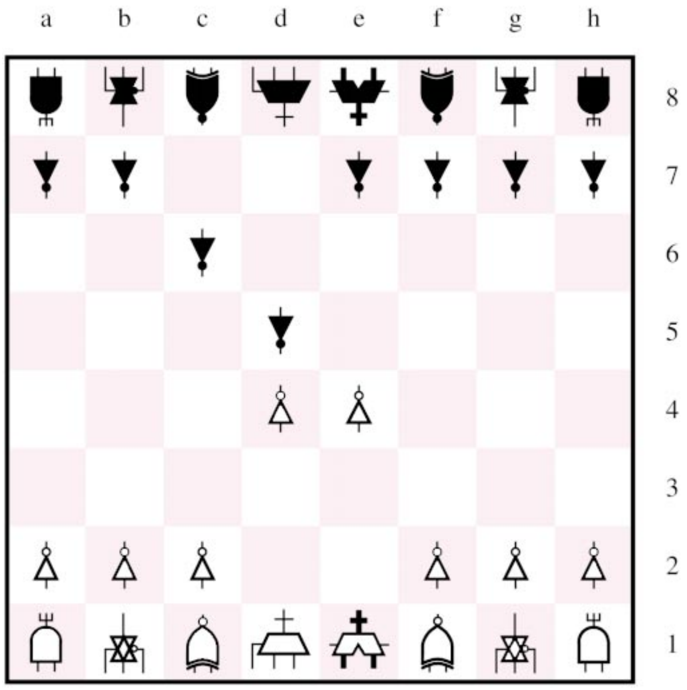
chapter

2

Introduction to Logic Circuits



2. d2–d4, d7–d5

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

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

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

2.1 VARIABLES AND FUNCTIONS

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

Consider a simple application of a switch, where the switch turns a small lightbulb on or off. This action is accomplished with the circuit in Figure 2.2*a*. A battery provides the power source. The lightbulb glows when sufficient current passes through its filament, which is an electrical resistance. The current flows when the switch is closed, that is, when $x = 1$. In this example the input that causes changes in the behavior of the circuit is the

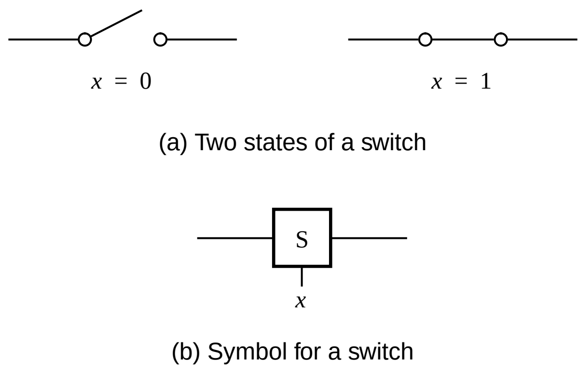
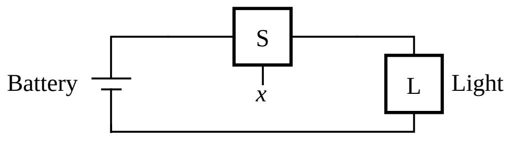
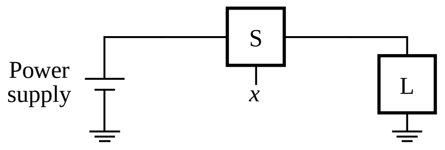


Figure 2.1 A binary switch.



(a) Simple connection to a battery



(b) Using a ground connection as the return path

Figure 2.2 A light controlled by a switch.

switch control x. The output is defined as the state (or condition) of the light L. If the light is on, we will say that $L = 1$. If the the light is off, we will say that $L = 0$. Using this convention, we can describe the state of the light L as a function of the input variable x. Since $L = 1$ if $x = 1$ and $L = 0$ if $x = 0$, we can say that

$$L(x) = x$$

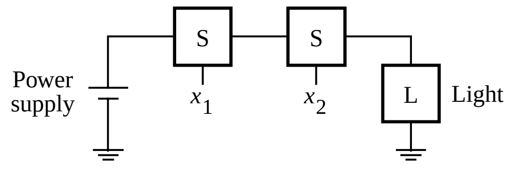
This simple *logic expression* describes the output as a function of the input. We say that $L(x) = x$ is a *logic function* and that x is an *input variable*.

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

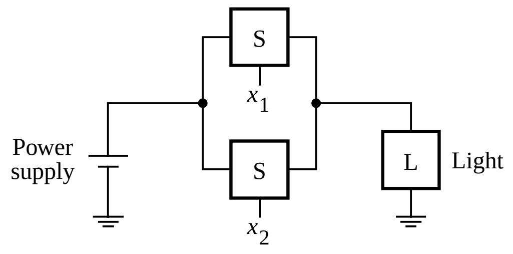
Consider now the possibility of using two switches to control the state of the light. Let x_1 and x_2 be the control inputs for these switches. The switches can be connected either in series or in parallel as shown in Figure 2.3. Using a series connection, the light will be turned on only if both switches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

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

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



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 2.3 Two basic functions.

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

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

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

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

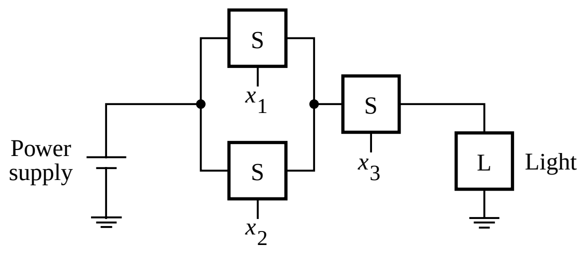


Figure 2.4 A series-parallel connection.

for the implementation of all logic circuits. Figure 2.4 illustrates how three switches can be used to control the light in a more complex way. This series-parallel connection of switches realizes the logic function

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

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

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

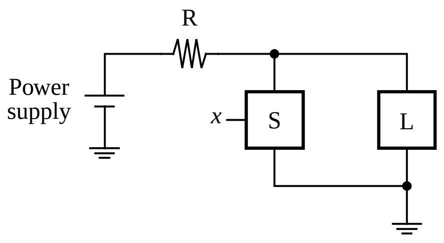


Figure 2.5 An inverting circuit.

placed after the variable, or the exclamation mark or the word NOT is placed in front of the variable to denote the complementation. Thus the following are equivalent:

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

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:

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

2.3 TRUTH TABLES

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

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.

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

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

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

Figure 2.7 Three-input AND and OR operations.

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

2.4 Logic Gates and Networks

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

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

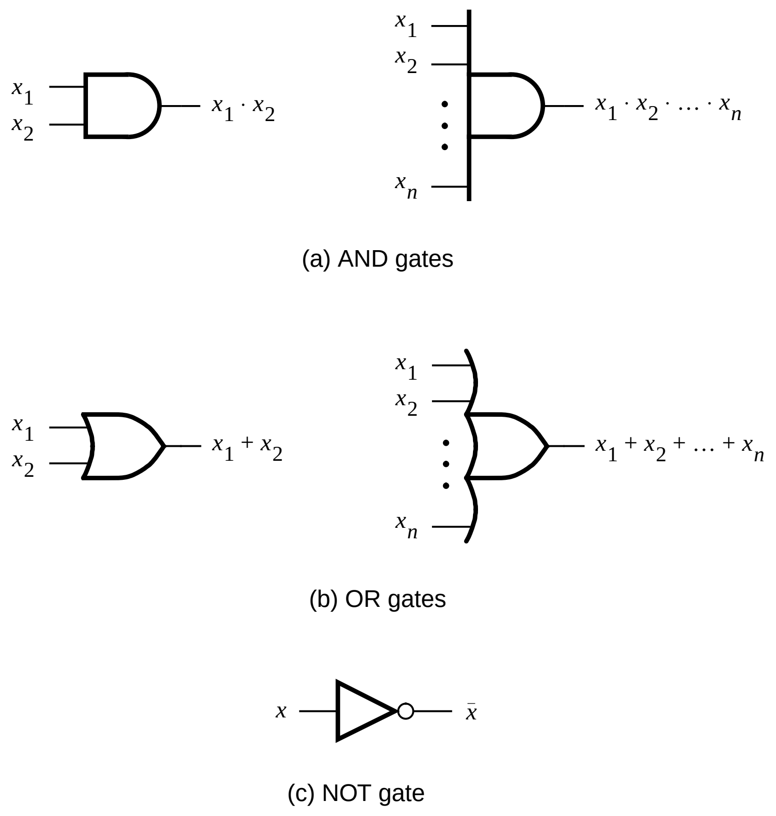


Figure 2.8 The basic gates.

2.4.1 Analysis of a Logic Network

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

Figure 2.10*a* shows a simple network consisting of three gates. To determine its functional behavior, we can consider what happens if we apply all possible input signals to it. Suppose that we start by making $x_1 = x_2 = 0$. This forces the output of the NOT gate

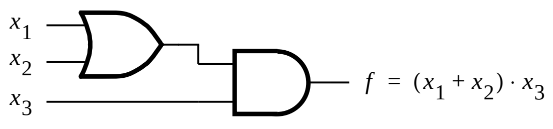
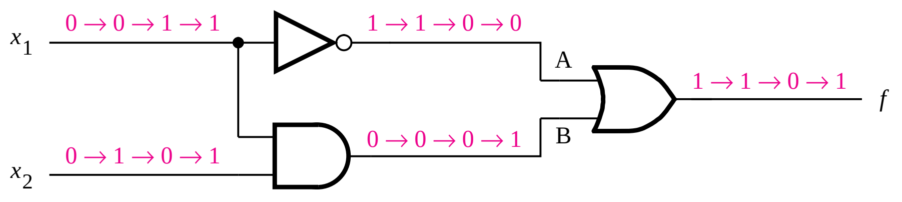


Figure 2.9 The function from Figure 2.4.

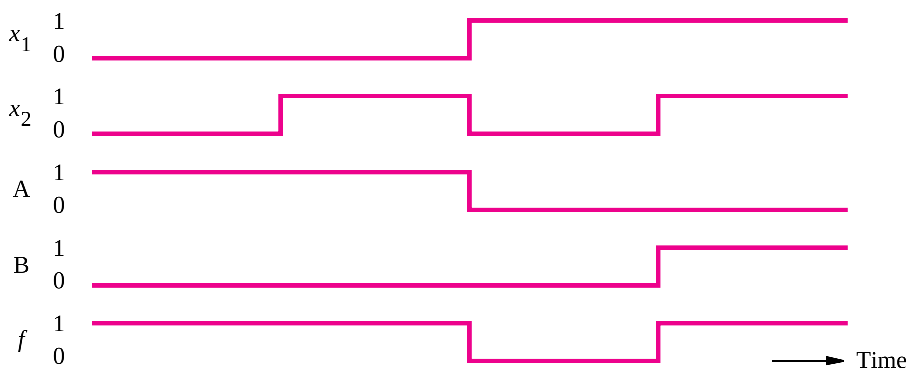


x_1
$0 \rightarrow 0 \rightarrow 1 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 0$
A
$1 \rightarrow 1 \rightarrow 0 \rightarrow 1$
f
B
$0 \rightarrow 0 \rightarrow 0 \rightarrow 1$
x_2
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$

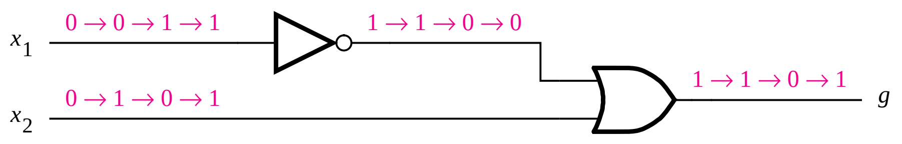
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

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

(b) Truth table for f



(c) Timing diagram



x_1
$0 \rightarrow 0 \rightarrow 1 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 0$
x_2
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 1$
g

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

Figure 2.10 An example of logic networks.

to be equal to 1 and the output of the AND gate to be 0. Because one of the inputs to the OR gate is 1, the output of this gate will be 1. Therefore, $f = 1$ if $x_1 = x_2 = 0$. If we let $x_1 = 0$ and $x_2 = 1$, then no change in the value of f will take place, because the outputs of the NOT and AND gates will still be 1 and 0, respectively. Next, if we apply $x_1 = 1$ and $x_2 = 0$, then the output of the NOT gate changes to 0 while the output of the AND gate remains at 0. Both inputs to the OR gate are then equal to 0; hence the value of f will be 0. Finally, let $x_1 = x_2 = 1$. Then the output of the AND gate goes to 1, which in turn causes f to be equal to 1. Our verbal explanation can be summarized in the form of the truth table shown in Figure 2.10*b*.

Timing Diagram

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

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? Many techniques exist for synthesizing logic functions. We will discuss the main approaches in Chapter 4. For now, we should note that some manipulation is needed to transform the more complex network in Figure 2.10*a* into the network in Figure 2.10*d*. Since $f(x_1, x_2) = \overline{x}_1 + x_1 \cdot x_2$ 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 the next section we will discuss a mathematical approach for dealing with logic functions, which provides the basis for modern design techniques.

2.5 Boolean Algebra

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

Axioms of Boolean Algebra

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

1a. $0 \cdot 0 = 0$

1b. $1 + 1 = 1$

2a. $1 \cdot 1 = 1$

2b. $0 + 0 = 0$

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

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

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

4b. 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 variable in B, then the following theorems hold:

5a. $x \cdot 0 = 0$

5b. $x + 1 = 1$

6a. $x \cdot 1 = x$

6b. $x + 0 = x$

7a. $x \cdot x = x$

7b. $x + x = x$

8a. $x \cdot \overline{x} = 0$

8b. $x + \overline{x} = 1$

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 5a, if $x = 0$, then the theorem states that $0 \cdot 0 = 0$, which is true according to axiom 1*a*. Similarly, if $x = 1$, then theorem 5*a* states that $1 \cdot 0 = 0$, which is also true according to axiom 3*a*. The reader should verify that theorems 5*a* to 9 can be proven in this way.

Duality

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

Two- and Three-Variable Properties

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

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

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

We have listed a number of axioms, theorems, and properties. Not all of these are necessary to define Boolean algebra. For example, assuming that the $+$ and $\cdot$ 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.

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

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

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

Example 2.1

Let us prove the validity of the logic equation

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

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

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

Applying the distributive property again yields

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

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 2.2

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

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

$$\begin{aligned} &= x_1 \cdot 1 + \overline{x}_2 \cdot 1 && \text{using } 8b \\ &= x_1 + \overline{x}_2 && \text{using } 6a \end{aligned}$$

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

2.5.1 THE VENN DIAGRAM

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

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

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

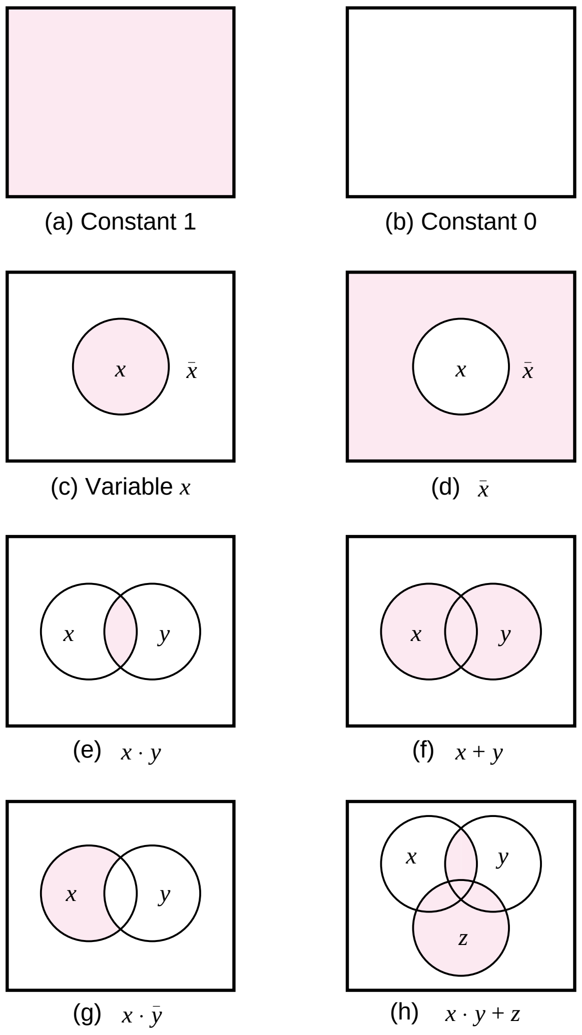


Figure 2.12 The Venn diagram representation.

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, namely, where at least one of x or y is equal to 1. Since this combines the areas in the circles, the OR operation is formally often called the *union* of x and y.

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

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

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

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

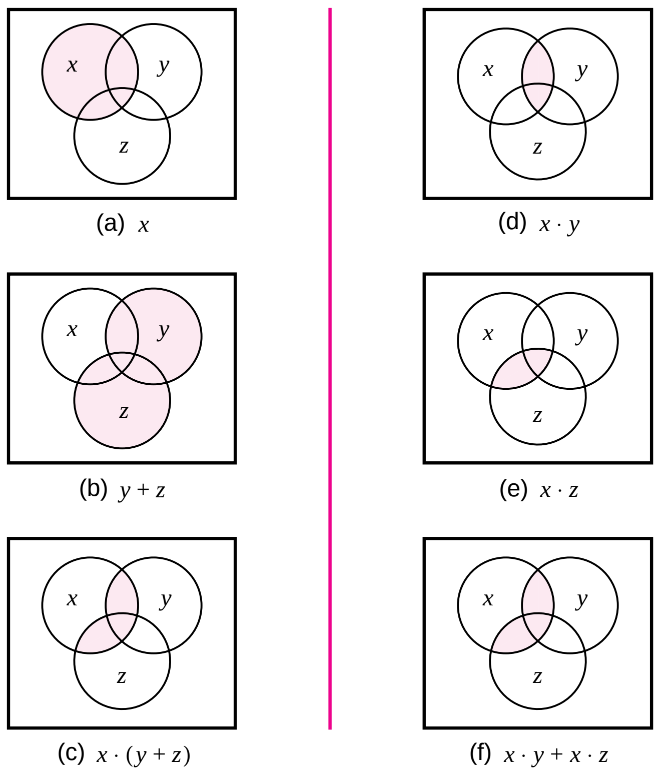


Figure 2.13 Verification of the distributive property $x \cdot (y + z) = x \cdot y + x \cdot z$.

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 2.14. 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.

The reader should use the Venn diagram to prove some other identities. It is particularly instructive to prove the validity of DeMorgan's theorem in this way.

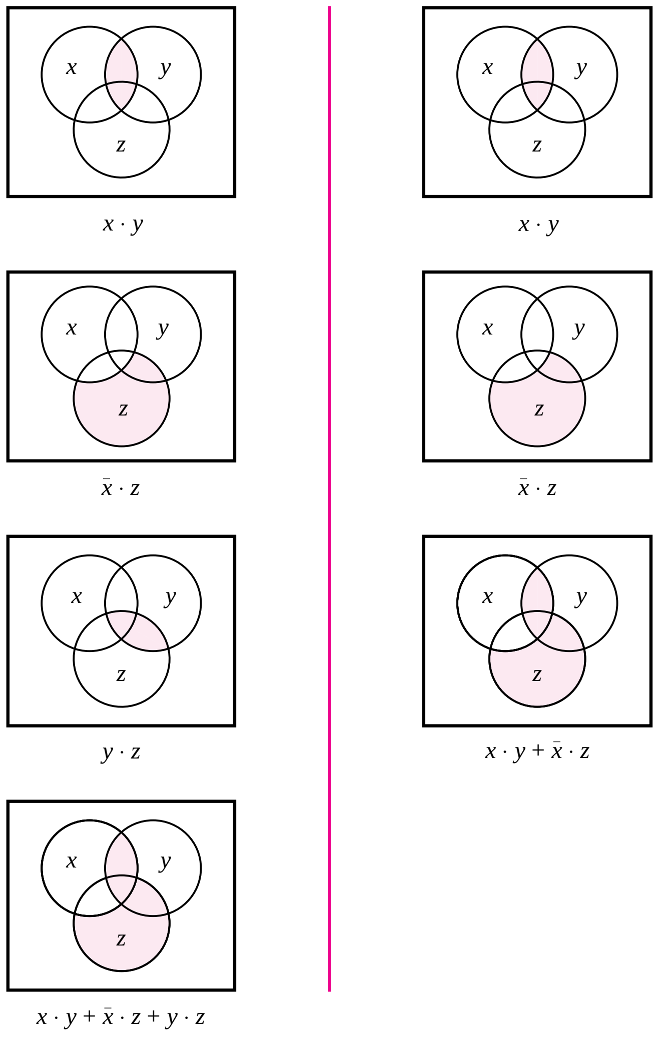

x
y
z
$x \cdot y$
$\bar{x} \cdot z$
$y \cdot z$
$x \cdot y + \bar{x} \cdot z + y \cdot z$
$x \cdot y$
$\bar{x} \cdot z$
$x \cdot y + \bar{x} \cdot z$

Figure 2.14 Verification of $x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$.

2.5.2 NOTATION AND TERMINOLOGY

Boolean algebra is based on the AND and OR operations. We have adopted the symbols $\cdot$ and $+$ to denote these operations. 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 7*b* in section 2.5.

When dealing with digital circuits, most of the time the $+$ symbol obviously represents the OR operation. However, when the task involves the design of logic circuits that perform arithmetic operations, some confusion may develop about the use of the $+$ symbol. To avoid such confusion, an alternative set of symbols exists for the AND and OR operations. It is quite common to use the $\wedge$ symbol to denote the AND operation, and the $\vee$ symbol for the OR operation. Thus, instead of $x_1 \cdot x_2$, we can write $x_1 \wedge x_2$, and instead of $x_1 + x_2$, we can write $x_1 \vee x_2$.

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 be open (0) or closed (1). The function of the circuit is to continuously monitor the state of the switches and to produce an output logic value 1 whenever the switches (x_1, x_2) are in states $(0, 0)$, $(0, 1)$, or $(1, 1)$. If the state of the switches is $(1, 0)$, the output should be 0. Another way of stating the required functional behavior of this circuit is that the output must be equal to 0 if the switch x_1 is closed and x_2 is open; otherwise, the output must be 1. We can express the required behavior using a truth table, as shown in Figure 2.15.

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

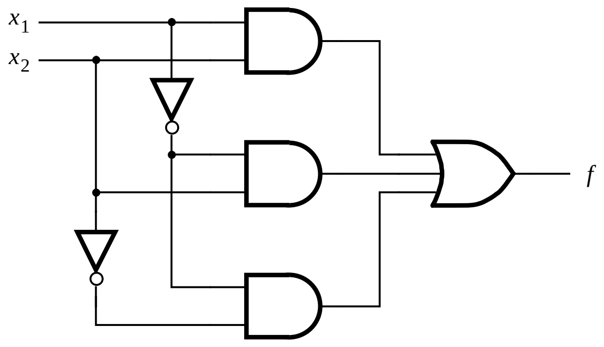
$$f(x_1, x_2) = x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2$$

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

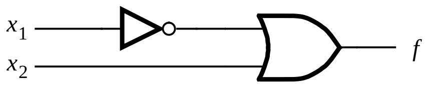
Although this network implements f correctly, it is not the simplest such network. To find a simpler network, we can manipulate the obtained expression using the theorems and

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

Figure 2.15 A function to be synthesized.



(a) Canonical sum-of-products



(b) Minimal-cost realization

Figure 2.16 Two implementations of the function in Figure 2.15.

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

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

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 2.16*b*. Obviously, the cost of this network is much less than the cost of the network in part (*a*) of the figure.

This simple example illustrates two things. First, a straightforward implementation of a function can be obtained by using a product term (AND gate) for each row of the truth table for which the function is equal to 1. Each product term contains all input variables, and it is formed such that if the input variable x_i is equal to 1 in the given row, then x_i is entered in the term; if $x_i = 0$, then $\overline{x}_i$ 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 2.16 from the truth table in Figure 2.15. Generation of AND-OR expressions from a truth table is just one of many types of synthesis techniques that we will encounter in this book.

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

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

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

Minterms

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

To illustrate this concept, consider the truth table in Figure 2.17. We have numbered the rows of the table from 0 to 7, so that we can refer to them easily. (The reader who is already familiar with the binary number representation will realize that the row numbers chosen are just the numbers represented by the bit patterns of variables x_1, x_2, and x_3; we will discuss number representation in Chapter 5.) The figure shows all minterms for the three-variable table. For example, in the first row the variables have the values $x_1 = x_2 = x_3 = 0$, which leads to the minterm $\overline{x}_1\overline{x}_2\overline{x}_3$. In the second row $x_1 = x_2 = 0$ and $x_3 = 1$, which gives the minterm $\overline{x}_1\overline{x}_2x_3$, and so on. To be able to refer to the individual minterms easily, it is convenient to identify each minterm by an index that corresponds to the row numbers shown in the figure. We will use the notation m_i to denote the minterm for row number i. Thus $m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$, $m_1 = \overline{x}_1\overline{x}_2x_3$, and so on.

Sum-of-Products Form

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

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

Figure 2.17 Three-variable minterms and maxterms.

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

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 of the *sum-of-products* form. If each product term is a minterm, then the expression is called a *canonical sum-of-products* for the function f. As we have seen in the example of Figure 2.16, the first step in the synthesis process is to derive a canonical sum-of-products expression for the given function. Then we can manipulate this expression, using the theorems and properties of section 2.5, with the goal of finding a functionally equivalent sum-of-products expression that has a lower cost.

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

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

This expression can be manipulated as follows

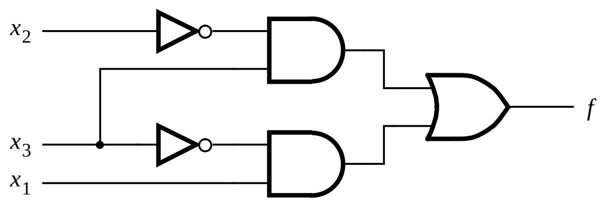
$$\begin{aligned} f &= (\overline{x}_1 + x_1)\overline{x}_2x_3 + x_1(\overline{x}_2 + x_2)\overline{x}_3 \\ &= 1 \cdot \overline{x}_2x_3 + x_1 \cdot 1 \cdot \overline{x}_3 \\ &= \overline{x}_2x_3 + x_1\overline{x}_3 \end{aligned}$$

This is the minimum-cost sum-of-products expression for f. It describes the circuit shown in Figure 2.19*a*. A good indication of the *cost* of a logic circuit is the total number of gates

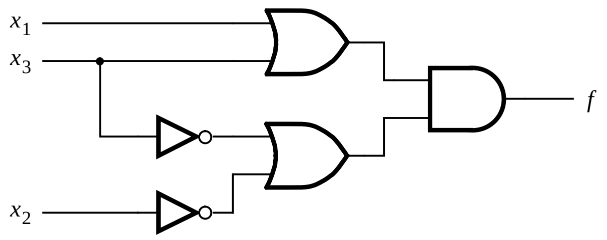
Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 2.18 A three-variable function.

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



(a) A minimal sum-of-products realization



(b) A minimal product-of-sums realization

Figure 2.19 Two realizations of the function in Figure 2.18.

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

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

or even more simply as

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

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

Maxterms

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

Product-of-Sums Form

If a given function f is specified by a truth table, then its complement $\overline{f}$ can be represented by a sum of minterms for which $\overline{f} = 1$, which are the rows where $f = 0$. For example, for the function in Figure 2.15

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

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

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

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

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

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

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

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

Then f can be expressed as

$$\begin{aligned}f &= \overline{m_0 + m_2 + m_3 + m_7}\\ &= \overline{m}_0 \cdot \overline{m}_2 \cdot \overline{m}_3 \cdot \overline{m}_7\end{aligned}$$

$$= M_0 \cdot M_2 \cdot M_3 \cdot M_7$$
$$= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$$

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* form. If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function. Any function f can be synthesized by finding its canonical product-of-sums. This involves taking the maxterm for each row in the truth table for which $f = 0$ and forming a product of these maxterms.

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

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

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

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

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

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

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

or more simply

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

The 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 circuits of a different structure, which usually implies a difference in cost. An important objective for a designer is to minimize the cost of the designed circuit. We will discuss the most important techniques for finding minimum-cost implementations in Chapter 4.

2.7 DESIGN EXAMPLES

Logic circuits provide a solution to a problem. They implement functions that are needed to carry out specific tasks. Within the framework of a computer, logic circuits provide complete capability for execution of programs and processing of data. Such circuits are complex and difficult to design. But regardless of the complexity of a given circuit, a designer of logic circuits is always confronted with the same basic issues. First, it is necessary to specify the desired behavior of the circuit. Second, the circuit has to be synthesized and implemented.

Finally, the implemented circuit has to be tested to verify that it meets the specifications. The desired behavior is often initially described in words, which then must be turned into a formal specification. In this section we give two simple examples of design.

2.7.1 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 2.20. 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 2.21*a*.

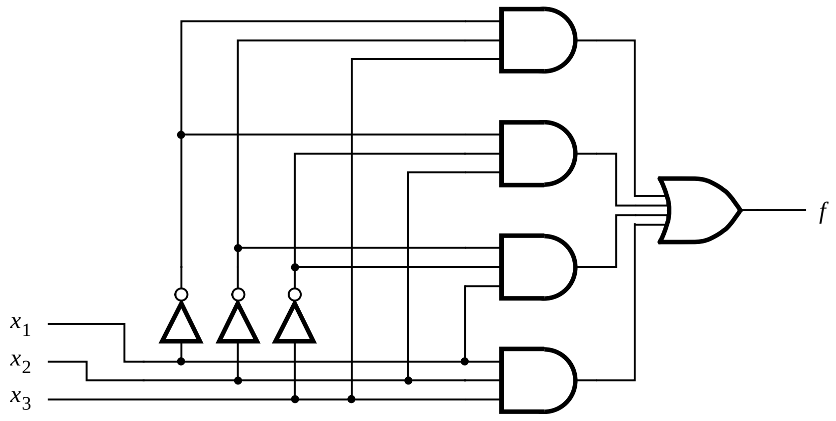
An alternative realization for this function is in the product-of-sums forms. The canonical expression of this type is

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

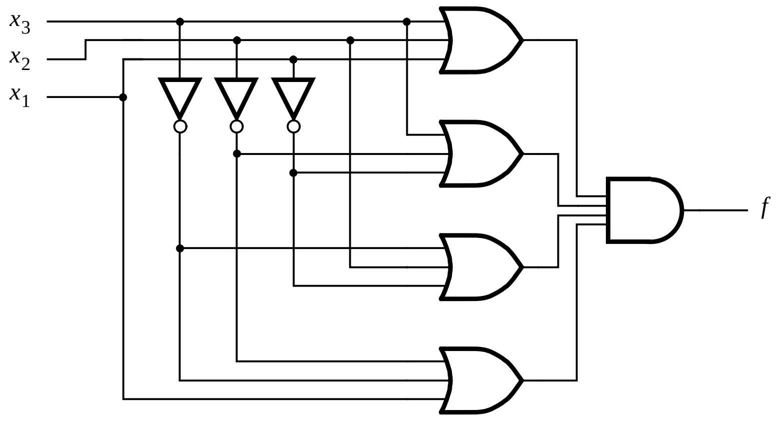
The resulting circuit is depicted in Figure 2.21*b*. It has the same cost as the circuit in part (*a*) of the figure.

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 2.20 Truth table for the three-way light control.



(a) Sum-of-products realization



(b) Product-of-sums realization

Figure 2.21 Implementation of the function in Figure 2.20.

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.

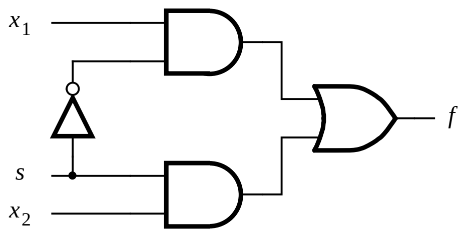
2.7.2 MULTIPLEXER CIRCUIT

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 $s = 1$.

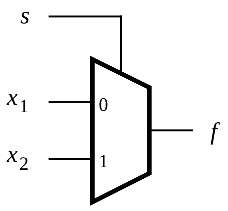
Based on these requirements, we can specify the desired circuit in the form of a truth table given in Figure 2.22*a*. From the truth table, we derive the canonical sum of products

$s\ x_1 x_2$	$f(s, x_1, x_2)$
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 2.22 Implementation of a multiplexer.

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

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

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

A circuit that implements this function is shown in Figure 2.22*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 2.22*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 2.22*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. In Chapter 3 we will show other possibilities for constructing multiplexers. In Chapter 6 we will discuss the use of multiplexers in considerable detail.

Designers of logic circuits rely heavily on CAD tools. We want to encourage the reader to become familiar with the CAD tool support provided with this book 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 2.9, a special language for describing logic circuits, called VHDL. This language is used to describe the circuits as an input to the CAD tools, which then proceed to derive a suitable implementation.

2.8 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, 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.

2.8.1 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 three design entry methods: using truth tables, using schematic capture, and writing source code in a hardware description language.

Design Entry with Truth Tables

We have already seen that any logic function of a few variables can be described conveniently by a truth table. Many CAD systems allow design entry using truth tables, where the table is specified as a plain text file. Alternatively, it may also be possible to specify a truth table as a set of waveforms in a timing diagram. We illustrated the equivalence of these two ways of representing truth tables in the discussion of Figure 2.10. The CAD system provided with this book supports both methods of using truth tables for design entry. Figure 2.23 shows an example in which the *Waveform Editor* is used to draw the timing diagram in Figure 2.10. The CAD system is capable of transforming this timing diagram automatically into a network of logic gates equivalent to that shown in Figure 2.10*d*.

Because truth tables are practical only for functions with a small number of variables, this design entry method is not appropriate for large circuits. It can, however, be applied for a small logic function that is part of a larger circuit. In this case the truth table becomes a subcircuit that can be interconnected to other subcircuits and logic gates. The most commonly used type of CAD tool for interconnecting such circuit elements 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.

Schematic Capture

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 basic gates in the schematic, the tool provides a collection of graphical symbols that represent gates

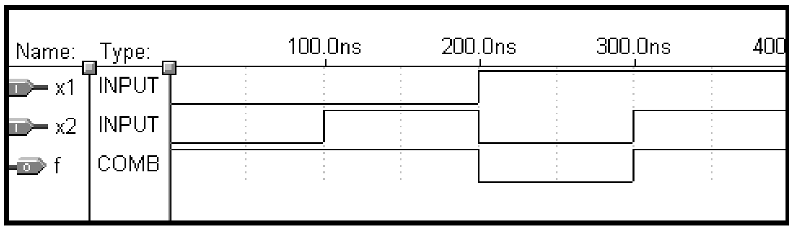


Figure 2.23 Screen capture of the Waveform Editor.

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, using either different design entry methods or the schematic capture tool itself, 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.

Figure 2.24 gives an example of a hierarchical design created with the schematic capture tool, provided with the CAD system, called the *Graphic Editor*. The circuit includes a subcircuit represented as a rectangular graphical symbol. This subcircuit represents the logic function entered by way of the timing diagram in Figure 2.23. Note that the complete circuit implements the function $f = \overline{x}_1 + x_2\overline{x}_3$.

In comparison to design entry with truth tables, the schematic-capture facility is more amenable for dealing with larger circuits. A disadvantage of using schematic capture is that every commercial tool of this type has a unique user interface and functionality. Therefore, extensive training is often required for a designer to learn how to use such a tool, and this training must be repeated if the designer switches to another tool at a later date. Another drawback is that the graphical user interface for schematic capture becomes awkward to use when the circuit being designed is large. A useful 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 provided 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.

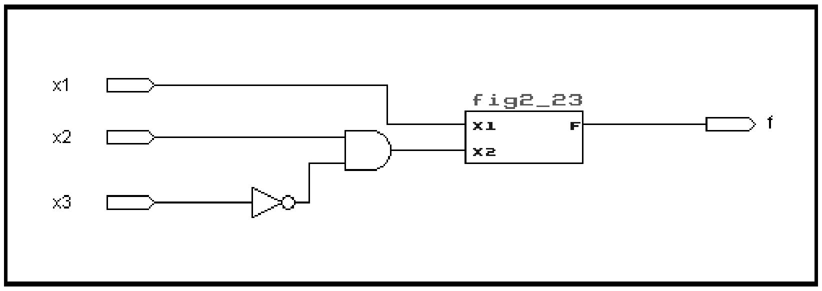


Figure 2.24 Screen capture of the Graphic Editor.

Two HDLs are IEEE standards: *VHDL (Very High Speed Integrated Circuit Hardware Description Language)* and *Verilog HDL*. Both languages are in widespread use in the industry. We use VHDL in this book because it is more popular than Verilog HDL. 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 VHDL can be directly applied when using Verilog HDL.

In comparison to performing schematic capture, using VHDL offers a number of advantages. Because it is supported by most companies that offer digital hardware technology, VHDL provides design *portability*. A circuit specified in VHDL can be implemented in different types of chips and with CAD tools provided by different companies, without having to change the VHDL specification. Design portability is an important advantage because digital circuit technology changes rapidly. By using a standard language, the designer can focus on the required 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 VHDL code. Signals in the circuit are represented as variables in the source code, and logic functions are expressed by assigning values to these variables. VHDL 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 VHDL is widely used, encourages sharing and reuse of VHDL-described circuits. This allows faster development of new products in cases where existing VHDL 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, VHDL code can be written in a modular way that facilitates hierarchical design. Both small and large logic circuit designs can be efficiently represented in VHDL code. VHDL has been used to define circuits such as microprocessors with millions of transistors.

VHDL 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 VHDL. We will introduce VHDL in section 2.9.

2.8.2 SYNTHESIS

In section 2.4.1 we said that synthesis is the process of generating a logic circuit from a truth table. Synthesis CAD tools perform this process automatically. However, the synthesis tools also handle many other tasks. The process of *translating*, or *compiling*, VHDL code into a network of logic gates is part of synthesis.

When the VHDL code representing a circuit is passed through initial synthesis tools, the output is a lower-level description of the circuit. For simplicity we will assume that this process produces a set of logic expressions that describe the logic functions needed to realize the circuit. These expressions are then manipulated further by the synthesis tools. If the design entry is performed using schematic capture, then the synthesis tools produce a set of logic equations representing the circuit from the schematic diagram. Similarly, if truth tables are used for design entry, then the synthesis tools generate expressions for the logic functions represented by the truth tables.

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 these expressions reflect the designer's input to the CAD tools, it is difficult for a designer to manually produce optimal results, especially for large circuits. One of the most important tasks of the synthesis tools is to manipulate the user's design to automatically produce an equivalent but better circuit. This step of synthesis is called *logic synthesis*, or *logic optimization*.

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. In section 2.6 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 will discuss implementation technologies in Chapter 3 and return to the issue of optimization goals in Chapter 4.

After logic synthesis the optimized circuit is still represented in the form of logic equations. The final task in the synthesis process is to determine exactly how the circuit will be realized in a specific hardware technology. This task involves deciding how each logic function, represented by an expression, should be implemented using whatever physical resources are available in the technology. The task involves two steps called *technology mapping*, followed by *layout synthesis*, or *physical design*. We will discuss these steps in detail in Chapter 4.

2.8.3 Functional Simulation

Once the design entry and synthesis are complete, it is useful to verify that the designed circuit functions as expected. The tool that performs this task is called a *functional simulator*, and it uses two types of information. First, the user's initial design is represented by the logic equations generated during synthesis. Second, the user specifies valuations of the circuit's inputs that should be applied to these equations during simulation. For each valuation, the simulator evaluates the outputs produced by the equations. The output of the simulation is provided either in truth-table form or as a timing diagram. The user examines this output to verify that the circuit operates as required.

The logic equations used by the simulator are those produced by the synthesis tools before any optimizations are applied during logic synthesis. There would be no advantage in using the optimized form of the equations, because the intent is to evaluate the basic functionality of the design, which does not change as a result of optimization. The functional simulator assumes that the time needed for signals to propagate through the logic gates is negligible. In real logic gates this assumption is not realistic, regardless of the hardware technology chosen for implementation of the circuit. However, the functional simulation provides a first step in validating the basic operation of a design without concern for the effects of implementation technology. Accurate simulations that account for the timing details related to technology can be obtained by using a *timing simulator*. We will discuss timing simulation in Chapter 4.

2.8.4 Summary

The CAD tools discussed in this section form a part of a CAD system. A typical design flow that the user follows is illustrated in Figure 2.25. After the design entry, initial synthesis tools perform various steps. For a function described by a truth table, the synthesis approach

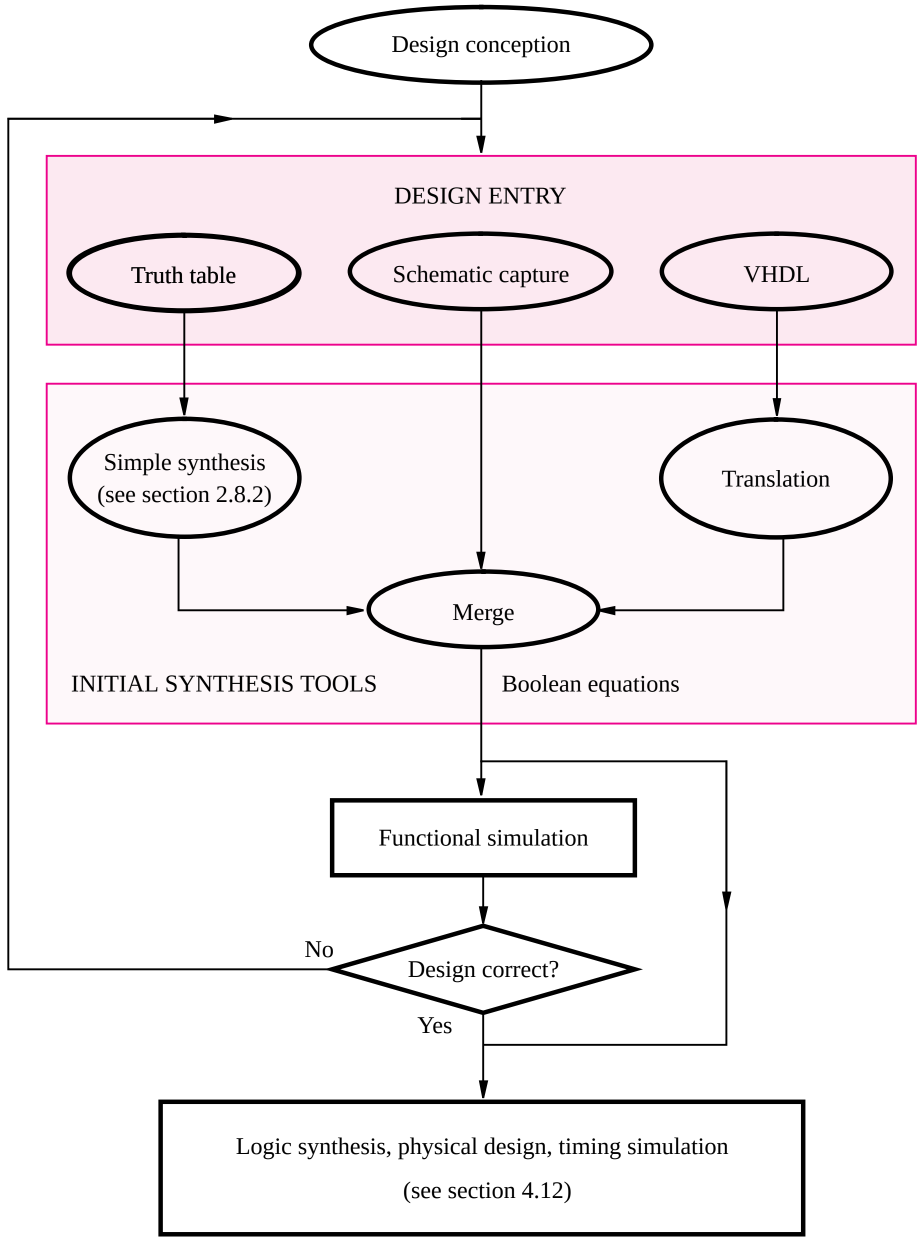


Figure 2.25 The first stages of a typical CAD system.

discussed in section 2.6 is applied to produce a logic expression for the function. For VHDL the translation process turns the VHDL source code into logic functions, which can be represented as logic expressions. As mentioned earlier, the designer can use a mixture of design entry methods. In Figure 2.25 this flexibility is reflected by the step labeled Merge, in which the components produced using any of the design entry methods are automatically merged into a single design. At this point the circuit is represented in the CAD system as a set of logic equations.

After the initial synthesis the correct operation of the designed circuit can be verified by using functional simulation. As shown in Figure 2.25, this step is not a requirement in the CAD flow and can be skipped at the designer's discretion. In practice, however, it is wise to verify that the designed circuit works as expected as early in the design process as possible. Any problems discovered during the simulation are fixed by returning to the design entry stage. Once errors are no longer apparent, the designer proceeds with the remaining tools in the CAD flow. These include logic synthesis, layout synthesis, timing simulation, and others. We have mentioned these tools only briefly thus far. The remaining CAD steps will be described in Chapter 4.

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. In Appendexes B to D, we provide step-by-step tutorials that illustrate how to use the MAX+plusII CAD system, which is included with this book. The tutorial in Appendix B covers design entry with both schematic capture and VHDL, as well as functional simulation. We strongly encourage the reader to work through the hands-on material. Because the tutorial uses VHDL for design entry, we provide an introduction to VHDL in the following section.

2.9 Introduction to VHDL

In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. VHDL was developed as a part of that effort. VHDL has become the industry standard language for describing digital circuits, largely because it is an official IEEE standard. The original standard for VHDL was adopted in 1987 and called IEEE 1076. A revised standard was adopted in 1993 and called IEEE 1164.

VHDL was originally intended to serve two main purposes. First, it was used as a documentation language for describing the structure of complex digital circuits. As an official IEEE standard, VHDL provided a common way of documenting circuits designed by numerous designers. Second, VHDL provided features for modeling the behavior of a digital circuit, which allowed its use as input to software programs that were then used to simulate the circuit's operation.

In recent years, in addition to its use for documentation and simulation, VHDL has also become popular for use in design entry in CAD systems. The CAD tools are used to synthesize the VHDL code into a hardware implementation of the described circuit. In this book our main use of VHDL will be for synthesis.

VHDL is an extremely 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 avoid confusion in learning this complex language, we will discuss only the features of VHDL that are actually used in the examples in the book. The material presented should be sufficient to allow the reader to design a wide range of circuits. The reader who wishes to learn the complete VHDL language can refer to one of the specialized texts [4–8].

To further simplify the task of learning VHDL, we will introduce the language 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. For convenience, in Appendix A we provide a complete listing of the VHDL features covered in the book. The reader may wish to refer to that material from time to time. In the remainder of this section, we discuss the most basic concepts needed to write simple VHDL code.

2.9.1 Representation of Digital Signals in VHDL

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 section 2.8.1. One convenient way is to write this description in the form of VHDL source code. The VHDL compiler translates this code into a logic circuit. Each logic signal in the circuit is represented in VHDL code as a data object. Just as the variables declared in any high-level programming language have associated types, such as integers or characters, data objects in VHDL can be of various types. The original VHDL standard, IEEE 1076, includes a data type called *BIT*. An object of this type is well suited for representing digital signals because BIT objects can have only two values, 0 and 1. In this chapter all signals in our examples will be of type BIT. Other data types are introduced in section 4.11 and are listed in Appendix A.

2.9.2 Writing Simple VHDL Code

We will use an example to illustrate how to write simple VHDL source code. Consider the logic circuit in Figure 2.26. If we wish to write VHDL code to represent this circuit, the first step is to declare the input and output signals. This is done using a construct called an *entity*. An appropriate entity for this example appears in Figure 2.27. An entity must

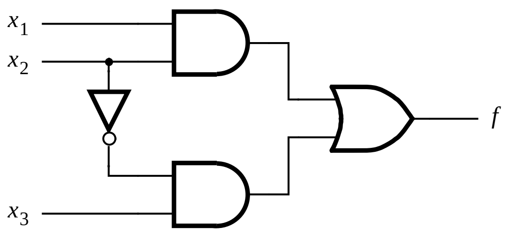


Figure 2.26 A simple logic function.

```
ENTITY example1 IS
    PORT ( x1, x2, x3  : IN   BIT ;
                f      : OUT  BIT ) ;
END example1 ;
```

Figure 2.27 VHDL entity declaration for the circuit in Figure 2.26.

be assigned a name; we have chosen the name *example1* for this first example. The input and output signals for the entity are called its *ports*, and they are identified by the keyword PORT. This name derives from the electrical jargon in which the word *port* refers to an input or output connection to an electronic circuit. Each port has an associated *mode* that specifies whether it is an input (IN) to the entity or an output (OUT) from the entity. Each port represents a signal, hence it has an associated type. The entity *example1* has four ports in total. The first three, x_1, x_2, and x_3, are input signals of type BIT. The port named f is an output of type BIT.

In Figure 2.27 we have used simple signal names *x1*, *x2*, *x3*, and *f* for the entity's ports. Similar to most computer programming languages, VHDL has rules that specify which characters are allowed in signal names. A simple guideline is that signal names can include any letter or number, as well as the underscore character '_'. There are two caveats: a signal name must begin with a letter, and a signal name cannot be a VHDL keyword.

An entity specifies the input and output signals for a circuit, but it does not give any details as to what the circuit represents. The circuit's functionality must be specified with a VHDL construct called an *architecture*. An architecture for our example appears in Figure 2.28. It must be given a name, and we have chosen the name *LogicFunc*. Although the name can be any text string, it is sensible to assign a name that is meaningful to the designer. In this case we have chosen the name *LogicFunc* because the architecture specifies the functionality of the design using a logic expression. VHDL has built-in support for the following Boolean operators: AND, OR, NOT, NAND, NOR, XOR, and XNOR. (So far we have introduced only AND, OR, and NOT operators; the others will be presented in Chapter 3.) Following the BEGIN keyword, our architecture specifies, using the VHDL signal assignment operator <=, that output f should be assigned the result of the logic expression on the right-hand side of the operator. Because VHDL does not assume any precedence of logic operators, parentheses are used in the expression. One might expect that an assignment statement such as

$$f <= x1 \text{ AND } x2 \text{ OR NOT } x2 \text{ AND } x3$$

would have implied parentheses

$$f <= (x1 \text{ AND } x2) \text{ OR } ((\text{NOT } x2) \text{ AND } x3)$$

But for VHDL code this assumption is not true. In fact, without the parentheses the VHDL compiler would produce a compile-time error for this expression.

Complete VHDL code for our example is given in Figure 2.29. This example has illustrated that a VHDL source code file has two main sections: an entity and an architecture.

```
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Figure 2.28 VHDL architecture for the entity in Figure 2.27.

```
ENTITY example1 IS
      PORT ( x1, x2, x3  : IN    BIT ;
                 f              : OUT  BIT ) ;
END example1 ;

ARCHITECTURE LogicFunc OF example1 IS
BEGIN
      f <= (x1 AND x2) OR (NOT x2 AND x3) ;
END LogicFunc ;
```

Figure 2.29 Complete VHDL code for the circuit in Figure 2.26.

A simple analogy for what each section represents is that the entity is equivalent to a symbol in a schematic diagram and the architecture specifies the logic circuitry inside the symbol.

A second example of VHDL code is given in Figure 2.30. This circuit has four input signals, called $x1$, $x2$, $x3$, and $x4$, and two output signals, named f and g. A logic expression is assigned to each output. A logic circuit produced by the VHDL compiler for this example is shown in Figure 2.31.

The preceding two examples indicate that one way to assign a value to a signal in VHDL code is by means of a logic expression. In VHDL terminology a logic expression is called a *simple assignment statement*. We will see later that VHDL also supports several other types of assignment statements and many other features that are useful for describing circuits that are much more complex.

2.9.3 How *not* to Write VHDL Code

When learning how to use VHDL or other hardware description languages, the tendency for the novice is to write code that resembles a computer program, containing many variables

```
ENTITY example2 IS
      PORT ( x1, x2, x3, x4  : IN    BIT ;
                 f, g                : OUT  BIT ) ;
END example2 ;

ARCHITECTURE LogicFunc OF example2 IS
BEGIN
      f <=  (x1 AND x3) OR (NOT x3 AND x2) ;
      g <= (NOT x3 OR x1) AND (NOT x3 OR x4) ;
END LogicFunc ;
```

Figure 2.30 VHDL code for a four-input function.

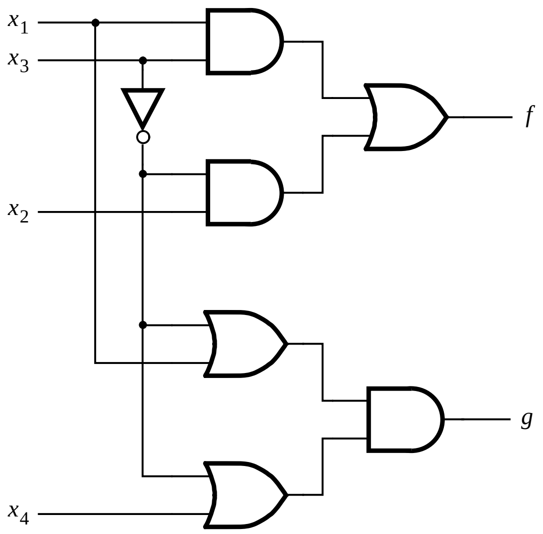


Figure 2.31 Logic circuit for the code in Figure 2.30.

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 VHDL 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 VHDL code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to describe.

Once complete VHDL code is written for a particular design, the reader is encouraged to analyze the resulting circuit synthesized by the CAD tools. Much can be learned about VHDL, logic circuits, and logic synthesis by studying the circuits that are produced automatically by the CAD tools.

2.10 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. This book is accompanied by the MAX+PlusII software, which is a CAD tool provided by Altera Corporation. We introduced a few basic features of this tool and urge the reader to start using this software as soon as possible.

Our discussion so far has been quite elementary. We will deal with both the logic circuits and the CAD tools in much more depth in the chapters that follow. But first, in

Chapter 3 we will examine the most important electronic technologies used to construct logic circuits. This material will give the reader an appreciation of practical constraints that a designer of logic circuits must face.

PROBLEMS

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 the Venn diagram to prove the identity in problem 1.

2.4 Use the Venn diagram to prove DeMorgan's theorem, as given in expressions 15*a* and 15*b* in section 2.5.

2.5 Use the Venn diagram to prove

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

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

2.7 Draw a timing diagram for the circuit in Figure 2.19*a*. Show the waveforms that can be observed on all wires in the circuit.

2.8 Repeat problem 2.7 for the circuit in Figure 2.19*b*.

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

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

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

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

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

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

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

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

2.15 (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 $f = x_1\overline{x}_2x_3 + x_1x_2 + \overline{x}_1x_3$. Use the Venn diagram to find the minimal sum-of-products form of f.

2.16 Represent the function in Figure 2.18 in the form of a Venn diagram and find its minimal sum-of-products form.

2.17 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.)

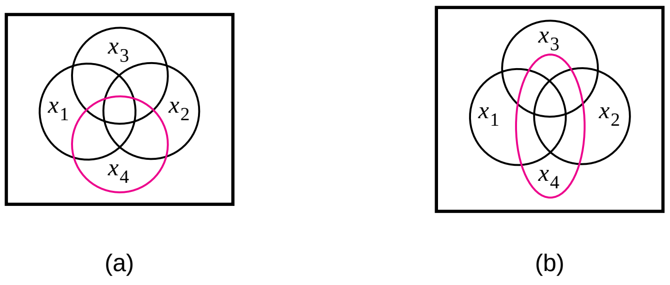


Figure P2.1 Two attempts to draw a four-variable Venn diagram.

2.18 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 $f = \overline{x}_1\overline{x}_2x_3\overline{x}_4 + x_1x_2x_3x_4 + \overline{x}_1x_2$ on this diagram.

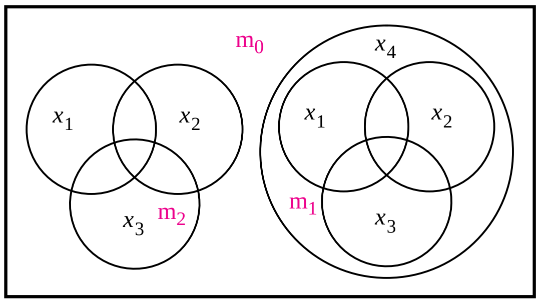


Figure P2.2 A four-variable Venn diagram.

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

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

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

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

2.23 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.

2.24 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.

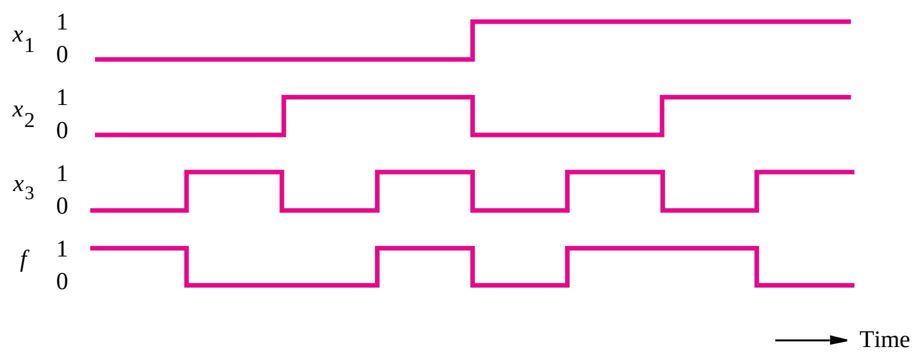


Figure P2.3 A timing diagram representing a logic function.

2.25 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.

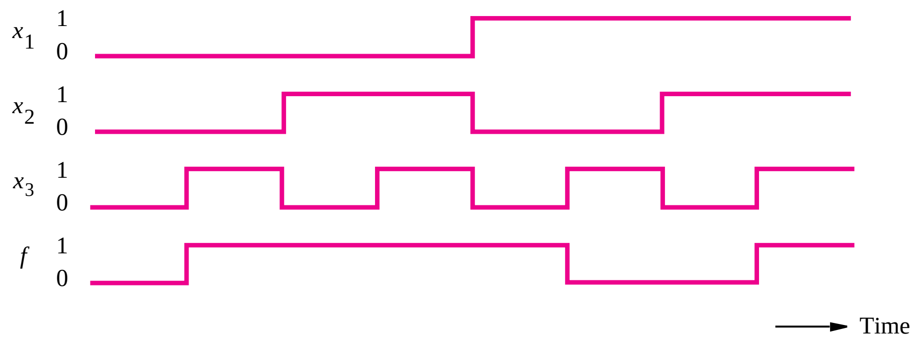


Figure P2.4 A timing diagram representing a logic function.

2.26 Design a circuit with output f and inputs x_1, x_0, y_1, and y_0. Let $X = x_1 x_0$ be a number, where the four possible values of X, namely, 00, 01, 10, and 11, represent the four numbers 0, 1, 2, and 3, respectively. (We discuss representation of numbers in Chapter 5.) Similarly, let $Y = y_1 y_0$ represent another number with the same four possible values. The output f should be 1 if the numbers represented by X and Y are not equal. Otherwise, f should be 0.
(a) Show the truth table for f.
(b) Synthesize the simplest possible product-of-sums expression for f.

2.27 Repeat problem 2.26 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.

2.28 (a) Use the Graphic Editor in MAX+plusII to draw schematics for the following functions

$$\begin{aligned} f_1 &= x_2\overline{x}_3\overline{x}_4 + \overline{x}_1 x_2 x_4 + \overline{x}_1 x_2 x_3 + x_1 x_2 x_3 \\ f_2 &= x_2\overline{x}_4 + \overline{x}_1 x_2 + x_2 x_3 \end{aligned}$$

(b) Use functional simulation in MAX+plusII to prove that $f_1 = f_2$.

2.29 (a) Use the Graphic Editor in MAX+plusII to draw schematics for the following functions

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

(b) Use functional simulation in MAX+plusII to prove that $f_1 = f_2$.

2.30 (a) Using the Text Editor in MAX+plusII, write VHDL code to describe the following functions

$$\begin{aligned} f_1 &= x_1\overline{x}_3 + x_2\overline{x}_3 + \overline{x}_3\overline{x}_4 + x_1 x_2 + x_1\overline{x}_4 \\ f_2 &= (x_1 + \overline{x}_3) \cdot (x_1 + x_2 + \overline{x}_4) \cdot (x_2 + \overline{x}_3 + \overline{x}_4) \end{aligned}$$

(b) Use functional simulation in MAX+plusII to prove that $f_1 = f_2$.

2.31 Consider the following VHDL assignment statements

```
f1 <= ((x1 AND x3) OR (NOT x1 AND NOT x3)) AND ((x2 AND x4) OR
        (NOT x2 AND NOT x4)) ;
f2 <= (x1 AND x2 AND NOT x3 AND NOT x4) OR (NOT x1 AND NOT x2 AND x3 AND x4)
        OR (x1 AND NOT x2 AND NOT x3 AND x4) OR
        (NOT x1 AND x2 AND x3 AND NOT x4) ;
```

(a) Write complete VHDL code to implement f1 and f2.
(b) Use functional simulation in MAX+plusII to prove that $f1 = \overline{f2}$.

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. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems*, 2nd ed. (McGraw-Hill: New York, 1998).
5. D. L. Perry, *VHDL*, 3rd ed. (McGraw-Hill: New York, 1998).
6. J. Bhasker, *A VHDL Primer* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
7. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
8. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, 1997).