

### LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- ◆ Understand the basic operations of **Boolean algebra**.
- ◆ Distinguish among the different types of **flip-flops**.
- ◆ Use a Karnaugh map to simplify a Boolean expression.
- ◆ Present an overview of **programmable logic devices**.

The operation of the digital computer is based on the storage and processing of binary data. Throughout this book, we have assumed the existence of storage elements that can exist in one of two stable states, and of circuits that can operate on binary data under the control of control signals to implement the various computer functions. In this chapter, we suggest how these storage elements and circuits can be implemented in digital logic, specifically with combinational and sequential circuits. The chapter begins with a brief review of Boolean algebra, which is the mathematical foundation of digital logic. Next, the concept of a gate is introduced. Finally, combinational and sequential circuits, which are constructed from **gates**, are described.

## 11.1 BOOLEAN ALGEBRA

The digital circuitry in digital computers and other digital systems is designed, and its behavior is analyzed, with the use of a mathematical discipline known as **Boolean algebra**. The name is in honor of an English mathematician George Boole, who proposed the basic principles of this algebra in 1854 in his treatise, *An Investigation of the Laws of Thought on Which to Found the Mathematical Theories of Logic and Probabilities*. In 1938, Claude Shannon, a research assistant in the Electrical Engineering Department at M.I.T., suggested that Boolean algebra could be used to solve problems in relay-switching circuit design [SHAN38].<sup>1</sup> Shannon's techniques were subsequently used in the analysis and design of electronic digital circuits. Boolean algebra turns out to be a convenient tool in two areas:

- **Analysis:** It is an economical way of describing the function of digital circuitry.
- **Design:** Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function.

As with any algebra, Boolean algebra makes use of variables and operations. In this case, the variables and operations are logical variables and operations. Thus, a variable may take on the value 1 (TRUE) or 0 (FALSE). The basic logical

<sup>1</sup>The paper is available at [box.com/COA10e](http://box.com/COA10e).

operations are AND, OR, and NOT, which are symbolically represented by dot, plus sign, and overbar:<sup>2</sup>

$$A \text{ AND } B = A \cdot B$$

$$A \text{ OR } B = A + B$$

$$\text{NOT } A = \overline{A}$$

The operation AND yields true (binary value 1) if and only if both of its operands are true. The operation OR yields true if either or both of its operands are true. The unary operation NOT inverts the value of its operand. For example, consider the equation

$$D = A + (\overline{B} \cdot C)$$

D is equal to 1 if A is 1 or if both B = 0 and C = 1. Otherwise D is equal to 0.

Several points concerning the notation are needed. In the absence of parentheses, the AND operation takes precedence over the OR operation. Also, when no ambiguity will occur, the AND operation is represented by simple concatenation instead of the dot operator. Thus,

$$A + B \cdot C = A + (B \cdot C) = A + BC$$

all mean: Take the AND of B and C; then take the OR of the result and A.

Table 11.1a defines the basic logical operations in a form known as a *truth table*, which lists the value of an operation for every possible combination of values of operands. The table also lists three other useful operators: XOR, **NAND**, and **NOR**. The exclusive-or (XOR) of two logical operands is 1 if and only if exactly one of the operands has the value 1. The NAND function is the complement (NOT) of the AND function, and the NOR is the complement of OR:

$$A \text{ NAND } B = \text{NOT } (A \text{ AND } B) = \overline{AB}$$

$$A \text{ NOR } B = \text{NOT } (A \text{ OR } B) = \overline{A + B}$$

As we shall see, these three new operations can be useful in implementing certain digital circuits.

The logical operations, with the exception of NOT, can be generalized to more than two variables, as shown in Table 11.1b.

Table 11.2 summarizes key identities of Boolean algebra. The equations have been arranged in two columns to show the complementary, or dual, nature of the AND and OR operations. There are two classes of identities: basic rules (or *postulates*), which are stated without proof, and other identities that can be derived from the basic postulates. The postulates define the way in which Boolean expressions are interpreted. One of the two distributive laws is worth noting because it differs from what we would find in ordinary algebra:

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

<sup>2</sup>Logical NOT is often indicated by an apostrophe: NOT A = A'.

**Table 11.1** Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P ( $\bar{P}$ )	P AND Q ( $P \cdot Q$ )	P OR Q ( $P + Q$ )	P NAND Q ( $\overline{P \cdot Q}$ )	P NOR Q ( $\overline{P + Q}$ )	P XOR Q ( $P \oplus Q$ )
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs (A, B, ...)

Operation	Expression	Output = 1 if
AND	$A \cdot B \cdot \dots$	All of the set {A, B, ...} are 1.
OR	$A + B + \dots$	Any of the set {A, B, ...} are 1.
NAND	$\overline{A \cdot B \cdot \dots}$	Any of the set {A, B, ...} are 0.
NOR	$\overline{A + B + \dots}$	All of the set {A, B, ...} are 0.
XOR	$A \oplus B \oplus \dots$	The set {A, B, ...} contains an odd number of ones.

The two bottommost expressions are referred to as DeMorgan's theorem. We can restate them as follows:

$$A \text{ NOR } B = \bar{A} \text{ AND } \bar{B}$$

$$A \text{ NAND } B = \bar{A} \text{ OR } \bar{B}$$

The reader is invited to verify the expressions in Table 11.2 by substituting actual values (1s and 0s) for the variables A, B, and C.

**Table 11.2** Basic Identities of Boolean Algebra

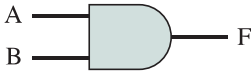
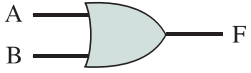
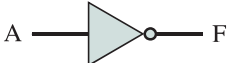

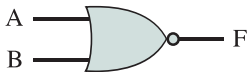
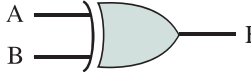
Basic Postulates		
$A \cdot B = B \cdot A$	$A + B = B + A$	Commutative Laws
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	Distributive Laws
$1 \cdot A = A$	$0 + A = A$	Identity Elements
$A \cdot \bar{A} = 0$	$A + \bar{A} = 1$	Inverse Elements
Other Identities		
$0 \cdot A = 0$	$1 + A = 1$	
$A \cdot A = A$	$A + A = A$	
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \cdot B} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A} \cdot \bar{B}$	DeMorgan's Theorem

## 11.2 GATES

The fundamental building block of all digital logic circuits is the gate. Logical functions are implemented by the interconnection of gates.

A gate is an electronic circuit that produces an output signal that is a simple Boolean operation on its input signals. The basic gates used in digital logic are AND, OR, NOT, NAND, NOR, and XOR. Figure 11.1 depicts these six gates. Each gate is defined in three ways: graphic symbol, algebraic notation, and truth table. The symbology used in this chapter is from the IEEE standard, IEEE Std 91. Note that the inversion (NOT) operation is indicated by a circle.

Each gate shown in Figure 11.1 has one or two inputs and one output. However, as indicated in Table 11.1b, all of the gates except NOT can have more than two inputs. Thus,  $(X + Y + Z)$  can be implemented with a single **OR gate** with three inputs. When one or more of the values at the input are changed, the correct output signal appears almost instantaneously, delayed only by the propagation time of signals through the gate (known as the *gate delay*). The significance of this delay is discussed in Section 11.3. In some cases, a gate is implemented with two outputs, one output being the negation of the other output.

Name	Graphical Symbol	Algebraic Function	Truth Table															
AND		$F = A \bullet B$ or $F = AB$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	F	0	0	0	0	1	0	1	0	0	1	1	1
A	B	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = A + B$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	F	0	0	0	0	1	1	1	0	1	1	1	1
A	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$F = \overline{A}$ or $F = A'$	<table><tr><th>A</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	F	0	1	1	0									
A	F																	
0	1																	
1	0																	
NAND		$F = \overline{AB}$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	1	1	0	1	1	1	0
A	B	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = \overline{A + B}$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	1	0	1	0	1	0	0	1	1	0
A	B	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR		$F = A \oplus B$	<table><tr><th>A</th><th>B</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	F	0	0	0	0	1	1	1	0	1	1	1	0
A	B	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

**Figure 11.1** Basic Logic Gates

Here we introduce a common term: we say that to **assert** a signal is to cause a signal line to make a transition from its logically false (0) state to its logically true (1) state. The true (1) state is either a high or low voltage state, depending on the type of electronic circuitry.

Typically, not all gate types are used in implementation. Design and fabrication are simpler if only one or two types of gates are used. Thus, it is important to identify *functionally complete* sets of gates. This means that any Boolean function can be implemented using only the gates in the set. The following are functionally complete sets:

- AND, OR, NOT
- AND, NOT
- OR, NOT
- NAND
- NOR

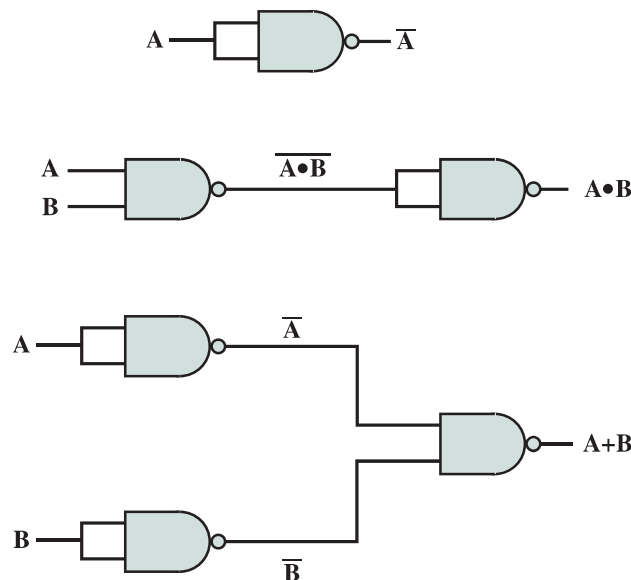
It should be clear that AND, OR, and NOT gates constitute a functionally complete set, because they represent the three operations of Boolean algebra. For the AND and NOT gates to form a functionally complete set, there must be a way to synthesize the OR operation from the AND and NOT operations. This can be done by applying DeMorgan's theorem:

$$A + B = \overline{\overline{A} \cdot \overline{B}}$$

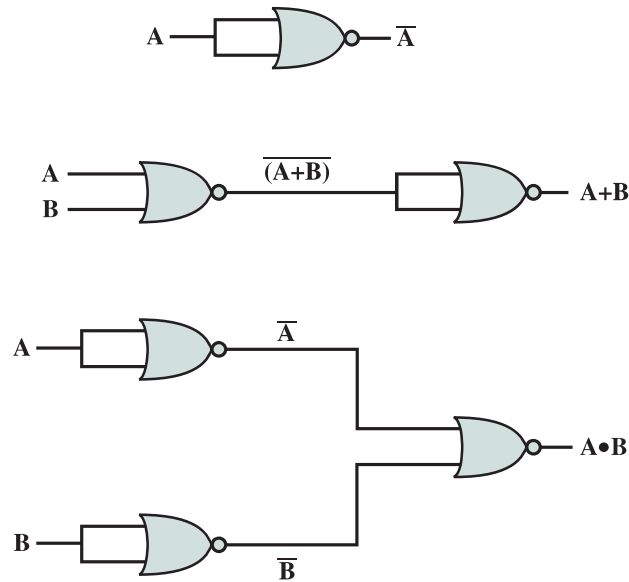
$$A \text{ OR } B = \text{NOT}((\text{NOT } A) \text{ AND } (\text{NOT } B))$$

Similarly, the OR and NOT operations are functionally complete because they can be used to synthesize the AND operation.

Figure 11.2 shows how the AND, OR, and NOT functions can be implemented solely with NAND gates, and Figure 11.3 shows the same thing for NOR gates. For this reason, digital circuits can be, and frequently are, implemented solely with NAND gates or solely with NOR gates.



**Figure 11.2** Some Uses of NAND Gates



**Figure 11.3** Some Uses of NOR Gates

With gates, we have reached the most primitive circuit level of computer hardware. An examination of the transistor combinations used to construct gates departs from that realm and enters the realm of electrical engineering. For our purposes, however, we are content to describe how gates can be used as building blocks to implement the essential logical circuits of a digital computer.

## 11.3 COMBINATIONAL CIRCUITS

A **combinational circuit** is an interconnected set of gates whose output at any time is a function only of the input at that time. As with a single gate, the appearance of the input is followed almost immediately by the appearance of the output, with only gate delays.

In general terms, a combinational circuit consists of  $n$  binary inputs and  $m$  binary outputs. As with a gate, a combinational circuit can be defined in three ways:

- **Truth table:** For each of the  $2^n$  possible combinations of input signals, the binary value of each of the  $m$  output signals is listed.
- **Graphical symbols:** The interconnected layout of gates is depicted.
- **Boolean equations:** Each output signal is expressed as a Boolean function of its input signals.

### Implementation of Boolean Functions

Any Boolean function can be implemented in electronic form as a network of gates. For any given function, there are a number of alternative realizations. Consider the Boolean function represented by the truth table in Table 11.3. We can express this function by simply itemizing the combinations of values of  $A$ ,  $B$ , and  $C$  that cause  $F$  to be 1:

$$F = \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C} \quad (11.1)$$

**Table 11.3** A Boolean Function of Three Variables

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

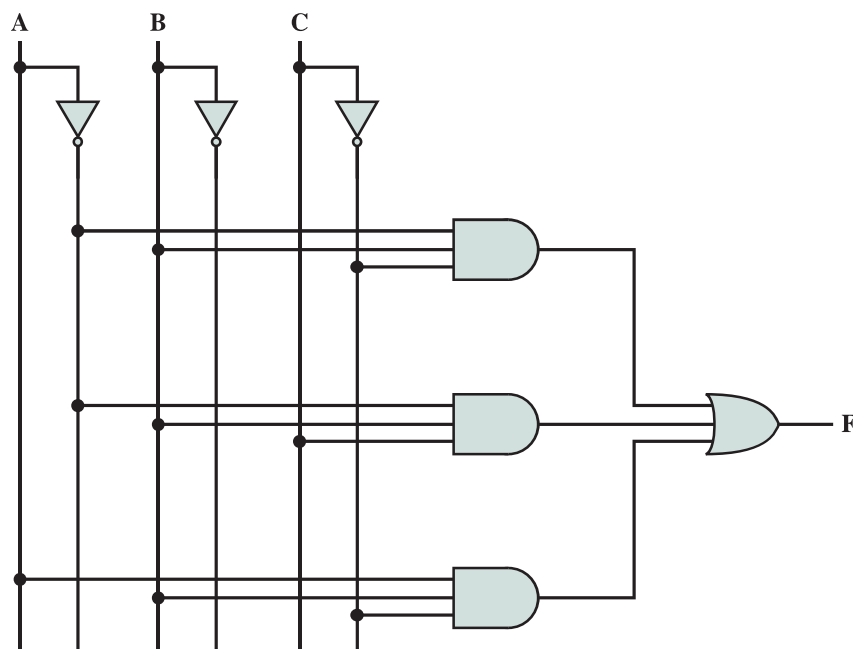
There are three combinations of input values that cause F to be 1, and if any one of these combinations occurs, the result is 1. This form of expression, for self-evident reasons, is known as the **sum of products (SOP)** form. Figure 11.4 shows a straightforward implementation with AND, OR, and NOT gates.

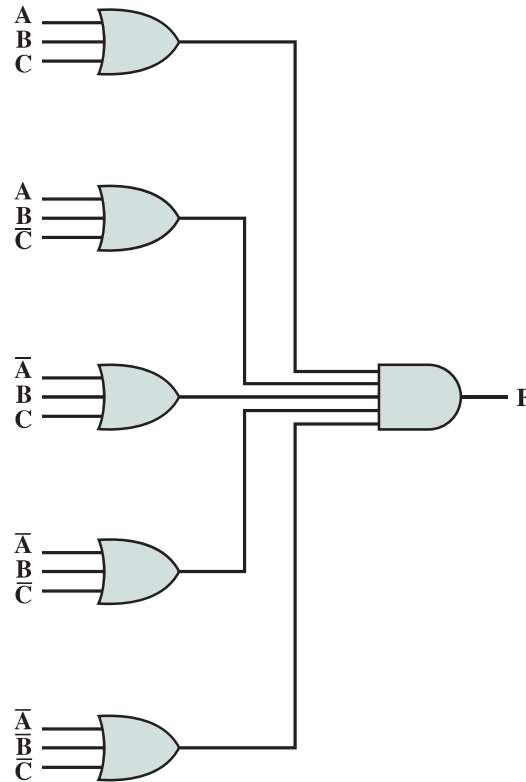
Another form can also be derived from the truth table. The SOP form expresses that the output is 1 if any of the input combinations that produce 1 is true. We can also say that the output is 1 if none of the input combinations that produce 0 is true. Thus,

$$F = (\overline{A} \overline{B} \overline{C}) \cdot (\overline{A} \overline{B} C) \cdot (\overline{A} B \overline{C}) \cdot (\overline{A} B C) \cdot (A \overline{B} \overline{C}) \cdot (A \overline{B} C) \cdot (A B \overline{C}) \cdot (A B C)$$

This can be rewritten using a generalization of DeMorgan's theorem:

$$\overline{(X \cdot Y \cdot Z)} = \overline{X} + \overline{Y} + \overline{Z}$$

**Figure 11.4** Sum-of-Products Implementation of Table 11.3



**Figure 11.5** Product-of-Sums Implementation of Table 11.3

Thus,

$$\begin{aligned}
 F &= (\bar{A} + \bar{B} + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C}) \quad (11.2) \\
 &= (A + B + C) \cdot (A + B + \bar{C}) \cdot (\bar{A} + B + C) \cdot (\bar{A} + B + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C})
 \end{aligned}$$

This is in the **product of sums (POS)** form, which is illustrated in Figure 11.5. For clarity, NOT gates are not shown. Rather, it is assumed that each input signal and its complement are available. This simplifies the logic diagram and makes the inputs to the gates more readily apparent.

Thus, a Boolean function can be realized in either SOP or POS form. At this point, it would seem that the choice would depend on whether the truth table contains more 1s or 0s for the output function: The SOP has one term for each 1, and the POS has one term for each 0. However, there are other considerations:

- It is often possible to derive a simpler Boolean expression from the truth table than either SOP or POS.
- It may be preferable to implement the function with a single gate type (NAND or NOR).

The significance of the first point is that, with a simpler Boolean expression, fewer gates will be needed to implement the function. Three methods that can be used to achieve simplification are

- Algebraic simplification
- Karnaugh maps
- Quine–McCluskey tables



**ALGEBRAIC SIMPLIFICATION** Algebraic simplification involves the application of the identities of Table 11.2 to reduce the Boolean expression to one with fewer elements. For example, consider again Equation (11.1). Some thought should convince the reader that an equivalent expression is

$$F = \overline{A}B + B\overline{C} \quad (11.3)$$

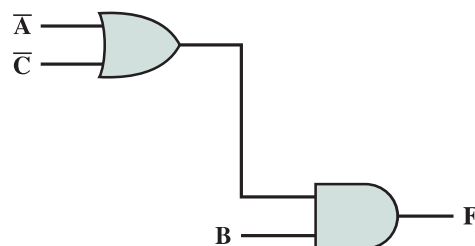
Or, even simpler,

$$F = B(\overline{A} + \overline{C})$$

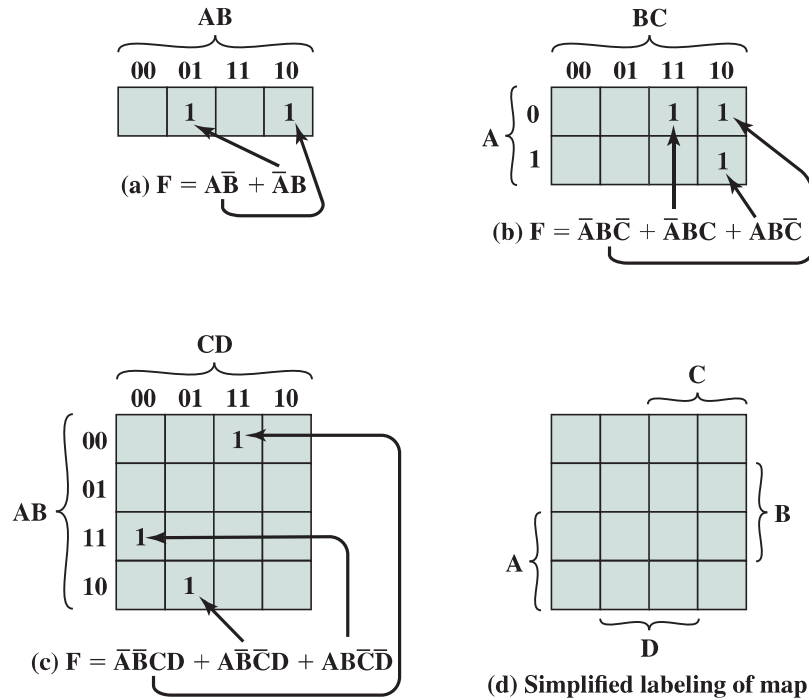
This expression can be implemented as shown in Figure 11.6. The simplification of Equation (11.1) was done essentially by observation. For more complex expressions, some more systematic approach is needed.

**KARNAUGH MAPS** For purposes of simplification, the **Karnaugh map** is a convenient way of representing a Boolean function of a small number (up to four) of variables. The map is an array of  $2^n$  squares, representing all possible combinations of values of  $n$  binary variables. Figure 11.7a shows the map of four squares for a function of two variables. It is essential for later purposes to list the combinations in the order 00, 01, 11, 10. Because the squares corresponding to the combinations are to be used for recording information, the combinations are customarily written above the squares. In the case of three variables, the representation is an arrangement of eight squares (Figure 11.7b), with the values for one of the variables to the left and for the other two variables above the squares. For four variables, 16 squares are needed, with the arrangement indicated in Figure 11.7c.

The map can be used to represent any Boolean function in the following way. Each square corresponds to a unique product in the sum-of-products form, with a 1 value corresponding to the variable and a 0 value corresponding to the NOT of that variable. Thus, the product  $\overline{A}B$  corresponds to the fourth square in Figure 11.7a. For each such product in the function, 1 is placed in the corresponding square. Thus, for the two-variable example, the map corresponds to  $\overline{A}B + \overline{A}\overline{B}$ . Given the truth table of a Boolean function, it is an easy matter to construct the map: for each combination of values of variables that produce a result of 1 in the truth table, fill in the corresponding square of the map with 1. Figure 11.7b shows the result for the truth table of Table 11.3. To convert from a Boolean expression to a map, it is first necessary to put the expression into what is referred to as *canonical* form: each term in the expression must contain each variable. So, for example, if we have Equation (11.3), we must first expand it into the full form of Equation (11.1) and then convert this to a map.



**Figure 11.6** Simplified Implementation of Table A.3



**Figure 11.7** The Use of Karnaugh Maps to Represent Boolean Functions

The labeling used in Figure 11.7d emphasizes the relationship between variables and the rows and columns of the map. Here the two rows embraced by the symbol  $A$  are those in which the variable  $A$  has the value 1; the rows not embraced by the symbol  $A$  are those in which  $A$  is 0; similarly for  $B$ ,  $C$ , and  $D$ .

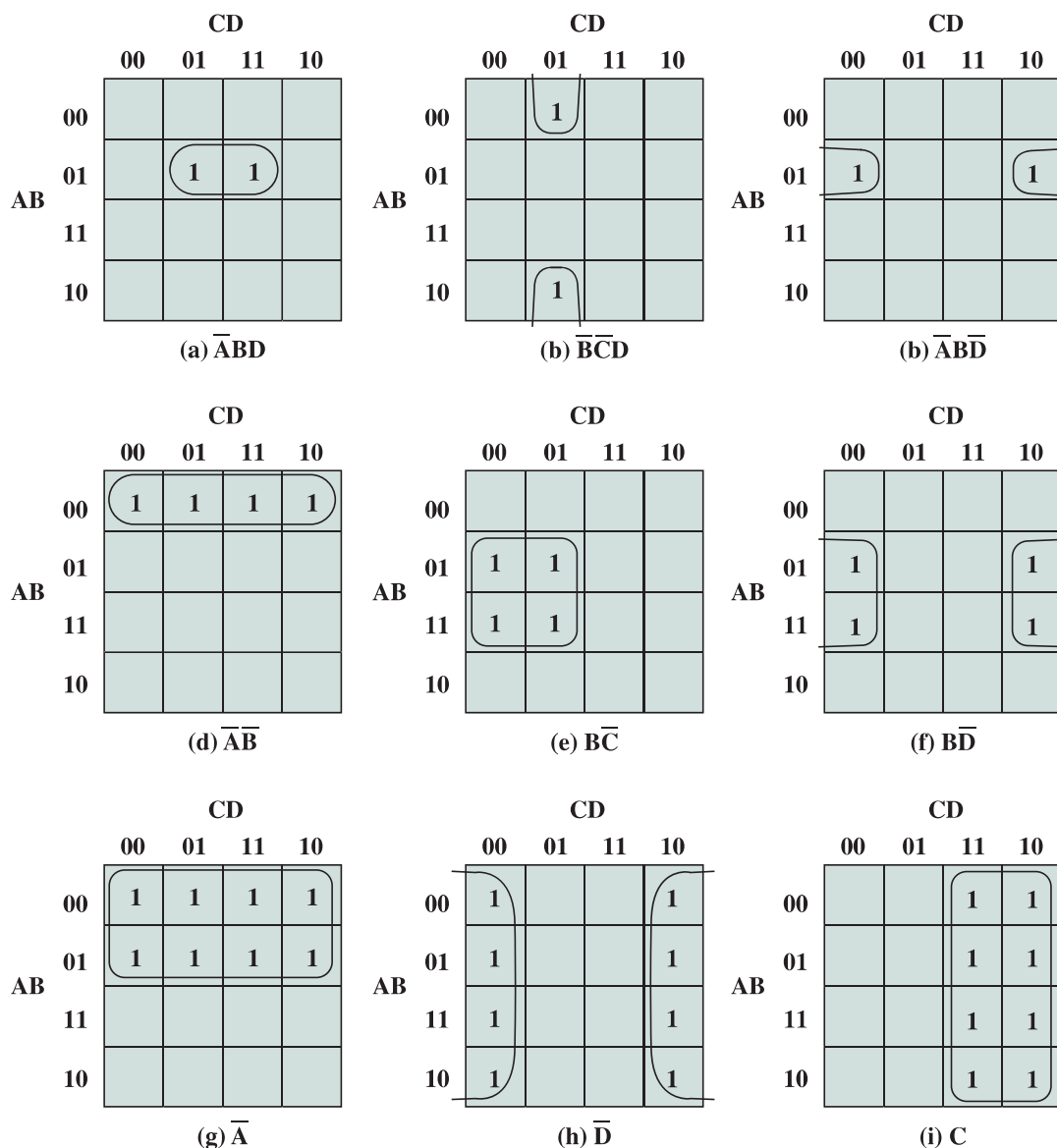
Once the map of a function is created, we can often write a simple algebraic expression for it by noting the arrangement of the 1s on the map. The principle is as follows. Any two squares that are adjacent differ in only one of the variables. If two adjacent squares both have an entry of one, then the corresponding product terms differ in only one variable. In such a case, the two terms can be merged by eliminating that variable. For example, in Figure 11.8a, the two adjacent squares correspond to the two terms  $\bar{A}\bar{B}CD$  and  $\bar{A}BCD$ . Thus, the function expressed is

$$\bar{A}\bar{B}CD + \bar{A}BCD = \bar{A}BD$$

This process can be extended in several ways. First, the concept of adjacency can be extended to include wrapping around the edge of the map. Thus, the top square of a column is adjacent to the bottom square, and the leftmost square of a row is adjacent to the rightmost square. These conditions are illustrated in Figures 11.8b and c. Second, we can group not just 2 squares but  $2^n$  adjacent squares (i.e., 2, 4, 8, etc.). The next three examples in Figure 11.8 show groupings of 4 squares. Note that in this case, two of the variables can be eliminated. The last three examples show groupings of 8 squares, which allow three variables to be eliminated.

We can summarize the rules for simplification as follows:

1. Among the marked squares (squares with a 1), find those that belong to a unique largest block of 1, 2, 4, or 8 and circle those blocks.



**Figure 11.8** The Use of Karnaugh Maps

2. Select additional blocks of marked squares that are as large as possible and as few in number as possible, but include every marked square at least once. The results may not be unique in some cases. For example, if a marked square combines with exactly two other squares, and there is no fourth marked square to complete a larger group, then there is a choice to be made as to which of the two groupings to choose. When you are circling groups, you are allowed to use the same 1 value more than once.
3. Continue to draw loops around single marked squares, or pairs of adjacent marked squares, or groups of four, eight, and so on in such a way that every marked square belongs to at least one loop; then use as few of these blocks as possible to include all marked squares.

Figure 11.9a, based on Table 11.3, illustrates the simplification process. If any isolated 1s remain after the groupings, then each of these is circled as a group of 1s.

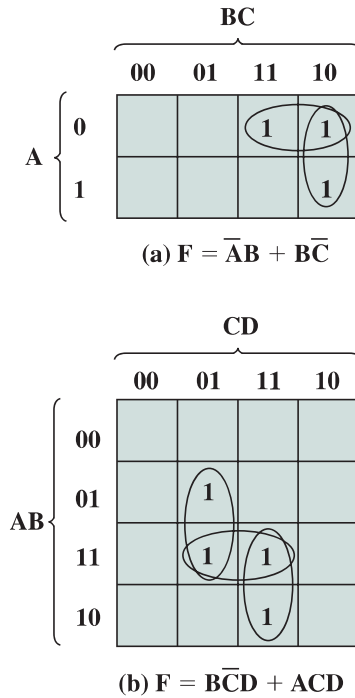


Figure 11.9 Overlapping Groups

Finally, before going from the map to a simplified Boolean expression, any group of 1s that is completely overlapped by other groups can be eliminated. This is shown in Figure 11.9b. In this case, the horizontal group is redundant and may be ignored in creating the Boolean expression.

One additional feature of Karnaugh maps needs to be mentioned. In some cases, certain combinations of values of variables never occur, and therefore the corresponding output never occurs. These are referred to as “don’t care” conditions. For each such condition, the letter “d” is entered into the corresponding square of the map. In doing the grouping and simplification, each “d” can be treated as a 1 or 0, whichever leads to the simplest expression.

An example, presented in [HAYE98], illustrates the points we have been discussing. We would like to develop the Boolean expressions for a circuit that adds 1 to a packed decimal digit. For packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way. Thus,  $0 = 0000$ ,  $1 = 0001$ , ...,  $8 = 1000$ , and  $9 = 1001$ . The remaining 4-bit values, from 1010 to 1111, are not used. This code is also referred to as **Binary Coded Decimal (BCD)**.

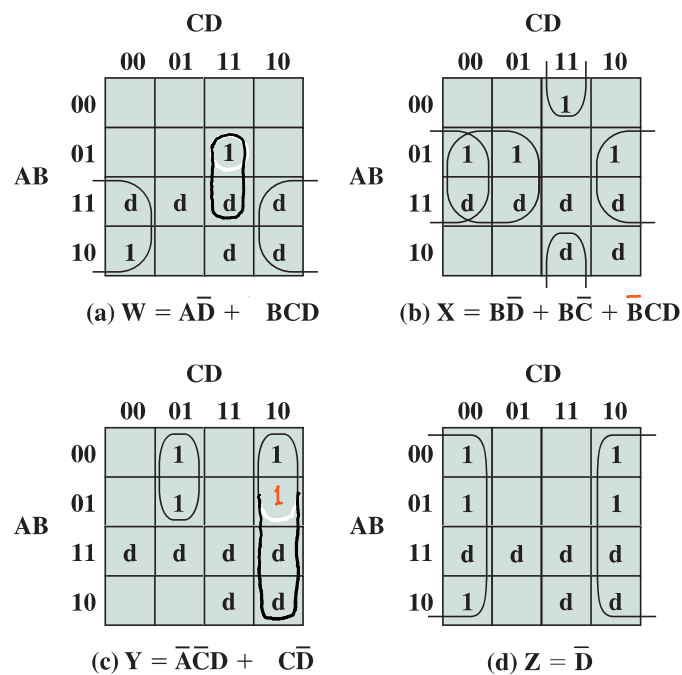
Table 11.4 shows the truth table for producing a 4-bit result that is one more than a 4-bit BCD input. The addition is modulo 10. Thus,  $9 + 1 = 0$ . Also, note that six of the input codes produce “don’t care” results, because those are not valid BCD inputs. Figure 11.10 shows the resulting Karnaugh maps for each of the output variables. The d squares are used to achieve the best possible groupings.

**THE QUINE–MCCLUSKEY METHOD** For more than four variables, the Karnaugh map method becomes increasingly cumbersome. With five variables, two  $16 \times 16$  maps are needed, with one map considered to be on top of the other in three dimensions to achieve adjacency. Six variables require the use of four  $16 \times 16$

**Table 11.4** Truth Table for the One-Digit Packed Decimal Incrementer

Input					Output				
Number	A	B	C	D	Number	W	X	Y	Z
0	0	0	0	0	1	0	0	0	1
1	0	0	0	1	2	0	0	1	0
2	0	0	1	0	3	0	0	1	1
3	0	0	1	1	4	0	1	0	0
4	0	1	0	0	5	0	1	0	1
5	0	1	0	1	6	0	1	1	0
6	0	1	1	0	7	0	1	1	1
7	0	1	1	1	8	1	0	0	0
8	1	0	0	0	9	1	0	0	1
9	1	0	0	1	0	0	0	0	0
Don't care condition	1	0	1	0		d	d	d	d
	1	0	1	1		d	d	d	d
	1	1	0	0		d	d	d	d
	1	1	0	1		d	d	d	d
	1	1	1	0		d	d	d	d
	1	1	1	1		d	d	d	d

tables in four dimensions! An alternative approach is a tabular technique, referred to as the Quine–McCluskey method. The method is suitable for programming on a computer to give an automatic tool for producing minimized Boolean expressions.

**Figure 11.10** Karnaugh Maps for the Incrementer