Chapter 3

Gate-Level Minimization

3.1 INTRODUCTION

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

3.2 THE MAP METHOD

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

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

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

Two-Variable Map

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

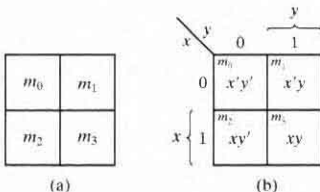


FIGURE 3.1
Two-variable map

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

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

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

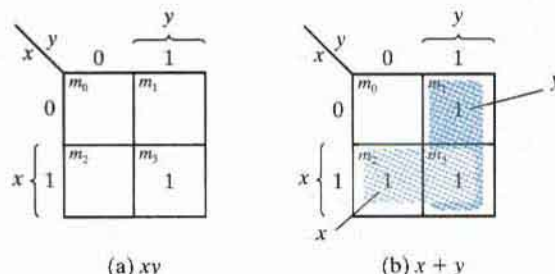


FIGURE 3.2
Representation of functions in the map

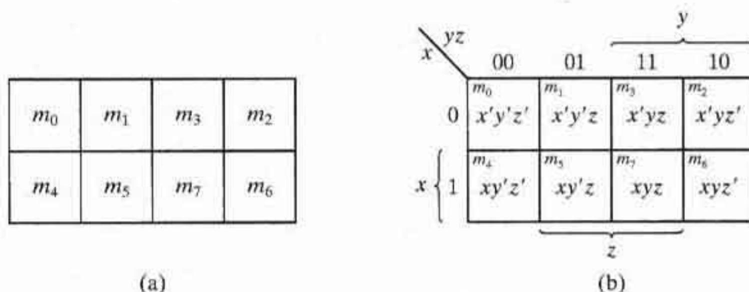


FIGURE 3.3
Three-variable map

Three-Variable Map

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

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

term consisting of only two literals. To clarify this concept, consider the sum of two adjacent squares such as m_5 and m_7:

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

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

EXAMPLE 3.1

Simplify the Boolean function

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

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

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

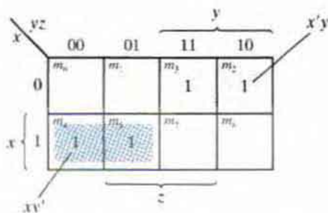


FIGURE 3.4
Map for Example 3.1, $F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$

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

$$m_0 + m_2 = x'y'z' + x'yz' = x'z'(y' + y) = x'z'$$
$$m_4 + m_6 = xy'z' + xyz' = xz' + (y' + y) = xz'$$

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

EXAMPLE 3.2

Simplify the Boolean function

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

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

$$F = yz + xz'$$

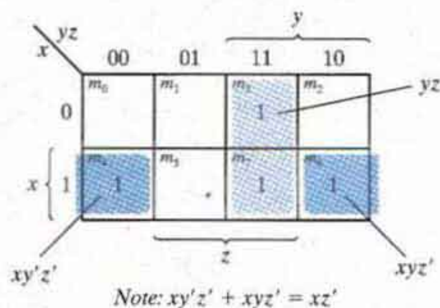


FIGURE 3.5
Map for Example 3.2, $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

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

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

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

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

Two adjacent squares represent a term with two literals.

Four adjacent squares represent a term with one literal.

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

EXAMPLE 3.3

Simplify the Boolean function

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

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

$$F = z' + xy'$$

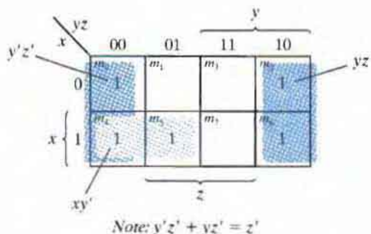


FIGURE 3.6
Map for Example 3.3, $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$

■

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

EXAMPLE 3.4

Let the Boolean function

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

(a) Express this function as a sum of minterms.

(b) Find the minimal sum-of-products expression.

Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term, $A'C$, are found in Fig. 3.7 from the coincidence of A' (first row) and C (two middle columns) to give squares 001

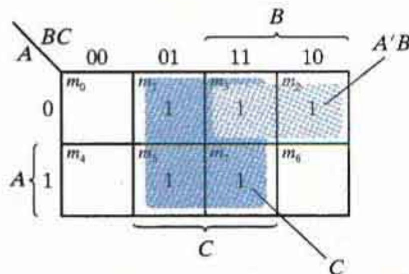


FIGURE 3.7
Map for Example 3.4, $A'C + A'B + AB'C + BC = C + A'B$

and 011. Note that, in marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term. This happens with the second term, $A'B$, which has 1's in squares 011 and 010. Square 011 is common with the first term, $A'C$, though, so only one 1 is marked in it. Continuing in this fashion, we determine that the term $AB'C$ belongs in square 101, corresponding to minterm 5, and the term BC has two 1's in squares 011 and 111. The function has a total of five minterms, as indicated by the five 1's in the map of Fig. 3.7. The minterms are read directly from the map to be 1, 2, 3, 5, and 7. The function can be expressed in sum-of-minterms form as

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

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

$$F = C + A'B$$

■

3.3 FOUR-VARIABLE MAP

The map for Boolean functions of four binary variables is shown in Fig. 3.8. In (a) are listed the 16 minterms and the squares assigned to each. In (b), the map is redrawn to show the relationship between the squares and the four variables. The rows and columns are numbered in a Gray code sequence, with only one digit changing value between two adjacent rows or columns. The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number. For example, the numbers of the third row (11) and the second column (01), when concatenated, give the binary number 1101, the binary equivalent of decimal 13. Thus, the square in the third row and second column represents minterm m_{13}.

The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions. Adjacent squares are defined to be squares next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares. For example,

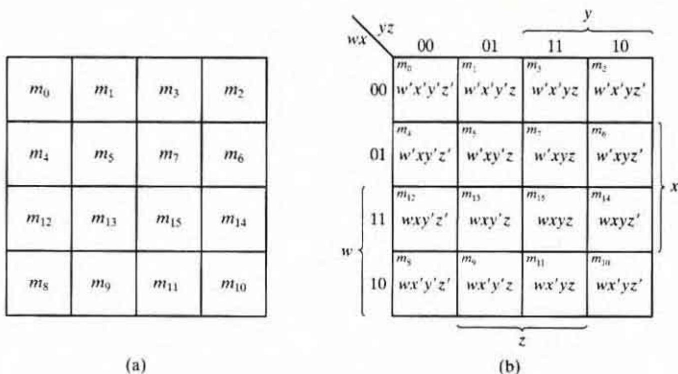


FIGURE 3.8
Four-variable map

m_0 and m_2 form adjacent squares, as do m_3 and m_{11}. The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:

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

Two adjacent squares represent a term with three literals.

Four adjacent squares represent a term with two literals.

Eight adjacent squares represent a term with one literal.

Sixteen adjacent squares produce a function that is always equal to 1.

No other combination of squares can simplify the function. The next two examples show the procedure used to simplify four-variable Boolean functions.

EXAMPLE 3.5

Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

Since the function has four variables, a four-variable map must be used. The minterms listed in the sum are marked by 1's in the map of Fig. 3.9. Eight adjacent squares marked with 1's can be combined to form the one literal term y'. The remaining three 1's on the right cannot be combined to give a simplified term; they must be combined as two or four adjacent squares. The larger the number of squares combined, the smaller is the number of literals in the term. In this example, the top two 1's on the right are combined with the top two 1's on the left to give the term $w'z'$. Note that it is permissible to use the same square more than once. We are

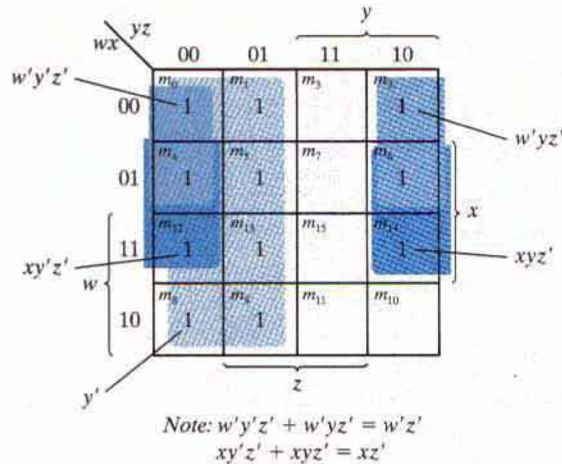


FIGURE 3.9
Map for Example 3.5, $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14) = y' + w'z' + xz'$

now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term with four literals), we combine it with squares already used to form an area of four adjacent squares. These squares make up the two middle rows and the two end columns, giving the term xz'. The simplified function is

$$F = y' + w'z' + xz'$$

■

EXAMPLE 3.6

Simplify the Boolean function

$$F = A'B'C' + B'CD' + A'BCD' + AB'C'$$

The area in the map covered by this function consists of the squares marked with 1's in Fig. 3.10. The function has four variables and, as expressed, consists of three terms with three literals each and one term with four literals. Each term with three literals is represented in the map by two squares. For example, $A'B'C'$ is represented in squares 0000 and 0001. The function can be simplified in the map by taking the 1's in the four corners to give the term $B'D'$. This is possible because these four squares are adjacent when the map is drawn in a surface with top and bottom edges, as well as left and right edges, touching one another. The two left-hand 1's in the top row are combined with the two 1's in the bottom row to give the term $B'C'$. The remaining 1 may be combined in a two-square area to give the term $A'CD'$. The simplified function is

$$F = B'D' + B'C' + A'CD'$$

■

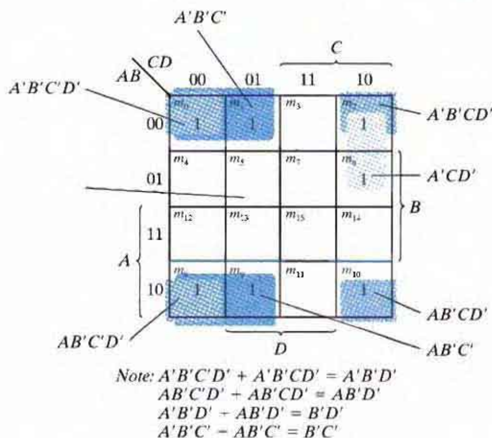


FIGURE 3.10
Map for Example 3.6, $A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$

Prime Implicants

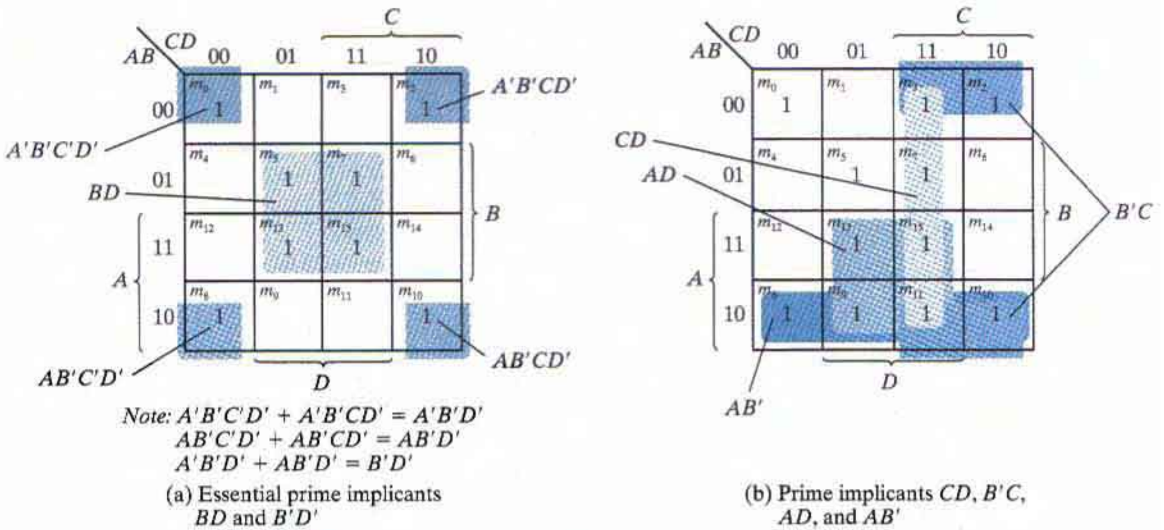
In choosing adjacent squares in a map, we must ensure that (1) all the minterms of the function are covered when we combine the squares, (2) the number of terms in the expression is minimized, and (3) there are no redundant terms (i.e., minterms already covered by other terms). Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of terms. A *prime implicant* is a product term obtained by combining the maximum possible number of adjacent squares in the map. If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be *essential.*

The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares. This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's form a prime implicant, provided that they are not within a group of four adjacent squares. Four adjacent 1's form a prime implicant if they are not within a group of eight adjacent squares, and so on. The essential prime implicants are found by looking at each square marked with a 1 and checking the number of prime implicants that cover it. The prime implicant is essential if it is the only prime implicant that covers the minterm.

Consider the following four-variable Boolean function:

$$F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

The minterms of the function are marked with 1's in the maps of Fig. 3.11. The partial map (part (a) of the figure) shows two essential prime implicants, each formed by collapsing four cells into a term having only two literals. One term is essential because there is only one way to include



(a) Essential prime implicants BD and $B'D'$

(b) Prime implicants CD, $B'C$, AD, and AB'

FIGURE 3.11
Simplification using prime implicants

minterm m_0 within four adjacent squares. These four squares define the term $B'D'$. Similarly, there is only one way that minterm m_5 can be combined with four adjacent squares, and this gives the second term BD. The two essential prime implicants cover eight minterms. The three minterms that were omitted from the partial map (m_3, m_9, and m_{11}) must be considered next.

Figure 3.11(b) shows all possible ways that the three minterms can be covered with prime implicants. Minterm m_3 can be covered with either prime implicant CD or prime implicant $B'C$. Minterm m_9 can be covered with either AD or AB'. Minterm m_{11} is covered with any one of the four prime implicants. The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants that cover minterms m_3, m_9, and m_{11}. There are four possible ways that the function can be expressed with four product terms of two literals each:

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

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

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

3.4 FIVE-VARIABLE MAP

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

The five-variable map is shown in Fig. 3.12. It consists of 2 four-variable maps with variables A, B, C, D, and E. Variable A distinguishes between the two maps, as indicated at the top of the diagram. The left-hand four-variable map represents the 16 squares in which $A = 0$, and the other four-variable map represents the squares in which $A = 1$. Minterms 0 through 15 belong with $A = 0$ and minterms 16 through 31 with $A = 1$. Each four-variable map retains the previously defined adjacency when taken separately. In addition, each square in the $A = 0$ map is adjacent to the corresponding square in the $A = 1$ map. For example, minterm 4 is adjacent to minterm 20 and minterm 15 to 31. The best way to visualize this new rule for adjacent squares is to consider the two half maps as being one on top of the other. Any two squares that fall one over the other are considered adjacent.

By following the procedure used for the five-variable map, it is possible to construct a six-variable map with 4 four-variable maps to obtain the required 64 squares. Maps with six or more variables need too many squares and are impractical to use. The alternative is to employ computer programs specifically written to facilitate the simplification of Boolean functions with a large number of variables.

By inspection, and taking into account the new definition of adjacent squares, it is possible to show that any 2^k adjacent squares, for $k = (0, 1, 2, \ldots, n)$ in an n-variable map, will represent an area that gives a term of $n - k$ literals. For this statement to have any meaning, however, n must be larger than k. When $n = k$, the entire area of the map is combined to give the

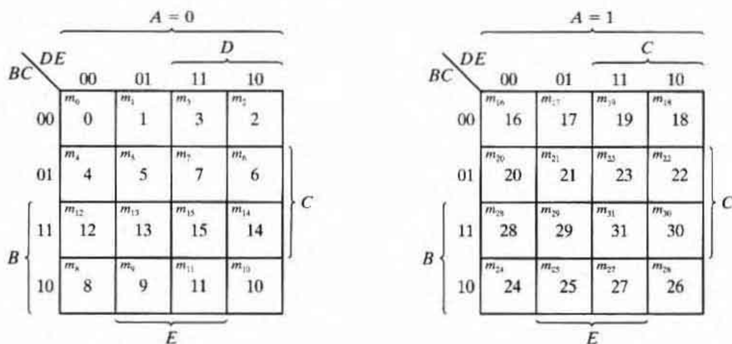


FIGURE 3.12
Five-variable map

Table 3.1
The Relationship between the Number of Adjacent Squares and the Number of Literals in the Term

	Number of Adjacent Squares	Number of Literals in a Term in an n-variable Map			
K	2^k	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0	1	2	3	4	5
1	2	1	2	3	4
2	4	0	1	2	3
3	8		0	1	2
4	16			0	1
5	32				0

identity function. Table 3.1 shows the relationship between the number of adjacent squares and the number of literals in the term. For example, eight adjacent squares combine an area in the five-variable map to give a term of two literals.

EXAMPLE 3.7

Simplify the Boolean function

$$F(A, B, C, D, E) = \Sigma(0, 2, 4, 6, 9, 13, 21, 23, 25, 29, 31)$$

The five-variable map for this function is shown in Fig. 3.13. There are six minterms from 0 to 15 that belong to the part of the map with $A = 0$. The other five minterms belong with $A = 1$. Four adjacent squares in the $A = 0$ map are combined to give the three-literal term $A'B'E'$. Note that it is necessary to include A' with the term because all the squares are associated with $A = 0$. The two squares in column 01 and the last two rows are common to both parts of the map. Therefore, they constitute four adjacent squares and give the three-literal term $BD'E$. Variable A is not included here because the adjacent squares belong to both $A = 0$ and $A = 1$. The term ACE is obtained from the four adjacent squares that are entirely within the $A = 1$ map. The simplified function is the logical sum of the three terms:

$$F = A'B'E' + BD'E + ACE$$

■

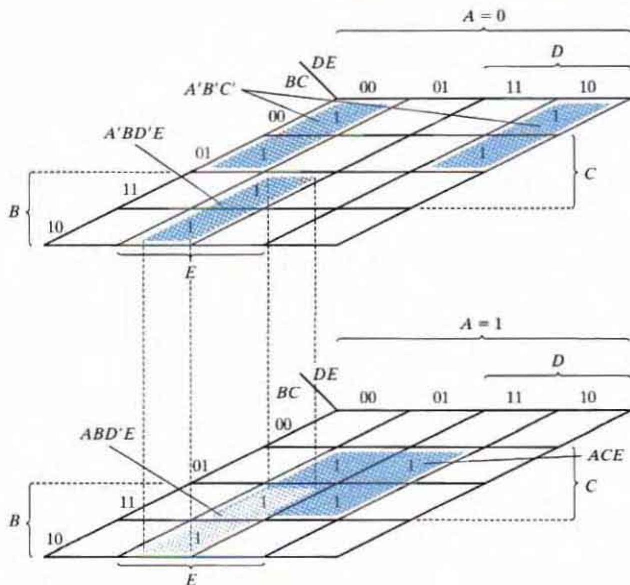


FIGURE 3.13
Map for Example 3.7, $F = A'B'E' + BD'E + ACE$

3.5 PRODUCT-OF-SUMS SIMPLIFICATION

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

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

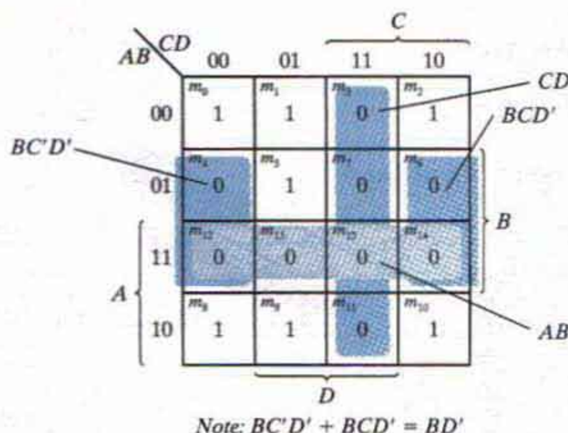


FIGURE 3.14
Map for Example 3.8, $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10) = B'D' + B'C' + A'C'D = (A' + B')(C' + D')(B' + D)$

EXAMPLE 3.8

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

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

The 1's marked in the map of Fig. 3.14 represent all the minterms of the function. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F. Combining the squares with 1's gives the simplified function in sum-of-products form:

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

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

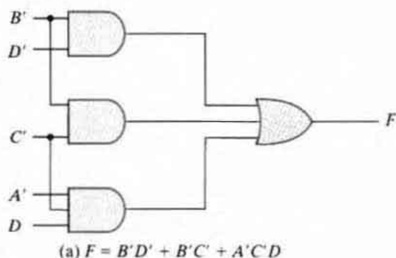
$$F' = AB + CD + BD'$$

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

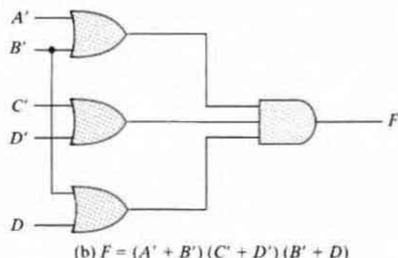
(b) $F = (A' + B')(C' + D')(B' + D)$

■

The implementation of the simplified expressions obtained in Example 3.8 is shown in Fig. 3.15. The sum-of-products expression is implemented in (a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in (b) in its product-of-sums form with a group of OR gates, one for each OR term. The outputs of the OR gates are connected to the inputs of a single AND gate. In each case, it is assumed that the input variables are directly



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



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

FIGURE 3.15
Gate implementations of the function of Example 3.8

Table 3.2
Truth Table of Function F

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

available in their complement, so inverters are not needed. The configuration pattern established in Fig. 3.15 is the general form by which any Boolean function is implemented when expressed in one of the standard forms. AND gates are connected to a single OR gate when in sum-of-products form; OR gates are connected to a single AND gate when in product-of-sums form. Either configuration forms two levels of gates. Thus, the implementation of a function in a standard form is said to be a two-level implementation.

Example 3.8 showed the procedure for obtaining the product-of-sums simplification when the function is originally expressed in the sum-of-minterms canonical form. The procedure is also valid when the function is originally expressed in the product-of-maxterms canonical form. Consider, for example, the truth table that defines the function F in Table 3.2. In sum-of-minterms form, this function is expressed as

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

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

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

In other words, the 1's of the function represent the minterms and the 0's represent the maxterms. The map for this function is shown in Fig. 3.16. One can start simplifying the function by first marking the 1's for each minterm that the function is a 1. The remaining squares are

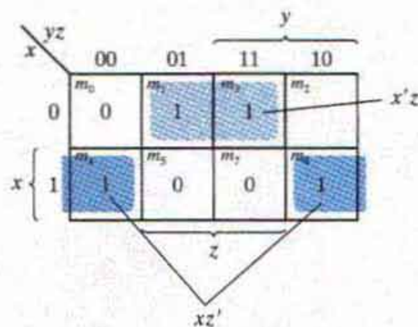


FIGURE 3.16
Map for the function of Table 3.2

marked by 0's. If, instead, the product of maxterms is initially given, one can start marking 0's in those squares listed in the function; the remaining squares are then marked by 1's. Once the 1's and 0's are marked, the function can be simplified in either one of the standard forms. For the sum of products, we combine the 1's to obtain

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

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

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

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

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

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

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

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

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

and then marking 0's in the squares representing the minterms of F'. The remaining squares are marked with 1's.

3.6 DON'T-CARE CONDITIONS

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. This pair of conditions assumes that all the combinations of the values for the variables of the function are valid. In practice, in some applications the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified.

Functions that have unspecified outputs for some input combinations are called *incompletely specified functions*. In most applications, we simply don't care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function *don't-care conditions*. These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

A don't-care minterm is a combination of variables whose logical value is not specified. Such a minterm cannot be marked with a 1 in the map, because it would require that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the particular minterm.

In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

EXAMPLE 3.9

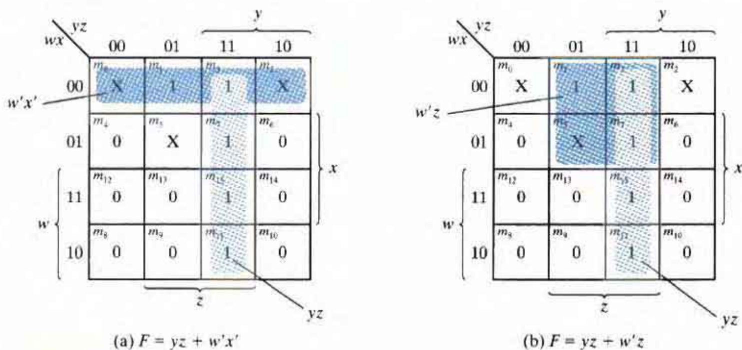
Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$$

which has the don't-care conditions

$$d(w, x, y, z) = \Sigma(0, 2, 5)$$

The minterms of F are the variable combinations that make the function equal to 1. The minterms of d are the don't-care minterms that may be assigned either 0 or 1. The map simplification is shown in Fig. 3.17. The minterms of F are marked by 1's, those of d are marked



(a) $F = yz + w'x'$

(b) $F = yz + w'z$

FIGURE 3.17
Example with don't-care conditions

by X's, and the remaining squares are filled with 0's. To get the simplified expression in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified. The term yz covers the four minterms in the third column. The remaining minterm, m_1, can be combined with minterm m_3 to give the three-literal term $w'x'z$. However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In part (a) of the diagram, don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

$$F = yz + w'x'$$

In part (b), don't-care minterm 5 is included with the 1's, and the simplified function is now

$$F = yz + w'z$$

Either one of the preceding two expressions satisfies the conditions stated for this example. ■

The previous example has shown that the don't-care minterms in the map are initially marked with X's and are considered as being either 0 or 1. The choice between 0 and 1 is made depending on the way the incompletely specified function is simplified. Once the choice is made, the simplified function obtained will consist of a sum of minterms that includes those minterms which were initially unspecified and have been chosen to be included with the 1's. Consider the two simplified expressions obtained in Example 3.9:

$$F(w, x, y, z) = yz + w'x' = \Sigma(0, 1, 2, 3, 7, 11, 15)$$
$$F(w, x, y, z) = yz + w'z = \Sigma(1, 3, 5, 7, 11, 15)$$

Both expressions include minterms 1, 3, 7, 11, and 15 that make the function F equal to 1. The don't-care minterms 0, 2, and 5 are treated differently in each expression. The first expression includes minterms 0 and 2 with the 1's and leaves minterm 5 with the 0's. The second expression includes minterm 5 with the 1's and leaves minterms 0 and 2 with the 0's. The two expressions represent two functions that are not algebraically equal. Both cover the specified minterms of the function, but each covers different don't-care minterms. As far as the incompletely specified function is concerned, either expression is acceptable because the only difference is in the value of F for the don't-care minterms.

It is also possible to obtain a simplified product-of-sums expression for the function of Fig. 3.17. In this case, the only way to combine the 0's is to include don't-care minterms 0 and 2 with the 0's to give a simplified complemented function:

$$F' = z' + wy'$$

Taking the complement of F' gives the simplified expression in product-of-sums form:

$$F(w, x, y, z) = z(w' + y) = \Sigma(1, 3, 5, 7, 11, 15)$$

In this case, we include minterms 0 and 2 with the 0's and minterm 5 with the 1's.

3.7 NAND AND NOR IMPLEMENTATION

Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

NAND Circuits

The NAND gate is said to be a universal gate because any digital system can be implemented with it. To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations of AND, OR, and complement can be obtained with NAND gates alone. This is indeed shown in Fig. 3.18. The complement operation is obtained from a one-input NAND gate that behaves exactly like an inverter. The AND operation requires two NAND gates. The first produces the NAND operation and the second inverts the logical sense of the signal. The OR operation is achieved through a NAND gate with additional inverters in each input.

A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit manipulation techniques that change AND–OR diagrams to NAND diagrams.

To facilitate the conversion to NAND logic, it is convenient to define an alternative graphic symbol for the gate. Two equivalent graphic symbols for the NAND gate are shown in Fig. 3.19.

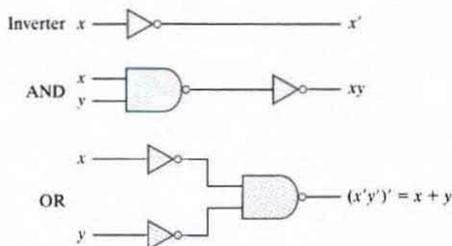


FIGURE 3.18
Logic operations with NAND gates

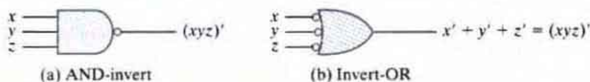


FIGURE 3.19
Two graphic symbols for the NAND gate

The AND-invert symbol has been defined previously and consists of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble. Alternatively, it is possible to represent a NAND gate by an OR graphic symbol that is preceded by a bubble in each input. The invert-OR symbol for the NAND gate follows DeMorgan's theorem and the convention that the negation indicator denotes complementation. The two graphic symbols' representations are useful in the analysis and design of NAND circuits. When both symbols are mixed in the same diagram, the circuit is said to be in mixed notation.

Two-Level Implementation

The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form. To see the relationship between a sum-of-product expression and its equivalent NAND implementation, consider the logic diagrams drawn in Fig. 3.20. All three diagrams are equivalent and implement the function

$$F = AB + CD$$

The function is implemented in (a) with AND and OR gates. In (b), the AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an OR-invert graphic symbol. Remember that a bubble denotes complementation and two bubbles along the same line represent double complementation, so both can be removed. Removing the bubbles on the gates of (b) produces the circuit of (a). Therefore, the two diagrams implement the same function and are equivalent.

In Fig. 3.20(c), the output NAND gate is redrawn with the AND-invert graphic symbol. In drawing NAND logic diagrams, the circuit shown in either (b) or (c) is acceptable. The

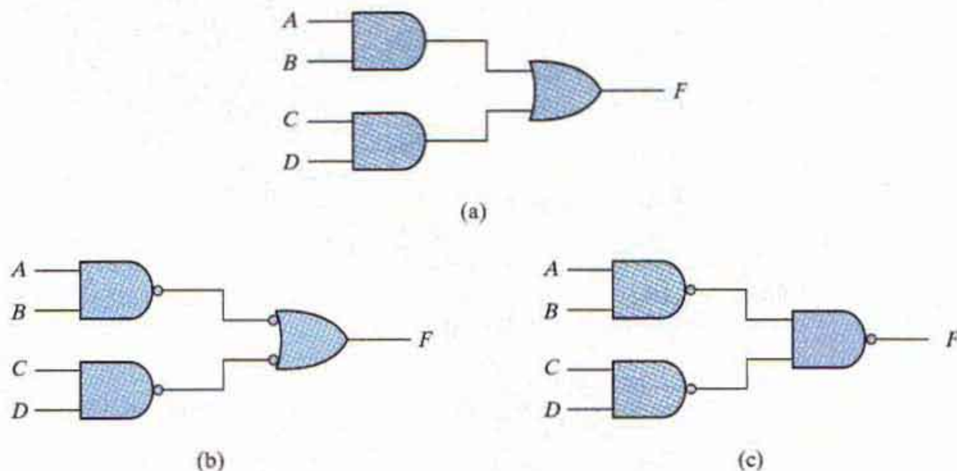


FIGURE 3.20
Three ways to implement $F = AB + CD$

one in (b) is in mixed notation and represents a more direct relationship to the Boolean expression it implements. The NAND implementation in Fig. 3.20(c) can be verified algebraically. The function it implements can easily be converted to sum-of-products form by DeMorgan's theorem:

$$F = ((AB)'(CD)')' = AB + CD$$

EXAMPLE 3.10

Implement the following Boolean function with NAND gates:

$$F(x, y, z) = (1, 2, 3, 4, 5, 7)$$

The first step is to simplify the function into sum-of-products form. This is done by means of the map of Fig. 3.21(a), from which the simplified function is obtained:

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

The two-level NAND implementation is shown in Fig. 3.21(b) in mixed notation. Note that input z must have a one-input NAND gate (an inverter) to compensate for the bubble in the second-level gate. An alternative way of drawing the logic diagram is given in Fig. 3.21(c). Here, all the NAND gates are drawn with the same graphic symbol. The inverter with input z has been removed, but the input variable is complemented and denoted by z'.

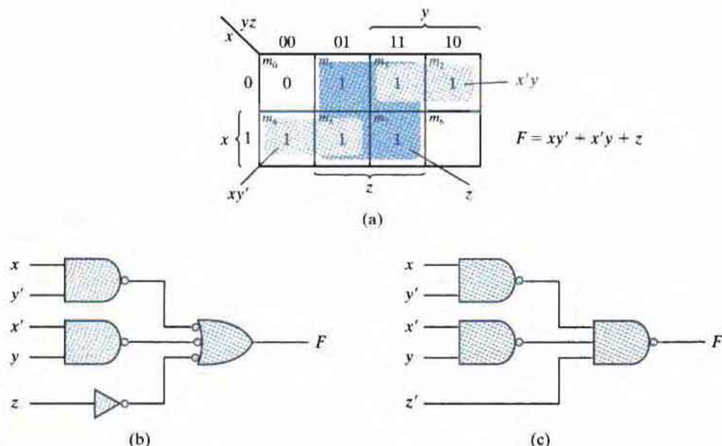


FIGURE 3.21
Solution to Example 3.10

The procedure described in the previous example indicates that a Boolean function can be implemented with two levels of NAND gates. The procedure for obtaining the logic diagram from a Boolean function is as follows:

1. Simplify the function and express it in sum-of-products form.
2. Draw a NAND gate for each product term of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates.
3. Draw a single gate using the AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from outputs of first-level gates.
4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second-level NAND gate.

Multilevel NAND Circuits

The standard form of expressing Boolean functions results in a two-level implementation. There are occasions, however, when the design of digital systems results in gating structures with three or more levels. The most common procedure in the design of multilevel circuits is to express the Boolean function in terms of AND, OR, and complement operations. The function can then be implemented with AND and OR gates. After that, if necessary, it can be converted into an all-NAND circuit. Consider, for example, the Boolean function

$$F = A(CD + B) + BC'$$

Although it is possible to remove the parentheses and reduce the expression into a standard sum-of-products form, we choose to implement it as a multilevel circuit for illustration. The AND–OR implementation is shown in Fig. 3.22(a). There are four levels of gating in the circuit. The first level has two AND gates. The second level has an OR gate followed by an AND gate in the third level and an OR gate in the fourth level. A logic diagram with a pattern of alternating levels of AND and OR gates can easily be converted into a NAND circuit with the use of mixed notation, shown in Fig. 3.22(b). The procedure is to change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol. The NAND circuit performs the same logic as the AND–OR diagram as long as there are two bubbles along the same line. The bubble associated with input B causes an extra complementation, which must be compensated for by changing the input literal to B'.

The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation is as follows:

1. Convert all AND gates to NAND gates with AND-invert graphic symbols.
2. Convert all OR gates to NAND gates with invert-OR graphic symbols.
3. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

As another example, consider the multilevel Boolean function

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

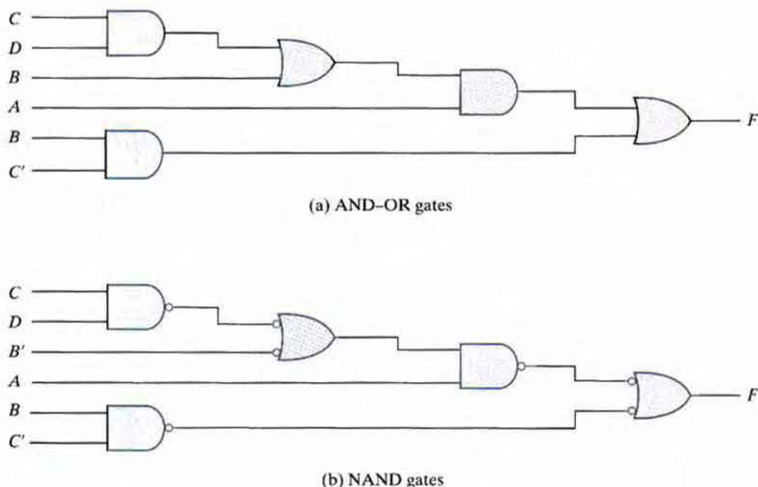


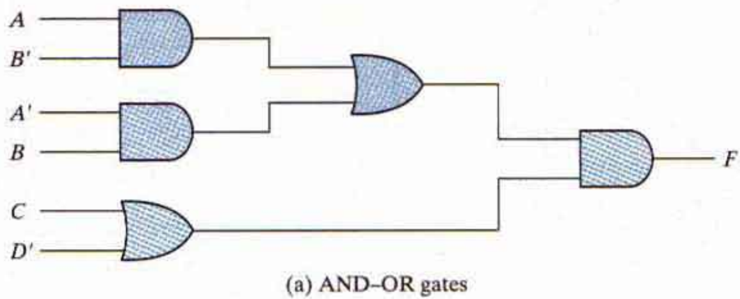
FIGURE 3.22
Implementing $F = A(CD + B) + BC'$

The AND–OR implementation of this function is shown in Fig. 3.23(a) with three levels of gating. The conversion to NAND with mixed notation is presented in part (b) of the diagram. The two additional bubbles associated with inputs C and D' cause these two literals to be complemented to C' and D. The bubble in the output NAND gate complements the output value, so we need to insert an inverter gate at the output in order to complement the signal again and get the original value back.

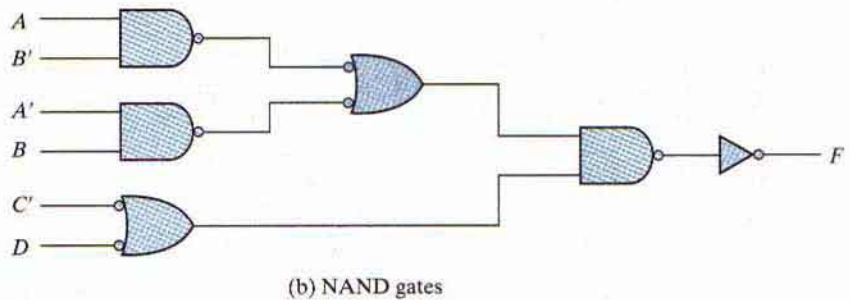
NOR Implementation

The NOR operation is the dual of the NAND operation. Therefore, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic. The NOR gate is another universal gate that can be used to implement any Boolean function. The implementation of the complement, OR, and AND operations with NOR gates is shown in Fig. 3.24. The complement operation is obtained from a one-input NOR gate that behaves exactly like an inverter. The OR operation requires two NOR gates, and the AND operation is obtained with a NOR gate that has inverters in each input.

The two graphic symbols for the mixed notation are shown in Fig. 3.25. The OR-invert symbol defines the NOR operation as an OR followed by a complement. The invert-AND symbol complements each input and then performs an AND operation. The two symbols designate the same NOR operation and are logically identical because of DeMorgan's theorem.



(a) AND–OR gates



(b) NAND gates

FIGURE 3.23
Implementing $F = (AB' + A'B)(C + D')$

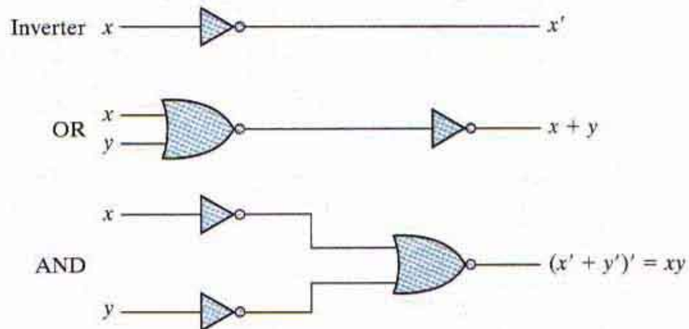
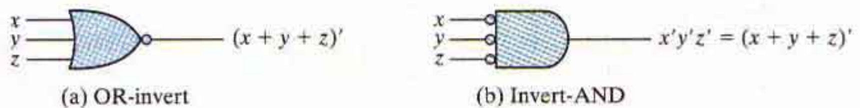


FIGURE 3.24
Logic operations with NOR gates



(a) OR-invert

(b) Invert-AND

FIGURE 3.25
Two graphic symbols for the NOR gate

A two-level implementation with NOR gates requires that the function be simplified into product-of-sums form. Remember that the simplified product-of-sums expression is obtained from the map by combining the 0's and complementing. A product-of-sums expression is implemented with a first level of OR gates that produce the sum terms followed by a second-level AND gate to produce the product. The transformation from the OR–AND diagram to a NOR diagram is achieved by changing the OR gates to NOR gates with OR-invert graphic symbols and the AND gate to a NOR gate with an invert-AND graphic symbol. A single literal term going into the second-level gate must be complemented. Fig. 3.26 shows the NOR implementation of a function expressed as a product of sums:

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

The OR–AND pattern can easily be detected by the removal of the bubbles along the same line. Variable E is complemented to compensate for the third bubble at the input of the second-level gate.

The procedure for converting a multilevel AND–OR diagram to an all-NOR diagram is similar to the one presented for NAND gates. For the NOR case, we must convert each OR gate to an OR-invert symbol and each AND gate to an invert-AND symbol. Any bubble that is not compensated by another bubble along the same line needs an inverter, or the complementation of the input literal.

The transformation of the AND–OR diagram of Fig. 3.23(a) into a NOR diagram is shown in Fig. 3.27. The Boolean function for this circuit is

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

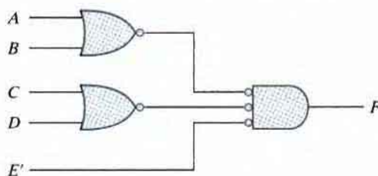


FIGURE 3.26
Implementing $F = (A + B)(C + D)E$

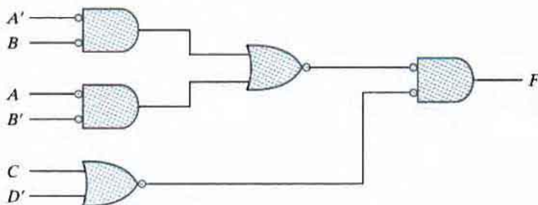


FIGURE 3.27
Implementing $F = (AB' + A'B)(C + D')$ with NOR gates

The equivalent AND–OR diagram can be recognized from the NOR diagram by removing all the bubbles. To compensate for the bubbles in four inputs, it is necessary to complement the corresponding input literals.

3.8 OTHER TWO-LEVEL IMPLEMENTATIONS

The types of gates most often found in integrated circuits are NAND and NOR gates. For this reason, NAND and NOR logic implementations are the most important from a practical point of view. Some (but not all) NAND or NOR gates allow the possibility of a wire connection between the outputs of two gates to provide a specific logic function. This type of logic is called *wired logic*. For example, open-collector TTL NAND gates, when tied together, perform wired-AND logic. (The open-collector TTL gate is shown in Chapter 10, Fig. 10.11.) The wired-AND logic performed with two NAND gates is depicted in Fig. 3.28(a). The AND gate is drawn with the lines going through the center of the gate to distinguish it from a conventional gate. The wired-AND gate is not a physical gate, but only a symbol to designate the function obtained from the indicated wired connection. The logic function implemented by the circuit of Fig. 3.28(a) is

$$F = (AB)' \cdots (CD)' = (AB + CD)' = (A' + B')(C' + D')$$

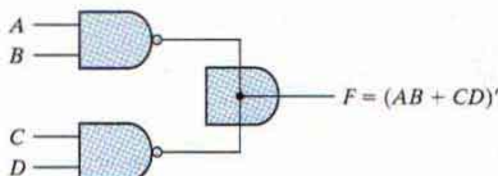
and is called an AND–OR–INVERT function.

Similarly, the NOR outputs of ECL gates (see Figure 10.17) can be tied together to perform a wired-OR function. The logic function implemented by the circuit of Fig. 3.28(b) is

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

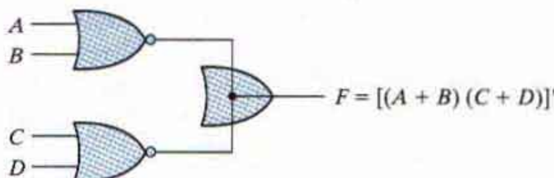
and is called an OR–AND–INVERT function.

A wired-logic gate does not produce a physical second-level gate, since it is just a wire connection. Nevertheless, for discussion purposes, we will consider the circuits of Fig. 3.28 as two-level implementations. The first level consists of NAND (or NOR) gates and the second level has a single AND (or OR) gate. The wired connection in the graphic symbol will be omitted in subsequent discussions.



(a) Wired-AND in open-collector TTL NAND gates.

(AND–OR–INVERT)



(b) Wired-OR in ECL gates

(OR–AND–INVERT)

FIGURE 3.28
Wired logic
(a) Wired-AND logic with two NAND gates
(b) Wired-OR in emitter-coupled logic (ECL) gates

Nondegenerate Forms

It will be instructive from a theoretical point of view to find out how many two-level combinations of gates are possible. We consider four types of gates: AND, OR, NAND, and NOR. If we assign one type of gate for the first level and one type for the second level, we find that there are 16 possible combinations of two-level forms. (The same type of gate can be in the first and second levels, as in a NAND–NAND implementation.) Eight of these combinations are said to be *degenerate* forms because they degenerate to a single operation. This can be seen from a circuit with AND gates in the first level and an AND gate in the second level. The output of the circuit is merely the AND function of all input variables. The remaining eight *nondegenerate* forms produce an implementation in sum-of-products form or product-of-sums form. The eight nondegenerate forms are as follows:

AND–OR	OR–AND
NAND–NAND	NOR–NOR
NOR–OR	NAND–AND
OR–NAND	AND–NOR

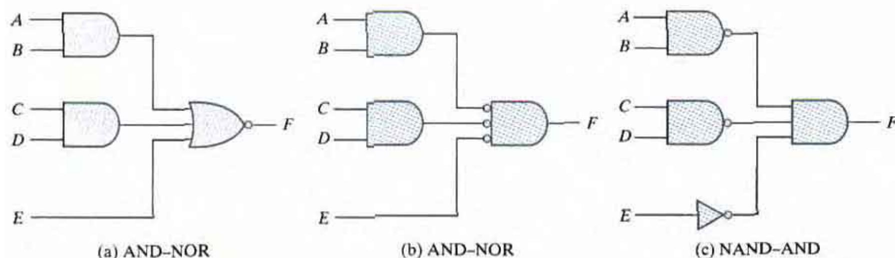
The first gate listed in each of the forms constitutes a first level in the implementation. The second gate listed is a single gate placed in the second level. Note that any two forms listed on the same line are duals of each other.

The AND–OR and OR–AND forms are the basic two-level forms discussed in Section 3.4. The NAND–NAND and NOR–NOR forms were presented in Section 3.6. The remaining four forms are investigated in this section.

AND–OR–INVERT Implementation

The two forms NAND–AND and AND–NOR are equivalent and can be treated together. Both perform the AND–OR–INVERT function, as shown in Fig. 3.29. The AND–NOR form resembles the AND–OR form, but with an inversion done by the bubble in the output of the NOR gate. It implements the function

$$F = (AB + CD + E)'$$



(a) AND-NOR (b) AND-NOR (c) NAND-AND

FIGURE 3.29
AND–OR–INVERT circuits, $F = (AB + CD + E)'$

By using the alternative graphic symbol for the NOR gate, we obtain the diagram of Fig. 3.29(b). Note that the single variable E is *not* complemented, because the only change made is in the graphic symbol of the NOR gate. Now we move the bubble from the input terminal of the second-level gate to the output terminals of the first-level gates. An inverter is needed for the single variable in order to compensate for the bubble. Alternatively, the inverter can be removed, provided that input E is complemented. The circuit of Fig. 3.29(c) is a NAND–AND form and was shown in Fig. 3.28 to implement the AND–OR–INVERT function.

An AND–OR implementation requires an expression in sum-of-products form. The AND–OR–INVERT implementation is similar, except for the inversion. Therefore, if the *complement* of the function is simplified into sum-of-products form (by combining the 0's in the map), it will be possible to implement F' with the AND–OR part of the function. When F' passes through the always present output inversion (the INVERT part), it will generate the output F of the function. An example for the AND–OR–INVERT implementation will be shown subsequently.

OR–AND–INVERT Implementation

The OR–NAND and NOR–OR forms perform the OR–AND–INVERT function, as shown in Fig. 3.30. The OR–NAND form resembles the OR–AND form, except for the inversion done by the bubble in the NAND gate. It implements the function

$$F = [(A + B)(C + D)E]'$$

By using the alternative graphic symbol for the NAND gate, we obtain the diagram of Fig. 3.30(b). The circuit in (c) is obtained by moving the small circles from the inputs of the second-level gate to the outputs of the first-level gates. The circuit of Fig. 3.30(c) is a NOR–OR form and was shown in Fig. 3.28 to implement the OR–AND–INVERT function.

The OR–AND–INVERT implementation requires an expression in product-of-sums form. If the complement of the function is simplified into that form, we can implement F' with the OR–AND part of the function. When F' passes through the INVERT part, we obtain the complement of F', or F, in the output.

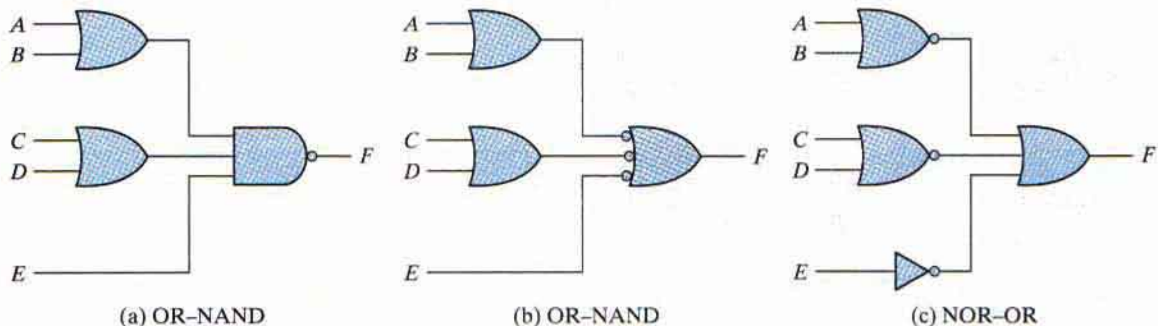


FIGURE 3.30
OR–AND–INVERT circuits, $F = [(A + B)(C + D)E]'$

Table 3.3
Implementation with Other Two-Level Forms

Equivalent Nondegenerate Form (a)	Equivalent Nondegenerate Form (b)*	Implements the Function	Simplify F' into	To Get an Output of
AND–NOR	NAND–AND	AND–OR–INVERT	Sum-of-products form by combining 0's in the map.	F
OR–NAND	NOR–OR	OR–AND–INVERT	Product-of-sums form by combining 1's in the map and then complementing.	F

*Form (b) requires an inverter for a single literal term.

Tabular Summary and Example

Table 3.3 summarizes the procedures for implementing a Boolean function in any one of the four 2-level forms. Because of the INVERT part in each case, it is convenient to use the simplification of F' (the complement) of the function. When F' is implemented in one of these forms, we obtain the complement of the function in the AND–OR or OR–AND form. The four 2-level forms invert this function, giving an output that is the complement of F'. This is the normal output F.

EXAMPLE 3.11

Implement the function of Fig. 3.31(a) with the four 2-level forms listed in Table 3.3.
The complement of the function is simplified into sum-of-products form by combining the 0's in the map:

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

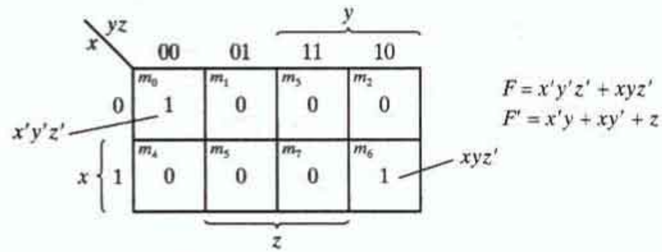
The normal output for this function can be expressed as

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

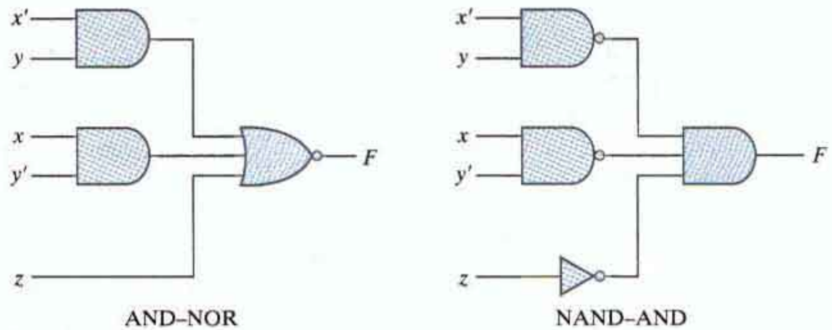
which is in the AND–OR–INVERT form. The AND–NOR and NAND–AND implementations are shown in Fig. 3.31(b). Note that a one-input NAND, or inverter, gate is needed in the NAND–AND implementation, but not in the AND–NOR case. The inverter can be removed if we apply the input variable z' instead of z.

The OR–AND–INVERT forms require a simplified expression of the complement of the function in product-of-sums form. To obtain this expression, we first combine the 1's in the map:

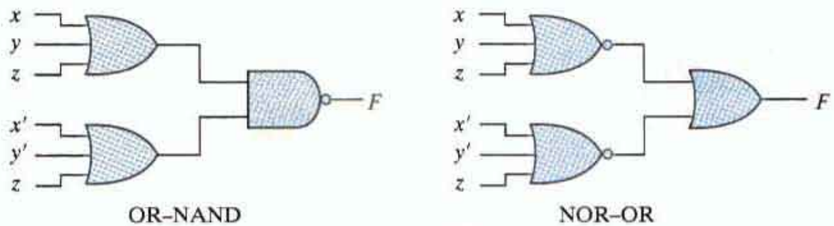
$$F = x'y'z' + xyz'$$



(a) Map simplification in sum of products



(b) $F = (x'y + xy' + z)'$



(c) $F = [(x + y + z)(x' + y' + z)]'$

FIGURE 3.31
Other two-level implementations

Then we take the complement of the function:

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

The normal output F can now be expressed in the form

$$F = [(x + y + z)(x' + y' + z)]'$$

which is the OR–AND–INVERT form. From this expression, we can implement the function in the OR–NAND and NOR–OR forms, as shown in Fig. 3.31(c).

3.9 EXCLUSIVE-OR FUNCTION

The exclusive-OR (XOR), denoted by the symbol $\oplus$, is a logical operation that performs the following Boolean operation:

$$x \oplus y = xy' + x'y$$

The exclusive-OR is equal to 1 if only x is equal to 1 or if only y is equal to 1 (i.e., x and y differ in value), but not when both are equal to 1 or when both are equal to 0. The exclusive-NOR, also known as equivalence, performs the following Boolean operation:

$$(x \oplus y)' = xy + x'y'$$

The exclusive-NOR is equal to 1 if both x and y are equal to 1 or if both are equal to 0. The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation:

$$(x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = xy + x'y'$$

The following identities apply to the exclusive-OR operation:

$$x \oplus 0 = x$$
$$x \oplus 1 = x'$$
$$x \oplus x = 0$$
$$x \oplus x' = 1$$
$$x \oplus y' = x' \oplus y = (x \oplus y)'$$

Any of these identities can be proven with a truth table or by replacing the $\oplus$ operation by its equivalent Boolean expression. Also, it can be shown that the exclusive-OR operation is both commutative and associative; that is,

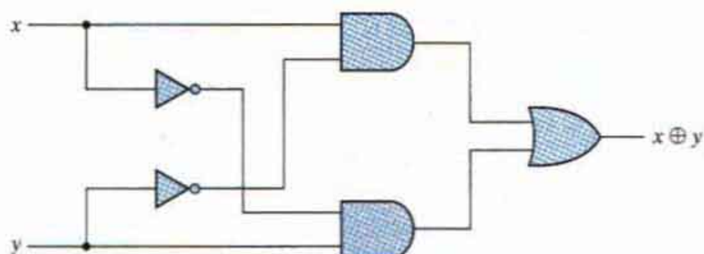
$$A \oplus B = B \oplus A$$

and

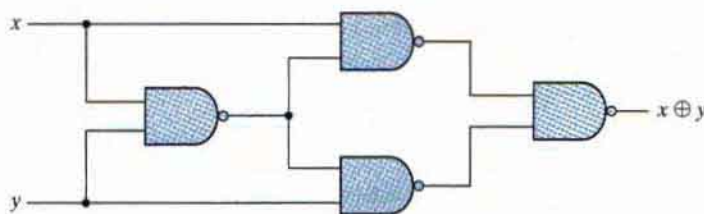
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation. It also means that we can evaluate a three-variable exclusive-OR operation in any order, and for this reason, three or more variables can be expressed without parentheses. This would imply the possibility of using exclusive-OR gates with three or more inputs. However, multiple-input exclusive-OR gates are difficult to fabricate with hardware. In fact, even a two-input function is usually constructed with other types of gates. A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate, as shown in Fig. 3.32(a). Figure 3.32(b) shows the implementation of the exclusive-OR with four NAND gates. The first NAND gate performs the operation $(xy)' = (x' + y')$. The other two-level NAND circuit produces the sum of products of its inputs:

$$(x' + y')x + (x' + y')y = xy' + x'y = x \oplus y$$



(a) With AND–OR–NOT gates



(b) With NAND gates

FIGURE 3.32
Exclusive-OR implementations

Only a limited number of Boolean functions can be expressed in terms of exclusive-OR operations. Nevertheless, this function emerges quite often during the design of digital systems. It is particularly useful in arithmetic operations and error detection and correction circuits.

Odd Function

The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the $\oplus$ symbol with its equivalent Boolean expression. In particular, the three-variable case can be converted to a Boolean expression as follows:

$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \Sigma(1, 2, 4, 7) \end{aligned}$$

The Boolean expression clearly indicates that the three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1. Contrary to the two-variable case, in which only one variable must be equal to 1, in the case of three or more variables the requirement is that an odd number of variables be equal to 1. As a consequence, the multiple-variable exclusive-OR operation is defined as an *odd function.*

The Boolean function derived from the three-variable exclusive-OR operation is expressed as the logical sum of four minterms whose binary numerical values are 001, 010, 100, and 111. Each of these binary numbers has an odd number of 1's. The remaining four minterms

not included in the function are 000, 011, 101, and 110, and they have an even number of 1's in their binary numerical values. In general, an n-variable exclusive-OR function is an odd function defined as the logical sum of the $2^n/2$ minterms whose binary numerical values have an odd number of 1's.

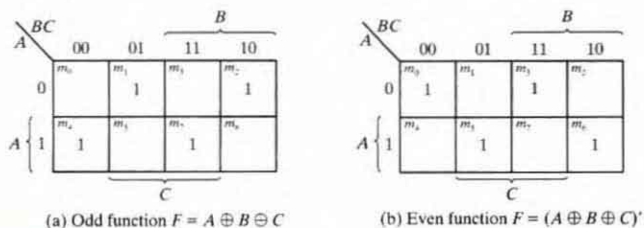
The definition of an odd function can be clarified by plotting it in a map. Figure 3.33(a) shows the map for the three-variable exclusive-OR function. The four minterms of the function are a unit distance apart from each other. The odd function is identified from the four minterms whose binary values have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig. 3.33(b), the three-variable even function is equal to 1 when an even number of its variables is equal to 1 (including the condition that none of the variables is equal to 1).

The three-input odd function is implemented by means of two-input exclusive-OR gates, as shown in Fig. 3.34(a). The complement of an odd function is obtained by replacing the output gate with an exclusive-NOR gate, as shown in Fig. 3.34(b).

Consider now the four-variable exclusive-OR operation. By algebraic manipulation, we can obtain the sum of minterms for this function:

$$
\begin{aligned}
A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\
&= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\
&= \Sigma(1, 2, 4, 7, 8, 11, 13, 14)
\end{aligned}
$$

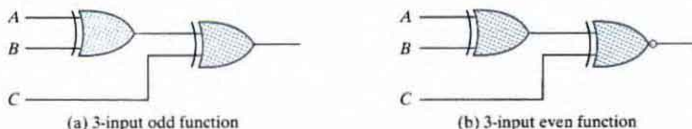
There are 16 minterms for a four-variable Boolean function. Half of the minterms have binary numerical values with an odd number of 1's; the other half of the minterms have binary numerical



(a) Odd function $F = A \oplus B \ominus C$

(b) Even function $F = (A \oplus B \oplus C)'$

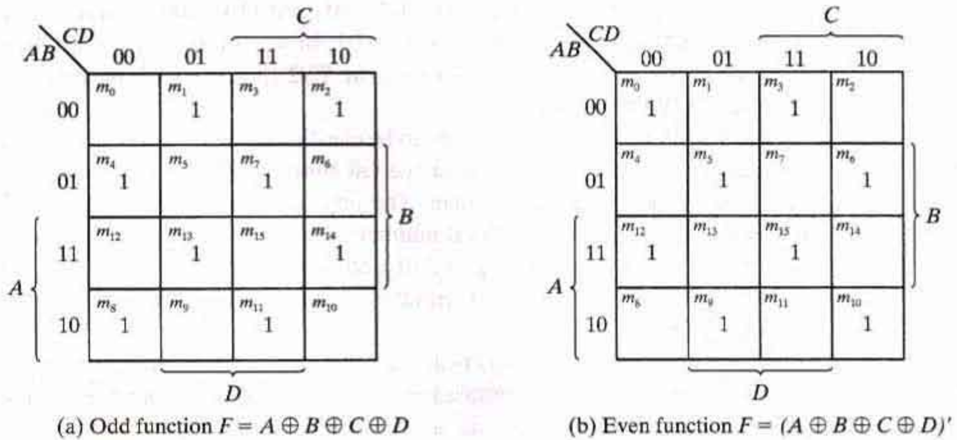
FIGURE 3.33
Map for a three-variable exclusive-OR function



(a) 3-input odd function

(b) 3-input even function

FIGURE 3.34
Logic diagram of odd and even functions



(a) Odd function $F = A \oplus B \oplus C \oplus D$

(b) Even function $F = (A \oplus B \oplus C \oplus D)'$

FIGURE 3.35
Map for a four-variable exclusive-OR function

values with an even number of 1's. In plotting the function in the map, the binary numerical value for a minterm is determined from the row and column numbers of the square that represents the minterm. The map of Fig. 3.35(a) is a plot of the four-variable exclusive-OR function. This is an odd function because the binary values of all the minterms have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig. 3.35(b), the four-variable even function is equal to 1 when an even number of its variables is equal to 1.

Parity Generation and Checking

Exclusive-OR functions are very useful in systems requiring error detection and correction codes. As discussed in Section 1.7, a parity bit is used for the purpose of detecting errors during the transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a *parity generator*. The circuit that checks the parity in the receiver is called a *parity checker*.

As an example, consider a three-bit message to be transmitted together with an even parity bit. Table 3.4 shows the truth table for the parity generator. The three bits—x, y, and z—constitute the message and are the inputs to the circuit. The parity bit P is the output. For even parity, the bit P must be generated to make the total number of 1's (including P) even. From the truth table, we see that P constitutes an odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's. Therefore, P can be expressed as a three-variable exclusive-OR function:

$$P = x \oplus y \oplus z$$

The logic diagram for the parity generator is shown in Fig. 3.36(a).

Table 3.4
Even-Parity-Generator Truth Table

Three-Bit Message			Parity Bit
x	y	z	P
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

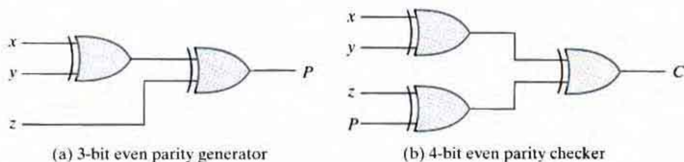


FIGURE 3.36
Logic diagram of a parity generator and checker

The three bits in the message, together with the parity bit, are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission. Since the information was transmitted with even parity, the four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission. The output of the parity checker, denoted by C, will be equal to 1 if an error occurs—that is, if the four bits received have an odd number of 1's. Table 3.5 is the truth table for the even-parity checker. From it, we see that the function C consists of the eight minterms with binary numerical values having an odd number of 1's. The table corresponds to the map of Fig. 3.35(a), which represents an odd function. The parity checker can be implemented with exclusive-OR gates:

$$C = x \oplus y \oplus z \oplus P$$

The logic diagram of the parity checker is shown in Fig. 3.36(b).

It is worth noting that the parity generator can be implemented with the circuit of Fig. 3.36(b) if the input P is connected to logic 0 and the output is marked with P. This is because $z \oplus 0 = z$, causing the value of z to pass through the gate unchanged. The advantage of this strategy is that the same circuit can be used for both parity generation and checking.

Table 3.5
Even-Parity-Checker Truth Table

Four Bits Received				Parity Error Check
x	*y*	*z*	*P*	*C*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

It is obvious from the foregoing example that parity generation and checking circuits always have an output function that includes half of the minterms whose numerical values have either an odd or even number of 1's. As a consequence, they can be implemented with exclusive-OR gates. A function with an even number of 1's is the complement of an odd function. It is implemented with exclusive-OR gates, except that the gate associated with the output must be an exclusive-NOR to provide the required complementation.

3.10 HARDWARE DESCRIPTION LANGUAGE

Manual methods for designing logic circuits are feasible only when the circuit is small. For anything else (i.e., a practical circuit), designers use computer-based design tools. Coupled with a correct-by-construction methodology, computer-based design tools leverage the creativity and effort of a designer and reduce the risk of producing a flawed design. Prototype integrated circuits are too expensive and time consuming to build, so all modern design tools rely on a hardware description language to describe, design, and test a circuit in software before it is ever manufactured.

A *hardware description language* (HDL) is a computer-based language that describes the hardware of digital systems in a textual form. It resembles an ordinary computer programming language, such as C, but is specifically oriented to describing hardware structures and the behavior of logic circuits. It can be used to represent logic diagrams, truth tables, Boolean

expressions, and complex abstractions of the behavior of a digital system. One way to view an HDL is to observe that it describes a relationship between signals that are the inputs to a circuit and the signals that are outputs of the circuit. For example, an HDL description of an AND gate describes how the logic value of the gate's output is determined by the logic values of its inputs.

As a *documentation* language, an HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers. The language content can be stored, retrieved, edited, and transmitted easily and processed by computer software in an efficient manner.

HDLs are used in several major steps in the design flow of an integrated circuit: design entry, functional simulation or verification, logic synthesis, timing verification, and fault simulation.

Design entry creates an HDL-based description of the functionality that is to be implemented in hardware. Depending on the HDL, the description can be in a variety of forms: Boolean logic equations, truth tables, a netlist of interconnected gates, or an abstract behavioral model. The HDL model may also represent a partition of a larger circuit into smaller interconnected and interacting functional units.

Logic simulation displays the behavior of a digital system through the use of a computer. A simulator interprets the HDL description and either produces readable output, such as a time-ordered sequence of input and output signal values, or displays waveforms of the signals. The simulation of a circuit predicts how the hardware will behave before it is actually fabricated. Simulation allows the detection of functional errors in a design without having to physically create and operate the circuit. Errors that are detected during a simulation can be corrected by modifying the appropriate HDL statements. The stimulus (i.e., the logic values of the inputs to a circuit) that tests the functionality of the design is called a *test bench*. Thus, to simulate a digital system, the design is first described in an HDL and then verified by simulating the design and checking it with a test bench, which is also written in the HDL. An alternative and more complex approach relies on formal mathematical methods to prove that a circuit is functionally correct. We will focus exclusively on simulation.

Logic synthesis is the process of deriving a list of physical components and their interconnections (called a *netlist*) from the model of a digital system described in an HDL. The netlist can be used to fabricate an integrated circuit or to lay out a printed circuit board with the hardware counterparts of the gates in the list. Logic synthesis is similar to compiling a program in a conventional high-level language. The difference is that, instead of producing an object code, logic synthesis produces a database describing the elements and structure of a circuit. The database specifies how to fabricate a physical integrated circuit that implements in silicon the functionality described by statements made in an HDL. Logic synthesis is based on formal exact procedures that implement digital circuits and addresses that part of a digital design which can be automated with computer software. The design of today's large, complex circuits is made possible by logic synthesis software.

Timing verification confirms that the fabricated integrated circuit will operate at a specified speed. Because each logic gate in a circuit has a propagation delay, a signal transition at the input of a circuit cannot immediately cause a change in the logic value of the output of a circuit. Propagation delays ultimately limit the speed at which a circuit can operate. Timing

verification checks each signal path to verify that it is not compromised by propagation delay. This step is done after logic synthesis specifies the actual devices that will compose a circuit and before the circuit is released for production.

In VLSI circuit design, *fault simulation* compares the behavior of an ideal circuit with the behavior of a circuit that contains a process-induced flaw. Dust and other particulates in the atmosphere of the clean room can cause a circuit to be fabricated with a fault. A circuit with a fault will not exhibit the same functionality as a fault-free circuit. Fault simulation is used to identify input stimuli that can be used to reveal the difference between the faulty circuit and the fault-free circuit. These test patterns will be used to test fabricated devices to ensure that only good devices are shipped to the customer. Test generation and fault simulation may occur at different steps in the design process, but they are always done before production in order to avoid the disaster of producing a circuit whose internal logic cannot be tested.

Companies that design integrated circuits use proprietary and public HDLs. In the public domain, there are two standard HDLs that are supported by the IEEE: VHDL and Verilog. VHDL is a Department of Defense–mandated language. (The *V* in VHDL stands for the first letter in VHSIC, an acronym for very high speed integrated circuit.) Verilog began as a proprietary HDL of Cadence Design Systems, but Cadence transferred control of Verilog to a consortium of companies and universities known as Open Verilog International (OVI) as a step leading to its adoption as an IEEE standard. VHDL is more difficult to learn than Verilog. Because Verilog is an easier language than VHDL to describe, learn, and use, we have chosen it for this book. However, the Verilog HDL descriptions listed throughout the book are not just about Verilog, but also serve to introduce a design methodology based on the concept of computer-aided modeling of digital systems by means of a typical hardware description language. Our emphasis will be on the modeling, verification, and synthesis (both manual and automated) of Verilog models of circuits having specified behavior. The Verilog HDL was initially approved as a standard HDL in 1995; revised and enhanced versions of the language were approved in 2001 and 2005. We will address only those features of Verilog, including the latest standard, that support our discussion of HDL-based design methodology for integrated circuits.

Module Declaration

The language reference manual for the Verilog HDL presents a syntax that describes precisely the constructs that can be used in the language. In particular, a Verilog model is composed of text using keywords, of which there are about 100. Keywords are predefined lowercase identifiers that define the language constructs. Examples of keywords are **module**, **endmodule**, **input**, **output**, **wire**, **and**, **or**, and **not**. For clarity, keywords will be displayed in boldface in the text in all examples of code and wherever it is appropriate to call attention to their use. Any text between two forward slashes (//) and the end of the line is interpreted as a comment and will have no effect on a simulation using the model. Multiline comments begin with /* and terminate with */. Blank spaces are ignored, but they may not appear within the text of a keyword, a user-specified identifier, an operator, or the representation of a number. Verilog is case sensitive, which means that uppercase and lowercase letters are distinguishable (e.g., **not** is not the same as NOT). The term *module* refers to the text enclosed

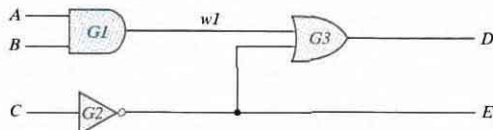


FIGURE 3.37
Circuit to demonstrate an HDL

by the keyword pair **module** ... **endmodule**. A module is the fundamental descriptive unit in the Verilog language. It is declared by the keyword **module** and must always be terminated by the keyword **endmodule**.

Combinational logic can be described by a schematic connection of gates, by a set of Boolean equations, or by a truth table. Each type of description can be developed in Verilog. We will demonstrate each style, beginning with a simple example of a Verilog gate-level description to illustrate some aspects of the language.

The HDL description of the circuit of Fig. 3.37 is shown in HDL Example 3.1. The first line of text is a comment (optional) providing useful information to the reader. The second line begins with the keyword **module** and starts the declaration (description) of the module; the last line completes the declaration with the keyword **endmodule**. The keyword **module** is followed by a name and a list of ports. The name (*Simple_Circuit* in this example) is an identifier. Identifiers are names given to modules, variables (e.g., a signal), and other elements of the language so that they can be referenced in the design. In general, we choose meaningful names for modules. Identifiers are composed of alphanumeric characters and the underscore (_), and are case sensitive. Identifiers must start with an alphabetic character or an underscore, but they cannot start with a number.

HDL Example 3.1 (Combinational logic modeled with primitives)

```
// Verilog model of circuit of Figure 3.37. IEEE 1364–1995 Syntax

module Simple_Circuit (A, B, C, D, E);
 output      D, E;
 input       A, B, C;
 wire        w1;

 and         G1 (w1, A, B); // Optional gate instance name
 not         G2 (E, C);
 or          G3 (D, w1, E);
endmodule
```

The port list of a module is the interface between the module and its environment. In this example, the ports are the inputs and outputs of the circuit. The logic values of the inputs to a circuit are determined by the environment; the logic values of the outputs are determined within the circuit and result from the action of the inputs on the circuit. The port list is enclosed in parentheses, and commas are used to separate elements of the list. The statement

is terminated with a semicolon (;). In our examples, all keywords (which must be in lowercase) are printed in bold for clarity, but that is not a requirement of the language. Next, the keywords **input** and **output** specify which of the ports are inputs and which are outputs. Internal connections are declared as wires. The circuit in this example has one internal connection, at terminal *w1*, and is declared with the keyword **wire**. The structure of the circuit is specified by a list of (predefined) *primitive* gates, each identified by a descriptive keyword (**and**, **not**, **or**). The elements of the list are referred to as *instantiations* of a gate, each of which is referred to as a *gate instance*. Each *gate instantiation* consists of an optional name (such as *G1*, *G2*, etc.) followed by the gate output and inputs separated by commas and enclosed in parentheses. The output of a primitive gate is always listed first, followed by the inputs. For example, the OR gate of the schematic is represented by the **or** primitive, is named *G3*, and has output *D* and inputs *w1* and *E*. (*Note*: The output of a primitive must be listed first, but the inputs and outputs of a module may be listed in any order.) The module description ends with the keyword **endmodule**. Each statement must be terminated with a semicolon, but there is no semicolon after **endmodule**.

It is important to understand the distinction between the terms *declaration* and *instantiation*. A Verilog module is declared. Its declaration specifies the input–output behavior of the hardware that it represents. Predefined primitives are not declared, because their definition is specified by the language and is not subject to change by the user. Primitives are used (i.e., instantiated), just as gates are used to populate a printed circuit board. We'll see that once a module has been declared, it may be used (instantiated) within a design. Note that *Simple_Circuit* is not a computational model like those developed in an ordinary programming language: The sequential ordering of the statements in the model does not specify a sequence of computations. A Verilog model is a *descriptive* model. *Simple_Circuit* describes what primitives form a circuit and how they are connected. The input–output behavior of the circuit is implicitly specified by the description because the behavior of each logic gate is defined. Thus, an HDL-based model can be used to simulate the circuit that it represents.

Gate Delays

All physical circuits exhibit a propagation delay between the transition of an input and a resulting transition of an output. When an HDL model of a circuit is simulated, it is sometimes necessary to specify the amount of delay from the input to the output of its gates. In Verilog, the propagation delay of a gate is specified in terms of *time units* and is specified by the symbol #. The numbers associated with time delays in Verilog are dimensionless. The association of a time unit with physical time is made with the `**timescale** compiler directive. (Compiler directives start with the (`) back quote, or grave accent, symbol.) Such a directive is specified before the declaration of a module and applies to all numerical values of time in the code that follows. An example of a timescale directive is

```
timescale 1ns/100ps
```

The first number specifies the unit of measurement for time delays. The second number specifies the precision for which the delays are rounded off, in this case to 0.1 ns. If no timescale is specified, a simulator may display dimensionless values or default to a certain time unit, usually 1 ns ($= 10^{-9}$ sec). Our examples will use only the default time unit.

Table 3.6
Output of Gates after Delay

	Time Units (ns)	Input ABC	Output E w1 D
Initial	—	000	1 0 1
Change	—	111	1 0 1
	10	111	0 0 1
	20	111	0 0 1
	30	111	0 1 0
	40	111	0 1 0
	50	111	0 1 1

HDL Example 3.2 repeats the description of the simple circuit of Example 3.1, but with propagation delays specified for each gate. The **and**, **or**, and **not** gates have a time delay of 30, 20, and 10 ns, respectively. If the circuit is simulated and the inputs change from $A, B, C = 0$ to $A, B, C = 1$, the outputs change as shown in Table 3.6 (calculated by hand or generated by a simulator). The output of the inverter at E changes from 1 to 0 after a 10-ns delay. The output of the AND gate at $w1$ changes from 0 to 1 after a 30-ns delay. The output of the OR gate at D changes from 1 to 0 at $t = 30$ ns and then changes back to 1 at $t = 50$ ns. In both cases, the change in the output of the OR gate results from a change in its inputs 20 ns earlier. It is clear from this result that although output D eventually returns to a final value of 1 after the input changes, the gate delays produce a negative spike that lasts 20 ns before the final value is reached.

HDL Example 3.2 (Gate-level model with propagation delays)

```
// Verilog model of simple circuit with propagation delay

module Simple_Circuit_prop_delay (A, B, C, D, E);
 output D, E;
 input  A, B, C;
 wire   w1;

 and          #(30) G1 (w1, A, B);
 not          #(10) G2 (E, C);
 or           #(20) G3 (D, w1, E);
endmodule
```

In order to simulate a circuit with an HDL, it is necessary to apply inputs to the circuit so that the simulator will generate an output response. An HDL description that provides the stimulus to a design is called a *test bench*. The writing of test benches is explained in more detail at the end of Section 4.12. Here, we demonstrate the procedure with a simple example without dwelling on too many details. HDL Example 3.3 shows a test bench for simulating the circuit with delay. (Note the distinguishing name *Simple_Circuit_prop_delay*.) In its simplest

form, a test bench is a module containing a signal generator and an instantiation of the model that is to be verified. Note that the test bench (*t_Simple_Circuit_prop_delay*) has no input or output ports, because it does not interact with its environment. In general, we prefer to name the test bench with the prefix *t_* concatenated with the name of the module that is to be tested by the test bench, but that choice is left to the designer. Within the test bench, the inputs to the circuit are declared with keyword **reg** and the outputs are declared with the keyword **wire**. The module *Simple_Circuit_prop_delay* is instantiated with the instance name M1. Every instantiation of a module must include a unique instance name. Note that using a test bench is similar to testing actual hardware by attaching signal generators to the inputs of a circuit and attaching probes (wires) to the outputs of the circuit. (The interaction between the signal generators of the stimulus module and the instantiated circuit module is illustrated in Fig. 4.33.)

HDL Example 3.3

```
// Test bench for Simple_Circuit_prop_delay

module t_Simple_Circuit_prop_delay;
 wire      D, E;
 reg       A, B, C;

 Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required

 initial
  begin
   A = 1'b0; B = 1'b0; C = 1'b0;
   #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

 initial #200 $finish;
endmodule
```

Hardware signal generators are not used to verify an HDL model: The entire simulation exercise is done with software models executing on a digital computer. The waveforms of the input signals are abstractly modeled (generated) by Verilog statements specifying waveform values and transitions. The **initial** keyword is used with a set of statements that begin executing when the simulation is initialized; **initial** terminates execution when the last statement has finished executing. **initial** statements are commonly used to describe waveforms in a test bench. The set of statements to be executed is called a *block statement* and consists of several statements enclosed by the keywords **begin** and **end**. The action specified by the statements begins when the simulation is launched, and the statements are executed in sequence, from top to bottom, by a simulator in order to provide the input to the circuit. Initially, $A, B, C = 0$. (A, B, and C are each set to 1′b0, which signifies one binary digit with a value of 0.) After 100 ns, the inputs change to $A, B, C = 1$. After another 100 ns, the simulation terminates at time 200 ns. A second **initial** statement uses the **$finish** system task to specify termination of the simulation. If a statement is preceded by a delay value (e.g., #100), the simulator postpones executing the statement until the specified time delay has elapsed. The timing diagram of waveforms that result

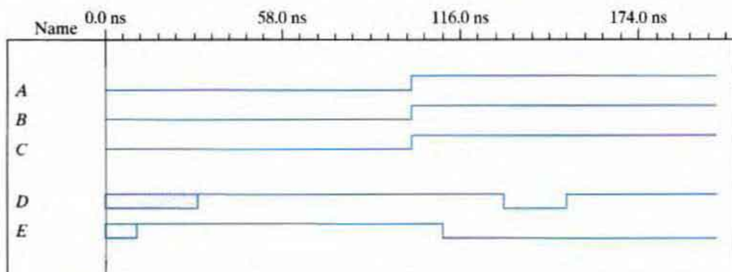


FIGURE 3.38
Simulation output of HDL Example 3.3

from the simulation is shown in Figure 3.38. The total simulation takes 200 ns. The inputs *A*, *B*, and *C* change from 0 to 1 after 100 ns. Output *E* is unknown for the first 10 ns (denoted by shading), and output *D* is unknown for the first 30 ns. Output *E* goes from 1 to 0 at 110 ns. Output *D* goes from 1 to 0 at 130 ns and back to 1 at 150 ns, just as we predicted in Table 3.6.

Boolean Expressions

Boolean equations describing combinational logic are specified in Verilog with a continuous assignment statement consisting of the keyword **assign** followed by a Boolean expression. To distinguish arithmetic operators from logical operators, Verilog uses the symbols (&), (/), and (~) for AND, OR, and NOT (complement), respectively. Thus, to describe the simple circuit of Fig. 3.37 with a Boolean expression, we use the statement

```
assign D = (A & B)|~C;
```

HDL Example 3.4 describes a circuit that is specified with the following two Boolean expressions:

$$E = A + BC + B'D$$

$$F = B'C + BC'D'$$

The equations specify how the logic values *E* and *F* are determined by the values of *A*, *B*, *C*, and *D*.

HDL Example 3.4 (Combinational logic modeled with Boolean equations)

```
// Verilog model: Circuit with Boolean expressions

module Circuit_Boolean_CA (E, F, A, B, C, D);
 output     E, F;
 input      A, B, C, D;

 assign E = A | (B & C) | (~B & D);
 assign F = (~B & C) | (B & ~C & ~D);
endmodule
```

The circuit has two outputs E and F and four inputs A, B, C, and D. The two **assign** statements describe the Boolean equations. The values of E and F during simulation are determined dynamically by the values of A, B, C, and D. The simulator detects when the test bench changes a value of one or more of the inputs. When this happens, the simulator updates the values of E and F. The continuous assignment mechanism is so named because the relationship between the assigned value and the variables is permanent. The mechanism acts just like combinational logic, has a gate-level equivalent circuit, and is referred to as *implicit combinational logic*.

We have shown that a digital circuit can be described with HDL statements, just as it can be drawn in a circuit diagram or specified with a Boolean expression. A third alternative is to describe combinational logic with a truth table.

User-Defined Primitives

The logic gates used in Verilog descriptions with keywords **and**, **or**, etc., are defined by the system and are referred to as *system primitives*. *(Caution: Other languages may use these words differently.)* The user can create additional primitives by defining them in tabular form. These types of circuits are referred to as *user-defined primitives* (UDPs). One way of specifying a digital circuit in tabular form is by means of a truth table. UDP descriptions do not use the keyword pair **module** … **endmodule**. Instead, they are declared with the keyword pair **primitive** … **endprimitive**. The best way to demonstrate a UDP declaration is by means of an example.

HDL Example 3.5 defines a UDP with a truth table. It proceeds according to the following general rules:

- It is declared with the keyword **primitive**, followed by a name and port list.
- There can be only one output, and it must be listed first in the port list and declared with keyword **output**.
- There can be any number of inputs. The order in which they are listed in the **input** declaration must conform to the order in which they are given values in the table that follows.
- The truth table is enclosed within the keywords **table** and **endtable**.
- The values of the inputs are listed in order, ending with a colon (:). The output is always the last entry in a row and is followed by a semicolon (;).
- The declaration of a UDP ends with the keyword **endprimitive**.

Note that the variables listed on top of the table are part of a comment and are shown only for clarity. The system recognizes the variables by the order in which they are listed in the input declaration. A user-defined primitive can be instantiated in the construction of other modules (digital circuits), just as the system primitives are used. For example, the declaration

```
Circuit_with_UDP_02467 (E, F, A, B, C, D);
```

will produce a circuit that implements the hardware shown in Figure 3.39.

Although Verilog HDL uses this kind of description for UDPs only, other HDLs and computer-aided design (CAD) systems use other procedures to specify digital circuits in tabular form. The tables can be processed by CAD software to derive an efficient gate structure of the design. None of Verilog's predefined primitives describes sequential logic. The

HDL Example 3.5

```
// Verilog model: User-defined Primitive

primitive UDP_02467 (D, A, B, C);
 output D;
 input   A, B, C;

// Truth table for D = f (A, B, C) = Σ (0, 2, 4, 6, 7);
 table
//     A      B      C      :      D     // Column header comment
       0      0      0      :      1;
       0      0      1      :      0;
       0      1      0      :      1;
       0      1      1      :      0;
       1      0      0      :      1;
       1      0      1      :      0;
       1      1      0      :      1;
       1      1      1      :      1;
 endtable
endprimitive

// Instantiate primitive

// Verilog model: Circuit instantiation of Circuit_UDP_02467

module Circuit_with_UDP_02467 (e, f, a, b, c, d);
 output      e, f;
 input       a, b, c, d;

 UDP_02467        (e, a, b, c);
 and              (f, e, d);        // Option gate instance name omitted
endmodule
```

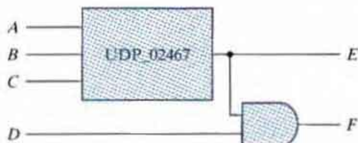


FIGURE 3.39
Schematic for *Circuit with_UDP_02467*

model of a sequential UDP requires that its output be declared as a **reg** data type, and that a column be added to the truth table to describe the next state. So the columns are organizes as inputs : state : next state.

In this section, we introduced the Verilog HDL and presented simple examples to illustrate alternatives for modeling combinational logic. A more detailed presentation of Verilog HDL can be found in the next chapter. The reader familiar with combinational circuits can go directly to Section 4.12 to continue with this subject.

PROBLEMS

Answers to problems marked with * appear at the end of the book.

3.1* Simplify the following Boolean functions, using three-variable maps:
(a) $F(x, y, z) = \Sigma(0, 2, 6, 7)$
(b) $F(x, y, z) = \Sigma(0, 2, 3, 4, 6)$
(c) $F(x, y, z) = \Sigma(0, 1, 2, 3, 7)$
(d) $F(x, y, z) = \Sigma(3, 5, 6, 7)$

3.2 Simplify the following Boolean functions, using three-variable maps:
(a)* $F(x, y, z) = \Sigma(0, 1, 5, 7)$
(b)* $F(x, y, z) = \Sigma(1, 2, 3, 6, 7)$
(c) $F(x, y, z) = \Sigma(0, 1, 6, 7)$
(d) $F(x, y, z) = \Sigma(0, 1, 3, 4, 5)$
(e) $F(x, y, z) = \Sigma(1, 3, 5, 7)$
(f) $F(x, y, z) = \Sigma(1, 4, 5, 6, 7)$

3.3* Simplify the following Boolean expressions, using three-variable maps:
(a)* $F(x,y,z) = xy + x'y'z' + x'yz'$
(b)* $F(x,y,z) = x'y' + yz + x'yz'$
(c)* $F(x, y, z) = x'y + yz' + y'z'$
(d) $F(x, y, z) = xyz + x'y'z + xy'z'$

3.4 Simplify the following Boolean functions, using *Karnaugh* maps:
(a)* $F(x, y, z) = \Sigma(2, 3, 6, 7)$
(b)* $F(A, B, C, D) = \Sigma(4, 6, 7, 15)$
(c)* $F(A, B, C, D) = \Sigma(3, 7, 11, 13, 14, 15)$
(d)* $F(w, x, y, z) = \Sigma(2, 3, 12, 13, 14, 15)$
(e) $F(w, x, y, z) = \Sigma(1, 4, 5, 6, 7, 13)$
(f) $F(w, x, y, z) = \Sigma(0, 1, 5, 8, 9)$

3.5 Simplify the following Boolean functions, using four-variable maps:
(a)* $F(w, x, y, z) = \Sigma(1, 4, 5, 6, 12, 14, 15)$
(b) $F(A, B, C, D) = \Sigma(1, 5, 9, 10, 11, 14, 15)$
(c) $F(w, x, y, z) = \Sigma(0, 1, 4, 5, 6, 7, 8, 9)$
(d)* $F(A, B, C, D) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$

3.6 Simplify the following Boolean expressions, using four-variable maps:
(a)* $A'B'C'D' + AC'D' + B'CD' + A'BCD + BC'D$
(b)* $x'z + w'xy' + w(x'y + xy')$
(c) $A'B'C'D' + A'CD' + AB'D' + ABCD + A'BD$
(d) $A'B'C'D' + AB'C + B'CD' + ABCD' + BC'D$

3.7 Simplify the following Boolean expressions, using four-variable maps:
(a)* $w'z + xz + x'y + wx'z$
(b) $C'D + A'B'C + ABC' + AB'C$
(c)* $AB'C + B'C'D' + BCD + ACD' + A'B'C + A'BC'D$
(d) $xyz + wy + wxy' + x'y$

3.8 Find the minterms of the following Boolean expressions by first plotting each function in a map:
(a)* $xy + yz + xy'z$
(b)* $C'D + ABC' + ABD' + A'B'D$
(c) $wyz + w'x' + wxz'$
(d) $A'B + A'CD + B'CD + BC'D'$

3.9 Find all the prime implicants for the following Boolean functions, and determine which are essential:

(a)* $F(w, x, y, z) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$
(b)* $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$
(c) $F(A, B, C, D) = \Sigma(1, 3, 4, 5, 10, 11, 12, 13, 14, 15)$
(d) $F(w, x, y, z) = \Sigma(1, 3, 6, 7, 8, 9, 12, 13, 14, 15)$
(e) $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 13, 15)$
(f) $F(w, x, y, z) = \Sigma(0, 2, 7, 8, 9, 10, 12, 13, 14, 15)$

3.10 Simplify the following Boolean functions by first finding the essential prime implicants:

(a) $F(w, x, y, z) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$
(b) $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$
(c)* $F(A, B, C, D) = \Sigma(1, 3, 4, 5, 10, 11, 12, 13, 14, 15)$
(d) $F(w, x, y, z) = \Sigma(1, 3, 6, 7, 8, 9, 12, 13, 14, 15)$
(e) $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 13, 15)$
(f) $F(w, x, y, z) = \Sigma(0, 2, 7, 8, 9, 10, 12, 13, 14, 15)$

3.11 Simplify the following Boolean functions, using five-variable maps:

(a)* $F(A, B, C, D, E) = \Sigma(0, 1, 4, 5, 16, 17, 21, 25, 29)$
(b) $F(A, B, C, D) = A'B'CE' + B'C'D'E' + A'B'D' + B'CD' + A'CD + A'BD$

3.12 Simplify the following Boolean functions to product-of-sums form:

(a) $F(w, x, y, z) = \Sigma(0, 1, 2, 5, 8, 10, 13)$
(b)* $F(A, B, C, D) = \Pi(1, 3, 5, 7, 13, 15)$
(c) $F(A, B, C, D) = \Pi(1, 3, 6, 9, 11, 12, 14)$

3.13 Simplify the following expressions to (1) sum-of-products and (2) products-of-sums:

(a)* $x'z' + y'z' + yz' + xy$
(b) $ACD' + C'D + AB' + ABCD$
(c) $(A + C' + D')(A' + B' + D')(A' + B + D')(A' + B + C')$
(d) $ABC' + AB'D + BCD$

3.14 Give three possible ways to express the following Boolean function with eight or fewer literals:

$$F = B'C'D' + AB'CD' + BC'D + A'BCD$$

3.15 Simplify the following Boolean function F, together with the don't-care conditions d, and then express the simplified function in sum-of-minterms form:

(a) $F(x, y, z) = \Sigma(2, 3, 4, 6, 7)$
$d(x, y, z) = \Sigma(0, 1, 5)$

(b)* $F(A, B, C, D) = \Sigma(0, 6, 8, 13, 14)$
$d(A, B, C, D) = \Sigma(2, 4, 10)$

(c) $F(A, B, C, D) = \Sigma(4, 5, 7, 12, 13, 14)$
$d(A, B, C, D) = \Sigma(1, 9, 11, 15)$

(d) $F(A, B, C, D) = \Sigma(1, 3, 8, 10, 15)$
$d(A, B, C, D) = \Sigma(0, 2, 9)$

3.16 Simplify the following functions, and implement them with two-level NAND gate circuits:

(a) $F(A, B, C, D) = A'B'C + AC' + ACD + ACD' + A'B'D'$
(b) $F(A, B, C, D) = AB + A'BC + A'B'C'D$
(c) $F(A, B, C) = (A' + B' + C')(A' + B')(A' + C')$
(d) $F(A, B, C, D) = A'B + A + C' + D'$

3.17* Draw a NAND logic diagram that implements the complement of the following function:

$$F(A, B, C, D) = \Sigma(0, 1, 2, 3, 4, 8, 9, 12)$$

3.18 Draw a logic diagram using only two-input NOR gates to implement the following function:

$$F(A, B, C, D) = (A \oplus B)'(C \oplus D)$$

3.19 Simplify the following functions, and implement them with two-level NOR gate circuits:
(a)* $F = wx' + y'z' + w'yz'$
(b) $F(w, x, y, z) = \Sigma(1, 2, 13, 14)$
(c) $F(x, y, z) = [(x + y)(x' + z)]'$

3.20 Draw the multi-level NOR and multi-level NAND circuits for the following expression:

$$(AB' + CD')E + BC(A + B)$$

3.21 Draw the multi-level NAND circuit for the following expression:

$$w(x + y + z) + xyz$$

3.22 Convert the logic diagram of the circuit shown in Fig. 4.4 into a multiple-level NAND circuit.

3.23 Implement the following Boolean function F, together with the don't-care conditions d, using no more than two NOR gates:

$$F(A, B, C, D) = \Sigma(2, 4, 6, 10, 12)$$
$$d(A, B, C, D) = \Sigma(0, 8, 9, 13)$$

Assume that both the normal and complement inputs are available.

3.24 Implement the following Boolean function F, using the two-level forms of logic (a) NAND-AND, (b) AND-NOR, (c) OR-NAND, and (d) NOR-OR:

$$F(A, B, C, D) = \Sigma(0, 4, 8, 9, 10, 11, 12, 14)$$

3.25 List the eight degenerate two-level forms and show that they reduce to a single operation. Explain how the degenerate two-level forms can be used to extend the number of inputs to a gate.

3.26 With the use of maps, find the simplest sum-of-products form of the function $F = fg$, where

$$f = abc' + c'd + a'cd' + b'cd'$$

and

$$g = (a + b + c' + d')(b' + c' + d)(a' + c + d')$$

3.27 Show that the dual of the exclusive-OR is also its complement.

3.28 Derive the circuits for a three-bit parity generator and four-bit parity checker using an odd parity bit.

3.29 Implement the following four Boolean expressions with three half adders

$$D = A \oplus B \oplus C$$
$$E = A'BC + AB'C$$
$$F = ABC' + (A' + B')C$$
$$G = ABC$$

3.30* Implement the following Boolean expression with exclusive-OR and AND gates:

$$F = AB'CD' + A'BCD' + AB'C'D + A'BC'D$$

3.31 Write a Verilog gate-level description of the circuit shown in
(a) Fig. 3.22(a) (b) Fig. 3.22(b) (c) Fig. 3.23(a)
(d) Fig. 3.23(b) (e) Fig. 3.26 (f) Fig. 3.27

3.32 Using continuous assignment statements, write a Verilog description of the circuit shown in
(a) Fig. 3.22(a) (b) Fig. 3.22(b) (c) Fig. 3.23(a)
(d) Fig. 3.23(b) (e) Fig. 3.26 (f) Fig. 3.27

3.33 The exclusive-OR circuit of Fig. 3.32(a) has gates with a delay of 4 ns for an inverter, a 8 ns delay for an AND gate, and a 10 ns delay for an OR gate. The input of the circuit goes from $xy = 00$ to $xy = 01$.
(a) Determine the signals at the output of each gate from $t = 0$ to $t = 50$ ns.
(b) Write a Verilog gate-level description of the circuit, including the delays.
(c) Write a stimulus module (i.e., a test bench similar to HDL Example 3.3), and simulate the circuit to verify the answer in part (a).

3.34 Using continuous assignments, write a Verilog description of the circuit specified by the following Boolean functions:

$$Out_1 = (C + B)(A' + D)B'$$
$$Out_2 = (CB' + ABC + C'B)(A + D')$$
$$Out_3 = C(AD + B) + BA'$$

Write a test bench and simulate the circuit's behavior.

3.35* Find the syntax errors in the following declarations (note that names for primitive gates are optional):

```
module Exmpl-3(A, B, C, D, F)          // Line 1
  inputs     A, B, C, Output D, F,     // Line 2
  output     B                         // Line 3
  and        g1(A, B, D);              // Line 4
  not        (D, A, C),                // Line 5
  OR         (F, B; C);                // Line 6
endofmodule;                           // Line 7
```

3.36 Draw the logic diagram of the digital circuit specified by the following Verilog description:

(a)
```
module Circuit_A (A, B, C, D, F);
  input   A, B, C, D;
  output        F;
  wire    w, x, y, z, a, d;
  and     (x, B, C, d);
  and     (y, a ,C);
  and     (w, z ,B);
  or      (z, y, A);
  or      (F, x, w);
  not     (a, A);
  not     (d, D);
endmodule
```

(b)
```
module Circuit_B (A_gtB, A_ltB, A_eqB, A0, A1, B0, B1);
  output A_gtB, A_ltB, A_eqB;
  input  A0, A1, B0, B1;
  nor    (A_gtB, A_ltB, A_eqB);
  or     (A_ltB, w1, w2, w3);
  and    (A_eqB, w4, w5);
  and    (w1, w6, B1);
  and    (w2, w6, w7, B0);
  and    (w3, w7, B0, B1);
  not    (w6, A1);
  not    (w7, A0);
  xnor   (w4, A1, B1);
  xnor   (w5, A0, B0);
endmodule
```

(c)
```
module Circuit_C (output y1, input a, b, output y2);
  assign y1 = a & b;
  or (y2, a, b);
endmodule
```

3.37 A majority logic function is a Boolean function that is equal to 1 if the majority of the variables are equal to 1, equal to 0 otherwise. Write a Verilog user-defined primitive for a four-bit majority function.

3.38 Simulate the behavior of *Circuit_with_UDP_02467*, using the stimulus waveforms shown in Fig. P3.38.

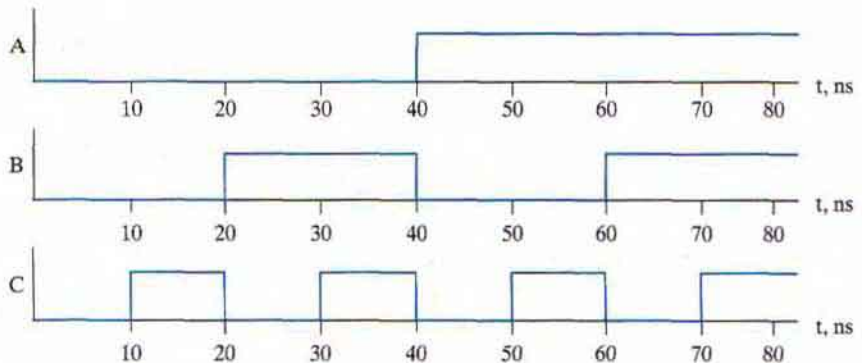


FIGURE P3.38
Stimulus waveforms for Problem 3.38

REFERENCES

1. BHASKER, J. 1997. *A Verilog HDL Primer.* Allentown, PA: Star Galaxy Press.
2. CILETTI, M.D. 1999. *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL.* Upper Saddle River, NJ: Prentice Hall.
3. HILL, F. J., and G. R. PETERSON. 1981. *Introduction to Switching Theory and Logical Design*, 3d ed. New York: John Wiley.
4. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995). 1995. New York: The Institute of Electrical and Electronics Engineers.
5. KARNAUGH, M. A Map Method for Synthesis of Combinational Logic Circuits. *Transactions of AIEE, Communication and Electronics.* 72, part I (Nov. 1953): 593–99.
6. KOHAVI, Z. 1978. *Switching and Automata Theory*, 2d ed. New York: McGraw-Hill.
7. MANO, M. M., and C. R. KIME. 2004. *Logic and Computer Design Fundamentals*, 3rd ed. Upper Saddle River, NJ: Prentice Hall.
8. MCCLUSKEY, E. J. 1986. *Logic Design Principles.* Englewood Cliffs, NJ: Prentice-Hall.
9. PALNITKAR, S. 1996. *Verilog HDL: A Guide to Digital Design and Synthesis.* Mountain View, CA: SunSoft Press (a Prentice Hall title).