chapter

2 Introduction to Logic Circuits

Chapter Objectives

In this chapter you will be introduced to:

- Logic functions and circuits
- Boolean algebra for dealing with logic functions
- Logic gates and synthesis of simple circuits
- CAD tools and the Verilog hardware description language
- Minimization of functions and Karnaugh maps

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, such as those that perform control applications or are involved in digital communications. All such applications are based on some simple logical operations that are performed on input information.

In Chapter 1 we showed that information in computers is represented as electronic signals that can have two discrete values. Although these values are implemented as voltage levels in a circuit, we refer to them simply as logic values, 0 and 1. Any circuit in which the signals are constrained to have only some number of discrete values is called a *logic circuit*. Logic circuits can be designed with different numbers of logic values, such as three, four, or even more, but in this book we deal only with the *binary* logic circuits that have two logic values.

Binary logic circuits 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 Appendix B we 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 a sufficient amount of current passes through it.

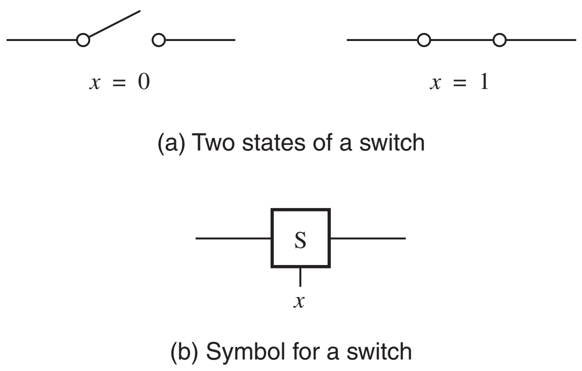


Figure 2.1 A binary switch.

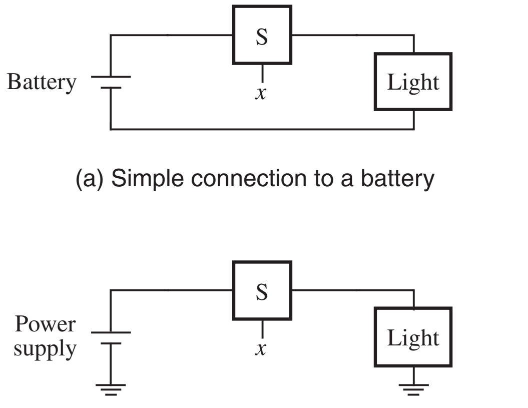

S
Battery
x
Light
(a) Simple connection to a battery
S
Power supply
x
Light
(b) Using a ground connection as the return path

Figure 2.2 A light controlled by a switch.

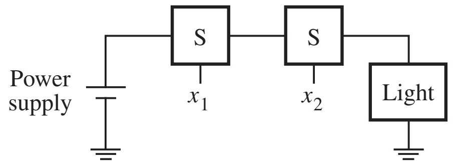
The current flows when the switch is closed, that is, when $x = 1$. In this example the input that causes changes in the behavior of the circuit is the switch control x. The output is defined as the state (or condition) of the light, which we will denote by the letter L. If the light is on, we will say that $L = 1$. If the 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

$$L(x) = x$$

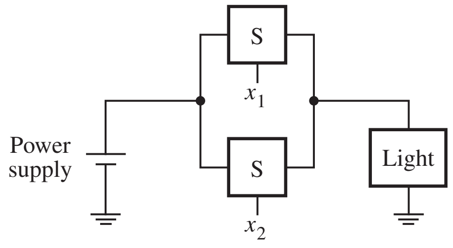
This simple *logic expression* describes the output as a function of the input. We say that $L(x) = x$ is a *logic function* and that x is an *input variable*.

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, usually in the range of 1 to 5 volts. One side of the power supply provides the *circuit ground*, as illustrated in Figure 2.2*b*. The circuit ground is a common reference point for voltages in the circuit. Rather than drawing wires in a circuit diagram for all nodes that return to the circuit ground, the diagram can be simplified by showing a connection to a ground symbol, as we have done for the bottom terminal of the light L in the figure. In the circuit diagrams that follow we will use this convention, because it makes the diagrams look simpler.

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.



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 2.3 Two basic functions.

This behavior can be described by the expression

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

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 the 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

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

The + symbol is called the *OR operator*, and the circuit in Figure 2.3*b* is said to implement a *logical OR function*. It is important not to confuse the use of the + symbol with its more common meaning, which is for arithmetic addition. In this chapter the + symbol represents the logical OR operation unless otherwise stated.

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

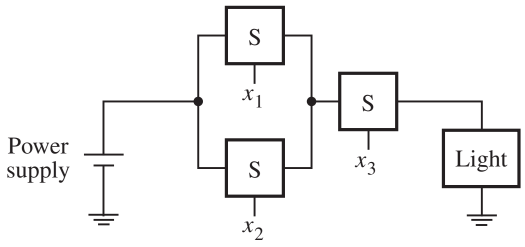


Figure 2.4 A series-parallel connection.

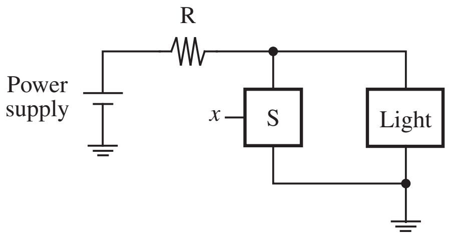


Figure 2.5 An inverting circuit.

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.

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

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

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 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 an exclamation mark (!), the tilde character (∼), or the word NOT is placed in front of the variable to denote the complementation. Thus the following are equivalent:

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

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

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

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 double vertical line) give all four possible combinations of logic values that the variables x_1 and x_2

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.

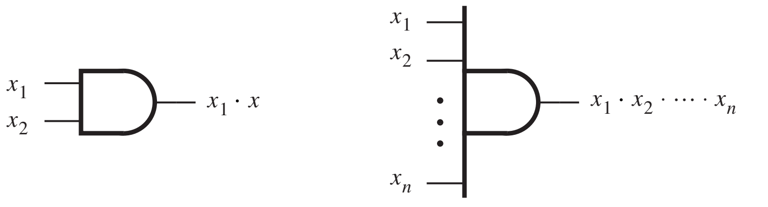
can have. The next column defines the AND operation for each combination of values of x_1 and x_2, and the last column defines the OR operation. Because we will frequently need to refer to "combinations of logic values" applied to some variables, we will adopt a shorter term, *valuation*, to denote such a combination of logic values.

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.

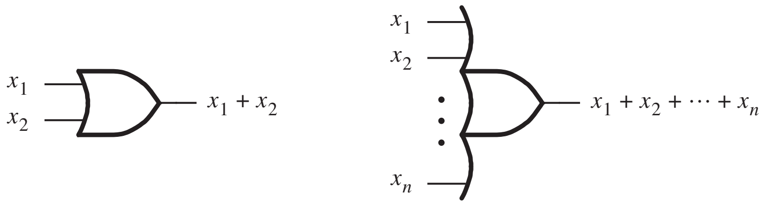
The AND and OR operations can be extended to n variables. An AND function of variables $x_1, x_2, \ldots, x_n$ has the value 1 only if all n variables are equal to 1. An OR function of variables $x_1, x_2, \ldots, x_n$ has the value 1 if one or more of the variables is equal to 1.

2.4 LOGIC GATES AND NETWORKS

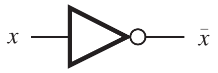
The three basic logic operations introduced in the previous sections can be used to implement logic functions of any complexity. A complex function may require many of these basic operations for its implementation. Each logic operation can be implemented electronically with transistors, resulting in 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



(a) AND gates



(b) OR gates



(c) NOT gate

Figure 2.8 The basic gates.

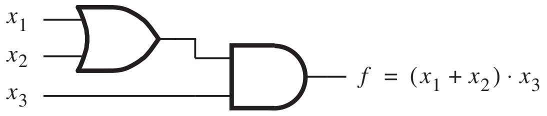


Figure 2.9 The function from Figure 2.4.

augmented to accommodate a greater number of inputs. We show how logic gates are built using transistors in Appendix B.

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 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

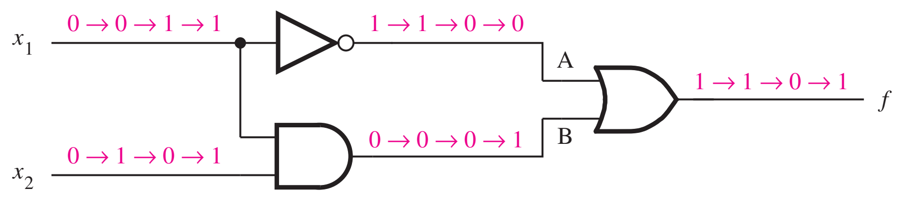
A designer of digital systems is faced with two basic issues. For an existing logic network, it must be possible to determine the function performed by the network. This task is referred to as the *analysis* process. The reverse task of designing a new network that implements a desired functional behavior 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 analyze its functional behavior, we can consider what happens if we apply all possible combinations of the 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 then let $x_1 = 0$ and $x_2 = 1$, 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 duration. 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.

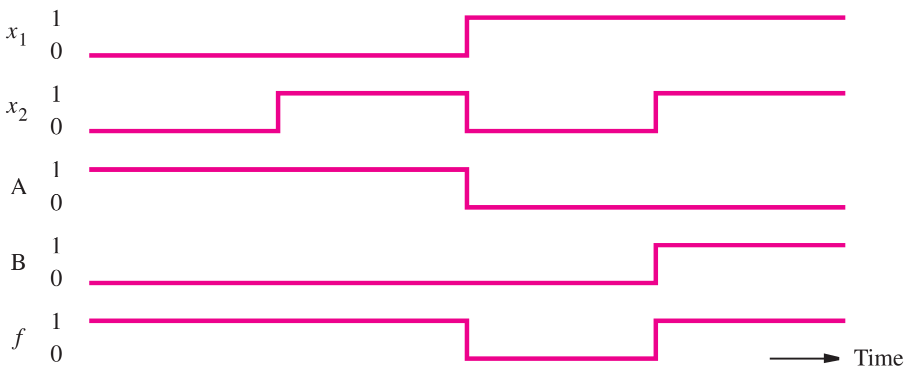
The timing diagram in Figure 2.10*c* shows that changes in the waveforms at points A and B and the output f take place instantaneously when the inputs x_1 and x_2 change their values. These idealized waveforms are based on the assumption that logic gates respond to changes on their inputs in zero time. 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.



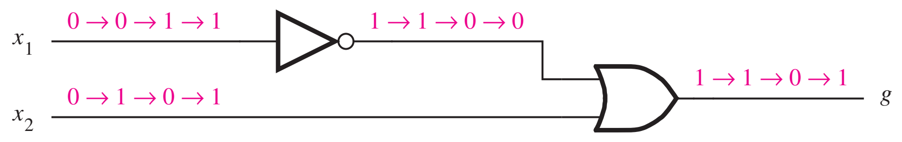
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

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.

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.

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? We will discuss some of the main approaches for synthesizing logic functions later in this chapter. 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 $f(x_1, x_2) = \overline{x}_1 + x_1 \cdot x_2$ and $g(x_1, x_2) = \overline{x}_1 + x_2$, there must exist some rules that can be used to show the equivalence

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

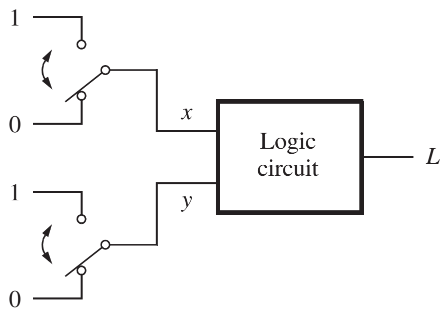
We have already established this equivalence through detailed analysis of the two circuits and construction of the truth table. But the same outcome can be achieved through algebraic manipulation of logic expressions. In Section 2.5 we will introduce a mathematical approach for dealing with logic functions, which provides the basis for modern design techniques.

Example 2.1

As an example of a logic function, consider the diagram in Figure 2.11*a*. It includes two toggle switches that control the values of signals x and y. Each toggle switch can be pushed down to the bottom position or up to the top position. When a toggle switch is in the bottom position it makes a connection to logic value 0 (ground), and when in the top position it connects to logic value 1 (power supply level). Thus, these toggle switches can be used to set x and y to either 0 or 1.

The signals x and y are inputs to a logic circuit that controls a light L. The required behavior is that the light should be on only if one, but not both, of the toggle switches is in the top position. This specification leads to the truth table in part (b) of the figure. Since $L = 1$ when $x = 0$ and $y = 1$ or when $x = 1$ and $y = 0$, we can implement this logic function using the network in Figure 2.11*c*.

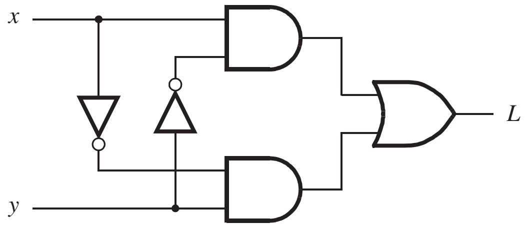
The reader may recognize the behavior of our light as being similar to that over a set of stairs in a house, where the light is controlled by two switches: one at the top of the stairs, and the other at the bottom. The light can be turned on or off by either switch because



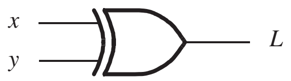
(a) Two switches that control a light

x	y	L
0	0	0
0	1	1
1	0	1
1	1	0

(b) Truth table



(c) Logic network



(d) XOR gate symbol

Figure 2.11 An example of a logic circuit.

it follows the truth table in Figure 2.11*b*. This logic function, which differs from the OR function only when both inputs are equal to 1, is useful for other applications as well. It is called the *exclusive-OR* (XOR) function and is indicated in logic expressions by the symbol $\oplus$. Thus, rather than writing $L = \overline{x} \cdot y + x \cdot \overline{y}$, we can write $L = x \oplus y$. The XOR function has the logic-gate symbol illustrated in Figure 2.11*d*.

Example 2.2 In Chapter 1 we showed how numbers are represented in computers by using binary digits. As another example of logic functions, consider the addition of two one-digit binary numbers a and b. The four possible valuations of a, b and the resulting sums are given in Figure 2.12*a* (in this figure the $+$ operator signifies *addition*). The sum $S = s_1s_0$ has to be a two-digit binary number, because when $a = b = 1$ then $S = 10$.

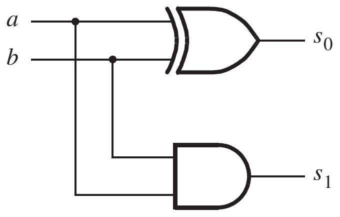
Figure 2.12*b* gives a truth table for the logic functions s_1 and s_0. From this table we can see that $s_1 = a \cdot b$ and $s_0 = a \oplus b$. The corresponding logic network is given in part (*c*) of the figure. This type of logic circuit, which adds binary numbers, is referred to as an *adder* circuit. We discuss circuits of this type in Chapter 3.

$$\begin{array}{r}a\\+\,b\\\hline s_1\,s_0\end{array}\qquad\begin{array}{r}0\\+\,0\\\hline 0\ 0\end{array}\quad\begin{array}{r}0\\+\,1\\\hline 0\ 1\end{array}\quad\begin{array}{r}1\\+\,0\\\hline 0\ 1\end{array}\quad\begin{array}{r}1\\+\,1\\\hline 1\ 0\end{array}$$

(a) Evaluation of $S = a + b$

a	b	s_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Logic network

Figure 2.12 Addition of binary numbers.

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 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 $\overline{x} = 1$

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

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 Boolean variable, 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 · 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 might 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 Boolean variables, 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*

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 2.13 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.

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 2.13 Proof of DeMorgan's theorem in 15*a*.

Example 2.3 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

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

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 $x_1 \cdot \overline{x}_1$ and $x_3 \cdot \overline{x}_3$ are both equal to 0. Therefore,

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

From 6*b* it follows that

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

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

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

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

Example 2.4 Consider the logic equation

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

The left-hand side can be manipulated as follows

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

The right-hand side can be manipulated as

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

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

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

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

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

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 2.3 and 2.4 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 2.14. 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

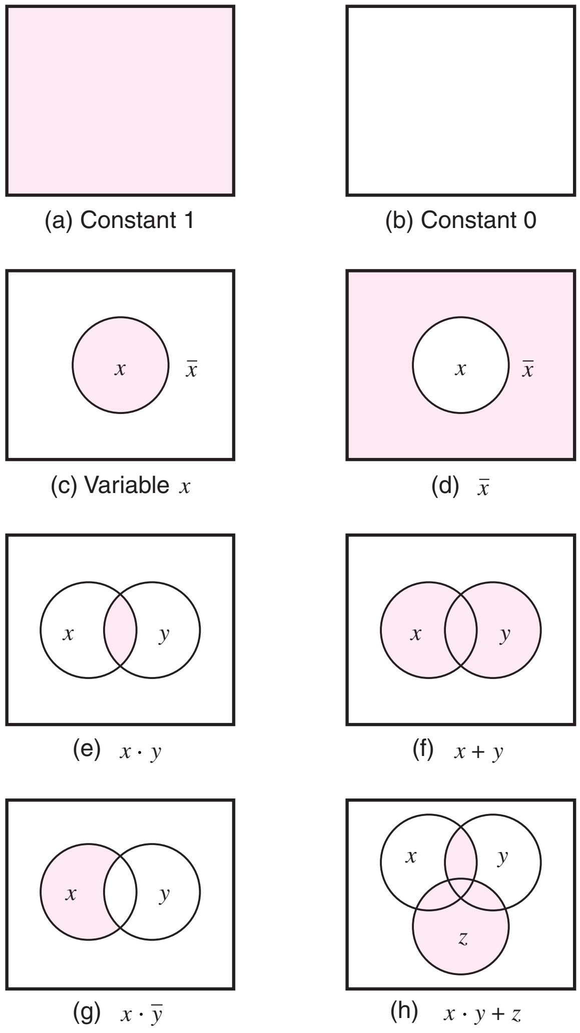


Figure 2.14 The Venn diagram representation.

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,

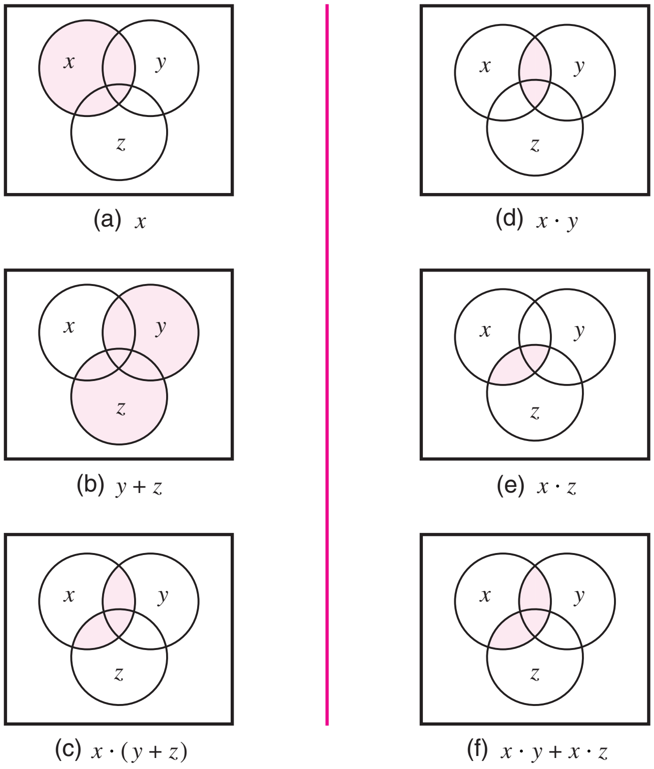


Figure 2.15 Verification of the distributive property $x \cdot (y + z) = x \cdot y + x \cdot z$.

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 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, 12a, in Section 2.5. Figure 2.15 gives the construction of the left and right sides of the identity that defines the property

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

Part (a) shows the area where $x = 1$. Part (b) indicates the area for $y + z$. Part (c) gives the diagram for $x \cdot (y + z)$, the intersection of shaded areas in parts (a) and (b). The right-hand side is constructed in parts (d), (e), and (f). Parts (d) and (e) describe the terms $x \cdot y$ and $x \cdot z$, respectively. The union of the shaded areas in these two diagrams then corresponds to the expression $x \cdot y + x \cdot z$, as seen in part (f). Since the shaded areas in parts (c) and (f) are identical, it follows that the distributive property is valid.

As another example, consider the identity

$$x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$$

which is illustrated in Figure 2.16. Notice that this identity states that the term $y \cdot z$ is fully covered by the terms $x \cdot y$ and $\overline{x} \cdot z$; therefore, this term can be omitted. This identity, which we listed earlier as property 17a, is often referred to as *consensus*.

The reader should use the Venn diagram to prove some other identities. The examples below prove the distributive property 12b, and DeMorgan's theorem, 15a.

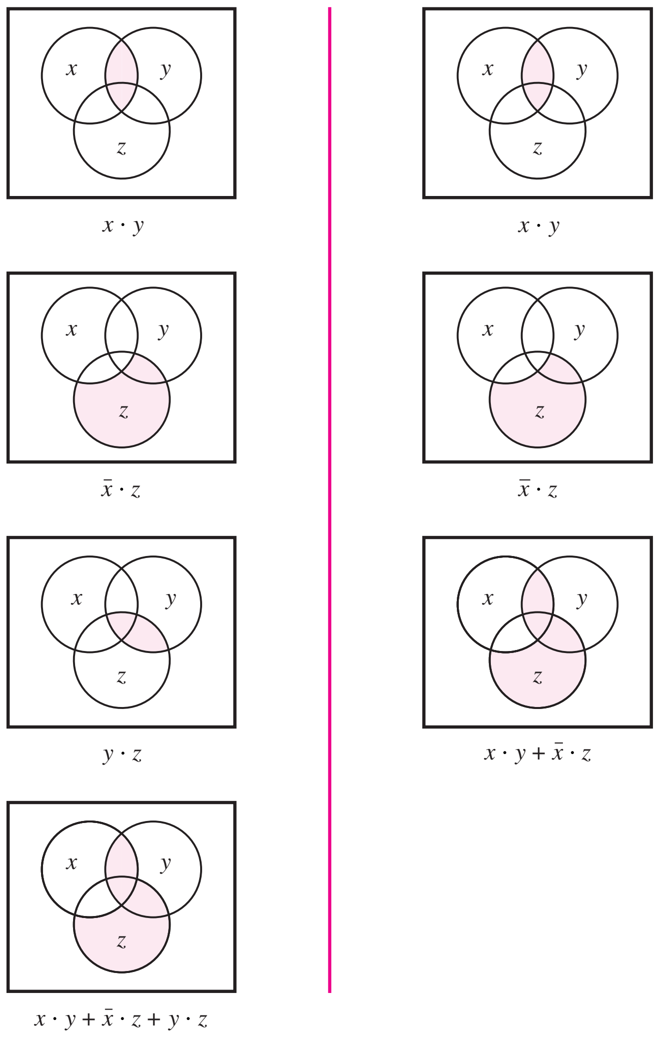

x
y
z
$x \cdot y$
$\overline{x} \cdot z$
$y \cdot z$
$x \cdot y + \overline{x} \cdot z + y \cdot z$
$x \cdot y + \overline{x} \cdot z$

Figure 2.16 Verification of $x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$.

Example 2.5

The distributive property 12*a* in Figure 2.15 will look familiar to the reader, because it is valid both for Boolean variables and for variables that are real numbers. In the case of real-number variables, the operations involved would be multiplication and addition, rather than logical AND and OR. However, the dual form 12*b* of this property, $x + y \cdot z = (x + y) \cdot (x + z)$, does not hold for real-number variables involving multiplication and addition operations. To prove that this identity is valid in Boolean algebra we can use the Venn diagrams in Figure 2.17. Parts (*a*) and (*b*) of the figure depict the terms x and $y \cdot z$, respectively, and part (*c*) gives the union of parts (*a*) and (*b*). Parts (*d*) and (*e*) depict the sum terms $(x + y)$ and $(x + z)$, and part (*f*) shows the intersection of (*d*) and (*e*). Since the diagrams in (*c*) and (*f*) are the same, this proves the identity.

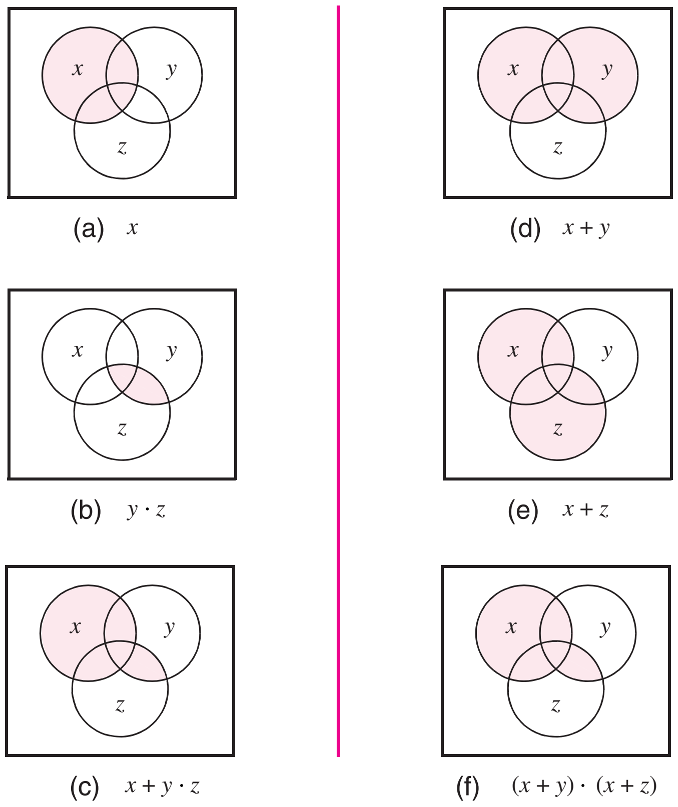


Figure 2.17 Proof of the distributive property 12*b*.

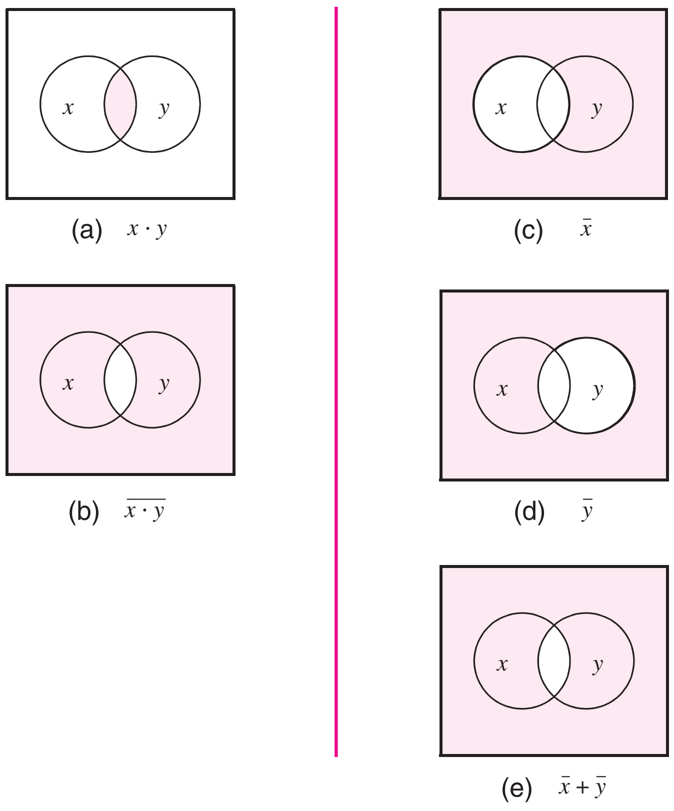


Figure 2.18 Proof of DeMorgan's theorem 15*a*.

Example 2.6 A proof of DeMorgan's theorem 15*a* by using Venn diagrams is illustrated in Figure 2.18. The diagram in part (*b*) of the figure, which is the complement of $x \cdot y$, is the same as the diagram in part (*e*), which is the union of part (*c*) with part (*d*), thus proving the theorem. We leave it as an exercise for the reader to prove the dual form of DeMorgan's theorem, 15*b*.

2.5.2 Notation and Terminology

Boolean algebra is based on the AND and OR operations, for which we have adopted the symbols $\cdot$ and $+$, respectively. These are also the standard symbols for the familiar arithmetic multiplication and addition operations. Considerable similarity exists between the Boolean operations and the arithmetic operations, which is the main reason why the same symbols are used. In fact, when single digits are involved there is only one significant difference; the result of $1 + 1$ is equal to 2 in ordinary arithmetic, whereas it is equal to 1 in Boolean algebra as defined by theorem $7b$ in Section 2.5.

Because of the similarity with the arithmetic addition and multiplication operations, the OR and AND operations are often called the *logical sum* and *product* operations. Thus $x_1 + x_2$ is the logical sum of x_1 and x_2, and $x_1 \cdot x_2$ is the logical product of x_1 and x_2. Instead of saying "logical product" and "logical sum," it is customary to say simply "product" and "sum." Thus we say that the expression

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

is a sum of three product terms, whereas the expression

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

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

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

it is first necessary to generate the complements of x_1 and x_2. Then the product terms $x_1 \cdot x_2$ and $\overline{x}_1 \cdot \overline{x}_2$ 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:

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

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

$$x_1x_2 + \overline{x}_1\overline{x}_2$$

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

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

Figure 2.19 A function to be synthesized.

produce a 0 or 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. We can express the required behavior using a truth table, as shown in Figure 2.19.

A possible procedure for designing a logic circuit that implements this 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 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 $\overline{x}_1 \cdot \overline{x}_2$. Similarly, the second row leads to the term $\overline{x}_1 \cdot x_2$. Thus f may be realized as

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

The logic network that corresponds to this expression is shown in Figure 2.20*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 sum expression. Replicating the third product term, the above expression becomes

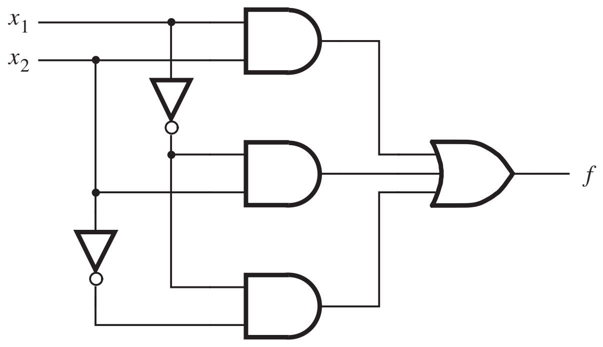
$$f(x_1, x_2) = x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + \overline{x}_1x_2$$

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

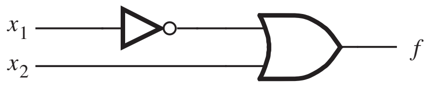
$$f(x_1, x_2) = x_1x_2 + \overline{x}_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2$$

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

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



(a) Canonical sum-of-products



(b) Minimal-cost realization

Figure 2.20 Two implementations of the function in Figure 2.19.

Applying theorem 8*b* we get

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

Finally, theorem 6*a* leads to

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

The network described by this expression is given in Figure 2.20*b*. Obviously, the cost of this network is much less than the cost of the network in part (*a*) of the figure.

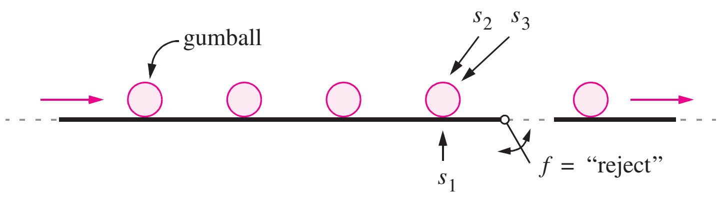
This simple example illustrates two things. First, a straightforward implementation of a function can be obtained by using a product term (AND gate) for each row of the truth table for which the function is equal to 1. Each product term contains all input variables, and it is formed such that if the input variable x_i is equal to 1 in the given row, then x_i is entered in the term; if $x_i = 0$ in that row, then $\overline{x}_i$ 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 2.20 from the truth table in Figure 2.19. 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.

Example 2.7 Figure 2.21*a* depicts a part of a factory that makes bubble gumballs. The gumballs travel on a conveyor that has three associated sensors s_1, s_2, and s_3. The sensor s_1 is connected to a scale that weighs each gumball, and if a gumball is not heavy enough to be acceptable then the sensor sets $s_1 = 1$. Sensors s_2 and s_3 examine the diameter of each gumball. If a gumball is too small to be acceptable, then $s_2 = 1$, and if it is too large, then $s_3 = 1$. If a gumball is of an acceptable weight and size, then the sensors give $s_1 = s_2 = s_3 = 0$. The conveyor pushes the gumballs over a "trap door" that it used to reject the ones that are not properly formed. A gumball should be rejected if it is too large, or both too small and too light. The trap door is opened by setting the logic function f to the value 1. By inspection, we can see that an appropriate logic expression is $f = s_1 s_2 + s_3$. We will use Boolean algebra to derive this logic expression from the truth table.

The truth table for f is given in Figure 2.21*b*. It sets f to 1 for each row in the table where s_3 has the value 1 (too large), as well as for each row where $s_1 = s_2 = 1$ (too light and too small). As described previously, a logic expression for f can be formed by including a product term for each row where $f = 1$. Thus, we can write

$$f = \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + s_1 s_2\bar{s}_3 + s_1 s_2 s_3$$



(a) Conveyor and sensors

s_1	s_2	s_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b) Truth table

Figure 2.21 A bubble gumball factory.

We can use algebraic manipulation to simplify this expression in a number of ways. For example, as shown below, we can first use rule 7*b* to repeat the term $s_1 s_2 s_3$, and then use the distributive property 12*a* and rule 8*b* to simplify the expression

$$\begin{aligned} f &= \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + s_1 s_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= \bar{s}_1 s_3(\bar{s}_2 + s_2) + s_1 s_3(\bar{s}_2 + s_2) + s_1 s_2(\bar{s}_3 + s_3) \\ &= \bar{s}_1 s_3 + s_1 s_3 + s_1 s_2 \end{aligned}$$

Now, using the combining property 14*a* on the first two product terms gives

$$f = s_3 + s_1 s_2$$

The observant reader will notice that using the combining property 14*a* is really just a short form of first using the distributive property 12*a* and then applying rule 8*b*, as we did in the previous step. Our simplified expression for f is the same as the one that we determined earlier, by inspection.

Example 2.8

There are different ways in which we can simplify the logic expression produced from the truth table in Figure 2.21*b*. Another approach is to first repeat the term $s_1 s_2 s_3$, as we did in Example 2.7, and then proceed as follows

$$\begin{aligned} f &= \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + s_1 s_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= s_3(\bar{s}_1\bar{s}_2 + \bar{s}_1 s_2 + s_1\bar{s}_2 + s_1 s_2) + s_1 s_2(\bar{s}_3 + s_3) \\ &= s_3 \cdot 1 + s_1 s_2 \\ &= s_3 + s_1 s_2 \end{aligned}$$

Here, we used the distributive property 12*a* to produce the expression $(\bar{s}_1\bar{s}_2 + \bar{s}_1 s_2 + s_1\bar{s}_2 + s_1 s_2)$. Since this expression includes all possible valuations of s_1, s_2, it is equal to 1, leading to the same expression for f that we derived before.

Example 2.9

Yet another way of producing the symplified logic expression is shown below.

$$\begin{aligned} f &= \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + \bar{s}_1\bar{s}_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= \bar{s}_1 s_3(\bar{s}_2 + s_2) + \bar{s}_2 s_3(s_1 + \bar{s}_1) + s_1 s_2(\bar{s}_3 + s_3) \\ &= \bar{s}_1 s_3 + \bar{s}_2 s_3 + s_1 s_2 \\ &= s_3(\bar{s}_1 + \bar{s}_2) + s_1 s_2 \\ &= s_3(\overline{s_1 s_2}) + s_1 s_2 \\ &= s_3 + s_1 s_2 \end{aligned}$$

In this solution, we first repeat the term $\bar{s}_1\bar{s}_2 s_3$, and then symplify to generate the expression $s_3(\bar{s}_1 + \bar{s}_2) + s_1 s_2$. Using DeMorgan's theorem 15*a* we can replace $(\bar{s}_1 + \bar{s}_2)$ with $(\overline{s_1 s_2})$, which can then be deleted by applying property 16*a*.

As illustrated by Examples 2.7 to 2.9, there are multiple ways in which a logic expression can be minimized by using Boolean algebra. This process can be daunting, because it is not obvious which rules, identities, and properties should be applied, and in what order. Later in this chapter, in Section 2.11, we will introduce a graphical technique, called the Karnaugh map, that clarifies this process by providing a systematic way of generating a minimal-cost logic expression for a function.

2.6.1 Sum-of-Products and Product-of-Sums Forms

Having introduced the synthesis process by means of simple examples, 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 $\bar{x}_i$ if $x_i = 0$.

To illustrate this concept, consider the truth table in Figure 2.22. 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 1.5, 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

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

Figure 2.22 Three-variable minterms and maxterms.

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 2.19 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 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 in 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 2.20, 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 2.23. To synthesize this function, we have to include the minterms m_1,

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 2.23 A three-variable function.

m_4, m_5, and m_6. Copying these minterms from Figure 2.22 leads to the following canonical sum-of-products expression for f

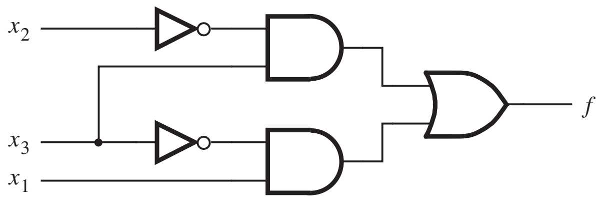
$$f(x_1, x_2, x_3) = \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3$$

This expression can be manipulated as follows

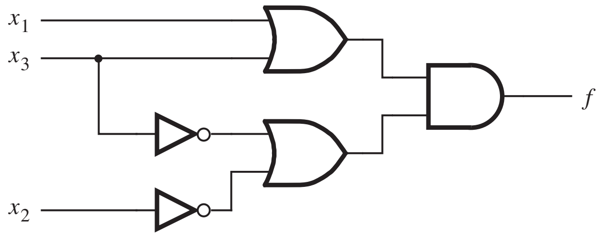
$$\begin{aligned} f &= (\overline{x}_1 + x_1)\overline{x}_2x_3 + x_1(\overline{x}_2 + x_2)\overline{x}_3 \\ &= 1 \cdot \overline{x}_2x_3 + x_1 \cdot 1 \cdot \overline{x}_3 \\ &= \overline{x}_2x_3 + x_1\overline{x}_3 \end{aligned}$$

This is the minimum-cost sum-of-products expression for f. It describes the circuit shown in Figure 2.24*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 2.24*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 2.23 can be specified



(a) A minimal sum-of-products realization



(b) A minimal product-of-sums realization

Figure 2.24 Two realizations of the function in Figure 2.23.

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.

Example 2.10

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

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

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

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

Example 2.11

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

$$f = \overline{x}_1 \overline{x}_2 x_3 x_4 + \overline{x}_1 x_2 x_3 x_4 + x_1 \overline{x}_2 \overline{x}_3 x_4 + x_1 x_2 \overline{x}_3 \overline{x}_4 + x_1 x_2 \overline{x}_3 x_4 + x_1 x_2 x_3 \overline{x}_4 + x_1 x_2 x_3 x_4$$

A simpler SOP expression can be obtained as follows

$$\begin{aligned} f &= \overline{x}_1 (\overline{x}_2 + x_2) x_3 x_4 + x_1 (\overline{x}_2 + x_2) \overline{x}_3 x_4 + x_1 x_2 \overline{x}_3 (\overline{x}_4 + x_4) + x_1 x_2 x_3 (\overline{x}_4 + x_4) \\ &= \overline{x}_1 x_3 x_4 + x_1 \overline{x}_3 x_4 + x_1 x_2 \overline{x}_3 + x_1 x_2 x_3 \\ &= \overline{x}_1 x_3 x_4 + x_1 \overline{x}_3 x_4 + x_1 x_2 (\overline{x}_3 + x_3) \\ &= \overline{x}_1 x_3 x_4 + x_1 \overline{x}_3 x_4 + x_1 x_2 \end{aligned}$$

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 2.22. 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 2.19

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

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 2.23. 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 2.24*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 2.24*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 Π 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 various different circuit structures, 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 strategies for finding minimum-cost implementations in Section 2.11.

Example 2.12

Consider again the function in Example 2.10. 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

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

A simplified POS expression can be derived as

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

Another way of deriving this product-of-sums expression is to use the sum-of-products form of $\overline{f}$. Thus,

$$\begin{aligned}\overline{f}(x_1, x_2, x_3) &= \sum m(0, 1, 5)\\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2x_3\\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2x_3\\ &= \overline{x}_1\overline{x}_2(\overline{x}_3 + x_3) + \overline{x}_2x_3(\overline{x}_1 + x_1)\\ &= \overline{x}_1\overline{x}_2 + \overline{x}_2x_3\end{aligned}$$

Now, first applying DeMorgan's theorem 15*b*, and then applying 15*a* (twice) gives

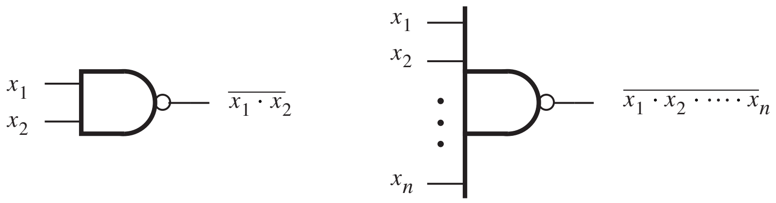
$$\begin{aligned}f &= \overline{\overline{f}}\\ &= \overline{(\overline{x}_1\overline{x}_2 + \overline{x}_2x_3)}\\ &= (\overline{\overline{x}_1\overline{x}_2})(\overline{\overline{x}_2x_3})\\ &= (x_1 + x_2)(x_2 + \overline{x}_3)\end{aligned}$$

To see that this product-of-sums expression for f is equivalent to the sum-of-products expression that we derived in Example 2.10, we can slightly rearrange our expression as $f = (x_2 + x_1)(x_2 + \overline{x}_3)$. Now, recognizing that this expression has the form of the righthand side of the distributive property 12*b*, we have the sum-of-products expression $f = x_2 + x_1\overline{x}_3$.

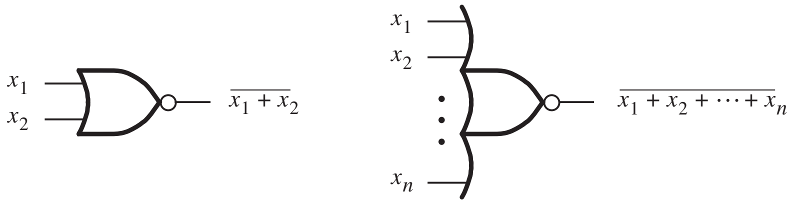
2.7 NAND AND NOR LOGIC NETWORKS

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 discuss in Appendix B. Figure 2.25 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 2.26. 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 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.

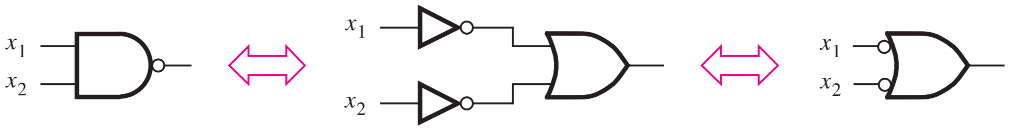

x_1
x_2
$\overline{x_1 \cdot x_2}$
x_1
x_2
x_n
$\overline{x_1 \cdot x_2 \cdot \cdots \cdot x_n}$

(a) NAND gates

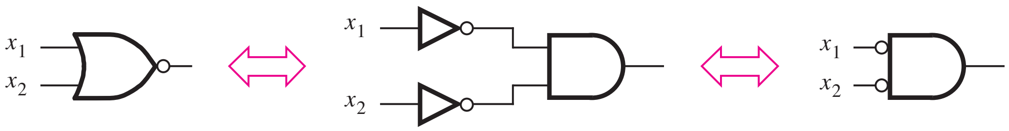


(b) NOR gates

Figure 2.25 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 2.26 DeMorgan's theorem in terms of logic gates.

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 2.27 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 2.26*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 2.27. This example shows that any AND-OR network can be implemented as a NAND-NAND network having the same topology.

Figure 2.28 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 2.27 except that now the identity in Figure 2.26*b* is applied. The conclusion is that any OR-AND network can be implemented as a NOR-NOR network having the same topology.

Example 2.13 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 2.12 we showed that the function can be represented by the POS expression

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

An OR-AND circuit that corresponds to this expression is shown in Figure 2.29*a*. Using the same structure of the circuit, a NOR-gate version is given in Figure 2.29*b*. Note that x_3 is inverted by a NOR gate that has its inputs tied together.

Example 2.14 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 2.10 we derived the SOP expression

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

which is realized using the circuit in Figure 2.30*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

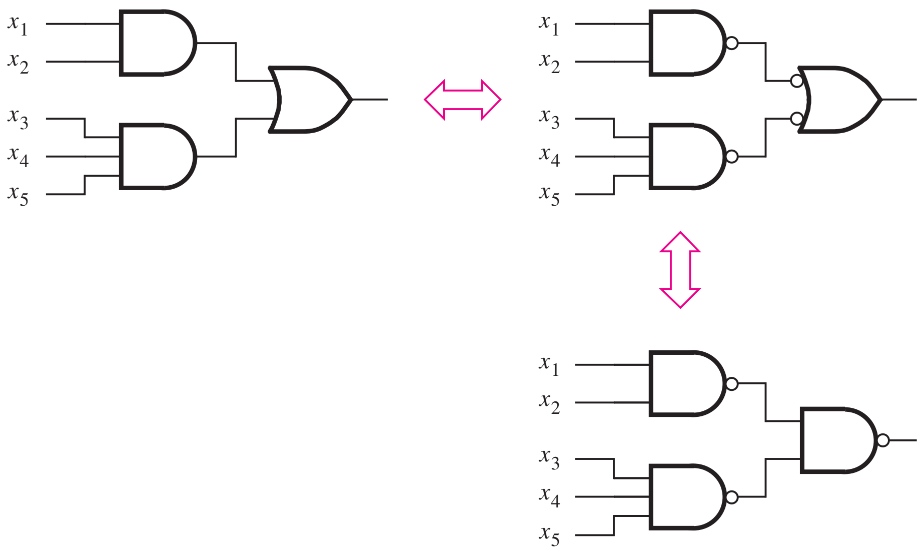


Figure 2.27 Using NAND gates to implement a sum-of-products.

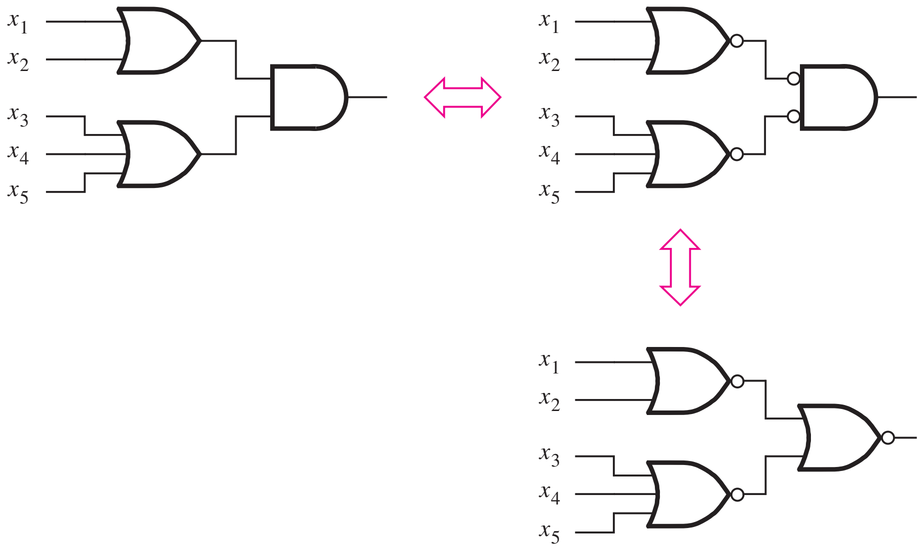
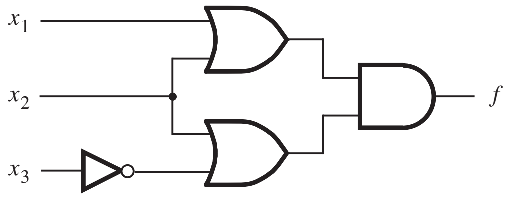
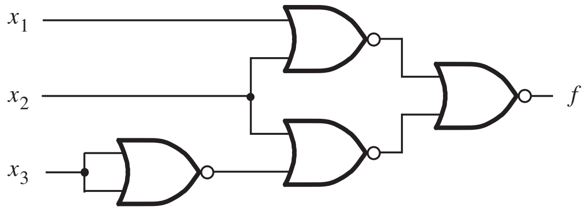


Figure 2.28 Using NOR gates to implement a product-of-sums.

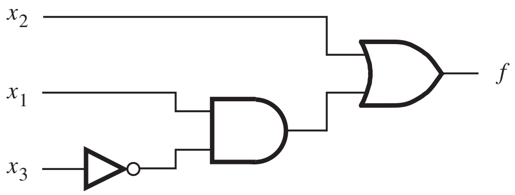


(a) POS implementation

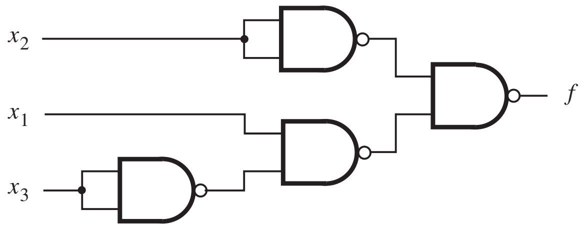


(b) NOR implementation

Figure 2.29 NOR-gate realization of the function in Example 2.13.



(a) SOP implementation



(b) NAND implementation

Figure 2.30 NAND-gate realization of the function in Example 2.10.

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, we can pass it through two NAND gates as depicted in Figure 2.30*b*. Observe that for this circuit the output *f* is

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

Applying DeMorgan's theorem, this expression becomes

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

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 three 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 2.31. The canonical sum-of-products expression for the specified function is

$$\begin{aligned} f &= m_1 + m_2 + m_4 + m_7 \\ &= \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_1x_2x_3 \end{aligned}$$

This expression cannot be simplified into a lower-cost sum-of-products expression. The resulting circuit is shown in Figure 2.32*a*.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2.31 Truth table for the three-way light control.

An alternative realization for this function is in the product-of-sums form. The canonical expression of this type is

$$\begin{aligned} f &= M_0 \cdot M_3 \cdot M_5 \cdot M_6 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + x_3) \end{aligned}$$

The resulting circuit is depicted in Figure 2.32*b*. It has the same cost as the circuit in part (*a*) of the figure.

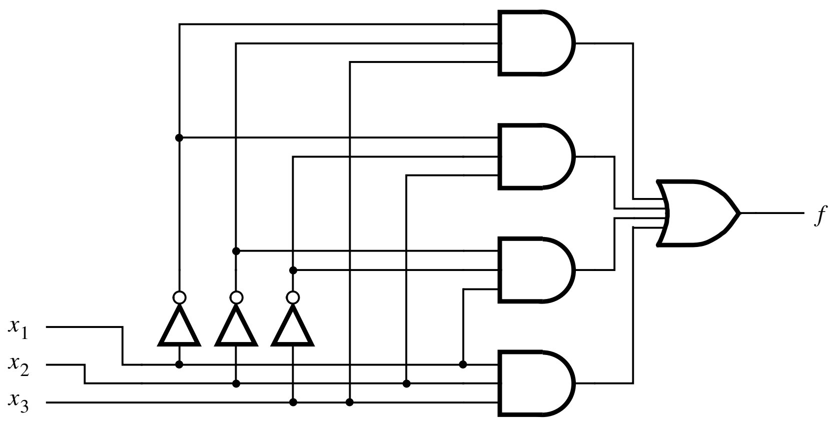
When the designed circuit is implemented, it can be tested by applying the various input valuations to the circuit and checking whether the output corresponds to the values specified in the truth table. A straightforward approach is to check that the correct output is produced for all eight possible input valuations.

2.8.2 MULTIPLEXER CIRCUIT

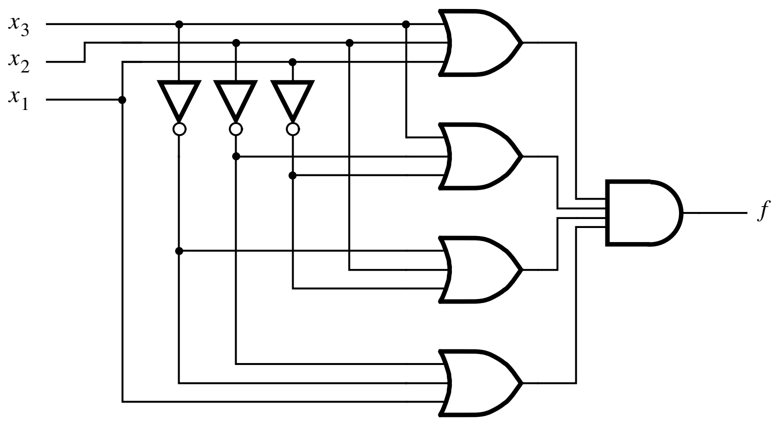
In computer systems it is often necessary to choose data from exactly one of a number of possible sources. Suppose that there are two sources of data, provided as input signals x_1 and x_2. The values of these signals change in time, perhaps at regular intervals. Thus sequences of 0s and 1s are applied on each of the inputs x_1 and x_2. We want to design a circuit that produces an output that has the same value as either x_1 or x_2, dependent on the value of a selection control signal s. Therefore, the circuit should have three inputs: x_1, x_2, and s. Assume that the output of the circuit will be the same as the value of input x_1 if $s = 0$, and it will be the same as x_2 if $s = 1$.

Based on these requirements, we can specify the desired circuit in the form of a truth table given in Figure 2.33*a*. From the truth table, we derive the canonical sum of products

$$f(s, x_1, x_2) = \overline{s}x_1\overline{x}_2 + \overline{s}x_1x_2 + s\overline{x}_1x_2 + sx_1x_2$$



(a) Sum-of-products realization



(b) Product-of-sums realization

Figure 2.32 Implementation of the function in Figure 2.31.

Using the distributive property, this expression can be written as

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

Applying theorem 8*b* yields

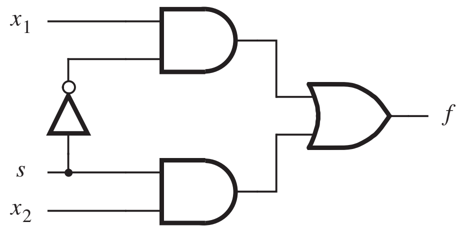
$$f = \bar{s}x_1 \cdot 1 + s \cdot 1 \cdot x_2$$

Finally, theorem 6*a* gives

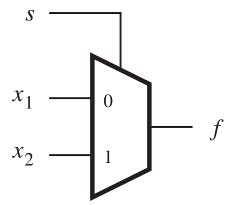
$$f = \bar{s}x_1 + sx_2$$

$s\ x_1\ x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

(a) Truth table



(b) Circuit



(c) Graphical symbol

s	$f(s, x_1, x_2)$
0	x_1
1	x_2

(d) More compact truth-table representation

Figure 2.33 Implementation of a multiplexer.

A circuit that implements this function is shown in Figure 2.33*b*. Circuits of this type are used so extensively that they are given a special name. A circuit that generates an output that exactly reflects the state of one of a number of data inputs, based on the value of one or more selection control inputs, is called a *multiplexer*. We say that a multiplexer circuit "multiplexes" input signals onto a single output.

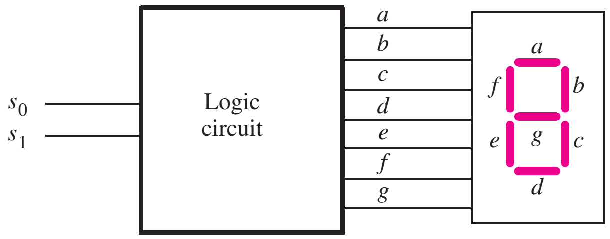
In this example we derived a multiplexer with two data inputs, which is referred to as a "2-to-1 multiplexer." A commonly used graphical symbol for the 2-to-1 multiplexer is shown in Figure 2.33*c*. The same idea can be extended to larger circuits. A 4-to-1 multiplexer has four data inputs and one output. In this case two selection control inputs are needed to choose one of the four data inputs that is transmitted as the output signal. An 8-to-1 multiplexer needs eight data inputs and three selection control inputs, and so on.

Note that the statement "$f = x_1$ if $s = 0$, and $f = x_2$ if $s = 1$" can be presented in a more compact form of a truth table, as indicated in Figure 2.33*d*. In later chapters we will have occasion to use such representation.

We showed how a multiplexer can be built using AND, OR, and NOT gates. The same circuit structure can be used to implement the multiplexer using NAND gates, as explained in Section 2.7. In Appendix B we will show other possibilities for constructing multiplexers. In Chapter 4 we will discuss the use of multiplexers in considerable detail.

2.8.3 Number Display

In Example 2.2 we designed an adder circuit that generates the arithmetic sum $S = a + b$, where a and b are one-bit numbers and $S = s_1 s_0$ provides the resulting two-bit sum, which is either 00, 01, or 10. In this design example we wish to create a logic circuit that drives a familiar seven-segment display, as illustrated in Figure 2.34*a*. This display allows us to show the value of S as a decimal number, either 0, 1, or 2. The display includes seven



(a) Logic circuit and 7-segment display

	s_1	s_0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1

(b) Truth table

Figure 2.34 Display of numbers.

segments, labeled $a, b, \ldots, g$ in the figure, where each segment is a light-emitting diode (LED). Our logic circuit has the two inputs s_1 and s_0. It produces seven outputs, one for each segment in the display. Setting an output to the value 1 illuminates the corresponding segment in the display. By illuminating specific segments for each valuation of $s_1 s_0$ we can make the display's appearance correspond to the shape of the appropriate decimal digit.

Part (*b*) of Figure 2.34 shows a truth table for the possible valuations of $s_1 s_0$ and indicates on the lefthand side how the display should appear in each case. The truth table specifies the logic values needed for each of the seven functions. For example, segment a in the 7-segment display needs to be turned on when S has the decimal values 0 or 2, but has to be off when S has the value 1. Hence, the corresponding logic function is set to 1 for minterms m_0 and m_2, giving $a = \overline{s}_1\overline{s}_0 + s_1\overline{s}_0 = \overline{s}_0$. Logic expressions for each of the seven functions are:

$$a = d = e = \overline{s}_0$$

$$b = 1$$

$$c = \overline{s}_1$$

$$f = \overline{s}_1\overline{s}_0$$

$$g = s_1\overline{s}_0$$

Designers of logic circuits rely heavily on CAD tools. We want to encourage the reader to become familiar with CAD tools as soon as possible. We have reached a point where an introduction to these tools is useful. The next section presents some basic concepts that are needed to use these tools. We will also introduce, in Section 2.10, a special language for describing logic circuits, called Verilog. This language is used to describe the circuits as an input to the CAD tools, which then proceed to derive a suitable implementation.

2.9 INTRODUCTION TO CAD TOOLS

The preceding sections introduced a basic approach for synthesis of logic circuits. A designer could use this approach manually for small circuits. However, logic circuits found in complex systems, such as today's computers, cannot be designed manually—they are designed using sophisticated CAD tools that automatically implement the synthesis techniques.

To design a logic circuit, a number of CAD tools are needed. They are usually packaged together into a *CAD system*, which typically includes tools for the following tasks: design entry, logic synthesis and optimization, simulation, and physical design. We will introduce some of these tools in this section and will provide additional discussion in later chapters.

2.9.1 DESIGN ENTRY

The starting point in the process of designing a logic circuit is the conception of what the circuit is supposed to do and the formulation of its general structure. This step is done manually by the designer because it requires design experience and intuition. The rest of the design process is done with the aid of CAD tools. The first stage of this process involves entering into the CAD system a description of the circuit being designed. This stage is called *design entry*. We will describe two design entry methods: using schematic capture and writing source code in a hardware description language.

Schematic Capture

A logic circuit can be defined by drawing logic gates and interconnecting them with wires. A CAD tool for entering a designed circuit in this way is called a *schematic capture* tool. The word *schematic* refers to a diagram of a circuit in which circuit elements, such as logic gates, are depicted as graphical symbols and connections between circuit elements are drawn as lines.

A schematic capture tool uses the graphics capabilities of a computer and a computer mouse to allow the user to draw a schematic diagram. To facilitate inclusion of gates in the schematic, the tool provides a collection of graphical symbols that represent gates of various types with different numbers of inputs. This collection of symbols is called a *library*. The gates in the library can be imported into the user's schematic, and the tool provides a graphical way of interconnecting the gates to create a logic network.

Any subcircuits that have been previously created can be represented as graphical symbols and included in the schematic. In practice it is common for a CAD system user to create a circuit that includes within it other smaller circuits. This methodology is known as *hierarchical design* and provides a good way of dealing with the complexities of large circuits.

The schematic-capture method is simple to use, but becomes awkward when large circuits are involved. A better method for dealing with large circuits is to write source code using a hardware description language to represent the circuit.

Hardware Description Languages

A *hardware description language (HDL)* is similar to a typical computer programming language except that an HDL is used to describe hardware rather than a program to be executed on a computer. Many commercial HDLs are available. Some are proprietary, meaning that they are provided by a particular company and can be used to implement circuits only in the technology offered by that company. We will not discuss the proprietary HDLs in this book. Instead, we will focus on a language that is supported by virtually all vendors that provide digital hardware technology and is officially endorsed as an *Institute of Electrical and Electronics Engineers (IEEE)* standard. The IEEE is a worldwide organization that promotes technical activities to the benefit of society in general. One of its activities involves the development of standards that define how certain technological concepts can be used in a way that is suitable for a large body of users.

Two HDLs are IEEE standards: *Verilog HDL* and *VHDL (Very High Speed Integrated Circuit Hardware Description Language)*. Both languages are in widespread use in the industry. We use Verilog in this book, but a VHDL version of the book is also available from the same publisher [4]. Although the two languages differ in many ways, the choice of using one or the other when studying logic circuits is not particularly important, because both offer similar features. Concepts illustrated in this book using Verilog can be directly applied when using VHDL.

In comparison to performing schematic capture, using Verilog offers a number of advantages. Because it is supported by most organizations that offer digital hardware technology, Verilog provides design *portability*. A circuit specified in Verilog can be implemented in different types of chips and with CAD tools provided by different companies, without having to change the Verilog specification. Design portability is an important advantage because digital circuit technology changes rapidly. By using a standard language, the designer can focus on the functionality of the desired circuit without being overly concerned about the details of the technology that will eventually be used for implementation.

Design entry of a logic circuit is done by writing Verilog code. Signals in the circuit can be represented as variables in the source code, and logic functions are expressed by assigning values to these variables. Verilog source code is plain text, which makes it easy for the designer to include within the code documentation that explains how the circuit works. This feature, coupled with the fact that Verilog is widely used, encourages sharing and reuse of Verilog-described circuits. This allows faster development of new products in cases where existing Verilog code can be adapted for use in the design of new circuits.

Similar to the way in which large circuits are handled in schematic capture, Verilog code can be written in a modular way that facilitates hierarchical design. Both small and large logic circuit designs can be efficiently represented in Verilog code.

Verilog design entry can be combined with other methods. For example, a schematic-capture tool can be used in which a subcircuit in the schematic is described using Verilog. We will introduce Verilog in Section 2.10.

2.9.2 Logic Synthesis

Synthesis is the process of generating a logic circuit from an initial specification that may be given in the form of a schematic diagram or code written in a hardware description language. Synthesis CAD tools generate efficient implementations of circuits from such specifications.

The process of translating, or *compiling*, Verilog code into a network of logic gates is part of synthesis. The output is a set of logic expressions that describe the logic functions needed to realize the circuit.

Regardless of what type of design entry is used, the initial logic expressions produced by the synthesis tools are not likely to be in an optimal form because they reflect the designer's input to the CAD tools. It is impossible for a designer to manually produce optimal results for large circuits. So, one of the important tasks of the synthesis tools is to manipulate the user's design to automatically generate an equivalent, but better circuit.

The measure of what makes one circuit better than another depends on the particular needs of a design project and the technology chosen for implementation. Earlier in this chapter we suggested that a good circuit might be one that has the lowest cost. There are other possible optimization goals, which are motivated by the type of hardware technology used for implementation of the circuit. We discuss implementation technologies in Appendix B.

The performance of a synthesized circuit can be assessed by physically constructing the circuit and testing it. But, its behavior can also be evaluated by means of simulation.

2.9.3 Functional Simulation

A circuit represented in the form of logic expressions can be simulated to verify that it will function as expected. The tool that performs this task is called a *functional simulator*. It uses the logic expressions (often referred to as equations) generated during synthesis, and assumes that these expressions will be implemented with perfect gates through which signals propagate instantaneously. The simulator requires the user to specify valuations of the circuit's inputs that should be applied during simulation. For each valuation, the simulator evaluates the outputs produced by the expressions. The results of simulation are usually provided in the form of a timing diagram which the user can examine to verify that the circuit operates as required.

2.9.4 Physical Design

After logic synthesis the next step in the design flow is to determine exactly how to implement the circuit on a given chip. This step is often called *physical design*. As we discuss in Appendix B, there are several different technologies that may be used to implement logic circuits. The physical design tools map a circuit specified in the form of logic expressions into a realization that makes use of the resources available on the target chip. They determine the placement of specific logic elements, which are not necessarily simple gates of the type we have encountered so far. They also determine the wiring connections that have to be made between these elements to implement the desired circuit.

2.9.5 Timing Simulation

Logic gates and other logic elements are implemented with electronic circuits, and these circuits cannot perform their function with zero delay. When the values of inputs to the circuit change, it takes a certain amount of time before a corresponding change occurs at the output. This is called a *propagation delay* of the circuit. The propagation delay consists of two kinds of delays. Each logic element needs some time to generate a valid output signal whenever there are changes in the values of its inputs. In addition to this delay, there is a delay caused by signals that must propagate along wires that connect various logic elements. The combined effect is that real circuits exhibit delays, which has a significant impact on their speed of operation.

A *timing simulator* evaluates the expected delays of a designed logic circuit. Its results can be used to determine if the generated circuit meets the timing requirements of the specification for the design. If the requirements are not met, the designer can ask the physical design tools to try again by indicating specific timing constraints that have to be met. If this does not succeed, then the designer has to try different optimizations in the synthesis step, or else improve the initial design that is presented to the synthesis tools.

2.9.6 Circuit Implementation

Having ascertained that the designed circuit meets all requirements of the specification, the circuit is implemented on an actual chip. If a custom-manufactured chip is created for this design, then this step is called *chip fabrication*. But if a programmable hardware device is used, then this step is called chip *configuration* or *programming*. Various types of chip technologies are described in Appendix B.

2.9.7 Complete Design Flow

The CAD tools discussed above are the essential parts of a CAD system. The complete design flow that we discussed is illustrated in Figure 2.35. This has been just a brief introductory discussion. A full presentation of the CAD tools is given in Chapter 10.

At this point the reader should have some appreciation for what is involved when using CAD tools. However, the tools can be fully appreciated only when they are used firsthand. We strongly encourage the reader to obtain access to suitable CAD tools and implement some examples of circuits by using these tools. Two examples of commonly-used CAD tools are the Quartus II tools available from Altera Corporation and the ISE tools provided by Xilinx Corporation. Both of these CAD systems can be obtained free-of-charge for educational use from their respective corporations' websites.

2.10 Introduction to Verilog

In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. Verilog was produced as a part of that effort. The original version of Verilog was developed by Gateway Design Automation, which was later acquired by Cadence Design Systems. In 1990 Verilog was put into the public domain, and it has since become one of the most popular languages for describing digital circuits. In 1995 Verilog was adopted as an official IEEE Standard, called 1364-1995. An enhanced version of Verilog, called Verilog 2001, was adopted as IEEE Standard 1364-2001 in 2001. While this version introduced a number of new features, it also supports all of the features in the original Verilog standard.

Verilog was originally intended for simulation and verification of digital circuits. Subsequently, with the addition of synthesis capability, Verilog has also become popular for use in design entry in CAD systems. The CAD tools are used to synthesize the Verilog code into a hardware implementation of the described circuit. In this book our main use of Verilog will be for synthesis.

Verilog is a complex, sophisticated language. Learning all of its features is a daunting task. However, for use in synthesis only a subset of these features is important. To simplify the presentation, we will focus the discussion on the features of the Verilog language that are actually used in the examples in the book. The material presented is sufficient to allow the reader to design a wide range of circuits. The reader who wishes to learn the complete Verilog language can refer to one of the specialized texts [5–11].

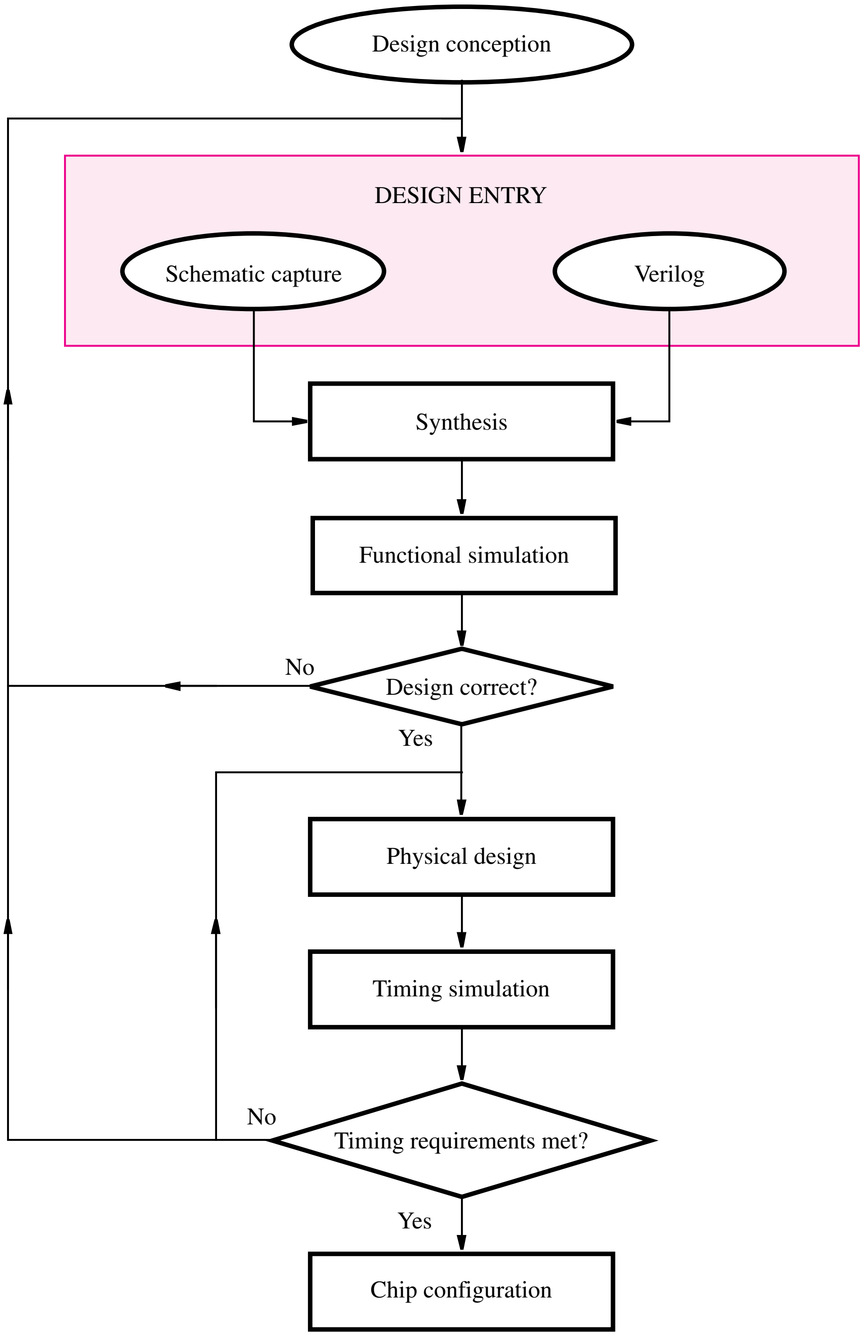


Figure 2.35 A typical CAD system.

Verilog is introduced in several stages throughout the book. Our general approach will be to introduce particular features only when they are relevant to the design topics covered in that part of the text. In Appendix A we provide a concise summary of the Verilog features covered in the book. The reader will find it convenient to refer to that material from time to time. In the remainder of this chapter we discuss the most basic concepts needed to write simple Verilog code.

Representation of Digital Circuits in Verilog

When using CAD tools to synthesize a logic circuit, the designer can provide the initial description of the circuit in several different ways, as we explained in the previous section. One efficient way is to write this description in the form of Verilog source code. The Verilog compiler translates this code into a logic circuit.

Verilog allows the designer to describe a desired circuit in a number of ways. One possibility is to use Verilog constructs that describe the structure of the circuit in terms of circuit elements, such as logic gates. A larger circuit is defined by writing code that connects such elements together. This approach is referred to as the *structural* representation of logic circuits. Another possibility is to describe a circuit more abstractly, by using logic expressions and Verilog programming constructs that define the desired behavior of the circuit, but not its actual structure in terms of gates. This is called the *behavioral* representation.

2.10.1 Structural Specification of Logic Circuits

Verilog includes a set of *gate-level primitives* that correspond to commonly-used logic gates. A gate is represented by indicating its functional name, output, and inputs. For example, a two-input AND gate, with output y and inputs x_1 and x_2, is denoted as

and (y, x1, x2);

A four-input OR gate is specified as

or (y, x1, x2, x3, x4);

The keywords **nand** and **nor** are used to define the NAND and NOR gates in the same way. The NOT gate given by

not (y, x);

implements $y = \overline{x}$. The gate-level primitives can be used to specify larger circuits. All of the available Verilog gate-level primitives are listed in Table A.2 in Appendix A.

A logic circuit is specified in the form of a *module* that contains the statements that define the circuit. A module has inputs and outputs, which are referred to as its *ports*. The word port is a commonly-used term that refers to an input or output connection to an electronic circuit. Consider the multiplexer circuit from Figure 2.33*b*, which is reproduced in Figure 2.36. This circuit can be represented by the Verilog code in Figure 2.37. The first statement gives the module a name, *example1*, and indicates that there are four port signals. The next two statements declare that x_1, x_2, and s are to be treated as **input** signals,

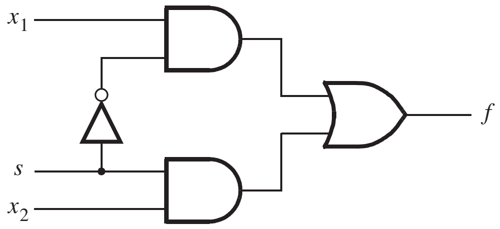


Figure 2.36 The logic circuit for a multiplexer.

```
module example1 (x1, x2, s, f);
   input x1, x2, s;
   output f;

   not (k, s);
   and (g, k, x1);
   and (h, s, x2);
   or (f, g, h);

endmodule
```

Figure 2.37 Verilog code for the circuit in Figure 2.36.

while f is the **output**. The actual structure of the circuit is specified in the four statements that follow. The NOT gate gives $k = \overline{s}$. The AND gates produce $g = \overline{s}x_1$ and $h = sx_2$. The outputs of AND gates are combined in the OR gate to form

$$\begin{aligned} f &= g + h \\ &= \overline{s}x_1 + sx_2 \end{aligned}$$

The module ends with the **endmodule** statement. We have written the Verilog keywords in bold type to make the text easier to read. We will continue this practice throughout the book.

A second example of Verilog code is given in Figure 2.38. It defines a circuit that has four input signals, x_1, x_2, x_3, and x_4, and three output signals, f, g, and h. It implements the logic functions

$$\begin{aligned} g &= x_1x_3 + x_2x_4 \\ h &= (x_1 + \overline{x}_3)(\overline{x}_2 + x_4) \\ f &= g + h \end{aligned}$$

```
module example2 (x1, x2, x3, x4, f, g, h);
  input x1, x2, x3, x4;
  output f, g, h;

  and (z1, x1, x3);
  and (z2, x2, x4);
  or (g, z1, z2);
  or (z3, x1, ~x3);
  or (z4, ~x2, x4);
  and (h, z3, z4);
  or (f, g, h);

endmodule
```

Figure 2.38 Verilog code for a four-input circuit.

Instead of using explicit NOT gates to define $\overline{x}_2$ and $\overline{x}_3$, we have used the Verilog operator "∼" (tilde character on the keyboard) to denote complementation. Thus, $\overline{x}_2$ is indicated as ∼x2 in the code. The circuit produced by the Verilog compiler for this example is shown in Figure 2.39.

Verilog Syntax

The names of modules and signals in Verilog code follow two simple rules: the name must start with a letter, and it can contain any letter or number plus the "_" underscore and "$" characters. Verilog is case sensitive. Thus, the name k is not the same as K and *Example1* is not the same as *example1*. The Verilog syntax does not enforce a particular style of code. For example, multiple statements can appear on a single line. White space characters, such as SPACE and TAB, and blank lines are ignored. As a matter of good style, code should be formatted in such a way that it is easy to read. Indentation and blank lines can be used to make separate parts of the code easily recognizable, as we have done in Figures 2.37 and 2.38. Comments may be included in the code to improve its readability. A comment begins with the double slash "//" and continues to the end of the line.

2.10.2 BEHAVIORAL SPECIFICATION OF LOGIC CIRCUITS

Using gate-level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the behavior of a logic circuit. One possibility is to define the circuit using logic expressions. Figure 2.40 shows how the circuit in Figure 2.36 can be defined with the expression

$$f = \overline{s}x_1 + sx_2$$

The AND and OR operations are indicated by the "&" and "|" Verilog operators, respectively. The *assign* keyword provides a *continuous assignment* for the signal f. The word continuous

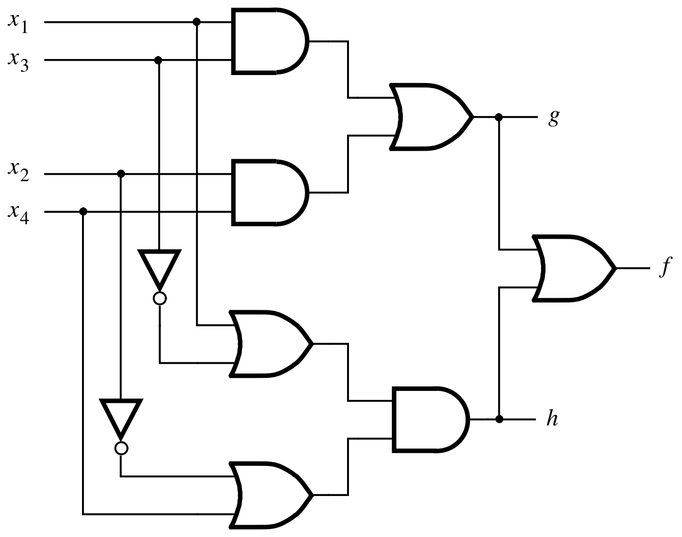


Figure 2.39 Logic circuit for the code in Figure 2.38.

```
module example3 (x1, x2, s, f);
   input x1, x2, s;
   output f;

   assign f = (~s & x1) | (s & x2);

endmodule
```

Figure 2.40 Using the continuous assignment to specify the circuit in Figure 2.36.

stems from the use of Verilog for simulation; whenever any signal on the right-hand side changes its state, the value of f will be re-evaluated. The effect is equivalent to using the gate-level primitives in Figure 2.37. Following this approach, the circuit in Figure 2.39 can be specified as shown in Figure 2.41.

Using logic expressions makes it easier to write Verilog code. But even higher levels of abstraction can often be used to advantage. Consider again the multiplexer circuit of Figure 2.36. The circuit can be described in words by saying that $f = x_1$ if $s = 0$ and $f = x_2$ if $s = 1$. In Verilog, this behavior can be defined with the **if-else** statement

```
if (s == 0)
   f = x1;
else
   f = x2;
```

```
module example4 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   assign g = (x1 & x3) | (x2 & x4);
   assign h = (x1 | ~x3) & (~x2 | x4);
   assign f = g | h;

endmodule
```

Figure 2.41 Using the continuous assignment to specify the circuit in Figure 2.39.

```
// Behavioral specification
module example5 (x1, x2, s, f);
   input x1, x2, s;
   output f;
   reg f;

   always @(x1 or x2 or s)
      if (s == 0)
         f = x1;
      else
         f = x2;

endmodule
```

Figure 2.42 Behavioral specification of the circuit in Figure 2.36.

The complete code is given in Figure 2.42. The first line illustrates how a comment can be inserted. The **if-else** statement is an example of a Verilog *procedural* statement. We will introduce other procedural statements, such as loop statements, in Chapters 3 and 4.

Verilog syntax requires that procedural statements be contained inside a construct called an **always** block, as shown in Figure 2.42. An **always** block can contain a single statement, as in this example, or it can contain multiple statements. A typical Verilog design module may include several **always** blocks, each representing a part of the circuit being modeled. An important property of the **always** block is that the statements it contains are evaluated in the order given in the code. This is in contrast to the continuous assignment statements, which are evaluated concurrently and hence have no meaningful order.

The part of the **always** block after the @ symbol, in parentheses, is called the *sensitivity list*. This list has its roots in the use of Verilog for simulation. The statements inside an **always** block are executed by the simulator only when one or more of the signals in the sensitivity list changes value. In this way, the complexity of a simulation process is simplified, because it is not necessary to execute every statement in the code at all times. When Verilog is being employed for synthesis of circuits, as in this book, the sensitivity list simply tells the Verilog compiler which signals can directly affect the outputs produced by the **always** block.

If a signal is assigned a value using procedural statements, then Verilog syntax requires that it be declared as a *variable*; this is accomplished by using the keyword **reg** in Figure 2.42. This term also derives from the simulation jargon: It means that, once a variable's value is assigned with a procedural statement, the simulator "registers" this value and it will not change until the **always** block is executed again. We will discuss this issue in detail in Chapter 3.

Instead of using a separate statement to declare that the variable f is of **reg** type in Figure 2.42, we can alternatively use the syntax

```
output reg f;
```

which combines these two statements. Also, Verilog 2001 adds the ability to declare a signal's direction and type directly in the module's list of ports. This style of code is illustrated in Figure 2.43. In the sensitivity list of the **always** statement we can use commas instead of the word **or**, which is also illustrated in Figure 2.43. Moreover, instead of listing the relevant signals in the sensitivity list, it is possible to write simply

```
always @(*)
```

or even more simply

```
always @*
```

assuming that the compiler will figure out which signals need to be considered.

Behavioral specification of a logic circuit defines only its behavior. CAD synthesis tools use this specification to construct the actual circuit. The detailed structure of the synthesized circuit will depend on the technology used.

```
// Behavioral specification
module example5 (input x1, x2, s, output reg f);

   always @(x1, x2, s)
      if (s == 0)
         f = x1;
      else
         f = x2;

endmodule
```

Figure 2.43 A more compact version of the code in Figure 2.42.

2.10.3 Hierarchical Verilog Code

The examples of Verilog code given so far include just a single module. For larger designs, it is often convenient to create a hierarchical structure in the Verilog code, in which there is a *top-level* module that includes multiple instances of *lower-level* modules. To see how hierarchical Verilog code can be written consider the circuit in Figure 2.44. This circuit comprises two lower-level modules: the adder module that we described in Figure 2.12, and the module that drives a 7-segment display which we showed in Figure 2.34. The purpose of the circuit is to generate the arithmetic sum of the two inputs x and y, using the *adder* module, and then to show the resulting decimal value on the 7-segment display.

Verilog code for the *adder* module from Figure 2.12 and the *display* module from Figure 2.34 is given in Figures 2.45 and 2.46, respectively. For the *adder* module continuous assignment statements are used to specify the two-bit sum $s_1 s_0$. The assignment statement for s_0 uses the Verilog XOR operator, which is specified as $s_0 = a \wedge b$. The code for the *display* module includes continuous assignment statements that correspond to the

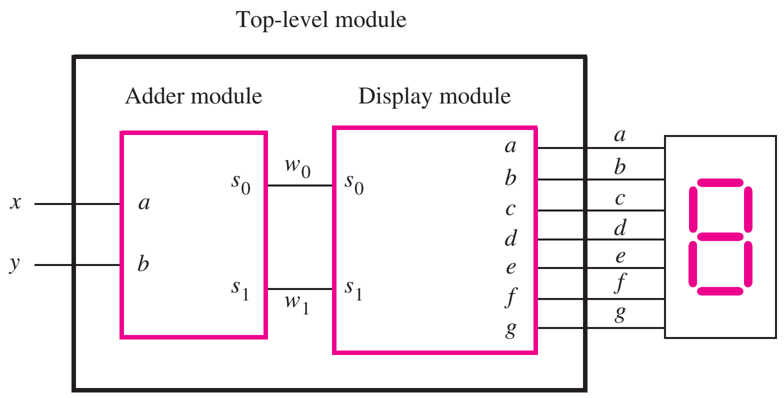


Figure 2.44 A logic circuit with two modules.

```
// An adder module
module adder (a, b, s1, s0);
   input a, b;
   output s1, s0;

   assign s1 = a & b;
   assign s0 = a ^ b;

endmodule
```

Figure 2.45 Verilog specification of the circuit in Figure 2.12.

```
// A module for driving a 7-segment display
module display (s1, s0, a, b, c, d, e, f, g);
   input s1, s0;
   output a, b, c, d, e, f, g;

   assign a = ~s0;
   assign b = 1;
   assign c = ~s1;
   assign d = ~s0;
   assign e = ~s0;
   assign f = ~s1 & ~s0;
   assign g = s1 & ~s0;

endmodule
```

Figure 2.46 Verilog specification of the circuit in Figure 2.34.

```
module adder_display (x, y, a, b, c, d, e, f, g);
   input x, y;
   output a, b, c, d, e, f, g;
   wire w1, w0;

   adder U1 (x, y, w1, w0);
   display U2 (w1, w0, a, b, c, d, e, f, g);

endmodule
```

Figure 2.47 Hierarchical Verilog code for the circuit in Figure 2.44.

logic expressions for each of the seven outputs of the display circuit, which are given in Section 2.8.3. The statement

assign b = 1;

assigns the output *b* of the display module to have the constant value 1. We discuss the specification of numbers in Verilog code in Chapter 3.

The top-level Verilog module, named *adder_display*, is given in Figure 2.47. This module has the inputs *x* and *y*, and the outputs $a, \ldots, g$. The statement

wire w1, w0;

is needed because the signals w_1 and w_0 are neither inputs nor outputs of the circuit in Figure 2.44. Since these signals cannot be declared as input or output ports in the Verilog code, they have to be declared as (internal) *wires*. The statement

```
adder U1 (x, y, w1, w0);
```

instantiates the *adder* module from Figure 2.45 as a submodule. The submodule is given a name, U1, which can be any valid Verilog name. In this instantiation statement the signals attached to the ports of the *adder* submodule are listed in the same order as those in Figure 2.45. Thus, the input ports x and y of the top-level module in Figure 2.47 are connected to the first two ports of *adder*, which are named a and b. The order in which signals are listed in the instantiation statement determines which signal is connected to each port in the submodule. The instantiation statement also attaches the last two ports of the *adder* submodule, which are its outputs, to the wires w1 and w0 in the top-level module. The statement

```
display U2 (w1, w0, a, b, c, d, e, f, g);
```

instantiates the other submodule in our circuit. Here, the wires w1 and w0, which have already been connected to the outputs of the *adder* submodule, are attached to the corresponding input ports of the *display* submodule. The *display* submodule's output ports are attached to the $a, \ldots, g$ output ports of the top-level module.

2.10.4 How *Not* to Write Verilog Code

When learning how to use Verilog or other hardware description languages, the tendency for the novice is to write code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesizing such code. This book contains more than 100 examples of complete Verilog code that represent a wide range of logic circuits. In these examples the code is easily related to the described logic circuit. The reader is advised to adopt the same style of code. A good general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the Verilog code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to model.

Once complete Verilog code is written for a particular design, the reader is encouraged to analyze the resulting circuit produced by the CAD synthesis tools; typical CAD systems provide graphical viewing tools that can display a logic circuit that corresponds to the output produced by the Verilog compiler. Much can be learned about Verilog, logic circuits, and logic synthesis through this process. We provide additional guidelines for writing Verilog code in Appendix A.

2.11 Minimization and Karnaugh Maps

In a number of our examples we have used algebraic manipulation to find a reduced-cost implementation of a function in either sum-of-products or product-of-sums form. In these examples, we made use of the rules, theorems, and properties of Boolean algebra that

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

were introduced in Section 2.5. For example, we often used the distributive property, DeMorgan's theorem, and the combining property. In general, it is not obvious when to apply these theorems and properties to find a minimum-cost circuit, and it is often tedious and impractical to do so. This section introduces a more manageable approach, call the *Karnaugh map*, which provides a systematic way of producing a minimum-cost logic expression.

The key to the Karnaugh map approach is that it allows the application of the combining property 14*a*, or 14*b*, as judiciously as possible. To understand how it works consider the function f in Figure 2.48. The canonical sum-of-products expression for f consists of minterms m_0, m_2, m_4, m_5, and m_6, so that

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

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 $\overline{x}_1$ and $\overline{x}_3$, but they differ in the value of x_2 because m_0 includes $\overline{x}_2$ while m_2 includes x_2. Thus

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

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

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

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

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

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 $\overline{x}_3$. In other words, the minterms m_0, m_2, m_4, and m_6 are all *included* in the term $\overline{x}_3$. The remaining minterm in f is m_5. It can be combined with m_4, which gives

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

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 $\overline{x}_3$ as explained above and also to combine with m_5 to yield the term $x_1\overline{x}_2$.

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

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

The expression has the product term $\overline{x}_3$ 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 2.48, 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

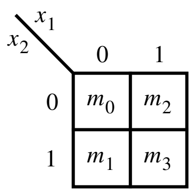
In a similar way, if we look at m_4 and m_5 as a group of two

	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.

x_1	x_2	
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3

(a) Truth table



(b) Karnaugh map

Figure 2.49 Location of two-variable minterms.

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 2.49. 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 14a 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

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

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 2.50. It corresponds to the function f of Figure 2.19. 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.

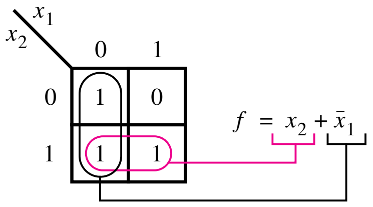


Figure 2.50 The function of Figure 2.19.

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 $f = 1$. 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 2.51*a* lists all of the three-variable minterms, and part (*b*) of the figure indicates the locations of these minterms in the Karnaugh map. 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 2.51*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 2.52*a* represents the function of Figure 2.23 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,

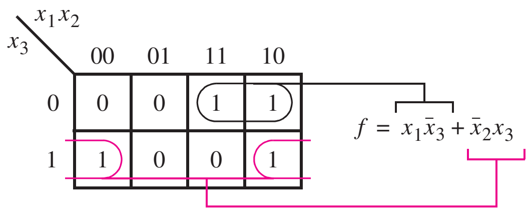
x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

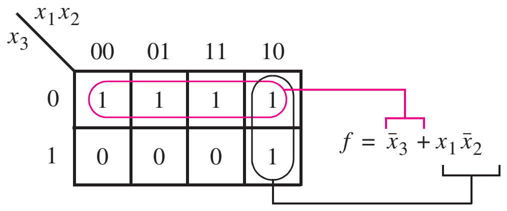
$x_3 \backslash x_1x_2$	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

(b) Karnaugh map

Figure 2.51 Location of three-variable minterms.



(a) The function of Figure 2.23



(b) The function of Figure 2.48

Figure 2.52 Examples of three-variable Karnaugh maps.

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 2.24*a*.

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 2.52*b*, using the function from Figure 2.48. 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 $\overline{x}_3$ represents these four cells. The remaining 1, corresponding to minterm m_5, is best covered by the term $x_1\overline{x}_2$, obtained by combining the two cells in the right-most column. The complete realization of f is

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

It is also possible to have a group of eight 1s in a three-variable map. This is the trivial case of a function 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 2.53 shows the structure of the four-variable map and the location 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 2.54 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

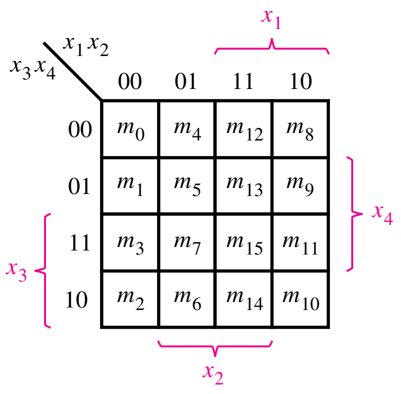


Figure 2.53 A four-variable Karnaugh map.

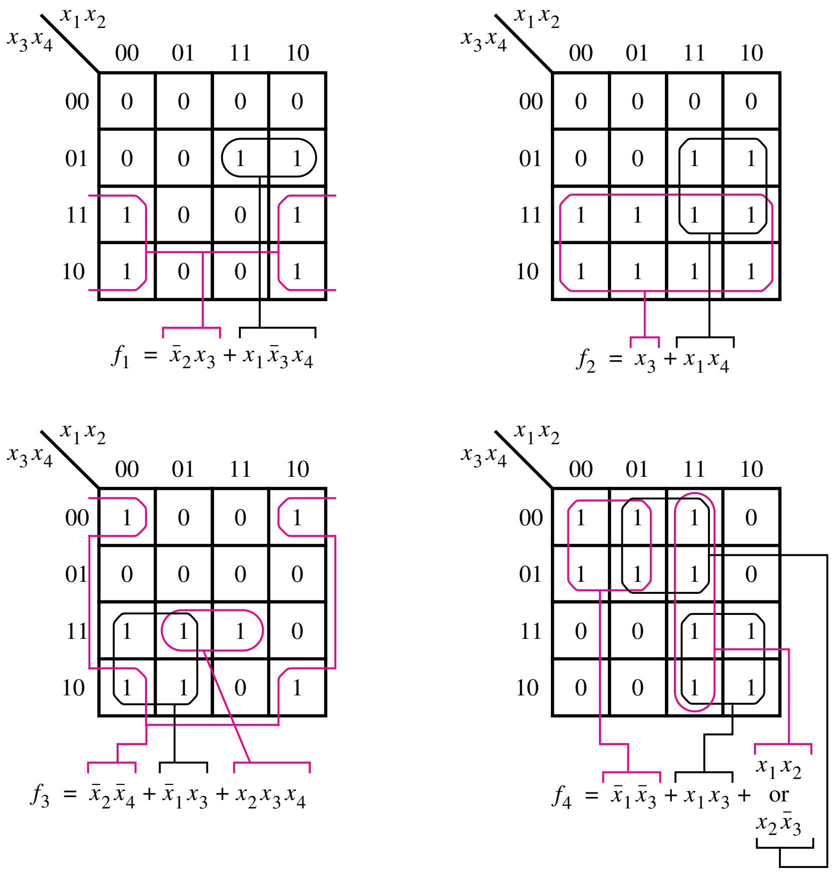

$f_1 = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$
$f_2 = x_3 + x_1 x_4$
$f_3 = \bar{x}_2 \bar{x}_4 + \bar{x}_1 x_3 + x_2 x_3 x_4$
$f_4 = \bar{x}_1 \bar{x}_3 + x_1 x_3 +$ $x_1 x_2$ or $x_2 \bar{x}_3$

Figure 2.54 Examples of four-variable Karnaugh maps.

represented by the product term $\bar{x}_2 x_3$. This leaves the two 1s in the second row to be covered, which can be accomplished with the term $x_1 \bar{x}_3 x_4$. Hence the minimum-cost implementation of the function is

$$f_1 = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$$

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 as a group of two, the result would be the product term $x_1 \bar{x}_3 x_4$. Implementing these 1s as a part of a group of four 1s, as shown in the figure, gives the less expensive product term $x_1 x_4$.

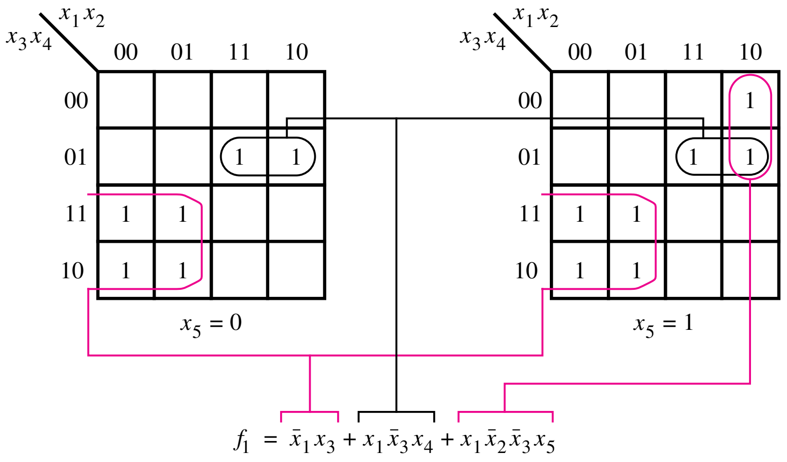
Just as the left and the right edges of the map are adjacent in terms of the assignment of the variables, so are the top and the bottom edges. Indeed, the four corners of the map are adjacent to each other and thus can form a group of four 1s, which may be implemented by the product term $\bar{x}_2 \bar{x}_4$. 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. 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 2.55. 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 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



x_1x_2
x_3x_4
00
01
11
10
1
$x_5 = 0$
$x_5 = 1$
$f_1 = \bar{x}_1x_3 + x_1\bar{x}_3x_4 + x_1\bar{x}_2\bar{x}_3x_5$

Figure 2.55 A five-variable Karnaugh map.

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.

2.12 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.

2.12.1 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 $x_1\overline{x}_2x_3$ has three literals, and the term $\overline{x}_1x_3\overline{x}_4x_6$ 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 2.56. There are 11 implicants for this function. This includes the five minterms: $\overline{x}_1\overline{x}_2\overline{x}_3$, $\overline{x}_1\overline{x}_2x_3$, $\overline{x}_1x_2\overline{x}_3$, $\overline{x}_1x_2x_3$, and $x_1x_2x_3$. Then there are the implicants that correspond to all possible pairs of minterms that can be combined, namely, $\overline{x}_1\overline{x}_2$ (m_0 and m_1), $\overline{x}_1\overline{x}_3$ (m_0 and m_2), $\overline{x}_1x_3$ (m_1 and m_3), $\overline{x}_1x_2$ (m_2 and m_3),

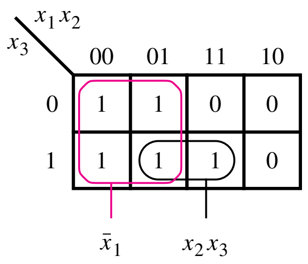


Figure 2.56 Three-variable function $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$.

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 $\bar{x}_1$.

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 2.56 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 $f = 0$. Another way of thinking about prime implicants is that they represent "the largest groups of 1s" that can be circled in the Karnaugh map.

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 2.56 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 Section 2.6.1 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

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

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

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

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

2.12.2 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.

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 2.56, both prime implicants are essential. The term x_2x_3 is the only prime implicant that covers the minterm m_7, and $\overline{x}_1$ 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

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

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 2.57. There are five prime implicants: $\overline{x}_1x_3$, $\overline{x}_2x_3$, $x_3\overline{x}_4$, $\overline{x}_1x_2x_4$, and $x_2\overline{x}_3x_4$. The essential ones (highlighted in blue) are $\overline{x}_2x_3$ (because of m_{11}), $x_3\overline{x}_4$ (because of m_{14}), and $x_2\overline{x}_3x_4$ (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 $\overline{x}_1x_3$ or $\overline{x}_1x_2x_4$.

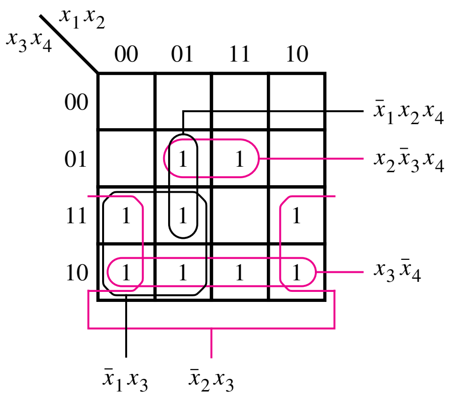


Figure 2.57 Four-variable function $f(x_1, \ldots, x_4) = \sum m(2, 3, 5, 6, 7, 10, 11, 13, 14)$.

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.
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 2.58. 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 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},

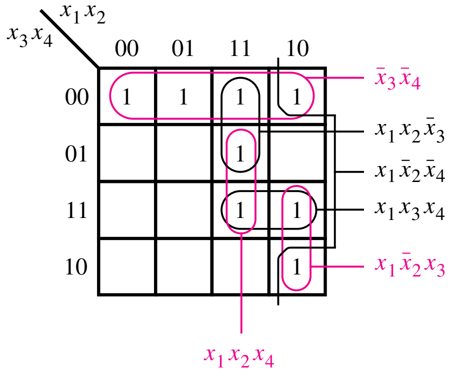


Figure 2.58 The function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$.

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 2.59. 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 Chapter 8.

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.

2.13 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

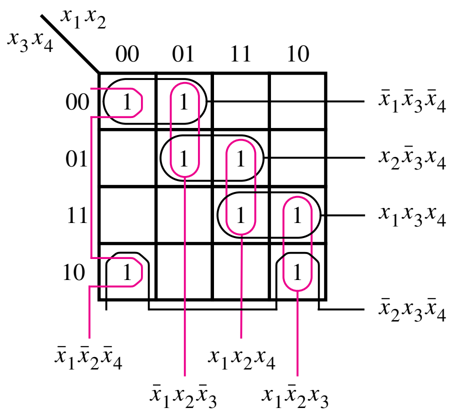


Figure 2.59 The function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 10, 11, 13, 15)$.

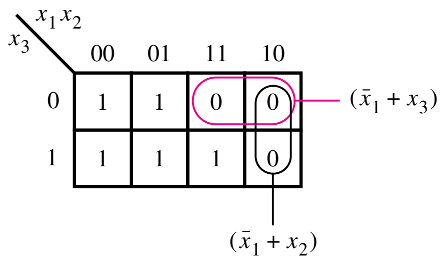


Figure 2.60 POS minimization of $f(x_1, x_2, x_3) = \Pi M(4, 5, 6)$.

$f = 0$ 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 2.60 depicts the same function as Figure 2.56 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 2.56, which requires only one OR gate and one AND gate.

The function from Figure 2.57 is reproduced in Figure 2.61. The maxterms for which $f = 0$ can be covered as shown, leading to the expression

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

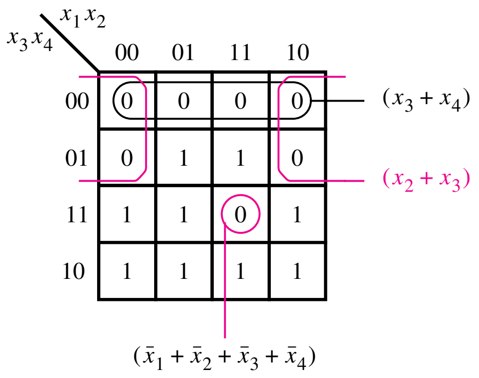


Figure 2.61 POS minimization of $f(x_1, \ldots, x_4) = \Pi M(0, 1, 4, 8, 9, 12, 15)$.

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 2.57, 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 2.58 and 2.59 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 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 $f = \overline{\overline{f}}$. For example, the simplest SOP implementation of $\overline{f}$ in Figure 2.60 is

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

Complementing this expression using DeMorgan's theorem yields

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

which is the same result as obtained above.

Using this approach for the function in Figure 2.61 gives

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

Complementing this expression produces

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

which matches the previously-derived implementation.

2.14 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*.

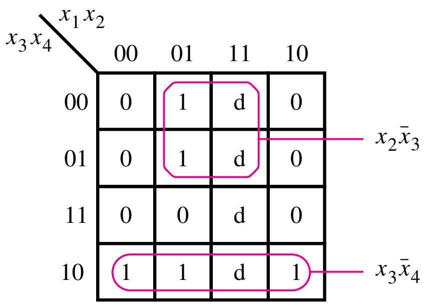
Don't-care conditions, or *don't-cares* for short, can be used to advantage in the design of logic circuits. Since these input valuations will never occur, the designer may assume that the function value for these valuations is either 1 or 0, whichever is more useful in trying to find a minimum-cost implementation. Figure 2.62 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-cares are denoted by the letter d in the map. Using the shorthand notation, the function f is specified as

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

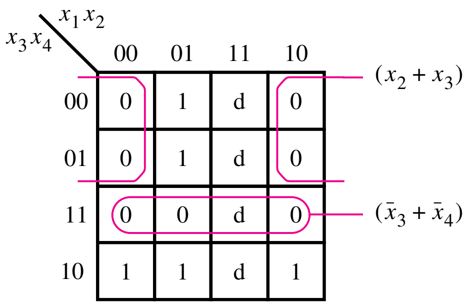
where D is the set of don't-cares.

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

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



(a) SOP implementation



(b) POS implementation

Figure 2.62 Two implementations of the function $f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$.

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

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

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 2.26 in Section 2.17 illustrates this possibility.

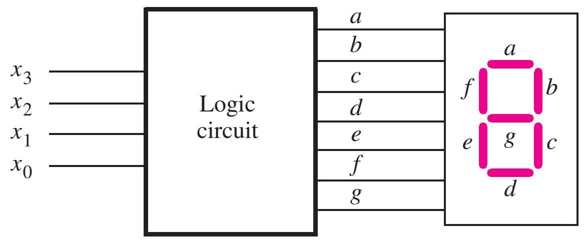
Using interlocked switches to illustrate how don't-care conditions can occur in a real system may seem to be somewhat contrived. A more practical example is shown below, and in future chapters we will encounter many examples of don't-cares that occur in the course of practical design of digital circuits.

Example 2.15 In Section 2.8.3 we designed a logic circuit that displays the decimal value of a two-bit number on a seven-segment display. In this example we will design a similar circuit, except that the input will be a four-bit number $X = x_3x_2x_1x_0$ that represents the decimal values 0, 1, . . . , 9. Using four bits to represent decimal digits in this way is often referred to as the *binary coded decimal* (BCD) representation. We discuss BCD numbers in detail in Chapter 3. The circuit is depicted in Figure 2.63*a*. Part (*b*) of the figure gives a truth table for each of the seven outputs $a, b, \ldots, g$, that control the display. It also indicates how the display should appear for each value of X. Since X is restricted to decimal digits, then the values of X from 1010 to 1111 are not used. These entries are omitted from the truth table in Figure 2.63*b*, and can be treated as don't cares in the design of the circuit.

To derive logic expressions for the outputs a to g, it is useful to draw Karnaugh maps. Figure 2.63*c* gives Karnaugh maps for the functions a and e. For the function a, the best result is obtained by setting all six don't-care cells in the map to the value of 1, which gives $a = \overline{x}_2\overline{x}_0 + x_1 + x_2x_0 + x_3$. However, for the function e a better choice is to set only two of the don't-care cells, corresponding to $x_3x_2x_1x_0 = 1010$ and $x_3x_2x_1x_0 = 1110$, to 1. The other don't-care cells should be set to 0 for this function to yield the minimum-cost expression $e = \overline{x}_2\overline{x}_0 + x_1\overline{x}_0$.

2.15 MULTIPLE-OUTPUT CIRCUITS

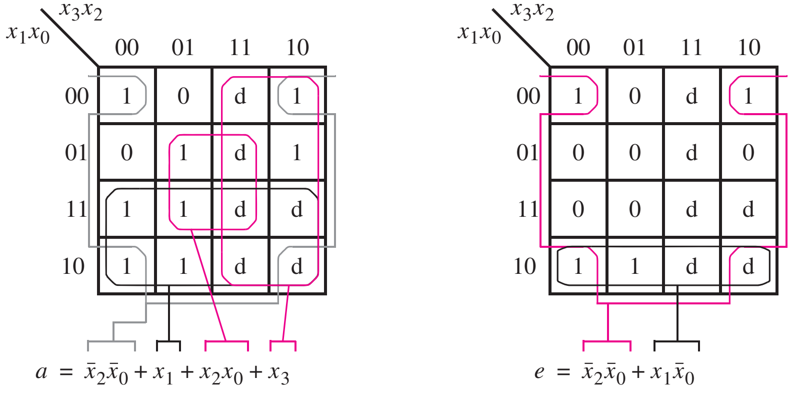
As illustrated in Example 2.15, in practical digital systems it is often necessary to implement a number of functions that are part of a larger logic circuit. Instead of implementing each of these functions separately, it may be possible to share some of the gates needed in the implementation of individual functions. For example, in Figure 2.63 the AND gate that produces $\overline{x}_2\overline{x}_0$ could be shared for use in both functions a and e.



(a) Logic circuit and 7-segment display

	x_3	x_2	x_1	x_0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

(b) Truth table


x_3x_2
x_1x_0
00 01 11 10
$a = \bar{x}_2\bar{x}_0 + x_1 + x_2x_0 + x_3$
$e = \bar{x}_2\bar{x}_0 + x_1\bar{x}_0$

(c) The Karnaugh maps for outputs a and e.

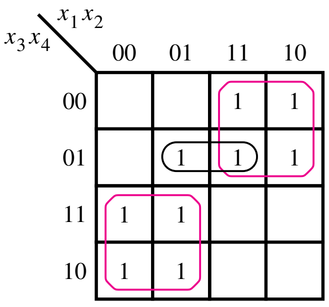
Figure 2.63 Using don't-care minterms when displaying BCD numbers.

Example 2.16 An example of gate sharing is given in Figure 2.64. Two functions, f_1 and f_2, of the same variables are to be implemented. The minimum-cost implementations for these functions are obtained as shown in parts (*a*) and (*b*) of the figure. This results in the expressions

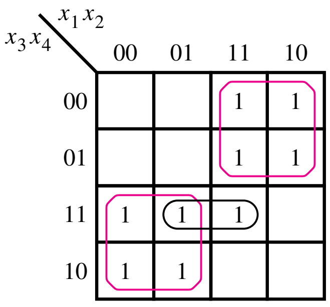
$$f_1 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2\bar{x}_3x_4$$

$$f_2 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2x_3x_4$$

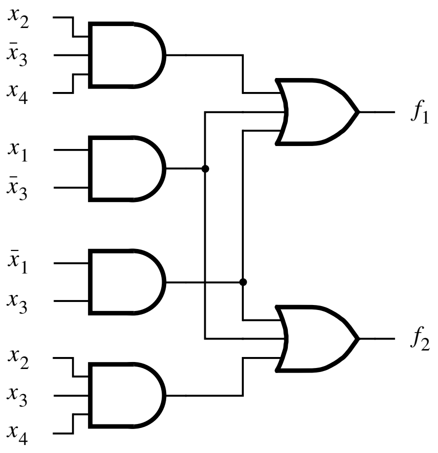
The cost of f_1 is four gates and 10 inputs, for a total of 14. The cost of f_2 is the same. Thus the total cost is 28 if both functions are implemented by separate circuits. A less-expensive realization is possible if the two circuits are combined into a single circuit with two outputs. Because the first two product terms are identical in both expressions, the AND gates that



(a) Function f_1



(b) Function f_2



(c) Combined circuit for f_1 and f_2

Figure 2.64 An example of multiple-output synthesis.

implement them need not be duplicated. The combined circuit is shown in Figure 2.64*c*. Its cost is six gates and 16 inputs, for a total of 22.

In this example we reduced the overall cost by finding minimum-cost realizations of f_1 and f_2 and then sharing the gates that implement the common product terms. This strategy does not necessarily always work the best, as the next example shows.

Example 2.17

Figure 2.65 shows two functions to be implemented by a single circuit. Minimum-cost realizations of the individual functions f_3 and f_4 are obtained from parts (*a*) and (*b*) of the figure.

$$f_3 = \overline{x}_1 x_4 + x_2 x_4 + \overline{x}_1 x_2 x_3$$
$$f_4 = x_1 x_4 + \overline{x}_2 x_4 + \overline{x}_1 x_2 x_3 \overline{x}_4$$

None of the AND gates can be shared, which means that the cost of the combined circuit would be six AND gates, two OR gates, and 21 inputs, for a total of 29.

But several alternative realizations are possible. Instead of deriving the expressions for f_3 and f_4 using only prime implicants, we can look for other implicants that may be shared advantageously in the combined realization of the functions. Figure 2.65*c* shows the best choice of implicants, which yields the realization

$$f_3 = x_1 x_2 x_4 + \overline{x}_1 x_2 x_3 \overline{x}_4 + \overline{x}_1 x_4$$
$$f_4 = x_1 x_2 x_4 + \overline{x}_1 x_2 x_3 \overline{x}_4 + \overline{x}_2 x_4$$

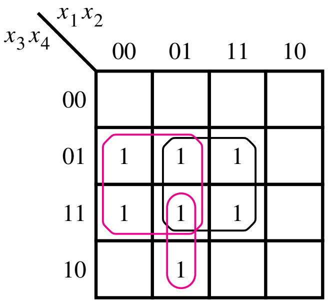
The first two implicants are identical in both expressions. The resulting circuit is given in Figure 2.65*d*. It has the cost of six gates and 17 inputs, for a total of 23.

Example 2.18

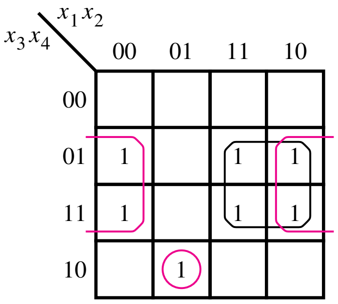
In Example 2.16 we sought the best SOP implementation for the functions f_1 and f_2 in Figure 2.64. We will now consider the POS implementation of the same functions. The minimum-cost POS expressions for f_1 and f_2 are

$$f_1 = (\overline{x}_1 + \overline{x}_3)(x_1 + x_2 + x_3)(x_1 + x_3 + x_4)$$
$$f_2 = (x_1 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_3 + x_4)$$

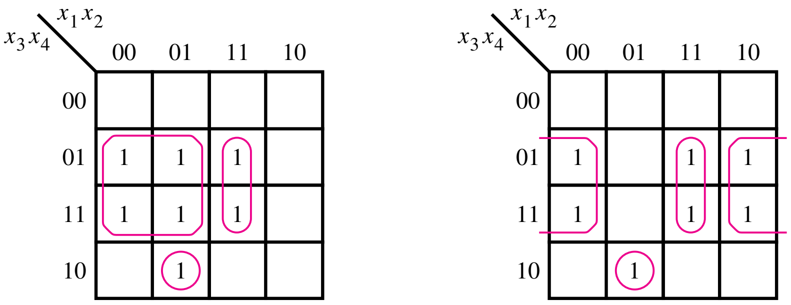
There are no common sum terms in these expressions that could be shared in the implementation. Moreover, from the Karnaugh maps in Figure 2.64, it is apparent that there is no sum term (covering the cells where $f_1 = f_2 = 0$) that can be profitably used in realizing both f_1 and f_2. Thus the best choice is to implement each function separately, according to the preceding expressions. Each function requires three OR gates, one AND gate, and 11 inputs. Therefore, the total cost of the circuit that implements both functions is 30. This realization is costlier than the SOP realization derived in Example 2.16.



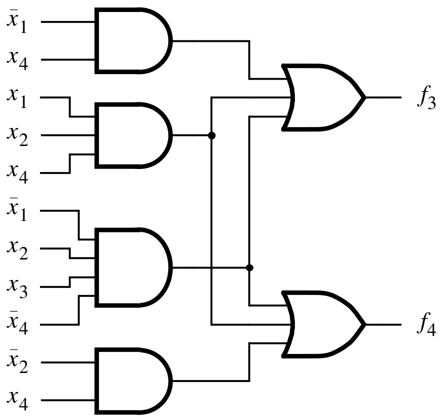
(a) Optimal realization of f_3



(b) Optimal realization of f_4



(c) Optimal realization of f_3 and f_4 together



(d) Combined circuit for f_3 and f_4

Figure 2.65 Another example of multiple-output synthesis.

Example 2.19

Consider now the POS realization of the functions f_3 and f_4 in Figure 2.65. The minimum-cost POS expressions for f_3 and f_4 are

$$f_3 = (x_3 + x_4)(x_2 + x_4)(\overline{x}_1 + x_4)(\overline{x}_1 + x_2)$$
$$f_4 = (x_3 + x_4)(x_2 + x_4)(\overline{x}_1 + x_4)(x_1 + \overline{x}_2 + \overline{x}_4)$$

The first three sum terms are the same in both f_3 and f_4; they can be shared in a combined circuit. These terms require three OR gates and six inputs. In addition, one 2-input OR gate and one 4-input AND gate are needed for f_3, and one 3-input OR gate and one 4-input AND gate are needed for f_4. Thus the combined circuit comprises five OR gates, two AND gates, and 19 inputs, for a total cost of 26. This cost is slightly higher than the cost of the circuit derived in Example 2.17.

These examples show that the complexities of the best SOP or POS implementations of given functions may be quite different. For the functions in Figures 2.64 and 2.65, the SOP form gives better results. But if we are interested in implementing the complements of the four functions in these figures, then the POS form would be less costly.

Sophisticated CAD tools used to synthesize logic functions will automatically perform the types of optimizations illustrated in the preceding examples.

2.16 Concluding Remarks

In this chapter we introduced the concept of logic circuits. We showed that such circuits can be implemented using logic gates and that they can be described using a mathematical model called Boolean algebra. Because practical logic circuits are often large, it is important to have good CAD tools to help the designer. We introduced the Verilog hardware description language that can be used to specify circuits for use in a CAD tool. We urge the reader to start using CAD software for the design of logic circuits as soon as possible.

This chapter has attempted to provide the reader with an understanding of some aspects of synthesis and optimization for logic functions. Now that the reader is comfortable with the fundamental concepts, we can examine digital circuits of a more sophisticated nature. The next chapter describes circuits that perform arithmetic operations, which are a key part of computers.

2.17 Examples of Solved Problems

This section presents some typical problems that the reader may encounter, and shows how such problems can be solved.

Example 2.20 **Problem:** Determine if the following equation is valid

$$\overline{x}_1\overline{x}_3 + x_2x_3 + x_1\overline{x}_2 = \overline{x}_1x_2 + x_1x_3 + \overline{x}_2\overline{x}_3$$

Solution: The equation is valid if the expressions on the left- and right-hand sides represent the same function. To perform the comparison, we could construct a truth table for each side and see if the truth tables are the same. An algebraic approach is to derive a canonical sum-of-products form for each expression.

Using the fact that $x + \overline{x} = 1$ (theorem 8*b*), we can manipulate the left-hand side as follows:

$$\begin{aligned}\text{LHS} &= \overline{x}_1\overline{x}_3 + x_2x_3 + x_1\overline{x}_2\\ &= \overline{x}_1(x_2+\overline{x}_2)\overline{x}_3 + (x_1+\overline{x}_1)x_2x_3 + x_1\overline{x}_2(x_3+\overline{x}_3)\\ &= \overline{x}_1x_2\overline{x}_3 + \overline{x}_1\overline{x}_2\overline{x}_3 + x_1x_2x_3 + \overline{x}_1x_2x_3 + x_1\overline{x}_2x_3 + x_1\overline{x}_2\overline{x}_3\end{aligned}$$

These product terms represent the minterms 2, 0, 7, 3, 5, and 4, respectively.

For the right-hand side we have

$$\begin{aligned}\text{RHS} &= \overline{x}_1x_2 + x_1x_3 + \overline{x}_2\overline{x}_3\\ &= \overline{x}_1x_2(x_3+\overline{x}_3) + x_1(x_2+\overline{x}_2)x_3 + (x_1+\overline{x}_1)\overline{x}_2\overline{x}_3\\ &= \overline{x}_1x_2x_3 + \overline{x}_1x_2\overline{x}_3 + x_1x_2x_3 + x_1\overline{x}_2x_3 + x_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2\overline{x}_3\end{aligned}$$

These product terms represent the minterms 3, 2, 7, 5, 4, and 0, respectively. Since both expressions specify the same minterms, they represent the same function; therefore, the equation is valid. Another way of representing this function is by $\sum m(0, 2, 3, 4, 5, 7)$.

Example 2.21 **Problem:** Design the minimum-cost product-of-sums expression for the function $f(x_1, x_2, x_3, x_4) = \sum m(0, 2, 4, 5, 6, 7, 8, 10, 12, 14, 15)$.

Solution: The function is defined in terms of its minterms. To find a POS expression we should start with the definition in terms of maxterms, which is $f = \Pi M(1, 3, 9, 11, 13)$. Thus,

$$\begin{aligned}f &= M_1 \cdot M_3 \cdot M_9 \cdot M_{11} \cdot M_{13}\\ &= (x_1+x_2+x_3+\overline{x}_4)(x_1+x_2+\overline{x}_3+\overline{x}_4)(\overline{x}_1+x_2+x_3+\overline{x}_4)(\overline{x}_1+x_2+\overline{x}_3+\overline{x}_4)(\overline{x}_1+\overline{x}_2+x_3+\overline{x}_4)\end{aligned}$$

We can rewrite the product of the first two maxterms as

$$\begin{aligned}M_1 \cdot M_3 &= (x_1+x_2+\overline{x}_4+x_3)(x_1+x_2+\overline{x}_4+\overline{x}_3) && \text{using commutative property } 10b\\ &= x_1+x_2+\overline{x}_4+x_3\overline{x}_3 && \text{using distributive property } 12b\\ &= x_1+x_2+\overline{x}_4+0 && \text{using theorem } 8a\\ &= x_1+x_2+\overline{x}_4 && \text{using theorem } 6b\end{aligned}$$

Similarly, $M_9 \cdot M_{11} = \overline{x}_1 + x_2 + \overline{x}_4$. Now, we can use M_9 again, according to property 7*a*, to derive $M_9 \cdot M_{13} = \overline{x}_1 + x_3 + \overline{x}_4$. Hence

$$f = (x_1+x_2+\overline{x}_4)(\overline{x}_1+x_2+\overline{x}_4)(\overline{x}_1+x_3+\overline{x}_4)$$

Applying 12b again, we get the final answer

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

Problem: A circuit that controls a given digital system has three inputs: x_1, x_2, and x_3. It has to recognize three different conditions: **Example 2.22**

- Condition A is true if x_3 is true and either x_1 is true or x_2 is false
- Condition B is true if x_1 is true and either x_2 or x_3 is false
- Condition C is true if x_2 is true and either x_1 is true or x_3 is false

The control circuit must produce an output of 1 if at least two of the conditions A, B, and C are true. Design the simplest circuit that can be used for this purpose.

Solution: Using 1 for true and 0 for false, we can express the three conditions as follows:

$$\begin{aligned} A = x_3(x_1 + \overline{x}_2) &= x_3x_1 + x_3\overline{x}_2 \\ B = x_1(\overline{x}_2 + \overline{x}_3) &= x_1\overline{x}_2 + x_1\overline{x}_3 \\ C = x_2(x_1 + \overline{x}_3) &= x_2x_1 + x_2\overline{x}_3 \end{aligned}$$

Then, the desired output of the circuit can be expressed as $f = AB + AC + BC$. These product terms can be determined as:

$$\begin{aligned} AB &= (x_3x_1 + x_3\overline{x}_2)(x_1\overline{x}_2 + x_1\overline{x}_3) \\ &= x_3x_1x_1\overline{x}_2 + x_3x_1x_1\overline{x}_3 + x_3\overline{x}_2x_1\overline{x}_2 + x_3\overline{x}_2x_1\overline{x}_3 \\ &= x_3x_1\overline{x}_2 + 0 + x_3\overline{x}_2x_1 + 0 \\ &= x_1\overline{x}_2x_3 \end{aligned}$$

$$\begin{aligned} AC &= (x_3x_1 + x_3\overline{x}_2)(x_2x_1 + x_2\overline{x}_3) \\ &= x_3x_1x_2x_1 + x_3x_1x_2\overline{x}_3 + x_3\overline{x}_2x_2x_1 + x_3\overline{x}_2x_2\overline{x}_3 \\ &= x_3x_1x_2 + 0 + 0 + 0 \\ &= x_1x_2x_3 \end{aligned}$$

$$\begin{aligned} BC &= (x_1\overline{x}_2 + x_1\overline{x}_3)(x_2x_1 + x_2\overline{x}_3) \\ &= x_1\overline{x}_2x_2x_1 + x_1\overline{x}_2x_2\overline{x}_3 + x_1\overline{x}_3x_2x_1 + x_1\overline{x}_3x_2\overline{x}_3 \\ &= 0 + 0 + x_1\overline{x}_3x_2 + x_1\overline{x}_3x_2 \\ &= x_1x_2\overline{x}_3 \end{aligned}$$

Therefore, f can be written as

$$\begin{aligned} f &= x_1\overline{x}_2x_3 + x_1x_2x_3 + x_1x_2\overline{x}_3 \\ &= x_1(\overline{x}_2 + x_2)x_3 + x_1x_2(x_3 + \overline{x}_3) \\ &= x_1x_3 + x_1x_2 \\ &= x_1(x_3 + x_2) \end{aligned}$$

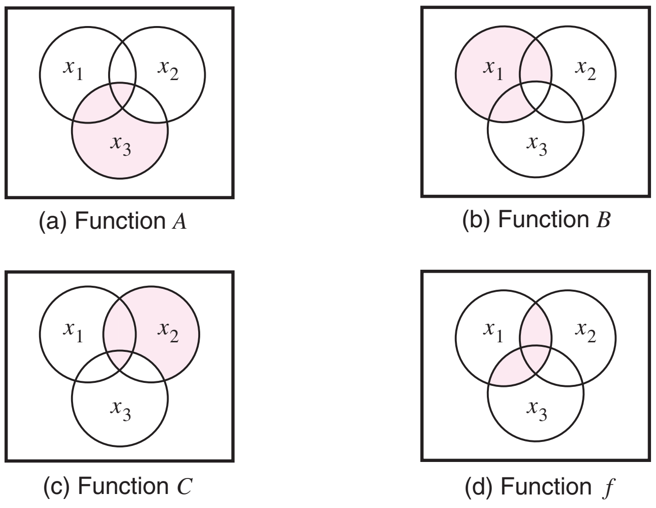


Figure 2.66 The Venn diagrams for Example 2.23.

Example 2.23 **Problem:** Solve the problem in Example 2.22 by using Venn diagrams.

Solution: The Venn diagrams for functions A, B, and C in Example 2.22 are shown in parts a to c of Figure 2.66. Since the function f has to be true when two or more of A, B, and C are true, then the Venn diagram for f is formed by identifying the common shaded areas in the Venn diagrams for A, B, and C. Any area that is shaded in two or more of these diagrams is also shaded in f, as shown in Figure 2.66d. This diagram corresponds to the function

$$f = x_1x_2 + x_1x_3 = x_1(x_2 + x_3)$$

Example 2.24 **Problem:** Use algebraic manipulation to derive the simplest sum-of-products expression for the function

$$f = x_2\overline{x}_3x_4 + x_1x_3x_4 + x_1\overline{x}_2x_4$$

Solution: Applying the consensus property 17a to the first two terms yields

$$\begin{aligned} f &= x_2\overline{x}_3x_4 + x_1x_3x_4 + x_2x_4x_1x_4 + x_1\overline{x}_2x_4 \\ &= x_2\overline{x}_3x_4 + x_1x_3x_4 + x_1x_2x_4 + x_1\overline{x}_2x_4 \end{aligned}$$

Now, using the combining property 14a for the last two terms gives

$$f = x_2\overline{x}_3x_4 + x_1x_3x_4 + x_1x_4$$

Finally, using the absorption property 13a produces

$$f = x_2\overline{x}_3x_4 + x_1x_4$$

Example 2.25

Problem: Use algebraic manipulation to derive the simplest product-of-sums expression for the function

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

Solution: Applying the consensus property 17*b* to the first two terms yields

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

Now, using the combining property 14*b* for the last two terms gives

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

Finally, using the absorption property 13*b* on the first and last terms produces

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

Example 2.26

Problem: Determine the minimum-cost SOP and POS expressions for the function $f(x_1, x_2, x_3, x_4) = \sum m(4, 6, 8, 10, 11, 12, 15) + D(3, 5, 7, 9)$.

Solution: The function can be represented in the form of a Karnaugh map as shown in Figure 2.67*a*. Note that the location of minterms in the map is as indicated in Figure 2.53. To find the minimum-cost SOP expression, it is necessary to find the prime implicants that cover all 1s in the map. The don't-cares may be used as desired. Minterm m_6 is covered only by the prime implicant $\overline{x}_1 x_2$, hence this prime implicant is essential and it must be included in the final expression. Similarly, the prime implicants $x_1\overline{x}_2$ and $x_3 x_4$ are essential because they are the only ones that cover m_{10} and m_{15}, respectively. These three prime implicants cover all minterms for which $f = 1$ except m_{12}. This minterm can be covered in two ways, by choosing either $x_1\overline{x}_3\overline{x}_4$ or $x_2\overline{x}_3\overline{x}_4$. Since both of these prime implicants have the same cost, we can choose either of them. Choosing the former, the desired SOP expression is

$$f = \overline{x}_1 x_2 + x_1\overline{x}_2 + x_3 x_4 + x_1\overline{x}_3\overline{x}_4$$

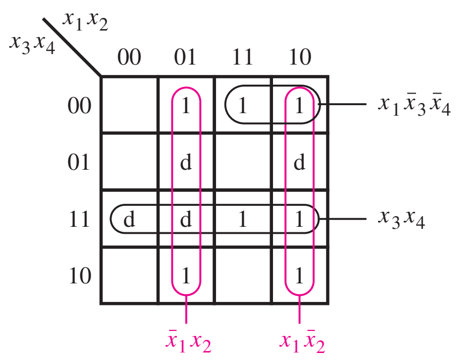
These prime implicants are encircled in the map.

The desired POS expression can be found as indicated in Figure 2.64*b*. In this case, we have to find the sum terms that cover all 0s in the function. Note that we have written 0s explicitly in the map to emphasize this fact. The term $(x_1 + x_2)$ is essential to cover the 0s in squares 0 and 2, which correspond to maxterms M_0 and M_2. The terms $(x_3 + \overline{x}_4)$ and $(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + x_4)$ must be used to cover the 0s in squares 13 and 14, respectively. Since these three sum terms cover all 0s in the map, the POS expression is

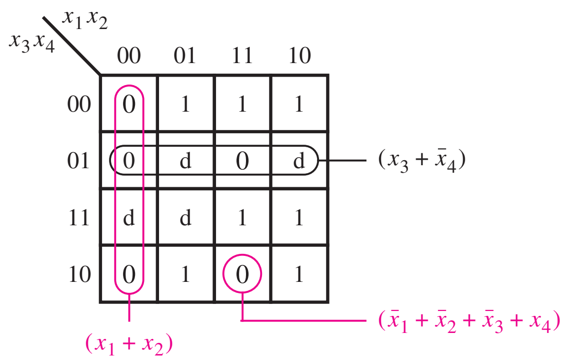
$$f = (x_1 + x_2)(x_3 + \overline{x}_4)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + x_4)$$

The chosen sum terms are encircled in the map.

Observe the use of don't-cares in this example. To get a minimum-cost SOP expression we assumed that all four don't-cares have the value 1. But, the minimum-cost POS expression becomes possible only if we assume that don't-cares 3, 5, and 9 have the value 0 while



(a) Determination of the SOP expression



(b) Determination of the POS expression

Figure 2.67 Karnaugh maps for Example 2.26.

the don't-care 7 has the value 1. This means that the resulting SOP and POS expressions are not identical in terms of the functions they represent. They cover identically all valuations for which f is specified as 1 or 0, but they differ in the valuations 3, 5, and 9. Of course, this difference does not matter, because the don't-care valuations will never be applied as inputs to the implemented circuits.

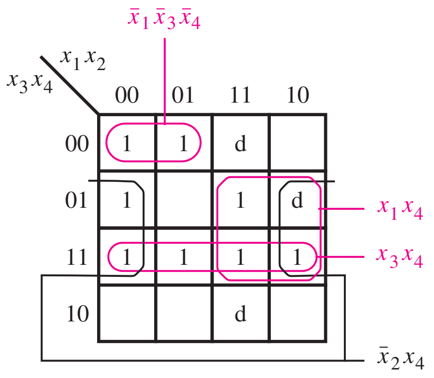
Example 2.27 **Problem:** Use Karnaugh maps to find the minimum-cost SOP and POS expressions for the function

$$f(x_1, \ldots, x_4) = \bar{x}_1\bar{x}_3\bar{x}_4 + x_3x_4 + \bar{x}_1\bar{x}_2x_4 + x_1x_2\bar{x}_3x_4$$

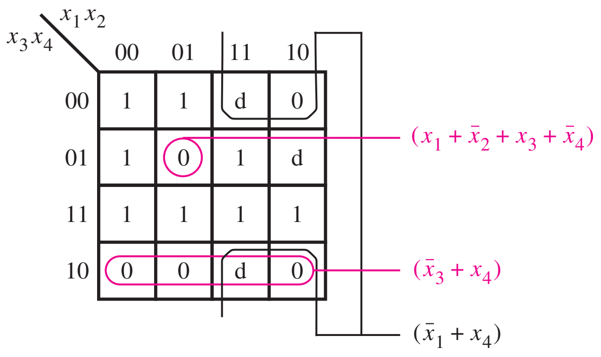
assuming that there are also don't-cares defined as $D = \sum(9, 12, 14)$.

Solution: The Karnaugh map that represents this function is shown in Figure 2.68*a*. The map is derived by placing 1s that correspond to each product term in the expression used to specify f. The term $\bar{x}_1\bar{x}_3\bar{x}_4$ corresponds to minterms 0 and 4. The term x_3x_4 represents the third row in the map, comprising minterms 3, 7, 11, and 15. The term $\bar{x}_1\bar{x}_2x_4$ specifies minterms 1 and 3. The fourth product term represents the minterm 13. The map also includes the three don't-care conditions.

To find the desired SOP expression, we must find the least-expensive set of prime implicants that covers all 1s in the map. The term x_3x_4 is a prime implicant which must be included because it is the only prime implicant that covers the minterm 7; it also covers minterms 3, 11, and 15. Minterm 4 can be covered with either $\bar{x}_1\bar{x}_3\bar{x}_4$ or $x_2\bar{x}_3\bar{x}_4$. Both of these terms have the same cost; we will choose $\bar{x}_1\bar{x}_3\bar{x}_4$ because it also covers the minterm 0. Minterm 1 may be covered with either $\bar{x}_1\bar{x}_2\bar{x}_3$ or $\bar{x}_2x_4$; we should choose the latter because its cost is lower. This leaves only the minterm 13 to be covered, which can be done with



(a) Determination of the SOP expression



(b) Determination of the POS expression

Figure 2.68 Karnaugh maps for Example 2.27.

either x_1x_4 or x_1x_2 at equal costs. Choosing x_1x_4, the minimum-cost SOP expression is

$$f = x_3x_4 + \bar{x}_1\bar{x}_3\bar{x}_4 + \bar{x}_2x_4 + x_1x_4$$

Figure 2.68*b* shows how we can find the POS expression. The sum term $(\bar{x}_3 + x_4)$ covers the 0s in the bottom row. To cover the 0 in square 8 we must include $(\bar{x}_1 + x_4)$. The remaining 0, in square 5, must be covered with $(x_1 + \bar{x}_2 + x_3 + \bar{x}_4)$. Thus, the minimum-cost POS expression is

$$f = (\bar{x}_3 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + x_3 + \bar{x}_4)$$

Example 2.28 **Problem:** Consider the expression

$$f = s_3(\bar{s}_1 + \bar{s}_2) + s_1s_2$$

Derive a minimum-cost SOP expression for f.

Solution: Applying the distributive property 12*a* we can write

$$f = \bar{s}_1s_3 + \bar{s}_2s_3 + s_1s_2$$

Now, it is easy to see how f can be represented in the form of a Karnaugh map, as depicted in Figure 2.69. The figure shows that minterms from each of the above three product terms cover the bottom row of the Karnaugh map, leading to the minimum-cost expression

$$f = s_3 + s_1s_2$$

Example 2.29 Write Verilog code that represents the logic circuit in Figure 2.70. Use only continuous assignment statements to specify the required functions.

Solution: An example of the required Verilog code is shown in Figure 2.71.

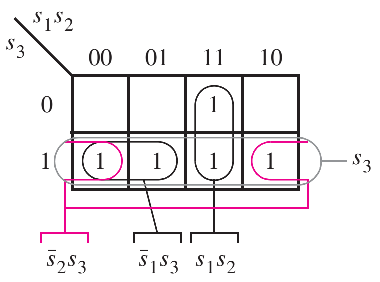


Figure 2.69 A Karnaugh map that represents the function in Example 2.28.

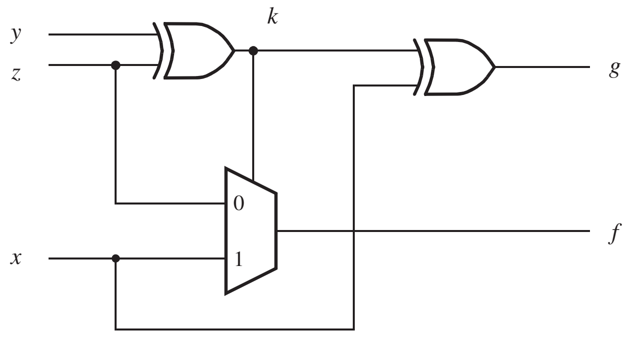


Figure 2.70 The logic circuit for Example 2.29.

```
module f_g (x, y, z, f, g);
    input x, y, z;
    output f, g;
    wire k;

    assign k = y ^ z;
    assign g = k ^ x;
    assign f = (~k & z) | (k & x);

endmodule
```

Figure 2.71 Verilog code for Example 2.29.

Example 2.30

Consider the circuit shown in Figure 2.72. It includes two copies of the 2-to-1 multiplexer circuit from Figure 2.33, and the adder circuit from Figure 2.12. If the multiplexers' select input $m = 0$, then this circuit produces the sum $S = a + b$. But if $m = 1$, then the circuit produces $S = c + d$. By using multiplexers, we are able to *share* one adder for generating the two different sums $a + b$ and $c + d$. Sharing a subcircuit for multiple purposes is commonly-used in practice, although usually with larger subcircuits than our one-bit adder. Write Verilog code for the circuit in Figure 2.72. Use the hierarchical style of Verilog code illustrated in Figure 2.47, including two instances of a Verilog module for the 2-to-1 multiplexer subcircuit, and one instance of the adder subcircuit.

Solution: An example of the required Verilog code is shown in Figure 2.73. We should mention that both the *shared* module and the *adder* module have input ports named *a* and *b*. This does not cause any conflict because the name of a signal declared in a Verilog module is limited to the scope of that module.

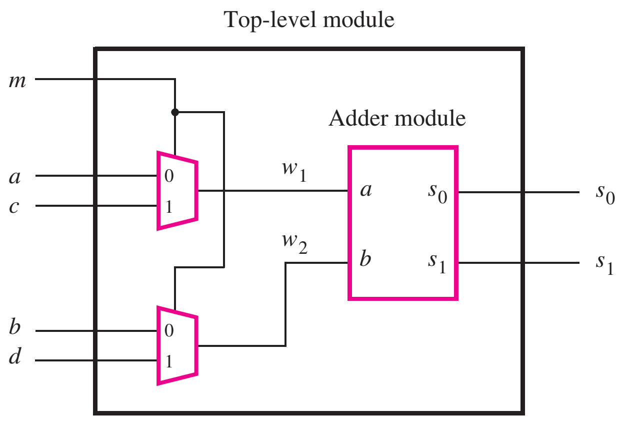


Figure 2.72 The circuit for Example 2.30.

```
module  shared (a, b, c, d, m, s1, s0);
    input a, b, c, d, m;
    output s1, s0;
    wire w1, w2;
    mux2to1 U1 (a, c, m, w1);
    mux2to1 U2 (b, d, m, w2);
    adder U3 (w1, w2, s1, s0);
endmodule

module  mux2to1 (x1, x2, s, f);
    input x1, x2, s;
    output f;
    assign f = (~s & x1) | (s & x2);
endmodule

module  adder (a, b, s1, s0);
    input a, b;
    output s1, s0;
    assign s1 = a & b;
    assign s0 = a ^ b;
endmodule
```

Figure 2.73 Verilog code for Example 2.30.

Example 2.31

In Chapter 1 we said that several types of integrated circuit chips are available for implementation of logic circuits, and that field-programmable gate arrays (FPGAs) are commonly used. In an FPGA, logic functions are not implemented directly using AND, OR, and NOT gates. Instead, an FPGA implements logic functions by using a type of circuit element called lookup tables (LUTs). A LUT can be programmed by the user to implement any logic function of its inputs. Thus, if a LUT has three inputs, then it can implement any logic function of these three inputs. We describe FPGAs and lookup tables in detail in Appendix B. Consider the four-input function

$$f = x_1x_2x_4 + x_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_3$$

Show how this function can be implemented in an FPGA that has three-input LUTs.

Solution: A straightforward implementation requires four 3-input LUTs. Three of these LUTs are used for the three-input AND operations, and the final LUT implements the three-input OR operation. It is also possible to apply logic synthesis techniques that result in fewer LUTs. We discuss these techniques in Chapter 8 (see Example 8.19).

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

2.1 Use algebraic manipulation to prove that $x + yz = (x + y) \cdot (x + z)$. Note that this is the distributive rule, as stated in identity 12*b* in Section 2.5.

2.2 Use algebraic manipulation to prove that $(x + y) \cdot (x + \overline{y}) = x$.

2.3 Use algebraic manipulation to prove that $xy + yz + \overline{x}z = xy + \overline{x}z$. Note that this is the consensus property 17*a* in Section 2.5.

2.4 Use the Venn diagram to prove the identity in Problem 2.3.

2.5 Use the Venn diagram to prove DeMorgan's theorem, as given in expression 15*b* in Section 2.5.

2.6 Use the Venn diagram to prove that

$$(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \overline{x}_3) = x_1 + x_2$$

***2.7** Determine whether or not the following expressions are valid, i.e., whether the left- and right-hand sides represent the same function.
(a) $\overline{x}_1x_3 + x_1x_2\overline{x}_3 + \overline{x}_1x_2 + x_1\overline{x}_2 = \overline{x}_2x_3 + x_1\overline{x}_3 + x_2\overline{x}_3 + \overline{x}_1x_2x_3$
(b) $x_1\overline{x}_3 + x_2x_3 + \overline{x}_2\overline{x}_3 = (x_1 + \overline{x}_2 + x_3)(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)$
(c) $(x_1 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2) = (x_1 + x_2)(x_2 + x_3)(\overline{x}_1 + \overline{x}_3)$

2.8 Draw a timing diagram for the circuit in Figure 2.24*a*. Show the waveforms that can be observed on all wires in the circuit.

2.9 Repeat Problem 2.8 for the circuit in Figure 2.24*b*.

2.10 Use algebraic manipulation to show that for three input variables x_1, x_2, and x_3

$$\sum m(1, 2, 3, 4, 5, 6, 7) = x_1 + x_2 + x_3$$

2.11 Use algebraic manipulation to show that for three input variables x_1, x_2, and x_3

$$\Pi M(0, 1, 2, 3, 4, 5, 6) = x_1 x_2 x_3$$

***2.12** Use algebraic manipulation to find the minimum sum-of-products expression for the function $f = x_1 x_3 + x_1 \overline{x}_2 + \overline{x}_1 x_2 x_3 + \overline{x}_1 \overline{x}_2 \overline{x}_3$.

2.13 Use algebraic manipulation to find the minimum sum-of-products expression for the function $f = x_1 \overline{x}_2 \overline{x}_3 + x_1 x_2 x_4 + x_1 \overline{x}_2 x_3 \overline{x}_4$.

2.14 Use algebraic manipulation to find the minimum product-of-sums expression for the function $f = (x_1 + x_3 + x_4) \cdot (x_1 + \overline{x}_2 + x_3) \cdot (x_1 + \overline{x}_2 + \overline{x}_3 + x_4)$.

***2.15** Use algebraic manipulation to find the minimum product-of-sums expression for the function $f = (x_1 + x_2 + x_3) \cdot (x_1 + \overline{x}_2 + x_3) \cdot (\overline{x}_1 + \overline{x}_2 + x_3) \cdot (x_1 + x_2 + \overline{x}_3)$.

2.16 (a) Show the location of all minterms in a three-variable Venn diagram.
(b) Show a separate Venn diagram for each product term in the function $f = x_1 \overline{x}_2 x_3 + x_1 x_2 + \overline{x}_1 x_3$. Use the Venn diagram to find the minimal sum-of-products form of f.

2.17 Represent the function in Figure 2.23 in the form of a Venn diagram and find its minimal sum-of-products form.

2.18 Figure P2.1 shows two attempts to draw a Venn diagram for four variables. For parts (*a*) and (*b*) of the figure, explain why the Venn diagram is not correct. (*Hint:* The Venn diagram must be able to represent all 16 minterms of the four variables.)

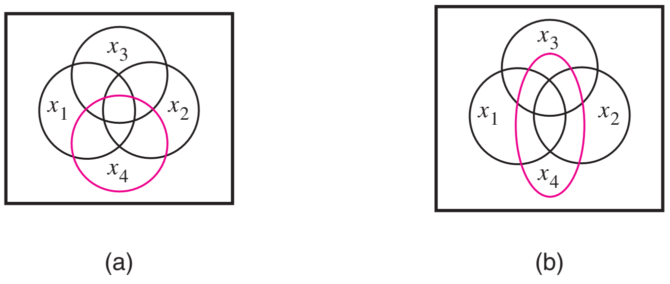


Figure P2.1 Two attempts to draw a four-variable Venn diagram.

2.19 Figure P2.2 gives a representation of a four-variable Venn diagram and shows the location of minterms m_0, m_1, and m_2. Show the location of the other minterms in the diagram. Represent the function $f = \overline{x}_1 \overline{x}_2 x_3 \overline{x}_4 + x_1 x_2 x_3 x_4 + \overline{x}_1 x_2$ on this diagram.

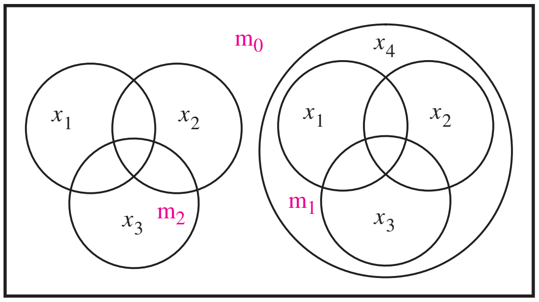


Figure P2.2 A four-variable Venn diagram.

***2.20** Design the simplest sum-of-products circuit that implements the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$.

2.21 Design the simplest sum-of-products circuit that implements the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$.

2.22 Design the simplest product-of-sums circuit that implements the function $f(x_1, x_2, x_3) = \Pi M(0, 2, 5)$.

***2.23** Design the simplest product-of-sums expression for the function $f(x_1, x_2, x_3) = \Pi M(0, 1, 5, 7)$.

2.24 Derive the simplest sum-of-products expression for the function $f(x_1, x_2, x_3, x_4) = x_1\overline{x}_3\overline{x}_4 + x_2\overline{x}_3x_4 + x_1\overline{x}_2\overline{x}_3$.

2.25 Use algebraic manipulation to derive the simplest sum-of-products expression for the function $f(x_1, x_2, x_3, x_4, x_5) = \overline{x}_1\overline{x}_3\overline{x}_5 + \overline{x}_1\overline{x}_3\overline{x}_4 + \overline{x}_1x_4x_5 + x_1\overline{x}_2\overline{x}_3x_5$. (*Hint:* Use the consensus property 17*a*.)

2.26 Use algebraic manipulation to derive the simplest product-of-sums expression for the function $f(x_1, x_2, x_3, x_4) = (\overline{x}_1 + \overline{x}_3 + \overline{x}_4)(\overline{x}_2 + \overline{x}_3 + x_4)(x_1 + \overline{x}_2 + \overline{x}_3)$. (*Hint:* Use the consensus property 17*b*.)

2.27 Use algebraic manipulation to derive the simplest product-of-sums expression for the function $f(x_1, x_2, x_3, x_4, x_5) = (\overline{x}_2 + x_3 + x_5)(x_1 + \overline{x}_3 + x_5)(x_1 + x_2 + x_5)(x_1 + \overline{x}_4 + \overline{x}_5)$. (*Hint:* Use the consensus property 17*b*.)

***2.28** Design the simplest circuit that has three inputs, x_1, x_2, and x_3, which produces an output value of 1 whenever two or more of the input variables have the value 1; otherwise, the output has to be 0.

2.29 Design the simplest circuit that has three inputs, x_1, x_2, and x_3, which produces an output value of 1 whenever exactly one or two of the input variables have the value 1; otherwise, the output has to be 0.

2.30 Design the simplest circuit that has four inputs, x_1, x_2, x_3, and x_4, which produces an output value of 1 whenever three or more of the input variables have the value 1; otherwise, the output has to be 0.

2.31 For the timing diagram in Figure P2.3, synthesize the function $f(x_1, x_2, x_3)$ in the simplest sum-of-products form.

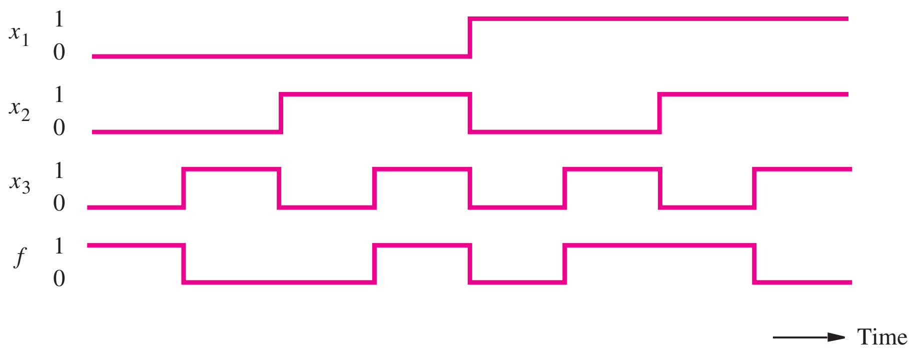


Figure P2.3 A timing diagram representing a logic function.

***2.32** For the timing diagram in Figure P2.3, synthesize the function $f(x_1, x_2, x_3)$ in the simplest product-of-sums form.

***2.33** For the timing diagram in Figure P2.4, synthesize the function $f(x_1, x_2, x_3)$ in the simplest sum-of-products form.

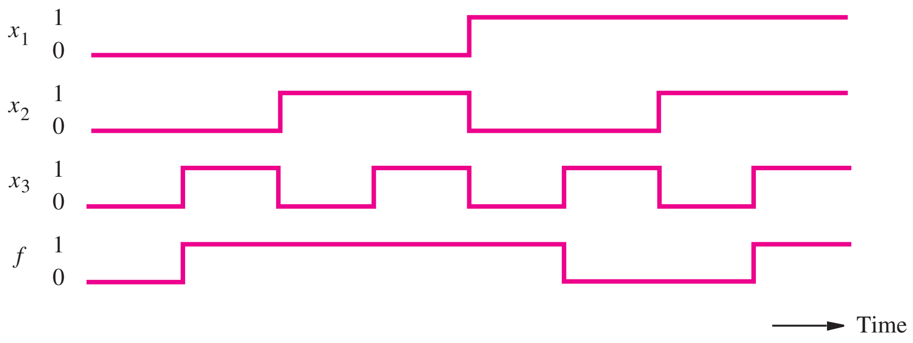


Figure P2.4 A timing diagram representing a logic function.

2.34 For the timing diagram in Figure P2.4, synthesize the function $f(x_1, x_2, x_3)$ in the simplest product-of-sums form.

2.35 Design a circuit with output f and inputs x_1, x_0, y_1, and y_0. Let $X = x_1x_0$ and $Y = y_1y_0$ represent two 2-digit binary numbers. The output f should be 1 if the numbers represented by X and Y are equal. Otherwise, f should be 0.

(a) Show the truth table for f.
(b) Synthesize the simplest possible product-of-sums expression for f.

2.36 Repeat Problem 2.35 for the case where f should be 1 only if $X \geq Y$.
(a) Show the truth table for f.
(b) Show the canonical sum-of-products expression for f.
(c) Show the simplest possible sum-of-products expression for f.

***2.37** Find the minimum-cost SOP and POS forms for the function $f(x_1, x_2, x_3) = \sum m(1, 2, 3, 5)$.

***2.38** Repeat Problem 2.37 for the function $f(x_1, x_2, x_3) = \sum m(1, 4, 7) + D(2, 5)$.

2.39 Repeat Problem 2.37 for the function $f(x_1, \ldots, x_4) = \Pi M(0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 14, 15)$.

2.40 Repeat Problem 2.37 for the function $f(x_1, \ldots, x_4) = \sum m(0, 2, 8, 9, 10, 15) + D(1, 3, 6, 7)$.

***2.41** Repeat Problem 2.37 for the function $f(x_1, \ldots, x_5) = \Pi M(1, 4, 6, 7, 9, 12, 15, 17, 20, 21, 22, 23, 28, 31)$.

2.42 Repeat Problem 2.37 for the function $f(x_1, \ldots, x_5) = \sum m(0, 1, 3, 4, 6, 8, 9, 11, 13, 14, 16, 19, 20, 21, 22, 24, 25) + D(5, 7, 12, 15, 17, 23)$.

2.43 Repeat Problem 2.37 for the function $f(x_1, \ldots, x_5) = \sum m(1, 4, 6, 7, 9, 10, 12, 15, 17, 19, 20, 23, 25, 26, 27, 28, 30, 31) + D(8, 16, 21, 22)$.

2.44 Find 5 three-variable functions for which the product-of-sums form has lower cost than the sum-of-products form.

***2.45** A four-variable logic function that is equal to 1 if any three or all four of its variables are equal to 1 is called a *majority* function. Design a minimum-cost SOP circuit that implements this majority function.

2.46 Derive a minimum-cost realization of the four-variable function that is equal to 1 if exactly two or exactly three of its variables are equal to 1; otherwise it is equal to 0.

***2.47** Prove or show a counter-example for the statement: If a function f has a unique minimum-cost SOP expression, then it also has a unique minimum-cost POS expression.

***2.48** A circuit with two outputs has to implement the following functions

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 6, 7, 9) + D(10, 11)$$

$$g(x_1, \ldots, x_4) = \sum m(2, 4, 9, 10, 15) + D(0, 13, 14)$$

Design the minimum-cost circuit and compare its cost with combined costs of two circuits that implement f and g separately. Assume that the input variables are available in both uncomplemented and complemented forms.

2.49 Repeat Problem 2.48 for the following functions

$$f(x_1, \ldots, x_5) = \sum m(1, 4, 5, 11, 27, 28) + D(10, 12, 14, 15, 20, 31)$$
$$g(x_1, \ldots, x_5) = \sum m(0, 1, 2, 4, 5, 8, 14, 15, 16, 18, 20, 24, 26, 28, 31) + D(10, 11, 12, 27)$$

***2.50** Consider the function $f = x_3x_5 + \overline{x}_1x_2x_4 + x_1\overline{x}_2\overline{x}_4 + x_1x_3\overline{x}_4 + \overline{x}_1x_3x_4 + \overline{x}_1x_2x_5 + x_1\overline{x}_2x_5$. Derive a minimum-cost POS expression for this function.

2.51 Implement the function in Figure 2.31 using only NAND gates.

2.52 Implement the function in Figure 2.31 using only NOR gates.

2.53 Implement the circuit in Figure 2.39 using NAND and NOR gates.

***2.54** Design the simplest circuit that implements the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$ using NAND gates.

2.55 Design the simplest circuit that implements the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$ using NAND gates.

***2.56** Repeat Problem 2.54 using NOR gates.

2.57 Repeat Problem 2.55 using NOR gates.

2.58 (a) Use a schematic capture tool (which can be downloaded from the Internet; for example, from www.altera.com) to draw schematics for the following functions

$$f_1 = x_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_2x_4 + \overline{x}_1x_2x_3 + x_1x_2x_3$$
$$f_2 = x_2\overline{x}_4 + \overline{x}_1x_2 + x_2x_3$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.59 (a) Use a schematic capture tool to draw schematics for the following functions

$$f_1 = (x_1 + x_2 + \overline{x}_4) \cdot (\overline{x}_2 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + x_3 + \overline{x}_4) \cdot (\overline{x}_1 + \overline{x}_3 + \overline{x}_4)$$
$$f_2 = (x_2 + \overline{x}_4) \cdot (x_3 + \overline{x}_4) \cdot (\overline{x}_1 + \overline{x}_4)$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.60 Write Verilog code to implement the circuit in Figure 2.32*a* using the gate-level primitives.

2.61 Repeat Problem 2.60 for the circuit in Figure 2.32*b*.

2.62 Write Verilog code to implement the function $f(x_1, x_2, x_3) = \sum m(1, 2, 3, 4, 5, 6)$ using the gate-level primitives. Ensure that the resulting circuit is as simple as possible.

2.63 Write Verilog code to implement the function $f(x_1, x_2, x_3) = \sum m(0, 1, 3, 4, 5, 6)$ using the continuous assignment.

2.64 (a) Write Verilog code to describe the following functions

$$f_1 = x_1\overline{x}_3 + x_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1x_2 + x_1\overline{x}_4$$
$$f_2 = (x_1 + \overline{x}_3) \cdot (x_1 + x_2 + \overline{x}_4) \cdot (x_2 + \overline{x}_3 + \overline{x}_4)$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.65 Consider the following Verilog statements

```
f1 = (x1 & x3) | (~x1 & ~x3) | (x2 & x4) | (~x2 & ~x4);
f2 = (x1 & x2 & ~x3 & ~x4) | (~x1 & ~x2 & x3 & x4) |
       (x1 & ~x2 & ~x3 & x4) | (~x1 & x2 & x3 & ~x4);
```

(a) Write complete Verilog code to implement f1 and f2.
(b) Use functional simulation to prove that f1 = $\overline{\text{f2}}$.

***2.66** Consider the logic expressions

$$f = x_1\overline{x}_2\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_4\overline{x}_5 + x_1x_2x_4x_5 + \overline{x}_1\overline{x}_2x_3\overline{x}_4 + x_1\overline{x}_2x_3x_5 + \overline{x}_2\overline{x}_3x_4\overline{x}_5 + x_1x_2x_3x_4\overline{x}_5$$
$$g = \overline{x}_2x_3\overline{x}_4 + \overline{x}_2\overline{x}_3\overline{x}_4\overline{x}_5 + x_1x_3x_4\overline{x}_5 + x_1\overline{x}_2x_4\overline{x}_5 + x_1x_3x_4x_5 + \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_5 + x_1x_2\overline{x}_3x_4x_5$$

Prove or disprove that $f = g$.

2.67 Repeat Problem 2.66 for the following expressions

$$f = x_1\overline{x}_2\overline{x}_3 + x_2x_4 + x_1\overline{x}_2\overline{x}_4 + \overline{x}_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_2x_3$$
$$g = (\overline{x}_2 + x_3 + x_4)(\overline{x}_1 + \overline{x}_2 + x_4)(x_2 + \overline{x}_3 + \overline{x}_4)(x_1 + x_2 + \overline{x}_3)(x_1 + x_2 + \overline{x}_4)$$

2.68 Repeat Problem 2.66 for the following expressions

$$f = x_2\overline{x}_3\overline{x}_4 + \overline{x}_2x_3 + \overline{x}_2x_4 + x_1x_2\overline{x}_4 + x_1x_2\overline{x}_3\overline{x}_5$$
$$g = (x_2 + x_3 + x_4)(\overline{x}_2 + \overline{x}_4 + x_5)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_2 + x_3 + \overline{x}_4 + \overline{x}_5)$$

2.69 A circuit with two outputs is defined by the logic functions

$$f = x_1\overline{x}_2\overline{x}_3 + \overline{x}_2\overline{x}_4 + \overline{x}_2\overline{x}_3x_4 + x_1x_2x_3x_4$$
$$g = x_1\overline{x}_3x_4 + x_1x_2x_4 + \overline{x}_1x_3x_4 + \overline{x}_2x_3\overline{x}_4$$

Derive a minimum-cost implementation of this circuit. What is the cost of your circuit?

2.70 Repeat Problem 2.69 for the functions

$$f = (\overline{x}_1 + x_2 + \overline{x}_3)(x_1 + x_3 + \overline{x}_4)(x_1 + \overline{x}_2 + x_3)(\overline{x}_1 + x_2 + x_4)(x_1 + \overline{x}_2 + \overline{x}_4)$$
$$g = (\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_4)(\overline{x}_2 + \overline{x}_3 + \overline{x}_4)(x_1 + \overline{x}_2 + x_3 + x_4)$$

2.71 A given system has four sensors that can produce an output of 0 or 1. The system operates properly when exactly one of the sensors has its output equal to 1. An alarm must be raised when two or more sensors have the output of 1. Design the simplest circuit that can be used to raise the alarm.

2.72 Repeat Problem 2.71 for a system that has seven sensors.

2.73 Find the minimum-cost circuit consisting only of two-input NAND gates for the function $f(x_1, \ldots, x_4) = \sum m(0, 1, 2, 3, 4, 6, 8, 9, 12)$. Assume that the input variables are available in both uncomplemented and complemented forms. (*Hint:* Consider the complement of the function.)

2.74 Repeat Problem 2.73 for the function $f(x_1, \ldots, x_4) = \sum m(2, 3, 6, 8, 9, 12)$.

2.75 Find the minimum-cost circuit consisting only of two-input NOR gates for the function $f(x_1, \ldots, x_4) = \sum m(6, 7, 8, 10, 12, 14, 15)$. Assume that the input variables are available in both uncomplemented and complemented forms. (*Hint:* Consider the complement of the function.)

2.76 Repeat Problem 2.75 for the function $f(x_1, \ldots, x_4) = \sum m(2, 3, 4, 5, 9, 10, 11, 12, 13, 15)$.

2.77 Consider the circuit in Figure P2.5, which implements functions f and g. What is the cost of this circuit, assuming that the input variables are available in both true and complemented forms? Redesign the circuit to implement the same functions, but at as low a cost as possible. What is the cost of your circuit?

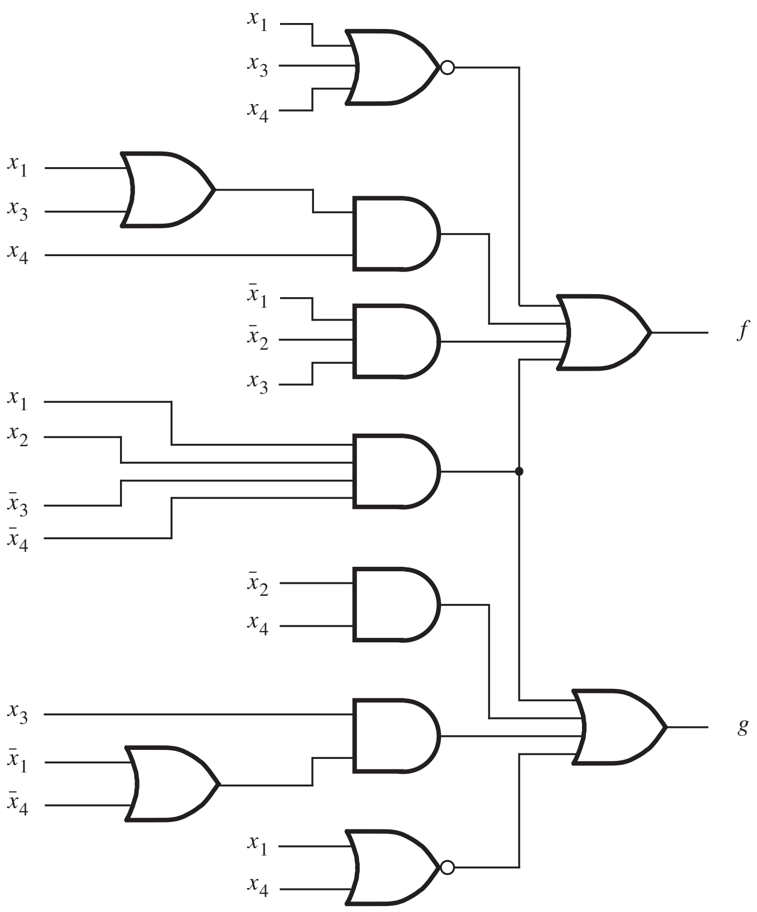


Figure P2.5 Circuit for Problem 2.78.

2.78 Repeat Problem 2.78 for the circuit in Figure P2.6. Use only NAND gates in your circuit.

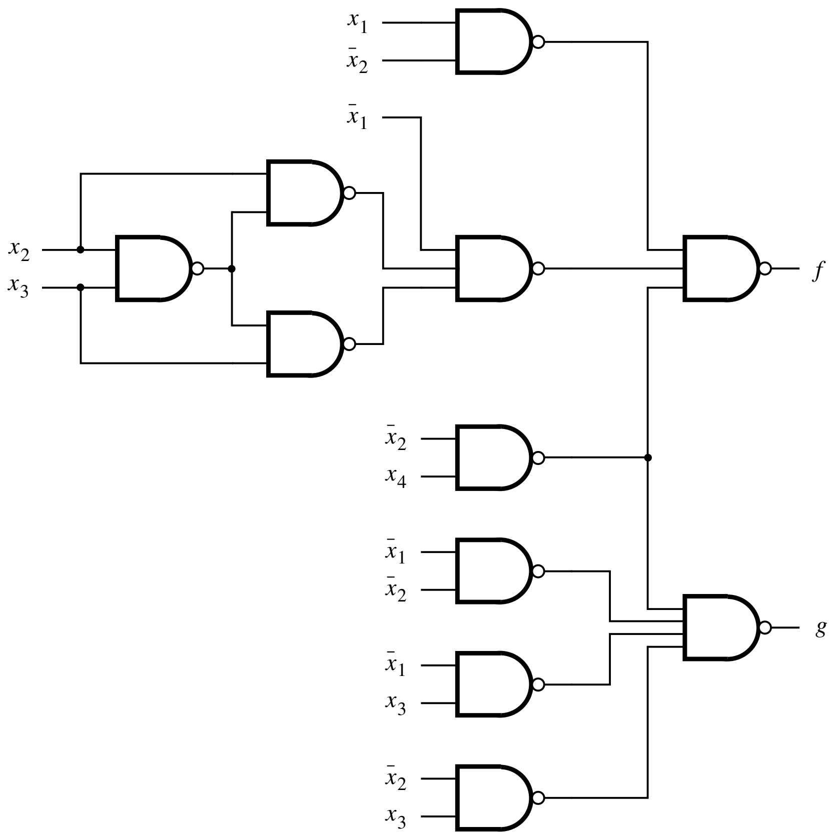


Figure P2.6 Circuit for Problem 2.79.

REFERENCES

1. G. Boole, *An Investigation of the Laws of Thought*, 1854, reprinted by Dover Publications, New York, 1954.
2. C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *Transactions of AIEE* 57 (1938), pp. 713–723.
3. E. V. Huntington, "Sets of Independent Postulates for the Algebra of Logic," *Transactions of the American Mathematical Society* 5 (1904), pp. 288–309.
4. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3rd ed. (McGraw-Hill: New York, 2009).
5. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
6. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
7. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed., (Prentice-Hall: Upper Saddle River, NJ, 2003).
8. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice Hall: Upper Saddle River, NJ, 2000).
9. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
10. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
11. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).