The study of logic circuits is motivated mostly by their use in digital computers. But such circuits also form the foundation of many other digital systems where performing arithmetic operations on numbers is not of primary interest. For example, in a myriad of control applications actions are determined by some simple logical operations on input information, without having to do extensive numerical computations.

Logic circuits perform operations on digital signals and are usually implemented as electronic circuits where the signal values are restricted to a few discrete values. In *binary* logic circuits there are only two values, 0 and 1. In *decimal* logic circuits there are 10 values, from 0 to 9. Since each signal value is naturally represented by a digit, such logic circuits are referred to as *digital circuits*. In contrast, there exist *analog circuits* where the signals may take on a continuous range of values between some minimum and maximum levels.

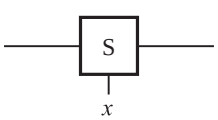
In this book we deal with binary circuits, which have the dominant role in digital technology. We hope to provide the reader with an understanding of how these circuits work, how are they represented in mathematical notation, and how are they designed using modern design automation techniques. We begin by introducing some basic concepts pertinent to the binary logic circuits.

## 2.1 VARIABLES AND FUNCTIONS

The dominance of binary circuits in digital systems is a consequence of their simplicity, which results from constraining the signals to assume only two possible values. The simplest binary element is a switch that has two states. If a given switch is controlled by an input variable $x$, then we will say that the switch is open if $x = 0$ and closed if $x = 1$, as illustrated in Figure 2.1*a*. We will use the graphical symbol in Figure 2.1*b* to represent such switches in the diagrams that follow. Note that the control input $x$ is shown explicitly in the symbol. In Chapter 3 we will explain how such switches are implemented with transistors.

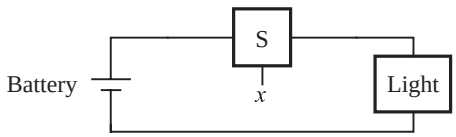
Consider a simple application of a switch, where the switch turns a small lightbulb on or off. This action is accomplished with the circuit in Figure 2.2*a*. A battery provides the power source. The lightbulb glows when sufficient current passes through its filament, which is an electrical resistance. The current flows when the switch is closed, that is, when


(a) Two states of a switch

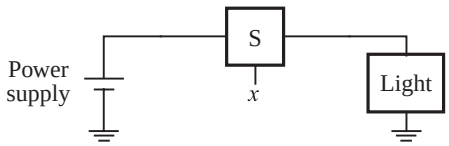



(b) Symbol for a switch

**Figure 2.1** A binary switch.




(a) Simple connection to a battery




(b) Using a ground connection as the return path

**Figure 2.2** A light controlled by a switch.

$x = 1$. In this example the input that causes changes in the behavior of the circuit is the switch control $x$. The output is defined as the state (or condition) of the light, which we will denote by the letter $L$. If the light is on, we will say that $L = 1$. If the the light is off, we will say that $L = 0$. Using this convention, we can describe the state of the light as a function of the input variable $x$. Since $L = 1$ if $x = 1$ and $L = 0$ if $x = 0$, we can say that

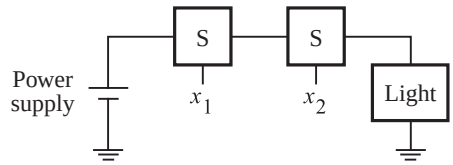
$$L(x) = x$$

This simple *logic expression* describes the output as a function of the input. We say that $L(x) = x$ is a *logic function* and that $x$ is an *input variable*.

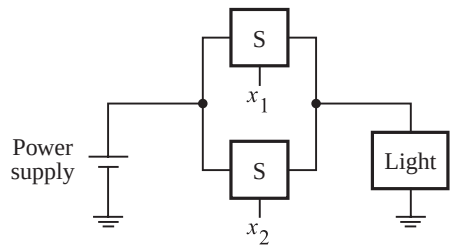
The circuit in Figure 2.2*a* can be found in an ordinary flashlight, where the switch is a simple mechanical device. In an electronic circuit the switch is implemented as a transistor and the light may be a light-emitting diode (LED). An electronic circuit is powered by a power supply of a certain voltage, perhaps 5 volts. One side of the power supply is connected to ground, as shown in Figure 2.2*b*. The ground connection may also be used as the return path for the current, to close the loop, which is achieved by connecting one side of the light to ground as indicated in the figure. Of course, the light can also be connected by a wire directly to the grounded side of the power supply, as in Figure 2.2*a*.

Consider now the possibility of using two switches to control the state of the light. Let $x_1$ and $x_2$ be the control inputs for these switches. The switches can be connected either in series or in parallel as shown in Figure 2.3. Using a series connection, the light will be turned on only if both switches are closed. If either switch is open, the light will be off. This behavior can be described by the expression

$$\begin{aligned} & L(x_1, x_2) = x_1 \cdot x_2 \\ \text{where} \quad & L = 1 \text{ if } x_1 = 1 \text{ and } x_2 = 1, \\ & L = 0 \text{ otherwise.} \end{aligned}$$




(a) The logical AND function (series connection)




(b) The logical OR function (parallel connection)

**Figure 2.3** Two basic functions.

The "·" symbol is called the *AND operator*, and the circuit in Figure 2.3*a* is said to implement a *logical AND function*.

The parallel connection of two switches is given in Figure 2.3*b*. In this case the light will be on if either $x_1$ or $x_2$ switch is closed. The light will also be on if both switches are closed. The light will be off only if both switches are open. This behavior can be stated as

$$L(x_1, x_2) = x_1 + x_2$$

where $L = 1$ if $x_1 = 1$ or $x_2 = 1$ or if $x_1 = x_2 = 1$,

$L = 0$ if $x_1 = x_2 = 0$.

The + symbol is called the *OR operator*, and the circuit in Figure 2.3*b* is said to implement a *logical OR function*.

In the above expressions for AND and OR, the output $L(x_1, x_2)$ is a logic function with input variables $x_1$ and $x_2$. The AND and OR functions are two of the most important logic functions. Together with some other simple functions, they can be used as building blocks for the implementation of all logic circuits. Figure 2.4 illustrates how three switches can be used to control the light in a more complex way. This series-parallel connection of switches realizes the logic function

$$L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$$

The light is on if $x_3 = 1$ and, at the same time, at least one of the $x_1$ or $x_2$ inputs is equal to 1.

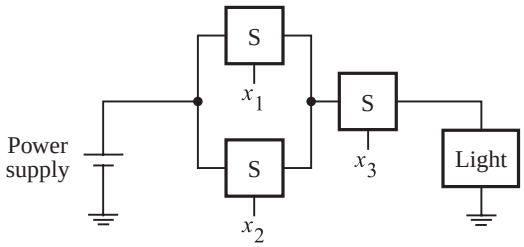



**Figure 2.4** A series-parallel connection.

## 2.2 INVERSION

So far we have assumed that some positive action takes place when a switch is closed, such as turning the light on. It is equally interesting and useful to consider the possibility that a positive action takes place when a switch is opened. Suppose that we connect the light as shown in Figure 2.5. In this case the switch is connected in parallel with the light, rather than in series. Consequently, a closed switch will short-circuit the light and prevent the current from flowing through it. Note that we have included an extra resistor in this circuit to ensure that the closed switch does not short-circuit the power supply. The light will be turned on when the switch is opened. Formally, we express this functional behavior as

$$L(x) = \overline{x}$$
$$\text{where} \quad L = 1 \text{ if } x = 0,$$
$$L = 0 \text{ if } x = 1$$

The value of this function is the inverse of the value of the input variable. Instead of using the word *inverse*, it is more common to use the term *complement*. Thus we say that $L(x)$ is a complement of $x$ in this example. Another frequently used term for the same operation is the *NOT operation*. There are several commonly used notations for indicating the complementation. In the preceding expression we placed an overbar on top of $x$. This notation is probably the best from the visual point of view. However, when complements

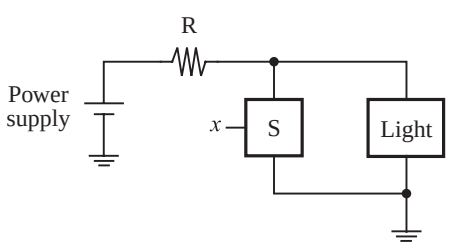



**Figure 2.5** An inverting circuit.

are needed in expressions that are typed using a computer keyboard, which is often done when using CAD tools, it is impractical to use overbars. Instead, either an apostrophe is placed after the variable, or the exclamation mark (!) or the tilde character (~) or the word NOT is placed in front of the variable to denote the complementation. Thus the following are equivalent:

$$\bar{x} = x' = !x = \sim x = \text{NOT } x$$

The complement operation can be applied to a single variable or to more complex operations. For example, if

$$f(x_1, x_2) = x_1 + x_2$$

then the complement of $f$ is

$$\bar{f}(x_1, x_2) = \overline{x_1 + x_2}$$

This expression yields the logic value 1 only when neither $x_1$ nor $x_2$ is equal to 1, that is, when $x_1 = x_2 = 0$. Again, the following notations are equivalent:

$$\overline{x_1 + x_2} = (x_1 + x_2)' = !(x_1 + x_2) = \sim(x_1 + x_2) = \text{NOT } (x_1 + x_2)$$

## 2.3 TRUTH TABLES

We have introduced the three most basic logic operations—AND, OR, and complement—by relating them to simple circuits built with switches. This approach gives these operations a certain "physical meaning." The same operations can also be defined in the form of a table, called a *truth table*, as shown in Figure 2.6. The first two columns (to the left of the heavy vertical line) give all four possible combinations of logic values that the variables $x_1$ and $x_2$ can have. The next column defines the AND operation for each combination of values of $x_1$ and $x_2$, and the last column defines the OR operation. Because we will frequently need to refer to "combinations of logic values" applied to some variables, we will adopt a shorter term, *valuation*, to denote such a combination of logic values.

| $x_1$ | $x_2$ | $x_1 \cdot x_2$ | $x_1 + x_2$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |
| | | AND | OR |

**Figure 2.6** A truth table for the AND and OR operations.

| $x_1$ | $x_2$ | $x_3$ | $x_1 \cdot x_2 \cdot x_3$ | $x_1 + x_2 + x_3$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**Figure 2.7** Three-input AND and OR operations.

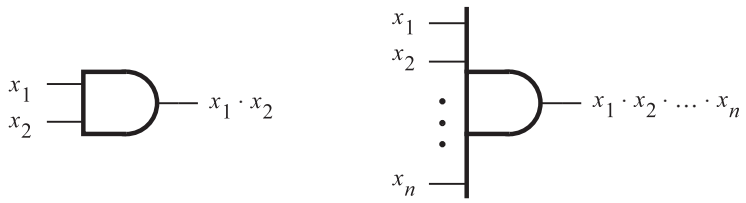
The truth table is a useful aid for depicting information involving logic functions. We will use it in this book to define specific functions and to show the validity of certain functional relations. Small truth tables are easy to deal with. However, they grow exponentially in size with the number of variables. A truth table for three input variables has eight rows because there are eight possible valuations of these variables. Such a table is given in Figure 2.7, which defines three-input AND and OR functions. For four input variables the truth table has 16 rows, and so on. In general, for $n$ input variables the truth table has $2^n$ rows.

The AND and OR operations can be extended to $n$ variables. An AND function of variables $x_1, x_2, \ldots, x_n$ has the value 1 only if all $n$ variables are equal to 1. An OR function of variables $x_1, x_2, \ldots, x_n$ has the value 1 if at least one, or more, of the variables is equal to 1.

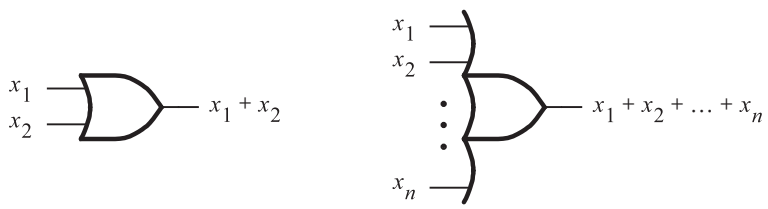
## 2.4 LOGIC GATES AND NETWORKS

The three basic logic operations introduced in the previous sections can be used to implement logic functions of any complexity. A complex function may require many of these basic operations for its implementation. Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a *logic gate*. A logic gate has one or more inputs and one output that is a function of its inputs. It is often convenient to describe a logic circuit by drawing a circuit diagram, or *schematic*, consisting of graphical symbols representing the logic gates. The graphical symbols for the AND, OR, and NOT gates are shown in Figure 2.8. The figure indicates on the left side how the AND and OR gates are drawn when there are only a few inputs. On the right side it shows how the symbols are augmented to accommodate a greater number of inputs. We will show how logic gates are built using transistors in Chapter 3.

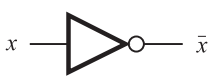
A larger circuit is implemented by a *network* of gates. For example, the logic function from Figure 2.4 can be implemented by the network in Figure 2.9. The complexity of a given network has a direct impact on its cost. Because it is always desirable to reduce




(a) AND gates




(b) OR gates




(c) NOT gate

**Figure 2.8** The basic gates.


**Figure 2.9** The function from Figure 2.4.

the cost of any manufactured product, it is important to find ways for implementing logic circuits as inexpensively as possible. We will see shortly that a given logic function can be implemented with a number of different networks. Some of these networks are simpler than others, hence searching for the solutions that entail minimum cost is prudent.

In technical jargon a network of gates is often called a *logic network* or simply a *logic circuit*. We will use these terms interchangeably.

### 2.4.1 ANALYSIS OF A LOGIC NETWORK

A designer of digital systems is faced with two basic issues. For an existing logic network, it must be possible to determine the function performed by the network. This task is referred to as the *analysis* process. The reverse task of designing a new network that implements a desired functional behavior is referred to as the *synthesis* process. The analysis process is rather straightforward and much simpler than the synthesis process.

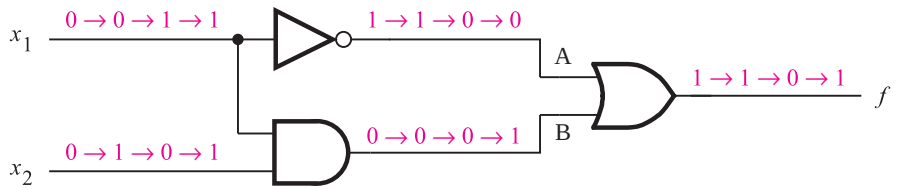
Figure 2.10*a* shows a simple network consisting of three gates. To determine its functional behavior, we can consider what happens if we apply all possible input signals to it. Suppose that we start by making $x_1 = x_2 = 0$. This forces the output of the NOT gate to be equal to 1 and the output of the AND gate to be 0. Because one of the inputs to the OR gate is 1, the output of this gate will be 1. Therefore, $f = 1$ if $x_1 = x_2 = 0$. If we let $x_1 = 0$ and $x_2 = 1$, then no change in the value of $f$ will take place, because the outputs of the NOT and AND gates will still be 1 and 0, respectively. Next, if we apply $x_1 = 1$ and $x_2 = 0$, then the output of the NOT gate changes to 0 while the output of the AND gate remains at 0. Both inputs to the OR gate are then equal to 0; hence the value of $f$ will be 0. Finally, let $x_1 = x_2 = 1$. Then the output of the AND gate goes to 1, which in turn causes $f$ to be equal to 1. Our verbal explanation can be summarized in the form of the truth table shown in Figure 2.10*b*.

#### Timing Diagram

We have determined the behavior of the network in Figure 2.10*a* by considering the four possible valuations of the inputs $x_1$ and $x_2$. Suppose that the signals that correspond to these valuations are applied to the network in the order of our discussion; that is, $(x_1, x_2) = (0, 0)$ followed by $(0, 1)$, $(1, 0)$, and $(1, 1)$. Then changes in the signals at various points in the network would be as indicated in blue in the figure. The same information can be presented in graphical form, known as a *timing diagram*, as shown in Figure 2.10*c*. The time runs from left to right, and each input valuation is held for some fixed period. The figure shows the waveforms for the inputs and output of the network, as well as for the internal signals at the points labeled $A$ and $B$.

The timing diagram in Figure 2.10*c* shows that changes in the waveforms at points $A$ and $B$ and the output $f$ take place instantaneously when the inputs $x_1$ and $x_2$ change their values. These idealized waveforms are based on the assumption that logic gates respond to changes on their inputs in zero time. Such timing diagrams are useful for indicating the *functional behavior* of logic circuits. However, practical logic gates are implemented using electronic circuits which need some time to change their states. Thus, there is a delay between a change in input values and a corresponding change in the output value of a gate. In chapters that follow we will use timing diagrams that incorporate such delays.

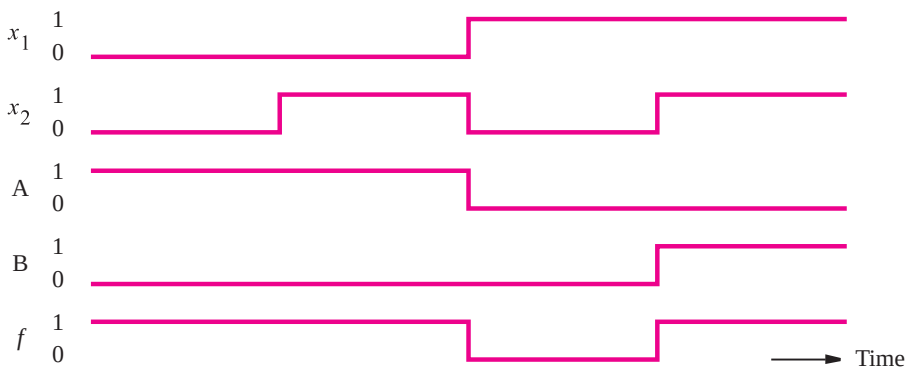
Timing diagrams are used for many purposes. They depict the behavior of a logic circuit in a form that can be observed when the circuit is tested using instruments such as logic analyzers and oscilloscopes. Also, they are often generated by CAD tools to show the designer how a given circuit is expected to behave before it is actually implemented electronically. We will introduce the CAD tools later in this chapter and will make use of them throughout the book.



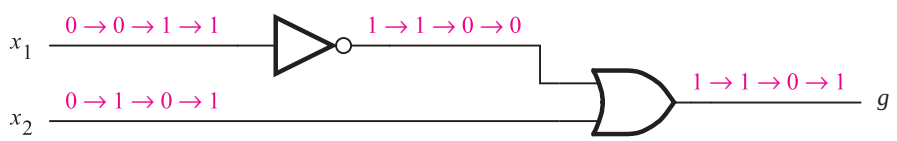

(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

| $x_1$ | $x_2$ | $f(x_1, x_2)$ | A | B |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 |

(b) Truth table




(c) Timing diagram




(d) Network that implements $g = \bar{x}_1 + x_2$

**Figure 2.10** An example of logic networks.

### Functionally Equivalent Networks

Now consider the network in Figure 2.10*d*. Going through the same analysis procedure, we find that the output $g$ changes in exactly the same way as $f$ does in part (*a*) of the figure. Therefore, $g(x_1, x_2) = f(x_1, x_2)$, which indicates that the two networks are functionally equivalent; the output behavior of both networks is represented by the truth table in Figure

2.10*b*. Since both networks realize the same function, it makes sense to use the simpler one, which is less costly to implement.

In general, a logic function can be implemented with a variety of different networks, probably having different costs. This raises an important question: How does one find the best implementation for a given function? Many techniques exist for synthesizing logic functions. We will discuss the main approaches in Chapter 4. For now, we should note that some manipulation is needed to transform the more complex network in Figure 2.10*a* into the network in Figure 2.10*d*. Since $f(x_1, x_2) = \bar{x}_1 + x_1 \cdot x_2$ and $g(x_1, x_2) = \bar{x}_1 + x_2$, there must exist some rules that can be used to show the equivalence

$$\bar{x}_1 + x_1 \cdot x_2 = \bar{x}_1 + x_2$$

We have already established this equivalence through detailed analysis of the two circuits and construction of the truth table. But the same outcome can be achieved through algebraic manipulation of logic expressions. In the next section we will discuss a mathematical approach for dealing with logic functions, which provides the basis for modern design techniques.

## 2.5 BOOLEAN ALGEBRA

In 1849 George Boole published a scheme for the algebraic description of processes involved in logical thought and reasoning [1]. Subsequently, this scheme and its further refinements became known as *Boolean algebra*. It was almost 100 years later that this algebra found application in the engineering sense. In the late 1930s Claude Shannon showed that Boolean algebra provides an effective means of describing circuits built with switches [2]. The algebra can, therefore, be used to describe logic circuits. We will show that this algebra is a powerful tool that can be used for designing and analyzing logic circuits. The reader will come to appreciate that it provides the foundation for much of our modern digital technology.

### Axioms of Boolean Algebra

Like any algebra, Boolean algebra is based on a set of rules that are derived from a small number of basic assumptions. These assumptions are called *axioms*. Let us assume that Boolean algebra $B$ involves elements that take on one of two values, 0 and 1. Assume that the following axioms are true:

1*a*. $0 \cdot 0 = 0$

1*b*. $1 + 1 = 1$

2*a*. $1 \cdot 1 = 1$

2*b*. $0 + 0 = 0$

3*a*. $0 \cdot 1 = 1 \cdot 0 = 0$

3*b*. $1 + 0 = 0 + 1 = 1$

4*a*. If $x = 0$, then $\bar{x} = 1$

4*b*. If $x = 1$, then $\bar{x} = 0$

### Single-Variable Theorems

From the axioms we can define some rules for dealing with single variables. These rules are often called *theorems*. If $x$ is a variable in $B$, then the following theorems hold:

5*a*. $x \cdot 0 = 0$
5*b*. $x + 1 = 1$
6*a*. $x \cdot 1 = x$
6*b*. $x + 0 = x$
7*a*. $x \cdot x = x$
7*b*. $x + x = x$
8*a*. $x \cdot \overline{x} = 0$
8*b*. $x + \overline{x} = 1$
9. $\overline{\overline{x}} = x$

It is easy to prove the validity of these theorems by perfect induction, that is, by substituting the values $x = 0$ and $x = 1$ into the expressions and using the axioms given above. For example, in theorem 5*a*, if $x = 0$, then the theorem states that $0 \cdot 0 = 0$, which is true according to axiom 1*a*. Similarly, if $x = 1$, then theorem 5*a* states that $1 \cdot 0 = 0$, which is also true according to axiom 3*a*. The reader should verify that theorems 5*a* to 9 can be proven in this way.

### Duality

Notice that we have listed the axioms and the single-variable theorems in pairs. This is done to reflect the important *principle of duality*. Given a logic expression, its *dual* is obtained by replacing all $+$ operators with $\cdot$ operators, and vice versa, and by replacing all 0s with 1s, and vice versa. The dual of any true statement (axiom or theorem) in Boolean algebra is also a true statement. At this point in the discussion, the reader will not appreciate why duality is a useful concept. However, this concept will become clear later in the chapter, when we will show that duality implies that at least two different ways exist to express every logic function with Boolean algebra. Often, one expression leads to a simpler physical implementation than the other and is thus preferable.

### Two- and Three-Variable Properties

To enable us to deal with a number of variables, it is useful to define some two- and three-variable algebraic identities. For each identity, its dual version is also given. These identities are often referred to as *properties*. They are known by the names indicated below. If $x$, $y$, and $z$ are the variables in $B$, then the following properties hold:

| | | |
|---|---|---|
| 10*a*. | $x \cdot y = y \cdot x$ | *Commutative* |
| 10*b*. | $x + y = y + x$ | |
| 11*a*. | $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ | *Associative* |
| 11*b*. | $x + (y + z) = (x + y) + z$ | |
| 12*a*. | $x \cdot (y + z) = x \cdot y + x \cdot z$ | *Distributive* |
| 12*b*. | $x + y \cdot z = (x + y) \cdot (x + z)$ | |
| 13*a*. | $x + x \cdot y = x$ | *Absorption* |

| $x$ | $y$ | $x \cdot y$ | $\overline{x \cdot y}$ | $\overline{x}$ | $\overline{y}$ | $\overline{x} + \overline{y}$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| | | LHS | | RHS | | |

**Figure 2.11** Proof of DeMorgan's theorem in 15*a*.

13*b*. $x \cdot (x + y) = x$

14*a*. $x \cdot y + x \cdot \overline{y} = x$ *Combining*

14*b*. $(x + y) \cdot (x + \overline{y}) = x$

15*a*. $\overline{x \cdot y} = \overline{x} + \overline{y}$ *DeMorgan's theorem*

15*b*. $\overline{x + y} = \overline{x} \cdot \overline{y}$

16*a*. $x + \overline{x} \cdot y = x + y$

16*b*. $x \cdot (\overline{x} + y) = x \cdot y$

17*a*. $x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} \cdot z$ *Consensus*

17*b*. $(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)$

Again, we can prove the validity of these properties either by perfect induction or by performing algebraic manipulation. Figure 2.11 illustrates how perfect induction can be used to prove DeMorgan's theorem, using the format of a truth table. The evaluation of left-hand and right-hand sides of the identity in 15*a* gives the same result.

We have listed a number of axioms, theorems, and properties. Not all of these are necessary to define Boolean algebra. For example, assuming that the + and · operations are defined, it is sufficient to include theorems 5 and 8 and properties 10 and 12. These are sometimes referred to as Huntington's basic postulates [3]. The other identities can be derived from these postulates.

The preceding axioms, theorems, and properties provide the information necessary for performing algebraic manipulation of more complex expressions.

---

**Example 2.1**

Let us prove the validity of the logic equation

$$(x_1 + x_3) \cdot (\overline{x}_1 + \overline{x}_3) = x_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_3$$

The left-hand side can be manipulated as follows. Using the distributive property, 12*a*, gives

$$\text{LHS} = (x_1 + x_3) \cdot \overline{x}_1 + (x_1 + x_3) \cdot \overline{x}_3$$

Applying the distributive property again yields

$$\text{LHS} = x_1 \cdot \bar{x}_1 + x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3 + x_3 \cdot \bar{x}_3$$

Note that the distributive property allows ANDing the terms in parenthesis in a way analogous to multiplication in ordinary algebra. Next, according to theorem 8*a*, the terms $x_1 \cdot \bar{x}_1$ and $x_3 \cdot \bar{x}_3$ are both equal to 0. Therefore,

$$\text{LHS} = 0 + x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3 + 0$$

From 6*b* it follows that

$$\text{LHS} = x_3 \cdot \bar{x}_1 + x_1 \cdot \bar{x}_3$$

Finally, using the commutative property, 10*a* and 10*b*, this becomes

$$\text{LHS} = x_1 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_3$$

which is the same as the right-hand side of the initial equation.

---

**Example 2.2** Consider the logic equation

$$x_1 \cdot \bar{x}_3 + \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot x_3 = \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

The left-hand side can be manipulated as follows

$$\begin{aligned}
\text{LHS} &= x_1 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_2 \cdot x_3 && \text{using } 10b \\
&= x_1 \cdot (\bar{x}_3 + x_3) + \bar{x}_2 \cdot (\bar{x}_3 + x_3) && \text{using } 12a \\
&= x_1 \cdot 1 + \bar{x}_2 \cdot 1 && \text{using } 8b \\
&= x_1 + \bar{x}_2 && \text{using } 6a
\end{aligned}$$

The right-hand side can be manipulated as

$$\begin{aligned}
\text{RHS} &= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot (x_2 + \bar{x}_2) && \text{using } 12a \\
&= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot 1 && \text{using } 8b \\
&= \bar{x}_1 \cdot \bar{x}_2 + x_1 && \text{using } 6a \\
&= x_1 + \bar{x}_1 \cdot \bar{x}_2 && \text{using } 10b \\
&= x_1 + \bar{x}_2 && \text{using } 16a
\end{aligned}$$

Being able to manipulate both sides of the initial equation into identical expressions establishes the validity of the equation. Note that the same logic function is represented by either the left- or the right-hand side of the above equation; namely

$$\begin{aligned}
f(x_1, x_2, x_3) &= x_1 \cdot \bar{x}_3 + \bar{x}_2 \cdot \bar{x}_3 + x_1 \cdot x_3 + \bar{x}_2 \cdot x_3 \\
&= \bar{x}_1 \cdot \bar{x}_2 + x_1 \cdot x_2 + x_1 \cdot \bar{x}_2
\end{aligned}$$

As a result of manipulation, we have found a much simpler expression

$$f(x_1, x_2, x_3) = x_1 + \bar{x}_2$$

which also represents the same function. This simpler expression would result in a lower-cost logic circuit that could be used to implement the function.

---

Examples 2.1 and 2.2 illustrate the purpose of the axioms, theorems, and properties as a mechanism for algebraic manipulation. Even these simple examples suggest that it is impractical to deal with highly complex expressions in this way. However, these theorems and properties provide the basis for automating the synthesis of logic functions in CAD tools. To understand what can be achieved using these tools, the designer needs to be aware of the fundamental concepts.

### 2.5.1 THE VENN DIAGRAM

We have suggested that perfect induction can be used to verify the theorems and properties. This procedure is quite tedious and not very informative from the conceptual point of view. A simple visual aid that can be used for this purpose also exists. It is called the Venn diagram, and the reader is likely to find that it provides for a more intuitive understanding of how two expressions may be equivalent.

The Venn diagram has traditionally been used in mathematics to provide a graphical illustration of various operations and relations in the algebra of sets. A set $s$ is a collection of elements that are said to be the members of $s$. In the Venn diagram the elements of a set are represented by the area enclosed by a contour such as a square, a circle, or an ellipse. For example, in a universe $N$ of integers from 1 to 10, the set of even numbers is $E = \{2, 4, 6, 8, 10\}$. A contour representing $E$ encloses the even numbers. The odd numbers form the complement of $E$; hence the area outside the contour represents $\overline{E} = \{1, 3, 5, 7, 9\}$.

Since in Boolean algebra there are only two values (elements) in the universe, $B = \{0, 1\}$, we will say that the area within a contour corresponding to a set $s$ denotes that $s = 1$, while the area outside the contour denotes $s = 0$. In the diagram we will shade the area where $s = 1$. The concept of the Venn diagram is illustrated in Figure 2.12. The universe $B$ is represented by a square. Then the constants 1 and 0 are represented as shown in parts ($a$) and ($b$) of the figure. A variable, say, $x$, is represented by a circle, such that the area inside the circle corresponds to $x = 1$, while the area outside the circle corresponds to $x = 0$. This is illustrated in part ($c$). An expression involving one or more variables is depicted by shading the area where the value of the expression is equal to 1. Part ($d$) indicates how the complement of $x$ is represented.

To represent two variables, $x$ and $y$, we draw two overlapping circles. Then the area where the circles overlap represents the case where $x = y = 1$, namely, the AND of $x$ and $y$, as shown in part ($e$). Since this common area consists of the intersecting portions of $x$ and $y$, the AND operation is often referred to formally as the *intersection* of $x$ and $y$. Part ($f$) illustrates the OR operation, where $x + y$ represents the total area within both circles, namely, where at least one of $x$ or $y$ is equal to 1. Since this combines the areas in the circles, the OR operation is formally often called the *union* of $x$ and $y$.

Part ($g$) depicts the product term $x \cdot \overline{y}$, which is represented by the intersection of the area for $x$ with that for $\overline{y}$. Part ($h$) gives a three-variable example; the expression $x \cdot y + z$ is the union of the area for $z$ with that of the intersection of $x$ and $y$.

To see how we can use Venn diagrams to verify the equivalence of two expressions, let us demonstrate the validity of the distributive property, 12$a$, in section 2.5. Figure 2.13 gives the construction of the left and right sides of the identity that defines the property

$$x \cdot (y + z) = x \cdot y + x \cdot z$$

"sum." Thus we say that the expression

$$x_1 \cdot \bar{x}_2 \cdot x_3 + \bar{x}_1 \cdot x_4 + x_2 \cdot x_3 \cdot \bar{x}_4$$

is a sum of three product terms, whereas the expression

$$(\bar{x}_1 + x_3) \cdot (x_1 + \bar{x}_3) \cdot (\bar{x}_2 + x_3 + x_4)$$

is a product of three sum terms.

### 2.5.3 PRECEDENCE OF OPERATIONS

Using the three basic operations—AND, OR, and NOT—it is possible to construct an infinite number of logic expressions. Parentheses can be used to indicate the order in which the operations should be performed. However, to avoid an excessive use of parentheses, another convention defines the precedence of the basic operations. It states that in the absence of parentheses, operations in a logic expression must be performed in the order: NOT, AND, and then OR. Thus in the expression

$$x_1 \cdot x_2 + \bar{x}_1 \cdot \bar{x}_2$$

it is first necessary to generate the complements of $x_1$ and $x_2$. Then the product terms $x_1 \cdot x_2$ and $\bar{x}_1 \cdot \bar{x}_2$ are formed, followed by the sum of the two product terms. Observe that in the absence of this convention, we would have to use parentheses to achieve the same effect as follows:

$$(x_1 \cdot x_2) + ((\bar{x}_1) \cdot (\bar{x}_2))$$

Finally, to simplify the appearance of logic expressions, it is customary to omit the $\cdot$ operator when there is no ambiguity. Therefore, the preceding expression can be written as

$$x_1 x_2 + \bar{x}_1 \bar{x}_2$$

We will use this style throughout the book.

## 2.6 SYNTHESIS USING AND, OR, AND NOT GATES

Armed with some basic ideas, we can now try to implement arbitrary functions using the AND, OR, and NOT gates. Suppose that we wish to design a logic circuit with two inputs, $x_1$ and $x_2$. Assume that $x_1$ and $x_2$ represent the states of two switches, either of which may be open (0) or closed (1). The function of the circuit is to continuously monitor the state of the switches and to produce an output logic value 1 whenever the switches $(x_1, x_2)$ are in states (0, 0), (0, 1), or (1, 1). If the state of the switches is (1, 0), the output should be 0. Another way of stating the required functional behavior of this circuit is that the output must be equal to 0 if the switch $x_1$ is closed and $x_2$ is open; otherwise, the output must be 1. We can express the required behavior using a truth table, as shown in Figure 2.15.

A possible procedure for designing a logic circuit that implements the truth table is to create a product term that has a value of 1 for each valuation for which the output function $f$ has to be 1. Then we can take a logical sum of these product terms to realize $f$. Let us

| $x_1$ | $x_2$ | $f(x_1, x_2)$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

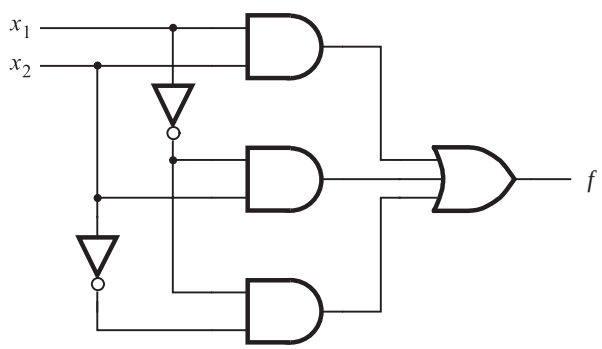
**Figure 2.15** A function to be synthesized.

begin with the fourth row of the truth table, which corresponds to $x_1 = x_2 = 1$. The product term that is equal to 1 for this valuation is $x_1 \cdot x_2$, which is just the AND of $x_1$ and $x_2$. Next consider the first row of the table, for which $x_1 = x_2 = 0$. For this valuation the value 1 is produced by the product term $\bar{x}_1 \cdot \bar{x}_2$. Similarly, the second row leads to the term $\bar{x}_1 \cdot x_2$. Thus $f$ may be realized as

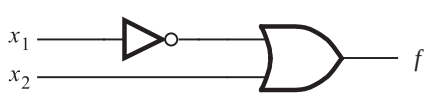
$$f(x_1, x_2) = x_1x_2 + \bar{x}_1\bar{x}_2 + \bar{x}_1x_2$$

The logic network that corresponds to this expression is shown in Figure 2.16*a*.

Although this network implements $f$ correctly, it is not the simplest such network. To find a simpler network, we can manipulate the obtained expression using the theorems and properties from section 2.5. According to theorem 7*b*, we can replicate any term in a logical




(a) Canonical sum-of-products




(b) Minimal-cost realization

**Figure 2.16** Two implementations of the function in Figure 2.15.

sum expression. Replicating the third product term, the above expression becomes

$$f(x_1, x_2) = x_1x_2 + \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 + \bar{x}_1x_2$$

Using the commutative property 10*b* to interchange the second and third product terms gives

$$f(x_1, x_2) = x_1x_2 + \bar{x}_1x_2 + \bar{x}_1\bar{x}_2 + \bar{x}_1x_2$$

Now the distributive property 12*a* allows us to write

$$f(x_1, x_2) = (x_1 + \bar{x}_1)x_2 + \bar{x}_1(\bar{x}_2 + x_2)$$

Applying theorem 8*b* we get

$$f(x_1, x_2) = 1 \cdot x_2 + \bar{x}_1 \cdot 1$$

Finally, theorem 6*a* leads to

$$f(x_1, x_2) = x_2 + \bar{x}_1$$

The network described by this expression is given in Figure 2.16*b*. Obviously, the cost of this network is much less than the cost of the network in part (*a*) of the figure.

This simple example illustrates two things. First, a straightforward implementation of a function can be obtained by using a product term (AND gate) for each row of the truth table for which the function is equal to 1. Each product term contains all input variables, and it is formed such that if the input variable $x_i$ is equal to 1 in the given row, then $x_i$ is entered in the term; if $x_i = 0$, then $\bar{x}_i$ is entered. The sum of these product terms realizes the desired function. Second, there are many different networks that can realize a given function. Some of these networks may be simpler than others. Algebraic manipulation can be used to derive simplified logic expressions and thus lower-cost networks.

The process whereby we begin with a description of the desired functional behavior and then generate a circuit that realizes this behavior is called *synthesis*. Thus we can say that we "synthesized" the networks in Figure 2.16 from the truth table in Figure 2.15. Generation of AND-OR expressions from a truth table is just one of many types of synthesis techniques that we will encounter in this book.

### 2.6.1 SUM-OF-PRODUCTS AND PRODUCT-OF-SUMS FORMS

Having introduced the synthesis process by means of a very simple example, we will now present it in more formal terms using the terminology that is encountered in the technical literature. We will also show how the principle of duality, which was introduced in section 2.5, applies broadly in the synthesis process.

If a function $f$ is specified in the form of a truth table, then an expression that realizes $f$ can be obtained by considering either the rows in the table for which $f = 1$, as we have already done, or by considering the rows for which $f = 0$, as we will explain shortly.

### Minterms

For a function of $n$ variables, a product term in which each of the $n$ variables appears once is called a *minterm*. The variables may appear in a minterm either in uncomplemented or complemented form. For a given row of the truth table, the minterm is formed by including $x_i$ if $x_i = 1$ and by including $\overline{x}_i$ if $x_i = 0$.

To illustrate this concept, consider the truth table in Figure 2.17. We have numbered the rows of the table from 0 to 7, so that we can refer to them easily. From the discussion of the binary number representation in section 1.6, we can observe that the row numbers chosen are just the numbers represented by the bit patterns of variables $x_1$, $x_2$, and $x_3$. The figure shows all minterms for the three-variable table. For example, in the first row the variables have the values $x_1 = x_2 = x_3 = 0$, which leads to the minterm $\overline{x}_1\overline{x}_2\overline{x}_3$. In the second row $x_1 = x_2 = 0$ and $x_3 = 1$, which gives the minterm $\overline{x}_1\overline{x}_2x_3$, and so on. To be able to refer to the individual minterms easily, it is convenient to identify each minterm by an index that corresponds to the row numbers shown in the figure. We will use the notation $m_i$ to denote the minterm for row number $i$. Thus $m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$, $m_1 = \overline{x}_1\overline{x}_2x_3$, and so on.

### Sum-of-Products Form

A function $f$ can be represented by an expression that is a sum of minterms, where each minterm is ANDed with the value of $f$ for the corresponding valuation of input variables. For example, the two-variable minterms are $m_0 = \overline{x}_1\overline{x}_2$, $m_1 = \overline{x}_1x_2$, $m_2 = x_1\overline{x}_2$, and $m_3 = x_1x_2$. The function in Figure 2.15 can be represented as

$$\begin{aligned} f &= m_0 \cdot 1 + m_1 \cdot 1 + m_2 \cdot 0 + m_3 \cdot 1 \\ &= m_0 + m_1 + m_3 \\ &= \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + x_1x_2 \end{aligned}$$

which is the form that we derived in the previous section using an intuitive approach. Only the minterms that correspond to the rows for which $f = 1$ appear in the resulting expression.

Any function $f$ can be represented by a sum of minterms that correspond to the rows in the truth table for which $f = 1$. The resulting implementation is functionally correct and

| Row number | $x_1$ | $x_2$ | $x_3$ | Minterm | Maxterm |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$ | $M_0 = x_1 + x_2 + x_3$ |
| 1 | 0 | 0 | 1 | $m_1 = \overline{x}_1\overline{x}_2x_3$ | $M_1 = x_1 + x_2 + \overline{x}_3$ |
| 2 | 0 | 1 | 0 | $m_2 = \overline{x}_1x_2\overline{x}_3$ | $M_2 = x_1 + \overline{x}_2 + x_3$ |
| 3 | 0 | 1 | 1 | $m_3 = \overline{x}_1x_2x_3$ | $M_3 = x_1 + \overline{x}_2 + \overline{x}_3$ |
| 4 | 1 | 0 | 0 | $m_4 = x_1\overline{x}_2\overline{x}_3$ | $M_4 = \overline{x}_1 + x_2 + x_3$ |
| 5 | 1 | 0 | 1 | $m_5 = x_1\overline{x}_2x_3$ | $M_5 = \overline{x}_1 + x_2 + \overline{x}_3$ |
| 6 | 1 | 1 | 0 | $m_6 = x_1x_2\overline{x}_3$ | $M_6 = \overline{x}_1 + \overline{x}_2 + x_3$ |
| 7 | 1 | 1 | 1 | $m_7 = x_1x_2x_3$ | $M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$ |

**Figure 2.17** Three-variable minterms and maxterms.

unique, but it is not necessarily the lowest-cost implementation of $f$. A logic expression consisting of product (AND) terms that are summed (ORed) is said to be of the *sum-of-products* (*SOP*) form. If each product term is a minterm, then the expression is called a *canonical sum-of-products* for the function $f$. As we have seen in the example of Figure 2.16, the first step in the synthesis process is to derive a canonical sum-of-products expression for the given function. Then we can manipulate this expression, using the theorems and properties of section 2.5, with the goal of finding a functionally equivalent sum-of-products expression that has a lower cost.

As another example, consider the three-variable function $f(x_1, x_2, x_3)$, specified by the truth table in Figure 2.18. To synthesize this function, we have to include the minterms $m_1$, $m_4$, $m_5$, and $m_6$. Copying these minterms from Figure 2.17 leads to the following canonical sum-of-products expression for $f$

$$f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

This expression can be manipulated as follows

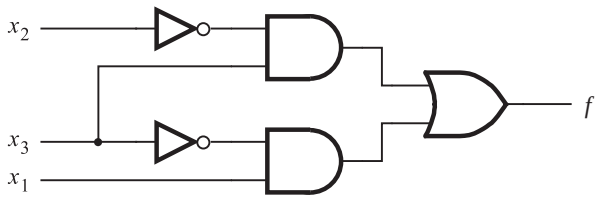
$$\begin{aligned} f &= (\bar{x}_1 + x_1)\bar{x}_2x_3 + x_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= 1 \cdot \bar{x}_2x_3 + x_1 \cdot 1 \cdot \bar{x}_3 \\ &= \bar{x}_2x_3 + x_1\bar{x}_3 \end{aligned}$$

This is the minimum-cost sum-of-products expression for $f$. It describes the circuit shown in Figure 2.19*a*. A good indication of the *cost* of a logic circuit is the total number of gates plus the total number of inputs to all gates in the circuit. Using this measure, the cost of the network in Figure 2.19*a* is 13, because there are five gates and eight inputs to the gates. By comparison, the network implemented on the basis of the canonical sum-of-products would have a cost of 27; from the preceding expression, the OR gate has four inputs, each of the four AND gates has three inputs, and each of the three NOT gates has one input.

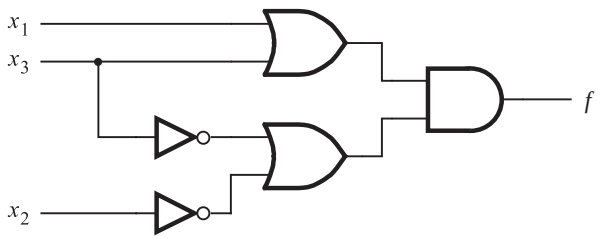
Minterms, with their row-number subscripts, can also be used to specify a given function in a more concise form. For example, the function in Figure 2.18 can be specified

| Row number | $x_1$ | $x_2$ | $x_3$ | $f(x_1, x_2, x_3)$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 1 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 0 |

**Figure 2.18** A three-variable function.




(a) A minimal sum-of-products realization




(b) A minimal product-of-sums realization

**Figure 2.19** Two realizations of the function in Figure 2.18.

as

$$f(x_1, x_2, x_3) = \sum (m_1, m_4, m_5, m_6)$$

or even more simply as

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

The $\sum$ sign denotes the logical sum operation. This shorthand notation is often used in practice.

### Maxterms

The principle of duality suggests that if it is possible to synthesize a function $f$ by considering the rows in the truth table for which $f = 1$, then it should also be possible to synthesize $f$ by considering the rows for which $f = 0$. This alternative approach uses the complements of minterms, which are called *maxterms*. All possible maxterms for three-variable functions are listed in Figure 2.17. We will refer to a maxterm $M_j$ by the same row number as its corresponding minterm $m_j$ as shown in the figure.

### Product-of-Sums Form

If a given function $f$ is specified by a truth table, then its complement $\overline{f}$ can be represented by a sum of minterms for which $\overline{f} = 1$, which are the rows where $f = 0$. For

example, for the function in Figure 2.15

$$\begin{aligned}\overline{f}(x_1, x_2) &= m_2 \\ &= x_1\overline{x}_2\end{aligned}$$

If we complement this expression using DeMorgan's theorem, the result is

$$\begin{aligned}\overline{\overline{f}} = f &= \overline{x_1\overline{x}_2} \\ &= \overline{x}_1 + x_2\end{aligned}$$

Note that we obtained this expression previously by algebraic manipulation of the canonical sum-of-products form for the function $f$. The key point here is that

$$f = \overline{m}_2 = M_2$$

where $M_2$ is the maxterm for row 2 in the truth table.

As another example, consider again the function in Figure 2.18. The complement of this function can be represented as

$$\begin{aligned}\overline{f}(x_1, x_2, x_3) &= m_0 + m_2 + m_3 + m_7 \\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1x_2x_3\end{aligned}$$

Then $f$ can be expressed as

$$\begin{aligned}f &= \overline{m_0 + m_2 + m_3 + m_7} \\ &= \overline{m}_0 \cdot \overline{m}_2 \cdot \overline{m}_3 \cdot \overline{m}_7 \\ &= M_0 \cdot M_2 \cdot M_3 \cdot M_7 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)\end{aligned}$$

This expression represents $f$ as a product of maxterms.

A logic expression consisting of sum (OR) terms that are the factors of a logical product (AND) is said to be of the *product-of-sums* (*POS*) form. If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function. Any function $f$ can be synthesized by finding its canonical product-of-sums. This involves taking the maxterm for each row in the truth table for which $f = 0$ and forming a product of these maxterms.

Returning to the preceding example, we can attempt to reduce the complexity of the derived expression that comprises a product of maxterms. Using the commutative property 10*b* and the associative property 11*b* from section 2.5, this expression can be written as

$$f = ((x_1 + x_3) + x_2)((x_1 + x_3) + \overline{x}_2)(x_1 + (\overline{x}_2 + \overline{x}_3))(\overline{x}_1 + (\overline{x}_2 + \overline{x}_3))$$

Then, using the combining property 14*b*, the expression reduces to

$$f = (x_1 + x_3)(\overline{x}_2 + \overline{x}_3)$$

The corresponding network is given in Figure 2.19*b*. The cost of this network is 13. While this cost happens to be the same as the cost of the sum-of-products version in Figure 2.19*a*, the reader should not assume that the cost of a network derived in the sum-of-products form

will in general be equal to the cost of a corresponding circuit derived in the product-of-sums form.

Using the shorthand notation, an alternative way of specifying our sample function is

$$f(x_1, x_2, x_3) = \Pi(M_0, M_2, M_3, M_7)$$

or more simply

$$f(x_1, x_2, x_3) = \Pi M(0, 2, 3, 7)$$

The $\Pi$ sign denotes the logical product operation.

The preceding discussion has shown how logic functions can be realized in the form of logic circuits, consisting of networks of gates that implement basic functions. A given function may be realized with circuits of a different structure, which usually implies a difference in cost. An important objective for a designer is to minimize the cost of the designed circuit. We will discuss the most important techniques for finding minimum-cost implementations in Chapter 4.

---

**Example 2.3** Consider the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

The canonical SOP expression for the function is derived using minterms

$$\begin{aligned} f &= m_2 + m_3 + m_4 + m_6 + m_7 \\ &= \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 \bar{x}_3 + x_1 x_2 x_3 \end{aligned}$$

This expression can be simplified using the identities in section 2.5 as follows

$$\begin{aligned} f &= \bar{x}_1 x_2 (\bar{x}_3 + x_3) + x_1 (\bar{x}_2 + x_2) \bar{x}_3 + x_1 x_2 (\bar{x}_3 + x_3) \\ &= \bar{x}_1 x_2 + x_1 \bar{x}_3 + x_1 x_2 \\ &= (\bar{x}_1 + x_1) x_2 + x_1 \bar{x}_3 \\ &= x_2 + x_1 \bar{x}_3 \end{aligned}$$

---

**Example 2.4** Consider again the function in Example 2.3. Instead of using the minterms, we can specify this function as a product of maxterms for which $f = 0$, namely

$$f(x_1, x_2, x_3) = \Pi M(0, 1, 5)$$

Then, the canonical POS expression is derived as

$$\begin{aligned} f &= M_0 \cdot M_1 \cdot M_5 \\ &= (x_1 + x_2 + x_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3) \end{aligned}$$

A simplified POS expression can be derived as

$$\begin{aligned} f &= ((x_1 + x_2) + x_3)((x_1 + x_2) + \bar{x}_3)(x_1 + (x_2 + \bar{x}_3))(\bar{x}_1 + (x_2 + \bar{x}_3)) \\ &= ((x_1 + x_2) + x_3\bar{x}_3)(x_1\bar{x}_1 + (x_2 + \bar{x}_3)) \\ &= (x_1 + x_2)(x_2 + \bar{x}_3) \end{aligned}$$

Note that by using the distributive property 12*b*, this expression leads to

$$f = x_2 + x_1\bar{x}_3$$

which is the same as the expression derived in Example 2.3.

---

**Example 2.5**

Suppose that a four-variable function is defined by

$$f(x_1, x_2, x_3, x_4) = \sum m(3, 7, 9, 12, 13, 14, 15)$$

The canonical SOP expression for this function is

$$f = \bar{x}_1\bar{x}_2x_3x_4 + \bar{x}_1x_2x_3x_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1x_2\bar{x}_3\bar{x}_4 + x_1x_2\bar{x}_3x_4 + x_1x_2x_3\bar{x}_4 + x_1x_2x_3x_4$$

A simpler SOP expression can be obtained as follows

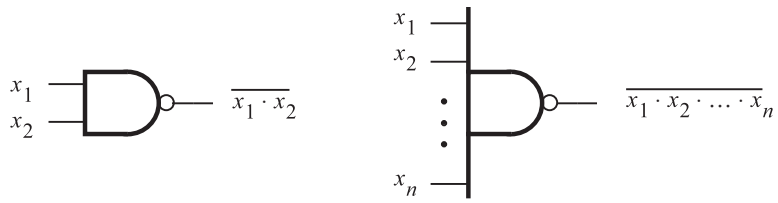
$$\begin{aligned} f &= \bar{x}_1(\bar{x}_2 + x_2)x_3x_4 + x_1(\bar{x}_2 + x_2)\bar{x}_3x_4 + x_1x_2\bar{x}_3(\bar{x}_4 + x_4) + x_1x_2x_3(\bar{x}_4 + x_4) \\ &= \bar{x}_1x_3x_4 + x_1\bar{x}_3x_4 + x_1x_2\bar{x}_3 + x_1x_2x_3 \\ &= \bar{x}_1x_3x_4 + x_1\bar{x}_3x_4 + x_1x_2(\bar{x}_3 + x_3) \\ &= \bar{x}_1x_3x_4 + x_1\bar{x}_3x_4 + x_1x_2 \end{aligned}$$

---

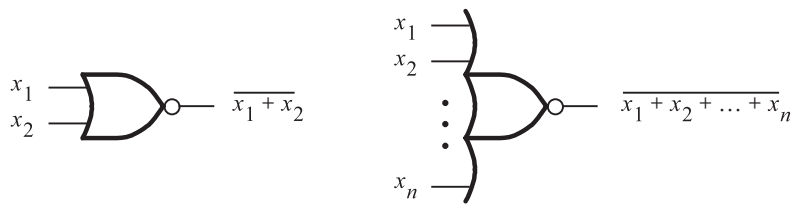
## 2.7 NAND AND NOR LOGIC NETWORKS

We have discussed the use of AND, OR, and NOT gates in the synthesis of logic circuits. There are other basic logic functions that are also used for this purpose. Particularly useful are the NAND and NOR functions which are obtained by complementing the output generated by AND and OR operations, respectively. These functions are attractive because they are implemented with simpler electronic circuits than the AND and OR functions, as we will see in Chapter 3. Figure 2.20 gives the graphical symbols for the NAND and NOR gates. A bubble is placed on the output side of the AND and OR gate symbols to represent the complemented output signal.

If NAND and NOR gates are realized with simpler circuits than AND and OR gates, then we should ask whether these gates can be used directly in the synthesis of logic circuits. In section 2.5 we introduced DeMorgan's theorem. Its logic gate interpretation is shown in Figure 2.21. Identity 15*a* is interpreted in part (*a*) of the figure. It specifies that a NAND of variables $x_1$ and $x_2$ is equivalent to first complementing each of the variables and then ORing them. Notice on the far-right side that we have indicated the NOT gates

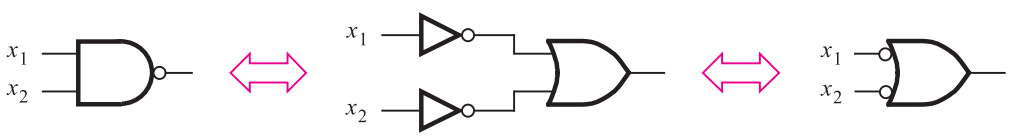



(a) NAND gates

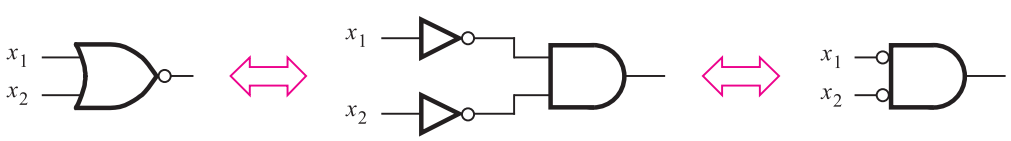



(b) NOR gates

**Figure 2.20** NAND and NOR gates.




(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$




(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

**Figure 2.21** DeMorgan's theorem in terms of logic gates.

simply as bubbles, which denote inversion of the logic value at that point. The other half of DeMorgan's theorem, identity 15*b*, appears in part (*b*) of the figure. It states that the NOR function is equivalent to first inverting the input variables and then ANDing them.

In section 2.6 we explained how any logic function can be implemented either in sum-of-products or product-of-sums form, which leads to logic networks that have either an AND-OR or an OR-AND structure, respectively. We will now show that such networks can be implemented using only NAND gates or only NOR gates.

Consider the network in Figure 2.22 as a representative of general AND-OR networks. This network can be transformed into a network of NAND gates as shown in the figure. First, each connection between the AND gate and an OR gate is replaced by a connection that includes two inversions of the signal: one inversion at the output of the AND gate and the other at the input of the OR gate. Such double inversion has no effect on the behavior of the network, as stated formally in theorem 9 in section 2.5. According to Figure 2.21*a*, the OR gate with inversions at its inputs is equivalent to a NAND gate. Thus we can redraw the network using only NAND gates, as shown in Figure 2.22. This example shows that any AND-OR network can be implemented as a NAND-NAND network having the same topology.

Figure 2.23 gives a similar construction for a product-of-sums network, which can be transformed into a circuit with only NOR gates. The procedure is exactly the same as the one described for Figure 2.22 except that now the identity in Figure 2.21*b* is applied. The conclusion is that any OR-AND network can be implemented as a NOR-NOR network having the same topology.

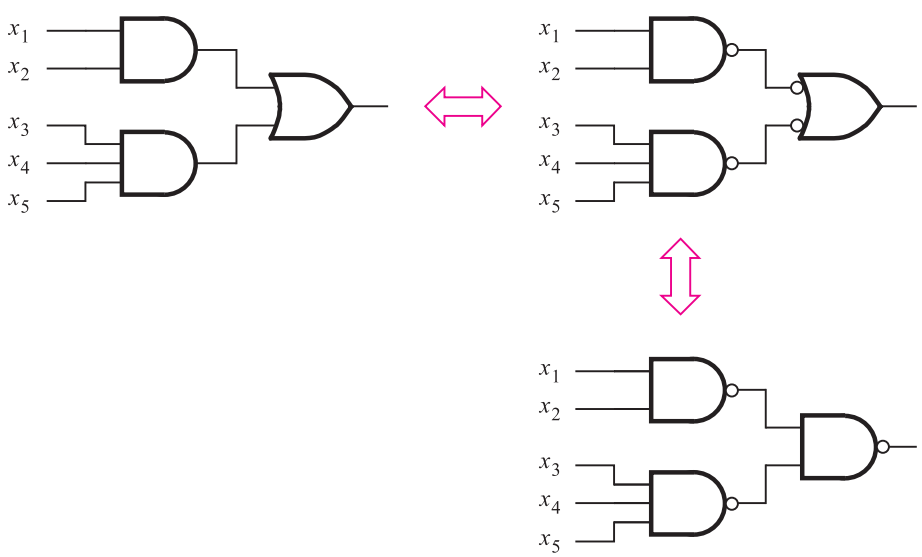



**Figure 2.22** Using NAND gates to implement a sum-of-products.

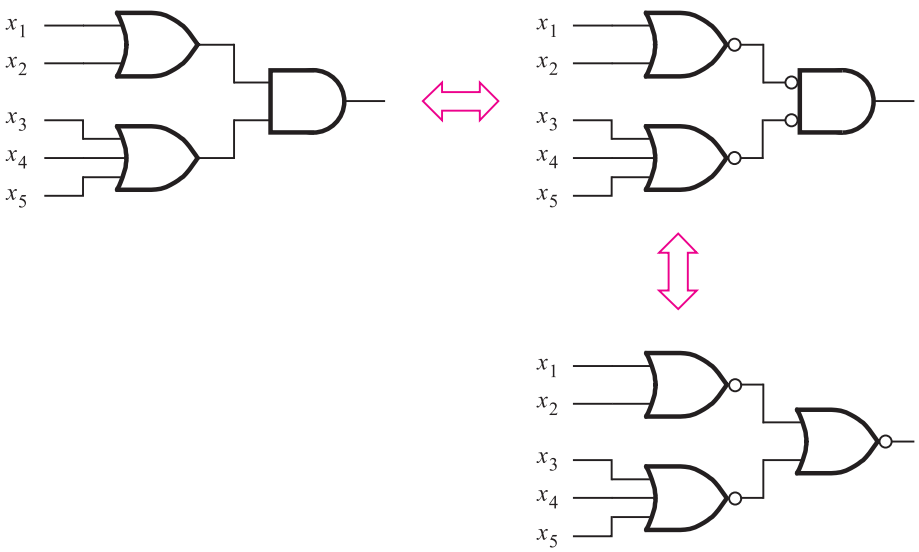



**Figure 2.23** Using NOR gates to implement a product-of-sums.

---

**Example 2.6** Let us implement the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

using NOR gates only. In Example 2.4 we showed that the function can be represented by the POS expression

$$f = (x_1 + x_2)(x_2 + \bar{x}_3)$$

An OR-AND circuit that corresponds to this expression is shown in Figure 2.24*a*. Using the same structure of the circuit, a NOR-gate version is given in Figure 2.24*b*. Note that $x_3$ is inverted by a NOR gate that has its inputs tied together.

---

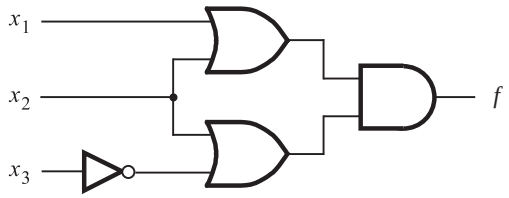
**Example 2.7** Let us now implement the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

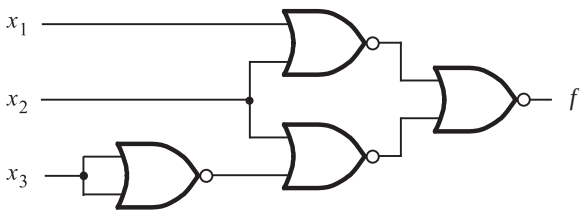
using NAND gates only. In Example 2.3 we derived the SOP expression

$$f = x_2 + x_1\bar{x}_3$$

which is realized using the circuit in Figure 2.25*a*. We can again use the same structure to obtain a circuit with NAND gates, but with one difference. The signal $x_2$ passes only through an OR gate, instead of passing through an AND gate and an OR gate. If we simply replace the OR gate with a NAND gate, this signal would be inverted which would result in a wrong output value. Since $x_2$ must either not be inverted, or it can be inverted twice,

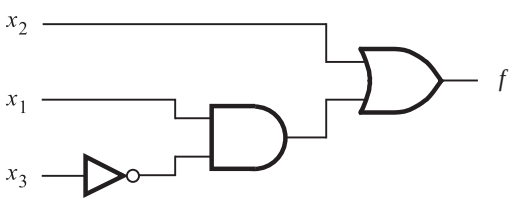



(a) POS implementation

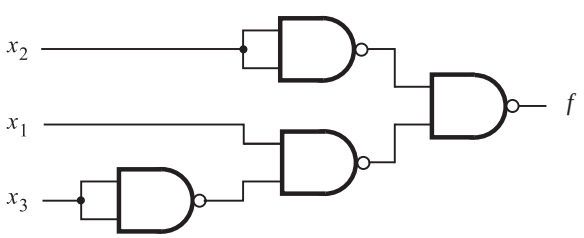



(b) NOR implementation

**Figure 2.24** NOR-gate realization of the function in Example 2.4.




(a) SOP implementation




(b) NAND implementation

**Figure 2.25** NAND-gate realization of the function in Example 2.3.

we can pass it through two NAND gates as depicted in Figure 2.25*b*. Observe that for this circuit the output $f$ is

$$f = \overline{\bar{x}_2 \cdot \overline{x_1 \bar{x}_3}}$$

Applying DeMorgan's theorem, this expression becomes

$$f = x_2 + x_1 \bar{x}_3$$

## 2.8 DESIGN EXAMPLES

Logic circuits provide a solution to a problem. They implement functions that are needed to carry out specific tasks. Within the framework of a computer, logic circuits provide complete capability for execution of programs and processing of data. Such circuits are complex and difficult to design. But regardless of the complexity of a given circuit, a designer of logic circuits is always confronted with the same basic issues. First, it is necessary to specify the desired behavior of the circuit. Second, the circuit has to be synthesized and implemented. Finally, the implemented circuit has to be tested to verify that it meets the specifications. The desired behavior is often initially described in words, which then must be turned into a formal specification. In this section we give two simple examples of design.

### 2.8.1 THREE-WAY LIGHT CONTROL

Assume that a large room has three doors and that a switch near each door controls a light in the room. It has to be possible to turn the light on or off by changing the state of any one of the switches.

As a first step, let us turn this word statement into a formal specification using a truth table. Let $x_1$, $x_2$, and $x_3$ be the input variables that denote the state of each switch. Assume that the light is off if all switches are open. Closing any one of the switches will turn the light on. Then turning on a second switch will have to turn off the light. Thus the light will be on if exactly one switch is closed, and it will be off if two (or no) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch. If $f(x_1, x_2, x_3)$ represents the state of the light, then the required functional behavior can be specified as shown in the truth table in Figure 2.26. The canonical sum-of-products expression for the specified function is

$$\begin{aligned} f &= m_1 + m_2 + m_4 + m_7 \\ &= \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_3 \end{aligned}$$

This expression cannot be simplified into a lower-cost sum-of-products expression. The resulting circuit is shown in Figure 2.27*a*.

In Chapter 2 we showed that algebraic manipulation can be used to find the lowest-cost implementations of logic functions. The purpose of that chapter was to introduce the basic concepts in the synthesis process. The reader is probably convinced that it is easy to derive a straightforward realization of a logic function in a canonical form, but it is not at all obvious how to choose and apply the theorems and properties of section 2.5 to find a minimum-cost circuit. Indeed, the algebraic manipulation is rather tedious and quite impractical for functions of many variables.

If CAD tools are used to design logic circuits, the task of minimizing the cost of implementation does not fall to the designer; the tools perform the necessary optimizations automatically. Even so, it is essential to know something about this process. Most CAD tools have many features and options that are under control of the user. To know when and how to apply these options, the user must have an understanding of what the tools do.

In this chapter we will introduce some of the optimization techniques implemented in CAD tools and show how these techniques can be automated. As a first step we will discuss a graphical approach, known as the Karnaugh map, which provides a neat way to manually derive minimum-cost implementations of simple logic functions. Although it is not suitable for implementation in CAD tools, it illustrates a number of key concepts. We will show how both two-level and multilevel circuits can be designed. Then we will describe a cubical representation for logic functions, which is suitable for use in CAD tools. We will also continue our discussion of the VHDL language.

## 4.1 KARNAUGH MAP

In section 2.6 we saw that the key to finding a minimum-cost expression for a given logic function is to reduce the number of product (or sum) terms needed in the expression, by applying the combining property 14*a* (or 14*b*) as judiciously as possible. The Karnaugh map approach provides a systematic way of performing this optimization. To understand how it works, it is useful to review the algebraic approach from Chapter 2. Consider the function $f$ in Figure 4.1. The canonical sum-of-products expression for $f$ consists of minterms $m_0$, $m_2$, $m_4$, $m_5$, and $m_6$, so that

$$f = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

The combining property 14*a* allows us to replace two minterms that differ in the value of only one variable with a single product term that does not include that variable at all. For example, both $m_0$ and $m_2$ include $\bar{x}_1$ and $\bar{x}_3$, but they differ in the value of $x_2$ because $m_0$ includes $\bar{x}_2$ while $m_2$ includes $x_2$. Thus

$$\begin{aligned}\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 &= \bar{x}_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= \bar{x}_1 \cdot 1 \cdot \bar{x}_3 \\ &= \bar{x}_1\bar{x}_3\end{aligned}$$

| Row number | $x_1$ | $x_2$ | $x_3$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 1 |
| 6 | 1 | 1 | 0 | 1 |
| 7 | 1 | 1 | 1 | 0 |

**Figure 4.1** The function $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

Hence $m_0$ and $m_2$ can be replaced by the single product term $\bar{x}_1\bar{x}_3$. Similarly, $m_4$ and $m_6$ differ only in the value of $x_2$ and can be combined using

$$\begin{aligned} x_1\bar{x}_2\bar{x}_3 + x_1x_2\bar{x}_3 &= x_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= x_1 \cdot 1 \cdot \bar{x}_3 \\ &= x_1\bar{x}_3 \end{aligned}$$

Now the two newly generated terms, $\bar{x}_1\bar{x}_3$ and $x_1\bar{x}_3$, can be combined further as

$$\begin{aligned} \bar{x}_1\bar{x}_3 + x_1\bar{x}_3 &= (\bar{x}_1 + x_1)\bar{x}_3 \\ &= 1 \cdot \bar{x}_3 \\ &= \bar{x}_3 \end{aligned}$$

These optimization steps indicate that we can replace the four minterms $m_0$, $m_2$, $m_4$, and $m_6$ with the single product term $\bar{x}_3$. In other words, the minterms $m_0$, $m_2$, $m_4$, and $m_6$ are all *included* in the term $\bar{x}_3$. The remaining minterm in $f$ is $m_5$. It can be combined with $m_4$, which gives

$$x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 = x_1\bar{x}_2$$

Recall that theorem 7*b* in section 2.5 indicates that

$$m_4 = m_4 + m_4$$

which means that we can use the minterm $m_4$ twice—to combine with minterms $m_0$, $m_2$, and $m_6$ to yield the term $\bar{x}_3$ as explained above and also to combine with $m_5$ to yield the term $x_1\bar{x}_2$.

We have now accounted for all the minterms in $f$; hence all five input valuations for which $f = 1$ are covered by the minimum-cost expression

$$f = \bar{x}_3 + x_1\bar{x}_2$$

The expression has the product term $\bar{x}_3$ because $f = 1$ when $x_3 = 0$ regardless of the values of $x_1$ and $x_2$. The four minterms $m_0$, $m_2$, $m_4$, and $m_6$ represent all possible minterms for which $x_3 = 0$; they include all four valuations, 00, 01, 10, and 11, of variables $x_1$ and $x_2$. Thus if $x_3 = 0$, then it is guaranteed that $f = 1$. This may not be easy to see directly from the truth table in Figure 4.1, but it is obvious if we write the corresponding valuations grouped together:

| | $x_1$ | $x_2$ | $x_3$ |
|---|---|---|---|
| $m_0$ | 0 | 0 | 0 |
| $m_2$ | 0 | 1 | 0 |
| $m_4$ | 1 | 0 | 0 |
| $m_6$ | 1 | 1 | 0 |

In a similar way, if we look at $m_4$ and $m_5$ as a group of two

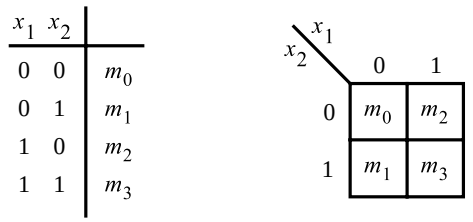
| | $x_1$ | $x_2$ | $x_3$ |
|---|---|---|---|
| $m_4$ | 1 | 0 | 0 |
| $m_5$ | 1 | 0 | 1 |

it is clear that when $x_1 = 1$ and $x_2 = 0$, then $f = 1$ regardless of the value of $x_3$.

The preceding discussion suggests that it would be advantageous to devise a method that allows easy discovery of groups of minterms for which $f = 1$ that can be combined into single terms. The Karnaugh map is a useful vehicle for this purpose.

The *Karnaugh map* [1] is an alternative to the truth-table form for representing a function. The map consists of *cells* that correspond to the rows of the truth table. Consider the two-variable example in Figure 4.2. Part (*a*) depicts the truth-table form, where each of the four rows is identified by a minterm. Part (*b*) shows the Karnaugh map, which has four cells. The columns of the map are labeled by the value of $x_1$, and the rows are labeled by $x_2$. This labeling leads to the locations of minterms as shown in the figure. Compared to the truth table, the advantage of the Karnaugh map is that it allows easy recognition of minterms that can be combined using property 14*a* from section 2.5. Minterms in any two cells that are adjacent, either in the same row or the same column, can be combined. For example, the minterms $m_2$ and $m_3$ can be combined as

$$\begin{aligned} m_2 + m_3 &= x_1\bar{x}_2 + x_1x_2 \\ &= x_1(\bar{x}_2 + x_2) \\ &= x_1 \cdot 1 \\ &= x_1 \end{aligned}$$




(a) Truth table (b) Karnaugh map

**Figure 4.2** Location of two-variable minterms.

The Karnaugh map is not just useful for combining pairs of minterms. As we will see in several larger examples, the Karnaugh map can be used directly to derive a minimum-cost circuit for a logic function.

### Two-Variable Map

A Karnaugh map for a two-variable function is given in Figure 4.3. It corresponds to the function $f$ of Figure 2.15. The value of $f$ for each valuation of the variables $x_1$ and $x_2$ is indicated in the corresponding cell of the map. Because a 1 appears in both cells of the bottom row and these cells are adjacent, there exists a single product term that can cause $f$ to be equal to 1 when the input variables have the values that correspond to either of these cells. To indicate this fact, we have circled the cell entries in the map. Rather than using the combining property formally, we can derive the product term intuitively. Both of the cells are identified by $x_2 = 1$, but $x_1 = 0$ for the left cell and $x_1 = 1$ for the right cell. Thus if $x_2 = 1$, then $f = 1$ regardless of whether $x_1$ is equal to 0 or 1. The product term representing the two cells is simply $x_2$.

Similarly, $f = 1$ for both cells in the first column. These cells are identified by $x_1 = 0$. Therefore, they lead to the product term $\bar{x}_1$. Since this takes care of all instances where $f = 1$, it follows that the minimum-cost realization of the function is

$$f = x_2 + \bar{x}_1$$

Evidently, to find a minimum-cost implementation of a given function, it is necessary to find the smallest number of product terms that produce a value of 1 for all cases where

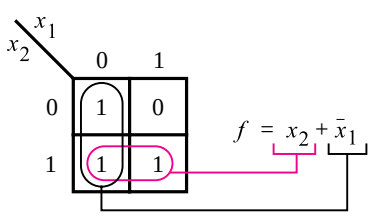



**Figure 4.3** The function of Figure 2.15.

$f = 1$. Moreover, the cost of these product terms should be as low as possible. Note that a product term that covers two adjacent cells is cheaper to implement than a term that covers only a single cell. For our example once the two cells in the bottom row have been covered by the product term $x_2$, only one cell (top left) remains. Although it could be covered by the term $\bar{x}_1\bar{x}_2$, it is better to combine the two cells in the left column to produce the product term $\bar{x}_1$ because this term is cheaper to implement.

### Three-Variable Map

A three-variable Karnaugh map is constructed by placing 2 two-variable maps side by side. Figure 4.4 shows the map and indicates the locations of minterms in it. In this case each valuation of $x_1$ and $x_2$ identifies a column in the map, while the value of $x_3$ distinguishes the two rows. To ensure that minterms in the adjacent cells in the map can always be combined into a single product term, the adjacent cells must differ in the value of only one variable. Thus the columns are identified by the sequence of $(x_1, x_2)$ values of 00, 01, 11, and 10, rather than the more obvious 00, 01, 10, and 11. This makes the second and third columns different only in variable $x_1$. Also, the first and the fourth columns differ only in variable $x_1$, which means that these columns can be considered as being adjacent. The reader may find it useful to visualize the map as a rectangle folded into a cylinder where the left and the right edges in Figure 4.4*b* are made to touch. (A sequence of codes, or valuations, where consecutive codes differ in one variable only is known as the *Gray code*. This code is used for a variety of purposes, some of which will be encountered later in the book.)

Figure 4.5*a* represents the function of Figure 2.18 in Karnaugh-map form. To synthesize this function, it is necessary to cover the four 1s in the map as efficiently as possible. It is not difficult to see that two product terms suffice. The first covers the 1s in the top row, which are represented by the term $x_1\bar{x}_3$. The second term is $\bar{x}_2x_3$, which covers the 1s in the bottom row. Hence the function is implemented as

$$f = x_1\bar{x}_3 + \bar{x}_2x_3$$

which describes the circuit obtained in Figure 2.19*a*.

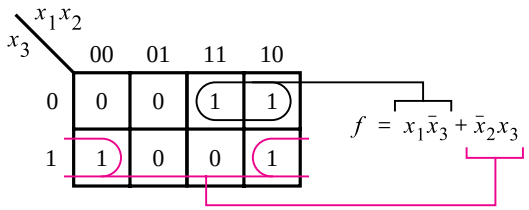
| $x_1$ | $x_2$ | $x_3$ | |
|---|---|---|---|
| 0 | 0 | 0 | $m_0$ |
| 0 | 0 | 1 | $m_1$ |
| 0 | 1 | 0 | $m_2$ |
| 0 | 1 | 1 | $m_3$ |
| 1 | 0 | 0 | $m_4$ |
| 1 | 0 | 1 | $m_5$ |
| 1 | 1 | 0 | $m_6$ |
| 1 | 1 | 1 | $m_7$ |

(a) Truth table

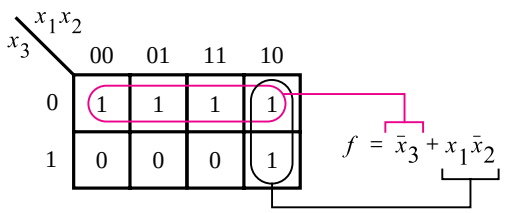
| $x_3$ \ $x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | $m_0$ | $m_2$ | $m_6$ | $m_4$ |
| 1 | $m_1$ | $m_3$ | $m_7$ | $m_5$ |

(b) Karnaugh map

**Figure 4.4** Location of three-variable minterms.




(a) The function of Figure 2.18




(b) The function of Figure 4.1

**Figure 4.5** Examples of three-variable Karnaugh maps.

In a three-variable map it is possible to combine cells to produce product terms that correspond to a single cell, two adjacent cells, or a group of four adjacent cells. Realization of a group of four adjacent cells using a single product term is illustrated in Figure 4.5*b*, using the function from Figure 4.1. The four cells in the top row correspond to the $(x_1, x_2, x_3)$ valuations 000, 010, 110, and 100. As we discussed before, this indicates that if $x_3 = 0$, then $f = 1$ for all four possible valuations of $x_1$ and $x_2$, which means that the only requirement is that $x_3 = 0$. Therefore, the product term $\bar{x}_3$ represents these four cells. The remaining 1, corresponding to minterm $m_5$, is best covered by the term $x_1\bar{x}_2$, obtained by combining the two cells in the right-most column. The complete realization of $f$ is

$$f = \bar{x}_3 + x_1\bar{x}_2$$

It is also possible to have a group of eight 1s in a three-variable map. This is the trivial case where $f = 1$ for all valuations of input variables; in other words, $f$ is equal to the constant 1.

The Karnaugh map provides a simple mechanism for generating the product terms that should be used to implement a given function. A product term must include only those variables that have the same value for all cells in the group represented by this term. If the variable is equal to 1 in the group, it appears uncomplemented in the product term; if it is equal to 0, it appears complemented. Each variable that is sometimes 1 and sometimes 0 in the group does not appear in the product term.

### Four-Variable Map

A four-variable map is constructed by placing 2 three-variable maps together to create four rows in the same fashion as we used 2 two-variable maps to form the four columns in a three-variable map. Figure 4.6 shows the structure of the four-variable map and the location

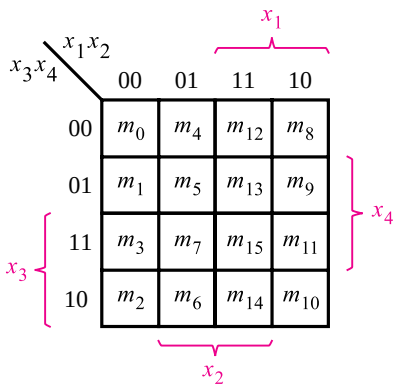



**Figure 4.6** A four-variable Karnaugh map.

of minterms. We have included in this figure another frequently used way of designating the rows and columns. As shown in blue, it is sufficient to indicate the rows and columns for which a given variable is equal to 1. Thus $x_1 = 1$ for the two right-most columns, $x_2 = 1$ for the two middle columns, $x_3 = 1$ for the bottom two rows, and $x_4 = 1$ for the two middle rows.

Figure 4.7 gives four examples of four-variable functions. The function $f_1$ has a group of four 1s in adjacent cells in the bottom two rows, for which $x_2 = 0$ and $x_3 = 1$—they are represented by the product term $\bar{x}_2x_3$. This leaves the two 1s in the second row to be covered, which can be accomplished with the term $x_1\bar{x}_3x_4$. Hence the minimum-cost implementation of the function is

$$f_1 = \bar{x}_2x_3 + x_1\bar{x}_3x_4$$

The function $f_2$ includes a group of eight 1s that can be implemented by a single term, $x_3$. Again, the reader should note that if the remaining two 1s were implemented separately, the result would be the product term $x_1\bar{x}_3x_4$. Implementing these 1s as a part of a group of four 1s, as shown in the figure, gives the less expensive product term $x_1x_4$.

Just as the left and the right edges of the map are adjacent in terms of the assignment of the variables, so are the top and the bottom edges. Indeed, the four corners of the map are adjacent to each other and thus can form a group of four 1s, which may be implemented by the product term $\bar{x}_2\bar{x}_4$. This case is depicted by the function $f_3$. In addition to this group of 1s, there are four other 1s that must be covered to implement $f_3$. This can be done as shown in the figure.

In all examples that we have considered so far, a unique solution exists that leads to a minimum-cost circuit. The function $f_4$ provides an example where there is some choice. The groups of four 1s in the top-left and bottom-right corners of the map are realized by the terms $\bar{x}_1\bar{x}_3$ and $x_1x_3$, respectively. This leaves the two 1s that correspond to the term $x_1x_2\bar{x}_3$. But these two 1s can be realized more economically by treating them as a part of a group of four 1s. They can be included in two different groups of four, as shown in the figure.

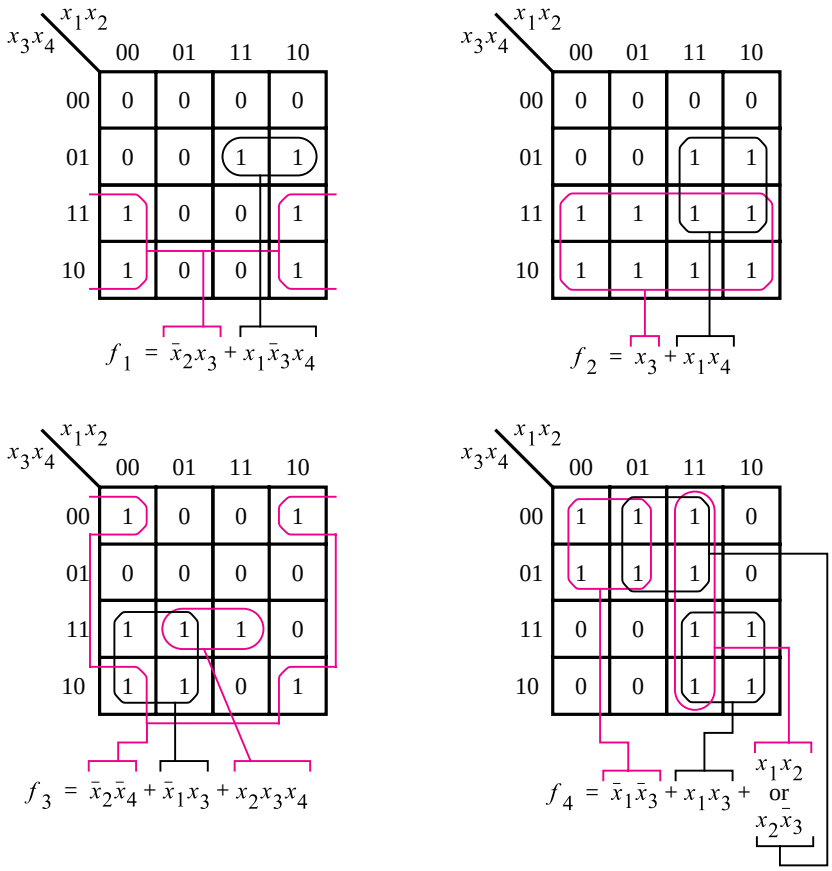



**Figure 4.7** Examples of four-variable Karnaugh maps.

One choice leads to the product term $x_1x_2$, and the other leads to $x_2\bar{x}_3$. Both of these terms have the same cost; hence it does not matter which one is chosen in the final circuit. Note that the complement of $x_3$ in the term $x_2\bar{x}_3$ does not imply an increased cost in comparison with $x_1x_2$, because this complement must be generated anyway to produce the term $\bar{x}_1\bar{x}_3$, which is included in the implementation.

### Five-Variable Map

We can use 2 four-variable maps to construct a five-variable map. It is easy to imagine a structure where one map is directly behind the other, and they are distinguished by $x_5 = 0$ for one map and $x_5 = 1$ for the other map. Since such a structure is awkward to draw, we can simply place the two maps side by side as shown in Figure 4.8. For the logic function given in this example, two groups of four 1s appear in the same place in both four-variable maps; hence their realization does not depend on the value of $x_5$. The same is true for the two groups of two 1s in the second row. The 1 in the top-right corner appears only in the

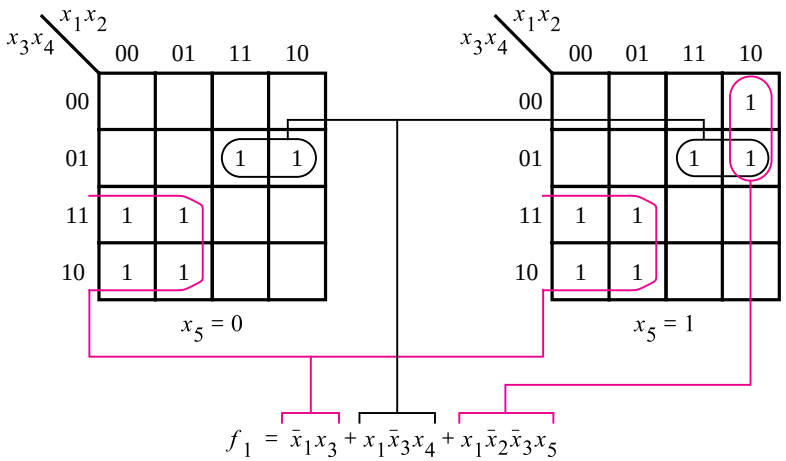



**Figure 4.8** A five-variable Karnaugh map.

right map, where $x_5 = 1$; it is a part of the group of two 1s realized by the term $x_1\bar{x}_2\bar{x}_3x_5$. Note that in this map we left blank those cells for which $f = 0$, to make the figure more readable. We will do likewise in a number of maps that follow.

Using a five-variable map is obviously more awkward than using maps with fewer variables. Extending the Karnaugh map concept to more variables is not useful from the practical point of view. This is not troublesome, because practical synthesis of logic functions is done with CAD tools that perform the necessary minimization automatically. Although Karnaugh maps are occasionally useful for designing small logic circuits, our main reason for introducing the Karnaugh maps is to provide a simple vehicle for illustrating the ideas involved in the minimization process.

## 4.2 STRATEGY FOR MINIMIZATION

For the examples in the preceding section, we used an intuitive approach to decide how the 1s in a Karnaugh map should be grouped together to obtain the minimum-cost implementation of a given function. Our intuitive strategy was to find as few as possible and as large as possible groups of 1s that cover all cases where the function has a value of 1. Each group of 1s has to comprise cells that can be represented by a single product term. The larger the group of 1s, the fewer the number of variables in the corresponding product term. This approach worked well because the Karnaugh maps in our examples were small. For larger logic functions, which have many variables, such intuitive approach is unsuitable. Instead, we must have an organized method for deriving a minimum-cost implementation. In this section we will introduce a possible method, which is similar to the techniques that are

automated in CAD tools. To illustrate the main ideas, we will use Karnaugh maps. Later, in section 4.8, we will describe a different way of representing logic functions, which is used in CAD tools.

## 4.2.1 TERMINOLOGY

A huge amount of research work has gone into the development of techniques for synthesis of logic functions. The results of this research have been published in numerous papers. To facilitate the presentation of the results, certain terminology has evolved that avoids the need for using highly descriptive phrases. We define some of this terminology in the following paragraphs because it is useful for describing the minimization process.

### Literal

A given product term consists of some number of variables, each of which may appear either in uncomplemented or complemented form. Each appearance of a variable, either uncomplemented or complemented, is called a *literal*. For example, the product term $x_1\bar{x}_2x_3$ has three literals, and the term $\bar{x}_1x_3\bar{x}_4x_6$ has four literals.

### Implicant

A product term that indicates the input valuation(s) for which a given function is equal to 1 is called an *implicant* of the function. The most basic implicants are the minterms, which we introduced in section 2.6.1. For an $n$-variable function, a minterm is an implicant that consists of $n$ literals.

Consider the three-variable function in Figure 4.9. There are 11 possible implicants for this function. This includes the five minterms: $\bar{x}_1\bar{x}_2\bar{x}_3$, $\bar{x}_1\bar{x}_2x_3$, $\bar{x}_1x_2\bar{x}_3$, $\bar{x}_1x_2x_3$, and $x_1x_2x_3$. Then there are the implicants that correspond to all possible pairs of minterms that can be combined, namely, $\bar{x}_1\bar{x}_2$ ($m_0$ and $m_1$), $\bar{x}_1\bar{x}_3$ ($m_0$ and $m_2$), $\bar{x}_1x_3$ ($m_1$ and $m_3$), $\bar{x}_1x_2$ ($m_2$ and $m_3$), and $x_2x_3$ ($m_3$ and $m_7$). Finally, there is one implicant that covers a group of four minterms, which consists of a single literal $\bar{x}_1$.

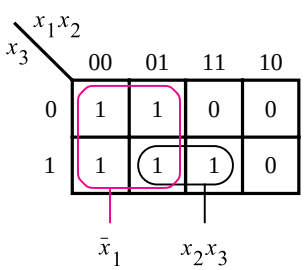



**Figure 4.9** Three-variable function $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$.

### Prime Implicant

An implicant is called a *prime implicant* if it cannot be combined into another implicant that has fewer literals. Another way of stating this definition is to say that it is impossible to delete any literal in a prime implicant and still have a valid implicant.

In Figure 4.9 there are two prime implicants: $\bar{x}_1$ and $x_2x_3$. It is not possible to delete a literal in either of them. Doing so for $\bar{x}_1$ would make it disappear. For $x_2x_3$, deleting a literal would leave either $x_2$ or $x_3$. But $x_2$ is not an implicant because it includes the valuation $(x_1, x_2, x_3) = 110$ for which $f = 0$, and $x_3$ is not an implicant because it includes $(x_1, x_2, x_3) = 101$ for which $f = 0$.

### Cover

A collection of implicants that account for all valuations for which a given function is equal to 1 is called a *cover* of that function. A number of different covers exist for most functions. Obviously, a set of all minterms for which $f = 1$ is a cover. It is also apparent that a set of all prime implicants is a cover.

A cover defines a particular implementation of the function. In Figure 4.9 a cover consisting of minterms leads to the expression

$$f = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_2x_3 + x_1x_2x_3$$

Another valid cover is given by the expression

$$f = \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 + x_2x_3$$

The cover comprising the prime implicants is

$$f = \bar{x}_1 + x_2x_3$$

While all of these expressions represent the function $f$ correctly, the cover consisting of prime implicants leads to the lowest-cost implementation.

### Cost

In Chapter 2 we suggested that a good indication of the cost of a logic circuit is the number of gates plus the total number of inputs to all gates in the circuit. We will use this definition of cost throughout the book. But we will assume that primary inputs, namely, the input variables, are available in both true and complemented forms at zero cost. Thus the expression

$$f = x_1\bar{x}_2 + x_3\bar{x}_4$$

has a cost of nine because it can be implemented using two AND gates and one OR gate, with six inputs to the AND and OR gates.

If an inversion is needed inside a circuit, then the corresponding NOT gate and its input are included in the cost. For example, the expression

$$g = \overline{x_1\bar{x}_2 + x_3}(\bar{x}_4 + x_5)$$

is implemented using two AND gates, two OR gates, and one NOT gate to complement $(x_1\bar{x}_2 + x_3)$, with nine inputs. Hence the total cost is 14.

### 4.2.2 MINIMIZATION PROCEDURE

We have seen that it is possible to implement a given logic function with various circuits. These circuits may have different structures and different costs. When designing a logic circuit, there are usually certain criteria that must be met. One such criterion is likely to be the cost of the circuit, which we considered in the previous discussion. In general, the larger the circuit, the more important the cost issue becomes. In this section we will assume that the main objective is to obtain a minimum-cost circuit.

Having said that cost is the primary concern, we should note that other optimization criteria may be more appropriate in some cases. For instance, in Chapter 3 we described several types of programmable-logic devices (PLDs) that have a predefined basic structure and can be programmed to realize a variety of different circuits. For such devices the main objective is to design a particular circuit so that it will fit into the target device. Whether or not this circuit has the minimum cost is not important if it can be realized successfully on the device. A CAD tool intended for design with a specific device in mind will automatically perform optimizations that are suitable for that device. We will show in section 4.6 that the way in which a circuit should be optimized may be different for different types of devices.

In the previous subsection we concluded that the lowest-cost implementation is achieved when the cover of a given function consists of prime implicants. The question then is how to determine the minimum-cost subset of prime implicants that will cover the function. Some prime implicants may have to be included in the cover, while for others there may be a choice. If a prime implicant includes a minterm for which $f = 1$ that is not included in any other prime implicant, then it must be included in the cover and is called an *essential prime implicant*. In the example in Figure 4.9, both prime implicants are essential. The term $x_2x_3$ is the only prime implicant that covers the minterm $m_7$, and $\bar{x}_1$ is the only one that covers the minterms $m_0$, $m_1$, and $m_2$. Notice that the minterm $m_3$ is covered by both of these prime implicants. The minimum-cost realization of the function is

$$f = \bar{x}_1 + x_2x_3$$

We will now present several examples in which there is a choice as to which prime implicants to include in the final cover. Consider the four-variable function in Figure 4.10. There are five prime implicants: $\bar{x}_1x_3$, $\bar{x}_2x_3$, $x_3\bar{x}_4$, $\bar{x}_1x_2x_4$, and $x_2\bar{x}_3x_4$. The essential ones (highlighted in blue) are $\bar{x}_2x_3$ (because of $m_{11}$), $x_3\bar{x}_4$ (because of $m_{14}$), and $x_2\bar{x}_3x_4$ (because of $m_{13}$). They must be included in the cover. These three prime implicants cover all minterms for which $f = 1$ except $m_7$. It is clear that $m_7$ can be covered by either $\bar{x}_1x_3$ or $\bar{x}_1x_2x_4$. Because $\bar{x}_1x_3$ has a lower cost, it is chosen for the cover. Therefore, the minimum-cost realization is

$$f = \bar{x}_2x_3 + x_3\bar{x}_4 + x_2\bar{x}_3x_4 + \bar{x}_1x_3$$

From the preceding discussion, the process of finding a minimum-cost circuit involves the following steps:

1. Generate all prime implicants for the given function $f$.
2. Find the set of essential prime implicants.

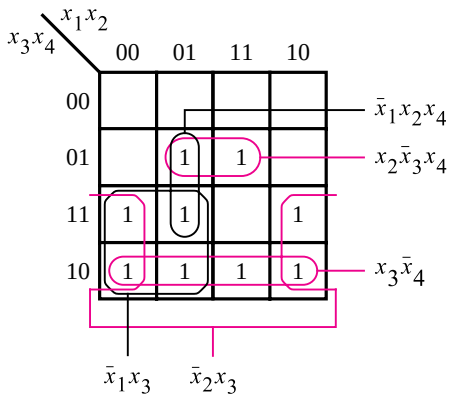



**Figure 4.10** Four-variable function $f(x_1, \ldots, x_4) = \sum m(2, 3, 5, 6, 7, 10, 11, 13, 14)$.

3. If the set of essential prime implicants covers all valuations for which $f = 1$, then this set is the desired cover of $f$. Otherwise, determine the nonessential prime implicants that should be added to form a complete minimum-cost cover.

The choice of nonessential prime implicants to be included in the cover is governed by the cost considerations. This choice is often not obvious. Indeed, for large functions there may exist many possibilities, and some *heuristic* approach (i.e., an approach that considers only a subset of possibilities but gives good results most of the time) has to be used. One such approach is to arbitrarily select one nonessential prime implicant and include it in the cover and then determine the rest of the cover. Next, another cover is determined assuming that this prime implicant is not in the cover. The costs of the resulting covers are compared, and the less-expensive cover is chosen for implementation.

We can illustrate the process by using the function in Figure 4.11. Of the six prime implicants, only $\bar{x}_3\bar{x}_4$ is essential. Consider next $x_1x_2\bar{x}_3$ and assume first that it will be

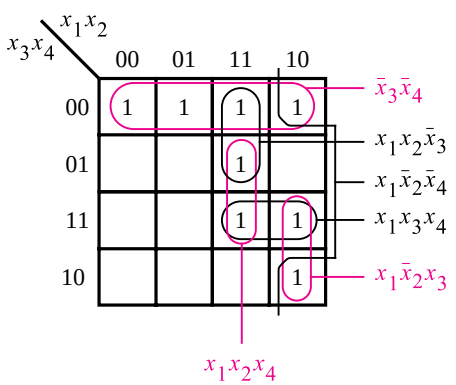



**Figure 4.11** The function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$.

included in the cover. Then the remaining three minterms, $m_{10}$, $m_{11}$, and $m_{15}$, will require two more prime implicants to be included in the cover. A possible implementation is

$$f = \bar{x}_3\bar{x}_4 + x_1x_2\bar{x}_3 + x_1x_3x_4 + x_1\bar{x}_2x_3$$

The second possibility is that $x_1x_2\bar{x}_3$ is not included in the cover. Then $x_1x_2x_4$ becomes essential because there is no other way of covering $m_{13}$. Because $x_1x_2x_4$ also covers $m_{15}$, only $m_{10}$ and $m_{11}$ remain to be covered, which can be achieved with $x_1\bar{x}_2x_3$. Therefore, the alternative implementation is

$$f = \bar{x}_3\bar{x}_4 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

Clearly, this implementation is a better choice.

Sometimes there may not be any essential prime implicants at all. An example is given in Figure 4.12. Choosing any of the prime implicants and first including it, then excluding it from the cover leads to two alternatives of equal cost. One includes the prime implicants indicated in black, which yields

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_1x_3x_4 + \bar{x}_2x_3\bar{x}_4$$

The other includes the prime implicants indicated in blue, which yields

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

This procedure can be used to find minimum-cost implementations of both small and large logic functions. For our small examples it was convenient to use Karnaugh maps to determine the prime implicants of a function and then choose the final cover. Other techniques based on the same principles are much more suitable for use in CAD tools; we will introduce such techniques in sections 4.9 and 4.10.

The previous examples have been based on the sum-of-products form. We will next illustrate that the same concepts apply for the product-of-sums form.

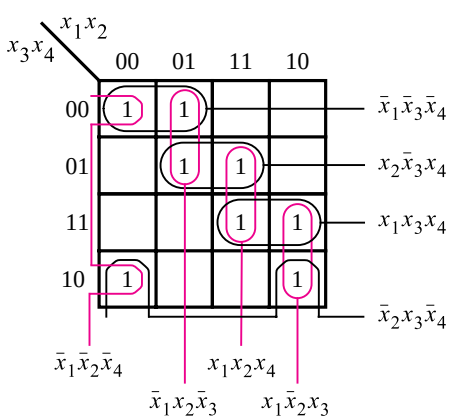



**Figure 4.12** The function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 10, 11, 13, 15)$.

## 4.3 MINIMIZATION OF PRODUCT-OF-SUMS FORMS

Now that we know how to find the minimum-cost sum-of-products (SOP) implementations of functions, we can use the same techniques and the principle of duality to obtain minimum-cost product-of-sums (POS) implementations. In this case it is the maxterms for which $f = 0$ that have to be combined into sum terms that are as large as possible. Again, a sum term is considered larger if it covers more maxterms, and the larger the term, the less costly it is to implement.

Figure 4.13 depicts the same function as Figure 4.9 depicts. There are three maxterms that must be covered: $M_4$, $M_5$, and $M_6$. They can be covered by two sum terms shown in the figure, leading to the following implementation:

$$f = (\bar{x}_1 + x_2)(\bar{x}_1 + x_3)$$

A circuit corresponding to this expression has two OR gates and one AND gate, with two inputs for each gate. Its cost is greater than the cost of the equivalent SOP implementation derived in Figure 4.9, which requires only one OR gate and one AND gate.

The function from Figure 4.10 is reproduced in Figure 4.14. The maxterms for which $f = 0$ can be covered as shown, leading to the expression

$$f = (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$$

This expression represents a circuit with three OR gates and one AND gate. Two of the OR gates have two inputs, and the third has four inputs; the AND gate has three inputs. Assuming that both the complemented and uncomplemented versions of the input variables $x_1$ to $x_4$ are available at no extra cost, the cost of this circuit is 15. This compares favorably with the SOP implementation derived from Figure 4.10, which requires five gates and 13 inputs at a total cost of 18.

In general, as we already know from section 2.6.1, the SOP and POS implementations of a given function may or may not entail the same cost. The reader is encouraged to find the POS implementations for the functions in Figures 4.11 and 4.12 and compare the costs with the SOP forms.

We have shown how to obtain minimum-cost POS implementations by finding the largest sum terms that cover all maxterms for which $f = 0$. Another way of obtaining

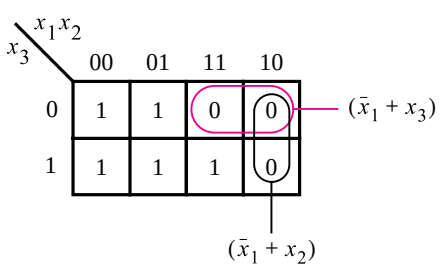



**Figure 4.13** POS minimization of $f(x_1, x_2, x_3) = \Pi M(4, 5, 6)$.

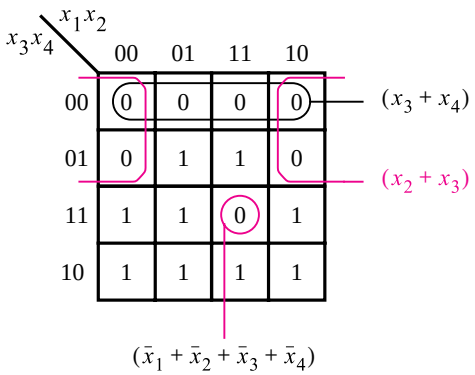



**Figure 4.14** POS minimization of $f(x_1, \ldots, x_4) = \Pi M(0, 1, 4, 8, 9, 12, 15)$.

the same result is by finding a minimum-cost SOP implementation of the complement of $f$. Then we can apply DeMorgan's theorem to this expression to obtain the simplest POS realization because $f = \bar{\bar{f}}$. For example, the simplest SOP implementation of $\bar{f}$ in Figure 4.13 is

$$\bar{f} = x_1\bar{x}_2 + x_1\bar{x}_3$$

Complementing this expression using DeMorgan's theorem yields

$$\begin{aligned} f = \bar{\bar{f}} &= \overline{x_1\bar{x}_2 + x_1\bar{x}_3} \\ &= \overline{x_1\bar{x}_2} \cdot \overline{x_1\bar{x}_3} \\ &= (\bar{x}_1 + x_2)(\bar{x}_1 + x_3) \end{aligned}$$

which is the same result as obtained above.

Using this approach for the function in Figure 4.14 gives

$$\bar{f} = \bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4$$

Complementing this expression produces

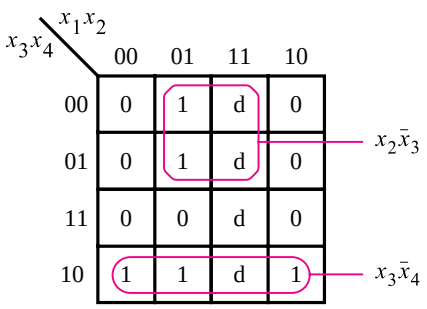
$$\begin{aligned} f = \bar{\bar{f}} &= \overline{\bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4} \\ &= \overline{\bar{x}_2\bar{x}_3} \cdot \overline{\bar{x}_3\bar{x}_4} \cdot \overline{x_1x_2x_3x_4} \\ &= (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4) \end{aligned}$$

which matches the previously derived implementation.

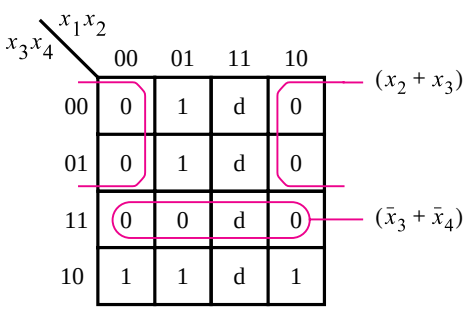
## 4.4 INCOMPLETELY SPECIFIED FUNCTIONS

In digital systems it often happens that certain input conditions can never occur. For example, suppose that $x_1$ and $x_2$ control two interlocked switches such that both switches cannot be closed at the same time. Thus the only three possible states of the switches are that both switches are open or that one switch is open and the other switch is closed. Namely, the input valuations $(x_1, x_2) = 00$, 01, and 10 are possible, but 11 is guaranteed not to occur. Then we say that $(x_1, x_2) = 11$ is a *don't-care condition*, meaning that a circuit with $x_1$ and $x_2$ as inputs can be designed by ignoring this condition. A function that has don't-care condition(s) is said to be *incompletely specified*.

Don't-care conditions, or *don't-cares* for short, can be used to advantage in the design of logic circuits. Since these input valuations will never occur, the designer may assume that the function value for these valuations is either 1 or 0, whichever is more useful in trying to find a minimum-cost implementation. Figure 4.15 illustrates this idea. The required function has a value of 1 for minterms $m_2$, $m_4$, $m_5$, $m_6$, and $m_{10}$. Assuming the above-mentioned interlocked switches, the $x_1$ and $x_2$ inputs will never be equal to 1 at the same time; hence the minterms $m_{12}$, $m_{13}$, $m_{14}$, and $m_{15}$ can all be used as don't-cares. The don't-




(a) SOP implementation




(b) POS implementation

**Figure 4.15** Two implementations of the function $f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$.

cares are denoted by the letter $d$ in the map. Using the shorthand notation, the function $f$ is specified as

$$f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

where $D$ is the set of don't-cares.

Part $(a)$ of the figure indicates the best sum-of-products implementation. To form the largest possible groups of 1s, thus generating the lowest-cost prime implicants, it is necessary to assume that the don't-cares $D_{12}$, $D_{13}$, and $D_{14}$ (corresponding to minterms $m_{12}$, $m_{13}$, and $m_{14}$) have the value of 1 while $D_{15}$ has the value of 0. Then there are only two prime implicants, which provide a complete cover of $f$. The resulting implementation is

$$f = x_2\overline{x}_3 + x_3\overline{x}_4$$

Part $(b)$ shows how the best product-of-sums implementation can be obtained. The same values are assumed for the don't cares. The result is

$$f = (x_2 + x_3)(\overline{x}_3 + \overline{x}_4)$$

The freedom in choosing the value of don't-cares leads to greatly simplified realizations. If we were to naively exclude the don't-cares from the synthesis of the function, by assuming that they always have a value of 0, the resulting SOP expression would be

$$f = \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_3\overline{x}_4 + \overline{x}_2x_3\overline{x}_4$$

and the POS expression would be

$$f = (x_2 + x_3)(\overline{x}_3 + \overline{x}_4)(\overline{x}_1 + \overline{x}_2)$$

Both of these expressions have higher costs than the expressions obtained with a more appropriate assignment of values to don't-cares.

Although don't-care values can be assigned arbitrarily, an arbitrary assignment may not lead to a minimum-cost implementation of a given function. If there are $k$ don't-cares, then there are $2^k$ possible ways of assigning 0 or 1 values to them. In the Karnaugh map we can usually see how best to do this assignment to find the simplest implementation.

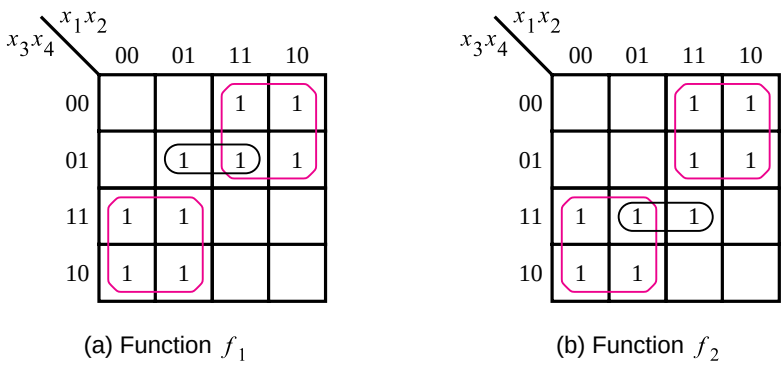
In the example above, we chose the don't-cares $D_{12}$, $D_{13}$, and $D_{14}$ to be equal to 1 and $D_{15}$ equal to 0 for both the SOP and POS implementations. Thus the derived expressions represent the same function, which could also be specified as $\sum m(2, 4, 5, 6, 10, 12, 13, 14)$. Assigning the same values to the don't-cares for both SOP and POS implementations is not always a good choice. Sometimes it may be advantageous to give a particular don't-care the value 1 for SOP implementation and the value 0 for POS implementation, or vice versa. In such cases the optimal SOP and POS expressions will represent different functions, but these functions will differ only for the valuations that correspond to these don't-cares. Example 4.24 in section 4.14 illustrates this possibility.

Using interlocked switches to illustrate how don't-care conditions can occur in a real system may seem to be somewhat contrived. However, in Chapters 6, 8, and 9 we will encounter many examples of don't-cares that occur in the course of practical design of digital circuits.

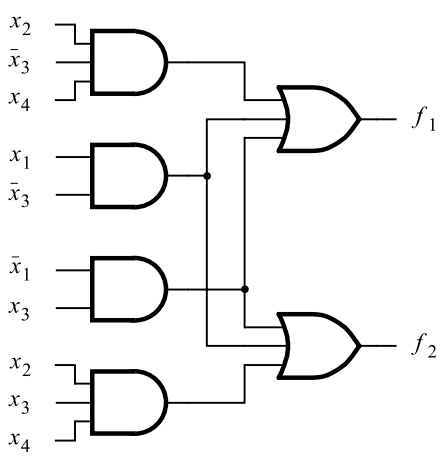
## 4.5 MULTIPLE-OUTPUT CIRCUITS

In all previous examples we have considered single functions and their circuit implementations. In practical digital systems it is necessary to implement a number of functions as part of some large logic circuit. Circuits that implement these functions can often be combined into a less-expensive single circuit with multiple outputs by sharing some of the gates needed in the implementation of individual functions.

**Example 4.1** An example of gate sharing is given in Figure 4.16. Two functions, $f_1$ and $f_2$, of the same variables are to be implemented. The minimum-cost implementations for these functions




(a) Function $f_1$ (b) Function $f_2$




(c) Combined circuit for $f_1$ and $f_2$

**Figure 4.16** An example of multiple-output synthesis.