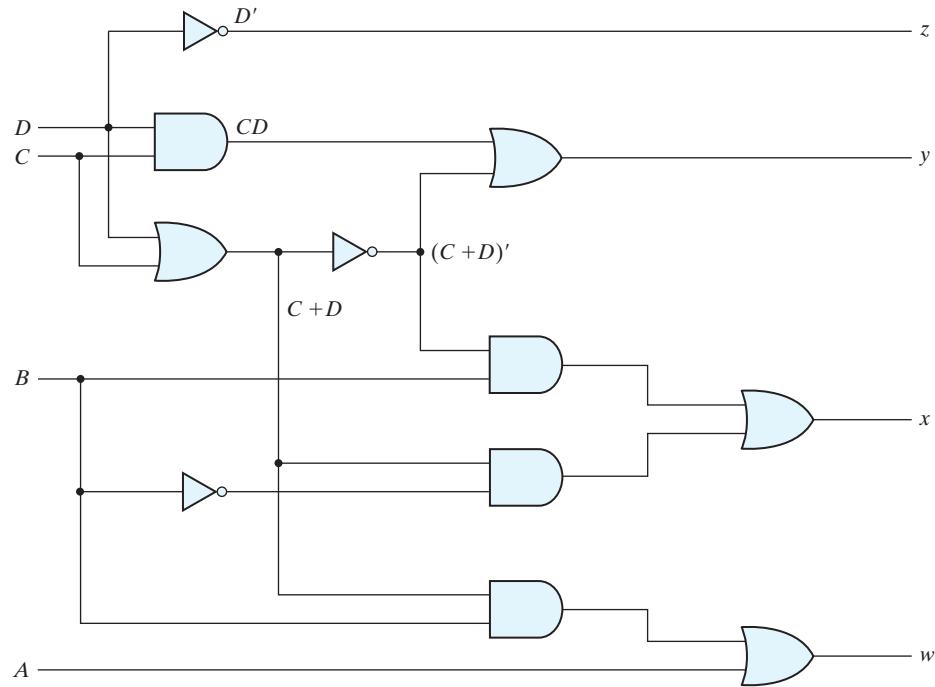
$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$

$$= B'(C + D) + B(C + D)'$$

$$w = A + BC + BD = A + B(C + D)$$

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

Not counting input inverters, the implementation in sum-of-products form requires seven AND gates and three OR gates. The implementation of Fig. 4.4 requires four AND gates, four OR gates, and one inverter. If only the normal inputs are available, the first



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

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.

## 4.5 BINARY ADDER–SUBTRACTOR

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

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

### Half Adder

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

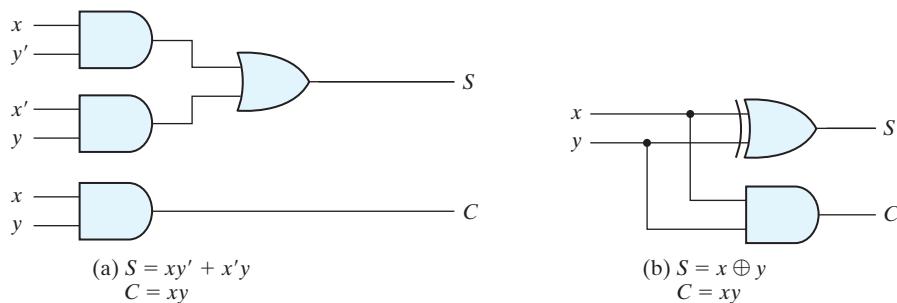
The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

$$\begin{aligned} S &= x'y + xy' \\ C &= xy \end{aligned}$$

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

**Table 4.3**  
*Half Adder*

<b>x</b>	<b>y</b>	<b>c</b>	<b>s</b>
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



**FIGURE 4.5**  
Implementation of half adder

### Full Adder

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 adds not only 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 value from 0 to 3, and binary representation of 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.

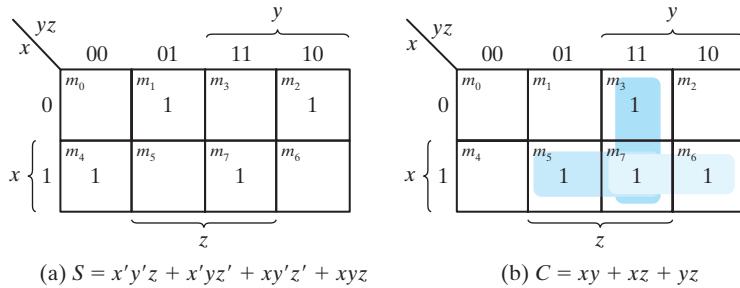
The input and output bits of the combinational circuit have different interpretations at various stages of the problem. On the one hand, physically, the binary signals of the inputs are considered binary digits to be added arithmetically to form a two-digit sum at the output. On the other hand, the same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

$$\begin{aligned} S &= x'y'z + x'yz' + xy'z' + xyz \\ C &= xy + xz + yz \end{aligned}$$

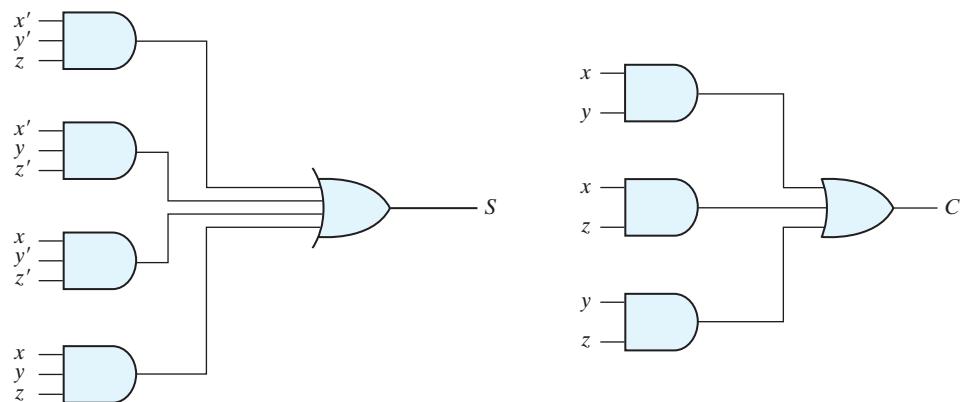
The logic diagram for the full adder implemented in sum-of-products form is shown in Fig. 4.7. It can also be implemented with two half adders and one OR gate, as shown

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

<b>x</b>	<b>y</b>	<b>z</b>	<b>C</b>	<b>S</b>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



**FIGURE 4.6**  
K-Maps for full adder



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

in Fig. 4.8. The  $S$  output from the second half adder is the exclusive-OR of  $z$  and the output of the first half adder, giving

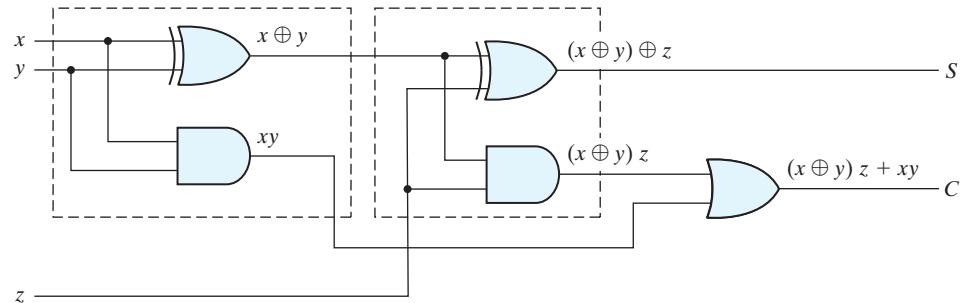
$$\begin{aligned}
 S &= z \oplus (x \oplus y) \\
 &= z'(xy' + x'y) + z(xy' + x'y)' \\
 &= z'(xy' + x'y) + z(xy + x'y') \\
 &= xy'z' + x'y'z + xyz + x'y'z
 \end{aligned}$$

The carry output is

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

### Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain.

**FIGURE 4.8****Implementation of full adder with two half adders and an OR gate**

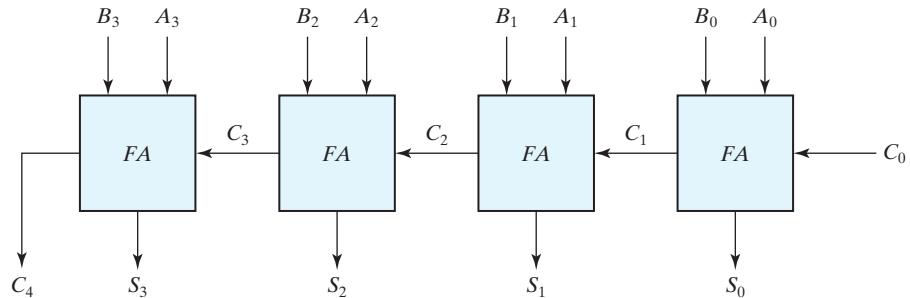
Addition of  $n$ -bit numbers requires a chain of  $n$  full adders or a chain of one-half adder and  $n-1$  full adders. In the former case, the input carry to the least significant position is fixed at 0. Figure 4.9 shows the interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder. The augend bits of  $A$  and the addend bits of  $B$  are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit. The carries are connected in a chain through the full adders. The input carry to the adder is  $C_0$ , and it ripples through the full adders to the output carry  $C_4$ . The  $S$  outputs generate the required sum bits. An  $n$ -bit adder requires  $n$  full adders, with each output carry connected to the input carry of the next higher order full adder.

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

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

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

The four-bit adder is a typical example of a standard component. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit



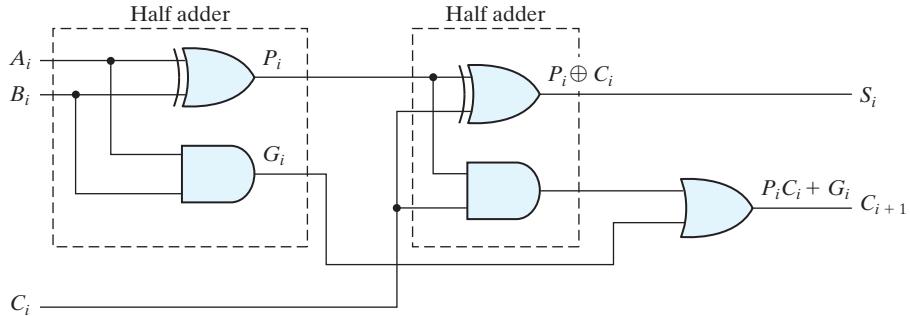
**FIGURE 4.9**  
Four-bit adder

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

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

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

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

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

$$\begin{aligned}P_i &= A_i \oplus B_i \\G_i &= A_i B_i\end{aligned}$$

the output sum and carry can respectively be expressed as

$$\begin{aligned}S_i &= P_i \oplus C_i \\C_{i+1} &= G_i + P_i C_i\end{aligned}$$

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

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

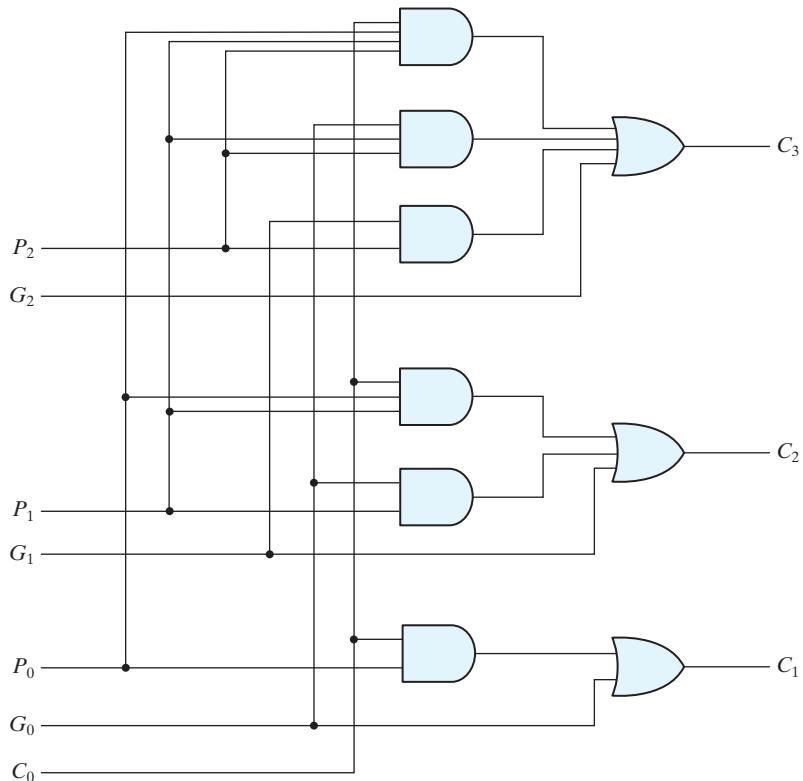
$$\begin{aligned}C_0 &= \text{input carry} \\C_1 &= G_0 + P_0 C_0\end{aligned}$$

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

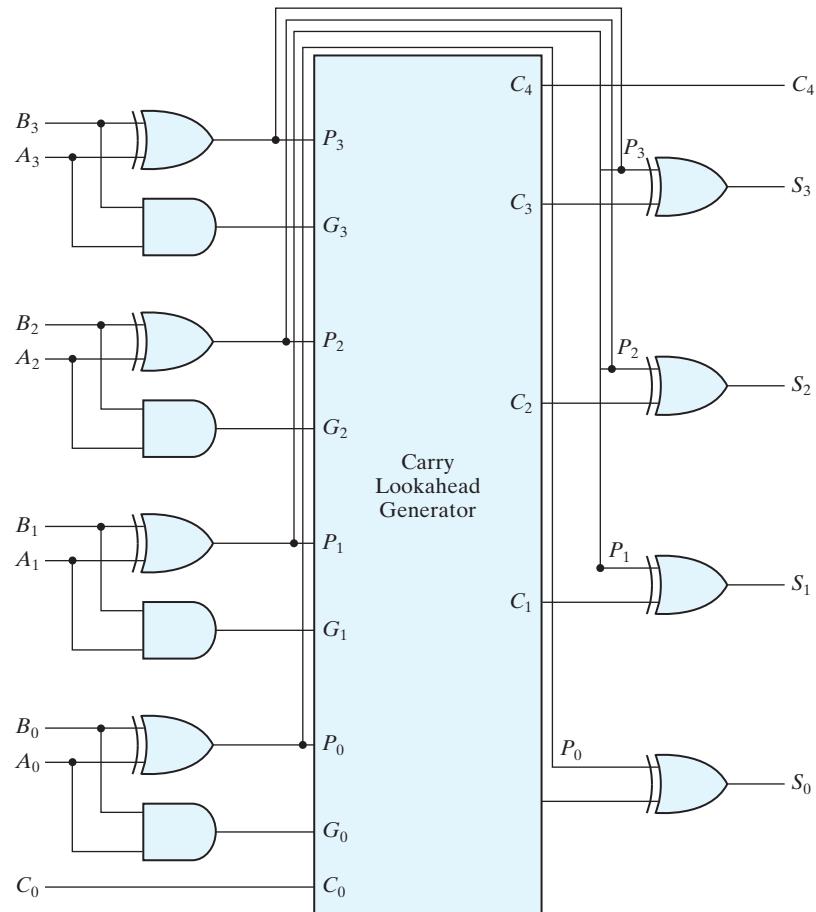
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$

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

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



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

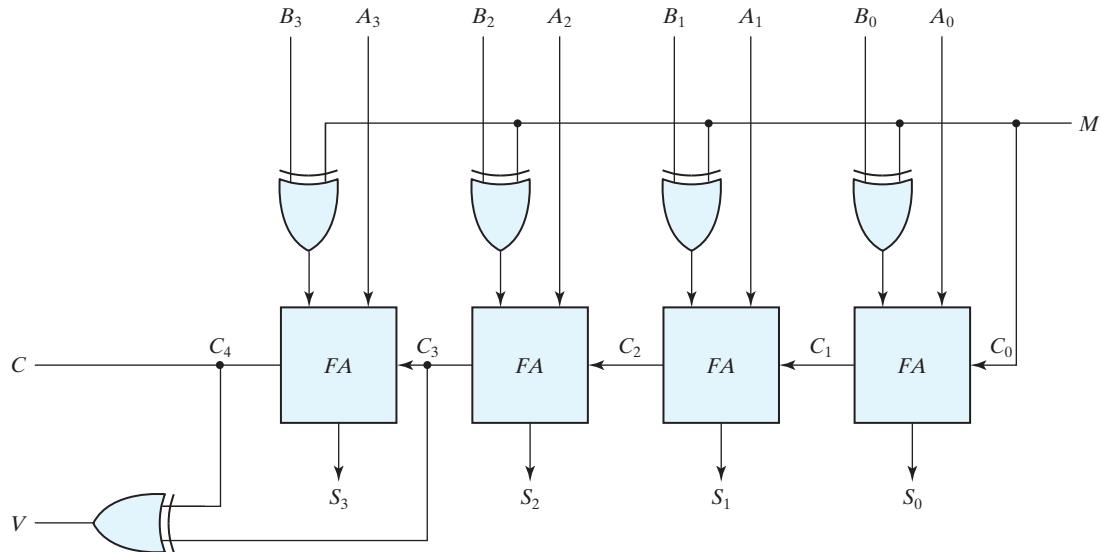


**FIGURE 4.12**  
Four-bit adder with carry lookahead

a delay through two levels of gates. Thus, outputs  $S_1$  through  $S_3$  have equal propagation delay times. The two-level circuit for the output carry  $C_4$  is not shown. This circuit can easily be derived by the equation-substitution method.

### Binary Subtractor

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



**FIGURE 4.13**  
Four-bit adder-subtractor (with overflow detection)

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.)

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

### Overflow

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

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, two details are important: the leftmost bit always represents the sign, and negative numbers are in 2's-complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

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

carries:	0 1	carries:	1 0
+70	0 1000110	–70	1 0111010
+80	<u>0 1010000</u>	–80	<u>1 0110000</u>
+150	1 0010110	–150	0 1101010

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the 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 output carry. There is a wide variety of possible decimal adder circuits, depending upon the code used to represent the decimal digits. Here we examine a decimal adder for the BCD code. (See Section 1.7.)

### BCD Adder

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

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