



MODULE 1 NOTES



MODULE 1

Introduction to Digital Design: Binary Logic, Basic Theorems And Properties Of Boolean Algebra, Boolean Functions, Digital Logic Gates, Introduction, The Map Method, Four-Variable Map, Don't-Care Conditions, NAND and NOR Implementation, Other Hardware Description Language – Verilog Model of a simple circuit

BINARY LOGIC

Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning. The two values the variables assume may be called by different names (true and false, yes and no, etc.), but for our purpose, it is convenient to think in terms of bits and assign the values 1 and 0. The binary logic introduced in this section is equivalent to an algebra called Boolean algebra. The formal presentation of Boolean algebra is covered in more detail in Chapter 2. The purpose of this section is to introduce Boolean algebra in a heuristic manner and relate it to digital logic circuits and binary signals.

Definition of Binary Logic

Binary logic consists of binary variables and a set of logical operations. The variables are designated by letters of the alphabet, such as A, B, C, x, y, z, etc., with each variable having two and only two distinct possible values: 1 and 0. There are three basic logical operations: AND, OR, and NOT. Each operation produces a binary result, denoted by z.

1. AND: This operation is represented by a dot or by the absence of an operator. For example, $x \cdot y = z$ or $xy = z$ is read "x AND y is equal to z." The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ and $y = 1$; otherwise $z = 0$. (Remember that x, y, and z are binary variables and can be equal either to 1 or 0, and nothing else.) The result of the operation $x \cdot y$ is z.

2. OR: This operation is represented by a plus sign. For example, $x + y = z$ is read "x OR y is equal to z," meaning that $z = 1$ if $x = 1$ or if $y = 1$ or if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.

Table 1.8
Truth Tables of Logical Operations

AND			OR			NOT	
x	y	$x \cdot y$	x	y	$x + y$	x	x'
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

3. NOT: This operation is represented by a prime (sometimes by an overbar). For example, $x' = z$ (or $x = Z$) is read "not x is equal to z," meaning that z is what x is not. In other words, if $x = 1$, then $z = 0$, but if $x = 0$, then $z = 1$. The NOT operation is also referred to as the complement operation, since it

Changes a 1 to 0 and a 0 to 1, that is, the result of complementing 1 is 0, and vice versa.

For each combination of the values of x and y, there is a value of z specified by the definition of the logical operation. Definitions of logical operations may be listed in a compact form called truth tables. A truth table is a table of all possible combinations of the variables, showing the relation between the values that the variables may take and the result of the operation. The truth tables for the operations AND and OR with variables x and y are obtained by listing all possible values that the variables may have when combined in pairs. For each combination, the result of the operation is then listed in a separate row. The truth tables for AND, OR, and NOT are given in Table 1.8. These tables clearly demonstrate the definition of the operations.

Logic Gates:

Logic gates are electronic circuits that operate on one or more physical input signals to produce an output signal. Electrical signals such as voltages or currents exist as analog signals having values over a given continuous range, say, 0-3 V, but in a digital system these voltages are interpreted to be either of two recognizable values, 0 or 1. Voltage-operated logic circuits respond to two separate voltage levels that represent a binary variable equal to logic 1 or logic 0. For example, a particular digital system may define logic 0 as a signal equal to 0 V and logic 1 as a signal equal to 3 V. In practice, each voltage level has an acceptable range, as shown in Fig. 1.3. The input terminals of digital circuits accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within the specified range. The intermediate region between the allowed regions is crossed only during a state transition. Any desired information for computing or control can be operated on by passing binary signals through various combinations of logic gates, with each signal representing a particular binary variable. When the physical signal is in a particular range it is interpreted to be either a 0 or a 1.

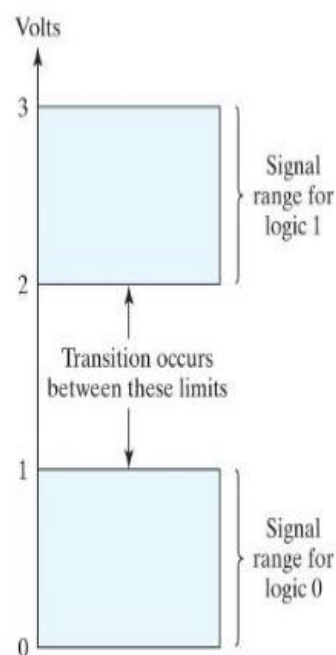


Figure: Signal levels for binary logic values

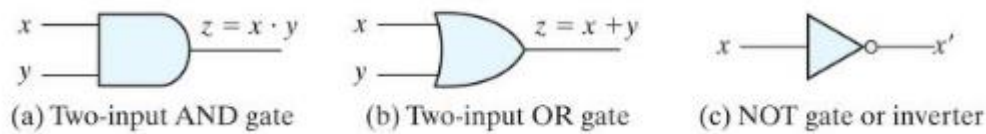


Figure : (a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter.

The graphic symbols used to designate the three types of gates are shown in Fig. 1.4. The gates are blocks of hardware that produce the equivalent of logic-1 or logic-0 output signals if input logic requirements are satisfied. The input signals x and y in the AND and OR gates may exist in one of four possible states: 00, 10, 11, or 01. These input signals are shown in Fig. 1.5 together with the corresponding output signal for each gate. The timing diagrams illustrate the idealized response of each gate to the four input signal combinations. The horizontal axis of the timing diagram represents the time, and the vertical axis shows the signal as it changes between the two possible voltage levels. In reality, the transitions between logic values occur quickly, but not instantaneously. The low level represents logic 0 and the high level logic 1. The AND gate responds with a logic 1 output signal when both input signals are logic 1. The OR gate responds with a logic 1 output signal if any input signal is logic 1. The NOT gate is commonly referred to as an inverter. The reason for this name is apparent from the signal response in the timing diagram, which shows that the output signal inverts the logic sense of the input signal.

AND and OR gates may have more than two inputs. An AND gate with three inputs and an OR gate with four inputs are shown in Fig. 1.6. The three-input AND gate responds with logic 1 output if all three inputs are logic 1. The output produces logic 0 if any input is logic 0. The four-input OR gate responds with logic 1 if any input is logic 1; its output becomes logic 0 only when all inputs are logic 0.

Problems 53

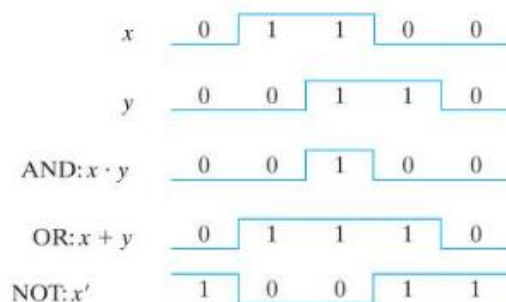
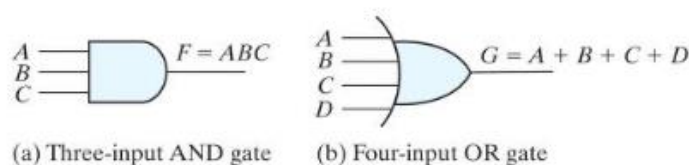


FIGURE 1.5
Input-output signals for gates



Basic Theorems And Properties Of Boolean Algebra:

Duality:

The important property of Boolean algebra is called the duality principle and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged. In a two-valued Boolean algebra, the identity elements and the elements of the set B are the same: 1 and 0. The duality principle has many applications. If the dual of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Basic Theorems:

Table 2.1
Postulates and Theorems of Boolean Algebra

Postulate 2	(a)	$x + 0 = x$	(b)	$x \cdot 1 = x$
Postulate 5	(a)	$x + x' = 1$	(b)	$x \cdot x' = 0$
Theorem 1	(a)	$x + x = x$	(b)	$x \cdot x = x$
Theorem 2	(a)	$x + 1 = 1$	(b)	$x \cdot 0 = 0$
Theorem 3, involution		$(x')' = x$		
Postulate 3, commutative	(a)	$x + y = y + x$	(b)	$xy = yx$
Theorem 4, associative	(a)	$x + (y + z) = (x + y) + z$	(b)	$x(yz) = (xy)z$
Postulate 4, distributive	(a)	$x(y + z) = xy + xz$	(b)	$x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a)	$(x + y)' = x'y'$	(b)	$(xy)' = x' + y'$
Theorem 6, absorption	(a)	$x + xy = x$	(b)	$x(x + y) = x$

Table 2.1 lists six theorems of Boolean algebra and four of its postulates. The notation is simplified by omitting the binary operator whenever doing so does not lead to confusion. The theorems and postulates listed are the most basic relationships in Boolean algebra. The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates. Proofs of the theorems with one variable are presented next. At the right is listed the number of the postulate which justifies that particular step of the proof.

THEOREM 1(a): $x + x = x$.

Statement	Justification
$x + x = (x + x) \cdot 1$	postulate 2(b)
$= (x + x)(x + x')$	5(a)
$= x + xx'$	4(b)
$= x + 0$	5(b)
$= x$	2(a)

THEOREM 1(b): $x \cdot x = x$.

Statement	Justification
$x \cdot x = xx + 0$	postulate 2(a)
$= xx + xx'$	5(b)
$= x(x + x')$	4(a)
$= x \cdot 1$	5(a)
$= x$	2(b)

Note that theorem 1(b) is the dual of theorem 1(a) and that each step of the proof in part (b) is the dual of its counterpart in part (a). Any dual theorem can be similarly derived from the proof of its corresponding theorem.

THEOREM 2(a): $x + 1 = 1$.

Statement	Justification
$x + 1 = 1 \cdot (x + 1)$	postulate 2(b)
$= (x + x')(x + 1)$	5(a)
$= x + x' \cdot 1$	4(b)
$= x + x'$	2(b)
$= 1$	5(a)

THEOREM 2(b): $x \cdot 0 = 0$ by duality.

THEOREM 3: $(x')' = x$. From postulate 5, we have $x + x' = 1$ and $x \cdot x' = 0$, which together define the complement of x . The complement of x' is x and is also $(x')'$.

Therefore, since the complement is unique, we have $(x')' = x$. The theorems involving two or three variables may be proven algebraically from the postulates and the theorems that have already been proven. Take, for example, the absorption theorem:

THEOREM 6(a): $x + xy = x$.

Statement	Justification
$x + xy = x \cdot 1 + xy$	postulate 2(b)
$= x(1 + y)$	4(a)
$= x(y + 1)$	3(a)
$= x \cdot 1$	2(a)
$= x$	2(b)

THEOREM 6(b): $x(x + y) = x$ by duality.

The theorems of Boolean algebra can be proven by means of truth tables. In truth tables, both sides of the relation are checked to see whether they yield identical results for all possible combinations of the variables involved. The following truth table verifies the first absorption theorem:

x	y	xy	$x + xy$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

x	y	$x + y$	$(x + y)'$	x'	y'	$x'y'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Operator Precedence

The operator precedence for evaluating Boolean expressions is (1) parentheses, (2) NOT, (3) AND, and (4) OR. In other words, expressions inside parentheses must be evaluated before all other operations. The next operation that holds precedence is the complement, and then follows the AND and, finally, the OR. As an example, consider the truth table for one of De Morgan's theorems. The left side of the expression is $(x + y)'$. Therefore, the expression inside

the parentheses is evaluated first and the result then complemented. The right side of the expression is $x'y'$, so the complement of x and the complement of y are both evaluated first and the result is then ANDed.

Note that in ordinary arithmetic, the same precedence holds (except for the complement) When multiplication and addition are replaced by AND and OR, respectively.

BOOLEAN FUNCTIONS:

Boolean algebra is an algebra that deals with binary variables and logic operations. A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols. For a given value of the binary variables, the function can be equal to either 1 or 0. As an example, consider the Boolean function,

$$F_1 = x + y'z$$

The function F_1 is equal to 1 if x is equal to 1 or if both y' and z are equal to 1. F_1 is equal to 0 otherwise. The complement operation dictates that when $y' = 1$, $y = 0$. Therefore, $F_1 = 1$ if $x = 1$ or if $y = 0$ and $z = 1$. A Boolean function expresses the logical relationship between binary variables and is evaluated by determining the binary value of the expression for all possible values of the variables.

A Boolean function can be represented in a truth table. The number of rows in the truth table is 2^n , where n is the number of variables in the function. The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through $2^n - 1$. Table 2.2 shows the truth table for the function F_1 . There are eight possible binary combinations for assigning bits to the three variables x , y , and z . The column labeled F_1 contains either 0 or 1 for each of these combinations. The table shows that the function is equal to 1 when $x = 1$ or when $yz = 01$ and 1s equal to 0 otherwise.

Table 2.2
Truth Tables for F_1 and F_2

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

combinations for the truth table are obtained from the binary numbers by counting from 0 through $2^n - 1$. Table 2.2 shows the truth table for the function F_1 . There are eight possible binary combinations for assigning bits to the three variables x , y , and z . The column labeled F_1 contains either 0 or 1 for each of these combinations. The table shows that the function is equal to 1 when $x = 1$ or when $yz = 01$ and 1s equal to 0 otherwise.

The logic-circuit diagram (also called a schematic) for F , is shown in Fig. 2.1. There is an inverter for input y to generate its complement. There is an AND gate for the term $y'z$ and an OR gate that combines x with $y'z$. In logic-circuit diagrams, the variables of the function are taken as the inputs of the circuit and the binary variable F , is taken as the output of the circuit. The schematic expresses the relationship between the output of the circuit and its inputs. Rather than listing each combination of inputs and outputs, it indicates how to compute the logic value of each output from the logic values of the inputs.

There is only one way that a Boolean function can be represented in a truth table. However, when the function is in algebraic form, it can be expressed in a variety of ways, all of which have equivalent logic. The particular expression used to represent the function will dictate the interconnection of gates in the logic-circuit diagram. Conversely, the interconnection of gates will dictate the logic expression. Here is a Key fact that motivates our use of Boolean algebra: By manipulating a Boolean expression according to the rules of Boolean algebra, it is sometimes possible to obtain a simpler expression for the same function and thus reduce the number of gates in the circuit and the number of inputs to the gate.

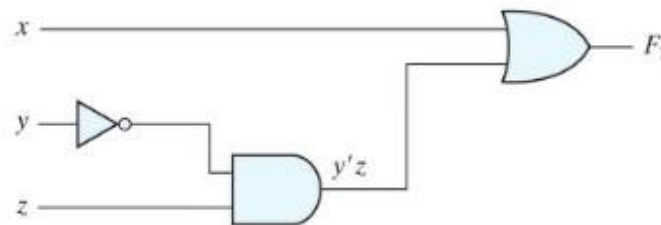


FIGURE 2.1

Logic diagram for the Boolean function $F_1 = x + y'z$

Designers are motivated to reduce the complexity and number of gates because their effort can significantly reduce the cost of a circuit. Consider, for example, the following Boolean function:

$$F_2 = x'y'z + x'yz + xy'$$

A schematic of an implementation of this function with logic gates is shown in Fig. 2.2(a). Input variables x and y are complemented with inverters to obtain x' and y' . The three terms in the expression are implemented with three AND gates. The OR gate forms the logical OR of the three terms. The truth table for F , is listed in Table 2.2. The function is equal to 1 when $xyz = 001$ or 011 or when $xy = 10$ (irrespective of the value of z) and is equal to 0 otherwise. This set of conditions produces four 1's and four 0's for F_5 .

Now consider the possible simplification of the function by applying some of the identities of Boolean algebra:

$$F_2 = x'y'z + x'yz + xy' = x'z(y' + y) + xy' = x'z + xy'$$

The function is reduced to only two terms and can be implemented with gates as shown in Fig. 2.2(b). It is obvious that the circuit in (b) is simpler than the one in (a), yet both implement

the same function. By means of a truth table, it is possible to verify that the two expressions are equivalent. The simplified expression is equal to 1 when $xz = 01$ or when $xy = 10$. This produces the same four 1's in the truth table. Since both expressions,

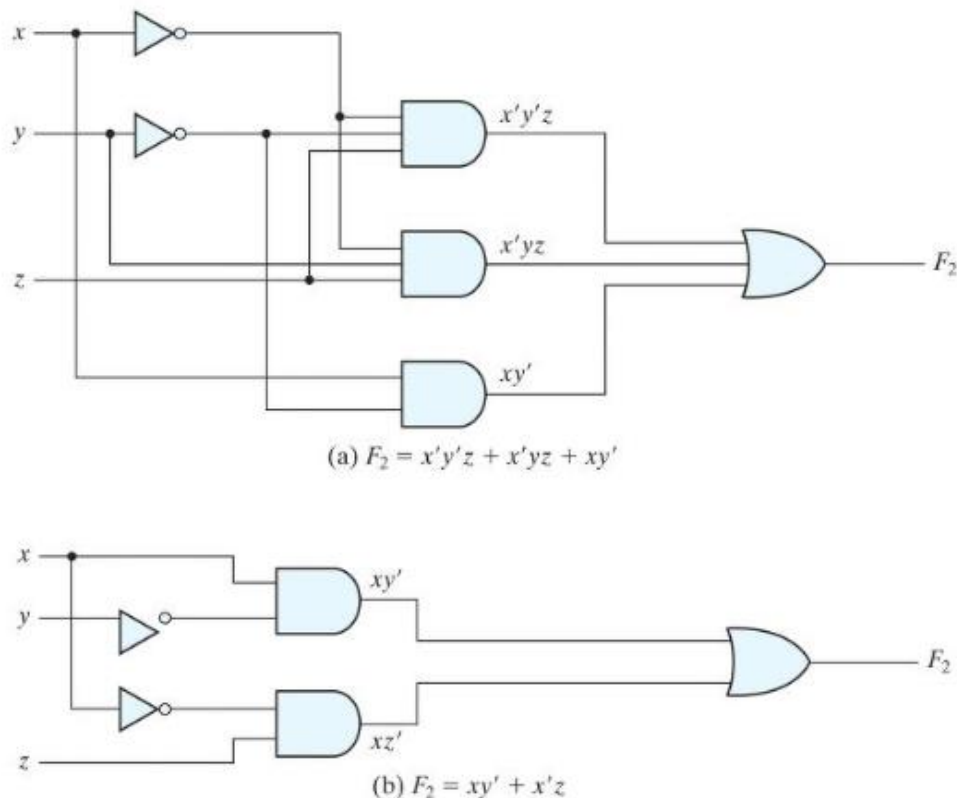


FIGURE 2.2
Implementation of Boolean function F_2 with gates

produce the same truth table, they are equivalent. Therefore, the two circuits have the same outputs for all possible binary combinations of inputs of the three variables. Each circuit implements the same identical function, but the one with fewer gates and fewer inputs to gates is preferable because it requires fewer wires and components. In general, there are many equivalent representations of a logic function. Finding the most economic representation of the logic is an important design task.

Algebraic Manipulation

When a Boolean expression is implemented with logic gates, each term requires a gate and each variable within the term designates an input to the gate. We define a literal to be a single variable within a term, in complemented or uncomplemented form. The function of Fig. 2.2(a) has three terms and eight literals, and the one in Fig. 2.2(b) has two terms and four literals. By reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit. The manipulation of Boolean algebra consists mostly of reducing an expression for the purpose of obtaining a simpler circuit. Functions of up to five variables can be simplified by the map method described in the next chapter. For complex Boolean functions and many different outputs, designers of digital circuits use

computer minimization programs that are capable of producing optimal circuits with millions of logic gates. The concepts introduced in this chapter provide the framework for those tools.

Simplify the following Boolean expressions to a minimum number of literals.

$$1. x(x' + y) = xx' + xy = 0 + xy = xy.$$

$$2. x + x'y = (x + x')(x + y) = 1(x + y) = x + y.$$

$$3. (x + y)(x + y') = x + xy + xy' + yy' = x(1 + y + y') = x.$$

$$\begin{aligned} 4. xy + x'z + yz &= xy + x'z + yz(x + x') \\ &= xy + x'z + xyz + x'yz \\ &= xy(1 + z) + x'z(1 + y) \\ &= xy + x'z. \end{aligned}$$

$$5. (x + y)(x' + z)(y + z) = (x + y)(x' + z), \text{ by duality from function 4.}$$

Expressions 1 and 2 are the dual of each other and use dual expressions in corresponding steps. An easier way to simplify function 3 is by means of postulate 4(b) from Table 2.1: $(x + y)(x + y') = x + yy' = x$. The fourth expression illustrates the fact that an increase in the number of literals sometimes leads to a simpler final expression. Expression 5 is not minimized directly, but can be derived from the dual of the steps used to derive expression 4. Expressions 4 and 5 are together known as the consensus theorem.

Complement of a Function

The complement of a function F is F' and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of F . The complement of a function may be derived algebraically through DeMorgan's theorems, listed in Table 2.1 for two variables. DeMorgan's theorems can be extended to three or more variables. The three-variable form of the first DeMorgan's theorem is derived as follows, from postulates and theorems listed in Table 2.1:

$$\begin{aligned} (A + B + C)' &= (A + x)' && \text{let } B + C = x \\ &= A'x' && \text{by theorem 5(a) (DeMorgan)} \\ &= A'(B + C)' && \text{substitute } B + C = x \\ &= A'(B'C') && \text{by theorem 5(a) (DeMorgan)} \\ &= A'B'C' && \text{by theorem 4(b) (associative)} \end{aligned}$$

DeMorgan's theorems for any number of variables resemble the two-variable case in form and can be derived by successive substitutions similar to the method used in the preceding derivation. These theorems can be generalized as follows:

$$\begin{aligned} (A + B + C + D + \cdots + F)' &= A'B'C'D' \cdots F' \\ (ABCD \cdots F)' &= A' + B' + C' + D' + \cdots + F' \end{aligned}$$

The generalized form of DeMorgan's theorems states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

EXAMPLE 2.2

Find the complement of the functions $F_1 = x'yz' + x'y'z$ and $F_2 = x(y'z' + yz)$. By applying DeMorgan's theorems as many times as necessary, the complements are obtained as follows:

$$F_1' = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$$

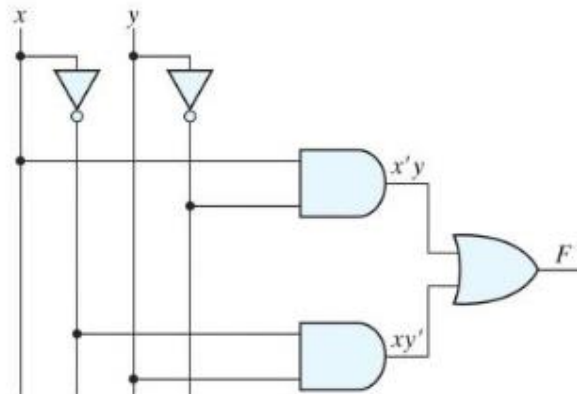
$$\begin{aligned} F_2' &= [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')'(yz)' \\ &= x' + (y + z)(y' + z') \\ &= x' + yz' + y'z \end{aligned}$$

A simpler procedure for deriving the complement of a function is to take the dual of the function and complement each literal. This method follows from the generalized forms of DeMorgan's theorems. Remember that the dual of a function is obtained from the interchange of AND and OR operators and 1's and 0's.

Practice Exercise 2.3

Draw a logic diagram for the Boolean function $F = x'y + xy'$.

Answer:



Practice Exercise 2.4

What Boolean expression is implemented by the following logic diagram?

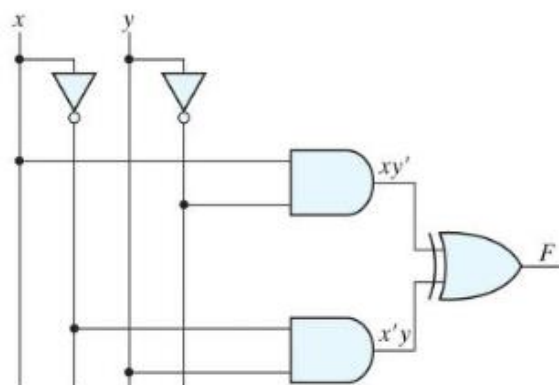


FIGURE PE2.4

Answer:

$$F = (x'y + xy')' = (x'y)'(xy')' = (x + y')(x' + y) = xx' + xy + y'x' + yy' = xy + x'y'$$

Practice Exercise 2.5

What truth table is implemented by the logic diagram in Fig. PE 2.4?

Answer:

<i>x</i>	<i>y</i>	<i>F</i>
0	0	1
0	1	0
1	0	0
1	1	1

Answer:

$$F = (x'y + xy')' = (x'y)'(xy')' = (x + y')(x' + y) = xx' + xy + y'x' + yy' = xy + x'y'$$

Practice Exercise 2.5

What truth table is implemented by the logic diagram in Fig. PE 2.4?

Answer:

<i>x</i>	<i>y</i>	<i>F</i>
0	0	1
0	1	0
1	0	0
1	1	1

Practice Exercise 2.6

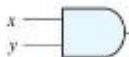







Find the complement of the Boolean function $F = A'BC' + A'B'C$.

Answer: $F' = A + BC + B'C'$

DIGITAL LOGIC GATES

Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these type of gates. Still, the possibility of constructing gates for the other logic operations is of practical interest. Factors to be weighed in considering the construction of other types of logic gates are (1) the feasibility and economy of producing the gate with physical components, (2) the possibility of extending the gate to more than two inputs, (3) the basic properties of the binary operator, such as commutativity and associativity, and (4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates. The graphic symbols and truth tables of the eight gates are shown in Fig. 2.5. Each gate has one or two binary input variables, designated by *x* and *y*, and one binary output variable, designated by *F*. The AND, OR, and inverter circuits were defined in Fig. 1.6. The inverter circuit inverts the logic sense of a binary variable, producing the NOT,

or complement, function. The small circle in the output of the graphic symbol of an inverter (referred to as a bubble) designates the logic complement. The triangle symbol by itself designates a buffer circuit. A buffer produces the transfer function, but does not produce a logic operation, since the binary value of the output is equal to the binary value of the input. This circuit is used for power amplification of the signal and is equivalent to two inverters connected in cascade.

Name	Graphic symbol	Algebraic function	Truth table															
AND		$F = x \cdot y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = x + y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	1
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$F = x'$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	x	F	0	1	1	0									
x	F																	
0	1																	
1	0																	
Buffer		$F = x$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	x	F	0	0	1	1									
x	F																	
0	0																	
1	1																	
NAND		$F = (xy)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (x + y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

The gates shown in Fig. 2.5—except for the inverter and buffer—can be extended to have more than two inputs. A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative. The AND and OR operations, defined in Boolean algebra, possess these two properties. For the OR function, we have

$$x+y = y+x \text{ (commutative)}$$

$$(x+y)+z = x+(y+z) = x+y+z \text{ (associative),}$$

which indicates that the gate inputs can be interchanged and that the OR function can be extended to three or more variables.

The NAND and NOR functions are commutative, and their gates can be extended to have more than two inputs, provided that the definition of the operation is modified slightly. The difficulty is that the NAND and NOR operators are not associative (i.e.,

$$(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)),$$

as shown in Fig. 2.6 and the following equations:

$$(x \downarrow y) \downarrow z = [(x + y)' + z]' = (x + y)z' = xz' + yz'$$

$$x \downarrow (y \downarrow z) = [x + (y + z)']' = x'(y + z) = x'y + x'z$$

To overcome this difficulty, we define the multiple NOR (or NAND) gate as a complemented OR (or AND) gate. Thus, by definition, we have

$$x \downarrow y \downarrow z = (x + y + z)'$$

$$x \uparrow y \uparrow z = (xyz)'$$

The graphic symbols for the three-input gates are shown in Fig. 2.7 In writing cascaded NOR and NAND operations, one must use the correct parentheses to signify the proper,

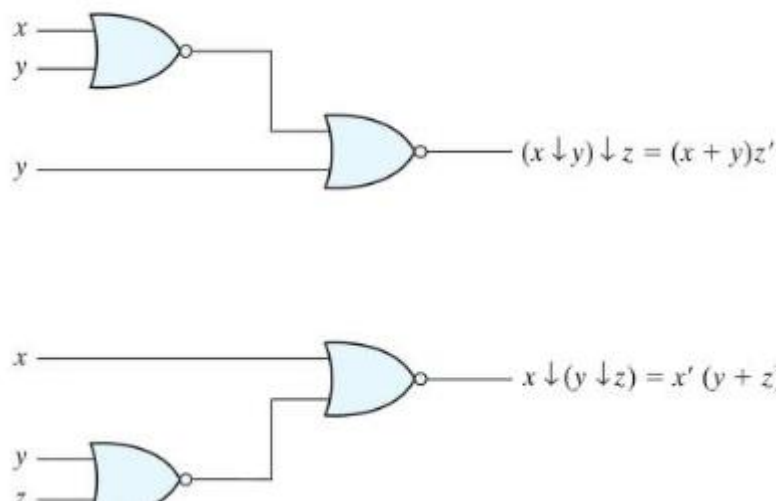


FIGURE 2.6 Demonstrating the nonassociativity of the NOR operator

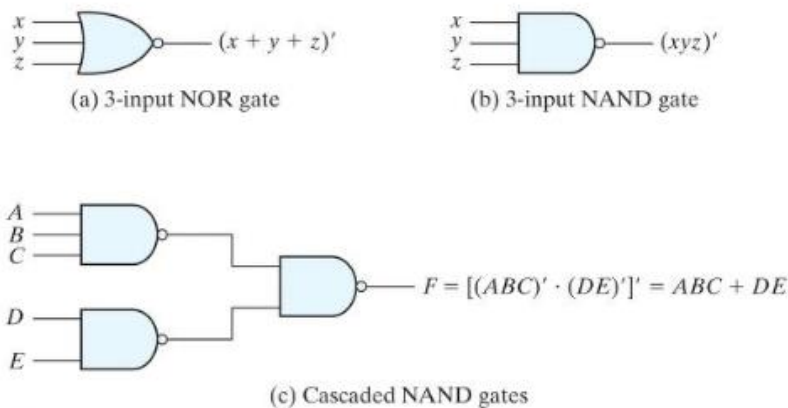


FIGURE 2.7
Multiple-input and cascaded NOR and NAND gates

Positive and Negative Logic

The binary signal at the inputs and outputs of any gate has one of two values, except during transition. One signal value represents logic 1 and the other logic 0. Since two signal values are assigned to two logic values, there exist two different assignments of signal level to logic value, as shown in Fig. 2.9. The higher signal level is designated by H and the lower signal level by L. Choosing the high-level H to represent logic 1 defines a positive logic system. Choosing the low-level L to represent logic 1 defines a negative logic system. The terms positive and negative are somewhat misleading, since both signals may be positive or both may be negative. It is not the actual values of the signals that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.

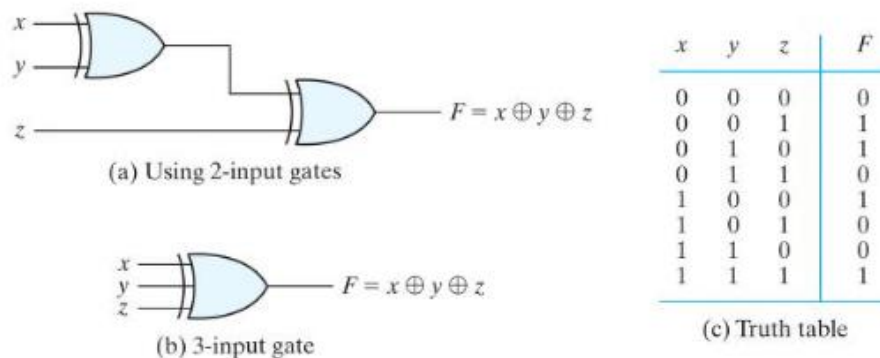


FIGURE 2.8
Three-input exclusive-OR gate

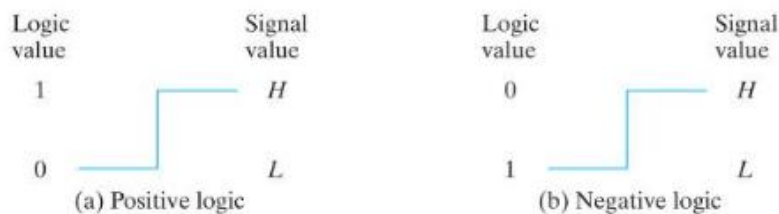
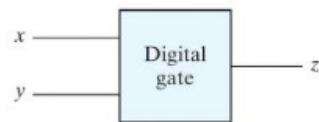


FIGURE 2.9
Signal assignment and logic polarity

x	y	z
L	L	L
L	H	L
H	L	L
H	H	H

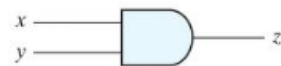
(a) Truth table with H and L



(b) Gate block diagram

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

(c) Truth table for positive logic



(d) Positive logic AND gate

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

(e) Truth table for negative logic



(f) Negative logic OR gate

FIGURE 2.10

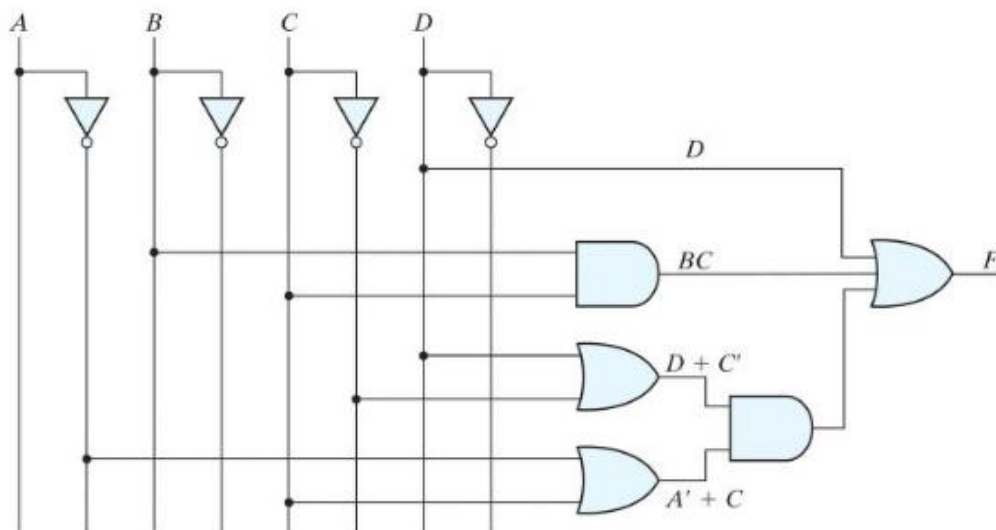
Demonstration of positive and negative logic

FIGURE :Demonstration of positive and negative logic

Draw the logic diagram corresponding to the following Boolean expression without simplifying it:

$$F = D + BC + (D + C')(A' + C).$$

Answer:



Practice Exercise 2.14

Implement the Boolean function $F = xz + x'z' + x'y$ with (a) NAND and inverter gates, and (b) NOR and inverter gates.

Answer:

$$\begin{aligned}F &= xz + x'z' + x'y \\F' &= (xz)' (x'z')' (x'y)' \\F &= [(xz)' (x'z')' (x'y)']'\end{aligned}$$

(a) NAND gates

$$\begin{aligned}F' &= (x' + z') (x + z)(x + y') \\F &= (x' + z')' + (x + z)' + (x + y')'\end{aligned}$$

(b) NOR gates

Gate-Level Minimization

INTRODUCTION

Gate-level minimization is 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, this dilemma has been solved by computer-based logic synthesis tools that minimize a large set of Boolean equations efficiently and quickly. Nevertheless, it is important that a designer understands the underlying mathematical description and solution of the gatelevel minimization problem. This chapter provides 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 the role and use of hardware description languages in modern logic design methodology.

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 describing the function. 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 in this section 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 method. 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 one that has a minimum number of terms 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 K-Map

The two-variable K-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.

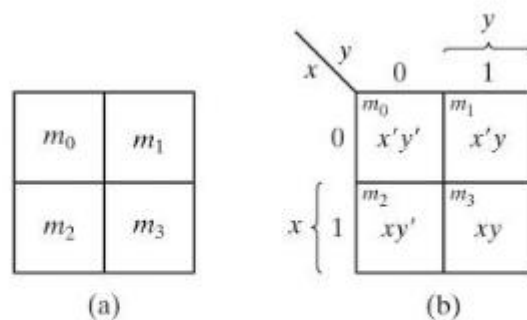


FIGURE 3.1
Two-variable K-map

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 minterm 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 union of the squares of variable x in the second row and those of 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.**

Three-Variable K-Map

A three-variable K-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 numbered minterms 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 above or beside the four squares in which it is unprimed.

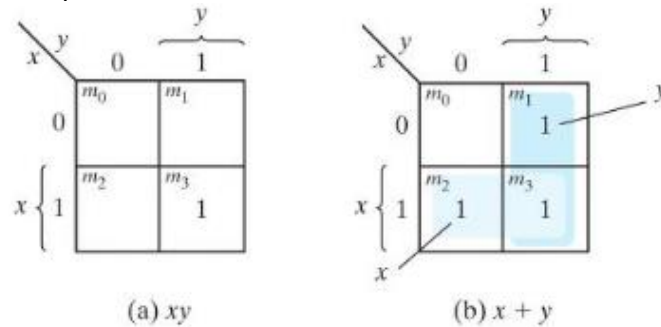


FIGURE 3.2
Representation of functions in the K-map

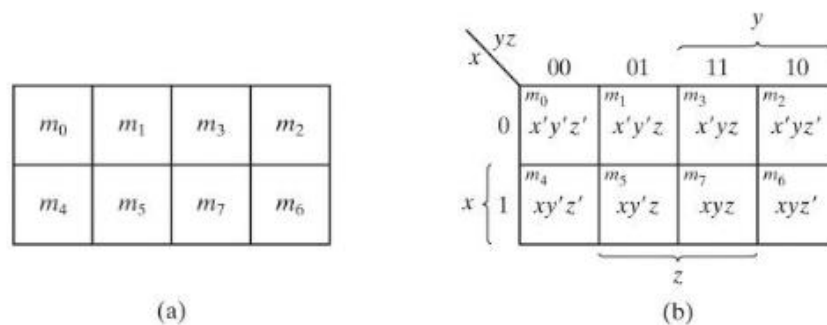


FIGURE 3.3
Three-variable K-map

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.¹ 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 product 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 K-map.

EXAMPLE

Simplify the Boolean function

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

First, a 1 is marked in each minterm square 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 shaded 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 sum of four minterms in the shaded squares can be replaced by a sum of only two product terms. The logical sum of these two product terms gives the simplified expression

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

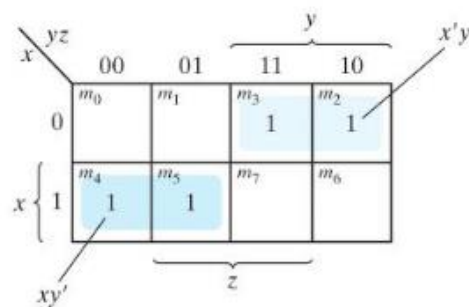


FIGURE 3.4

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(b), m_0 is adjacent to m_2 and m_4 is adjacent to m_6 because their 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.

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 shaded squares in the third column are combined to give a two-literal term yz . The remaining two squares with 1's are also adjacent by the new definition. These two shaded squares, when combined, give the two-literal term xz' . The simplified function then becomes

$$F = yz + xz'$$

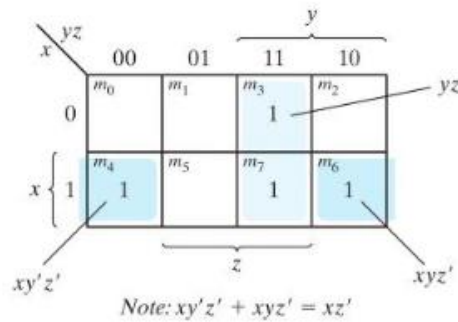


FIGURE 3.5

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' :

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

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 three-variable 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 also 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

$$F = z' + xy'$$

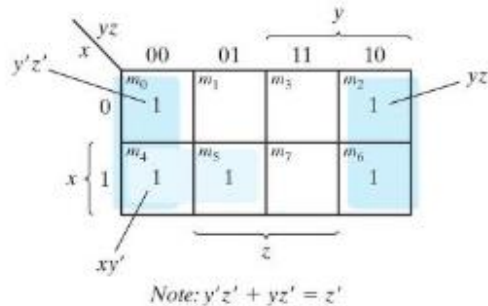


FIGURE 3.6

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

For the Boolean function

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

- Express this function as a sum of minterms.
- Find the minimal sum-of-products expression.

Note that F is a sum of products, but not a sum of minterms. 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 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 re-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 indicated by the shaded squares in the map, to an expression with only two terms:

$$F = C + A'B$$

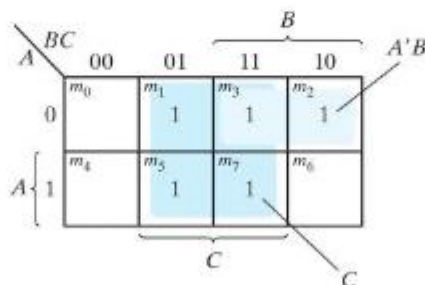


FIGURE 3.7

Map of Example 3.4, $A'C + A'B + AB'C + BC = C + A'B$

FOUR-VARIABLE K-MAP

The map for Boolean functions of four binary variables (w, x, y, z) is shown in Fig. 3.8, which lists the 16 minterms and the squares assigned to each. In Fig. 3.8(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} .

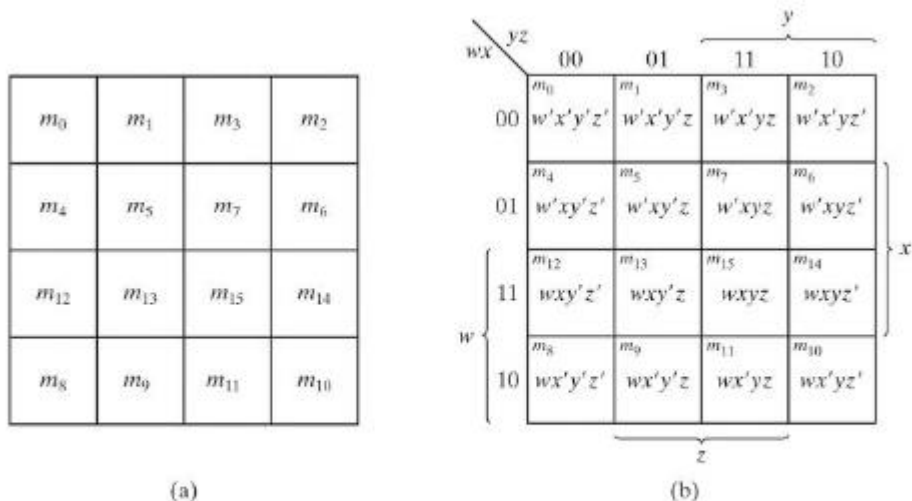


FIGURE 3.8
Four-variable map

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 (vertically or horizontally, but not diagonally). 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, 7) and m_{15} 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 shaded, adjacent squares marked with 1's can be combined to form the one literal term y' . The remaining three

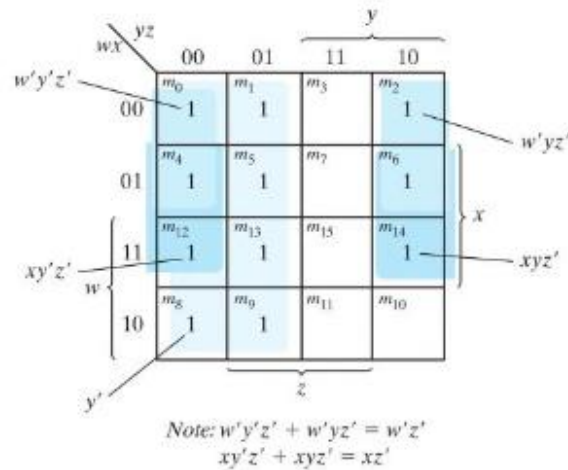


FIGURE 3.9

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

Don't-Care Conditions:

The "Don't Care" conditions allow us to replace the empty cell of a K-Map to form a grouping of the variables which is larger than that of forming groups without don't care. While forming groups of cells, we can consider a "Don't Care" cell as 1 or 0 or we can also ignore that cell.

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.

EXAMPLE 3.8

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. 3.15. The minterms of F are marked by 1's, those of d are marked 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

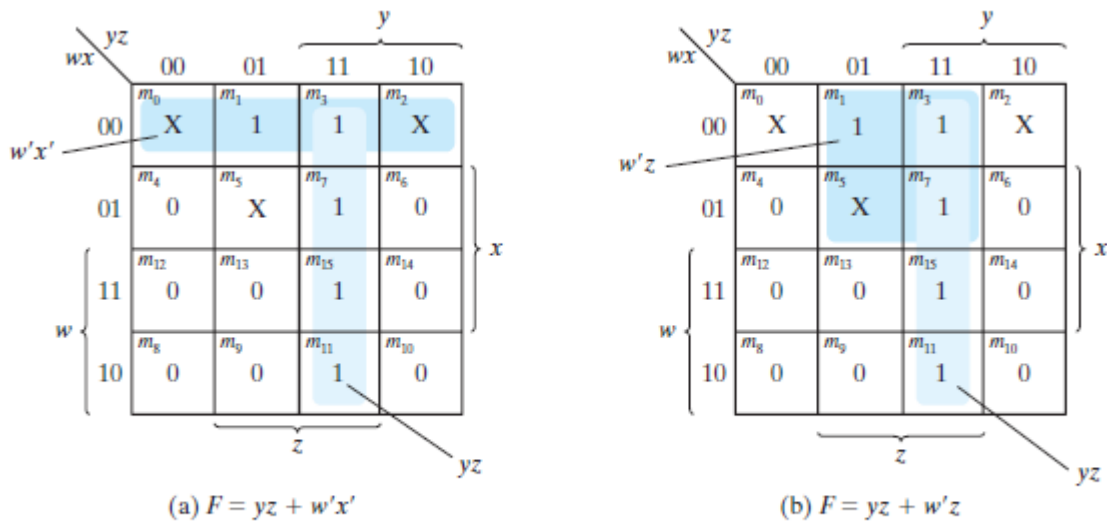


FIGURE 3.15
Example with don't-care conditions

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 Fig. 3.15(a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

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

In Fig. 3.15(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 3.8:

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

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

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. 3.15. 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.

NAND AND NOR IMPLEMENTATION:

Any logic function can be implemented using NAND and NOR gates. To achieve this, first the logic function has to be written in Sum of Product (SOP) form. Once logic function is converted to SOP, then

is very easy to implement using NAND gate. In other words any logic circuit with AND gates in first level and OR gates in second level can be converted into a NAND-NAND gate circuit.

SECTION 3.6 NAND and NOR Implementation 71

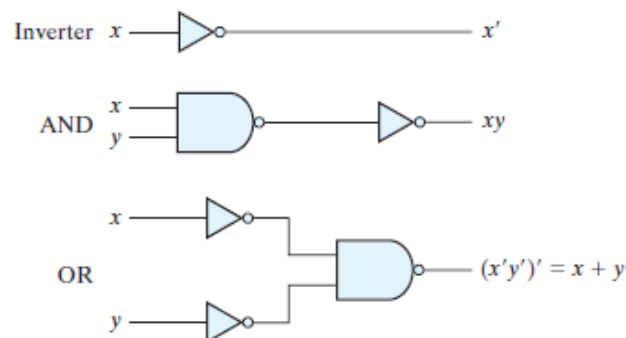


FIGURE 3.16
Logic operations with NAND gates

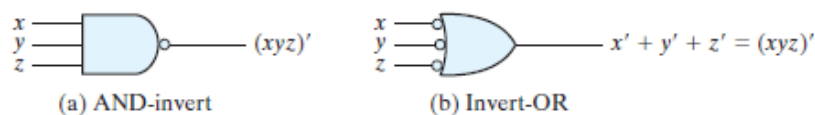


FIGURE 3.17
Two graphic symbols for a three-input NAND gate

NAND Circuits:

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.

Two-Level Implementation

The implementation of two-level Boolean functions with NAND gates requires that the functions be in sum-of-products form. To see the relationship between a sum-of-products expression and its equivalent NAND implementation, consider the logic diagrams drawn in Fig. 3.18. All three diagrams are equivalent and implement the function

$$F = AB + CD$$

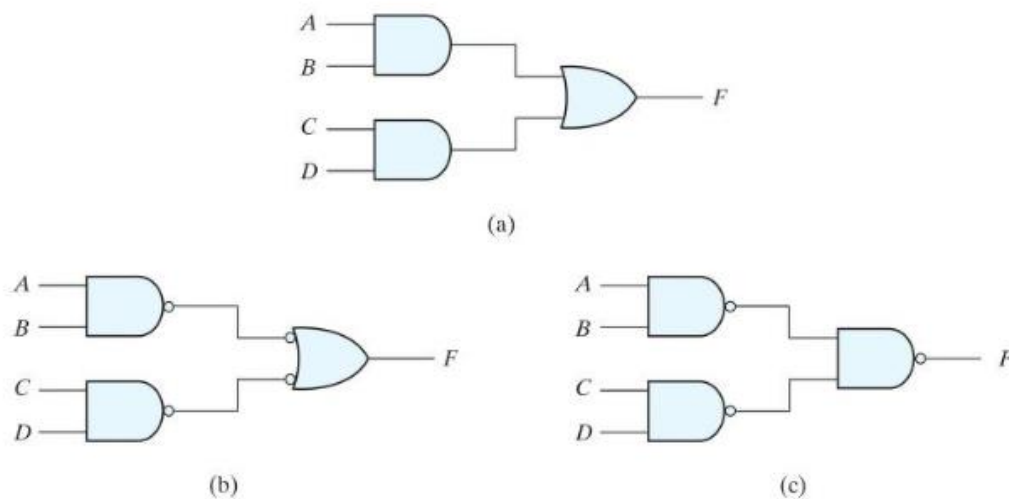


FIGURE 3.18
Three ways to implement $F = AB + CD$

Example:

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. 3.19(a), from which the simplified function is obtained:

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

The two-level NAND implementation is shown in Fig. 3.19(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. 3.19(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' .

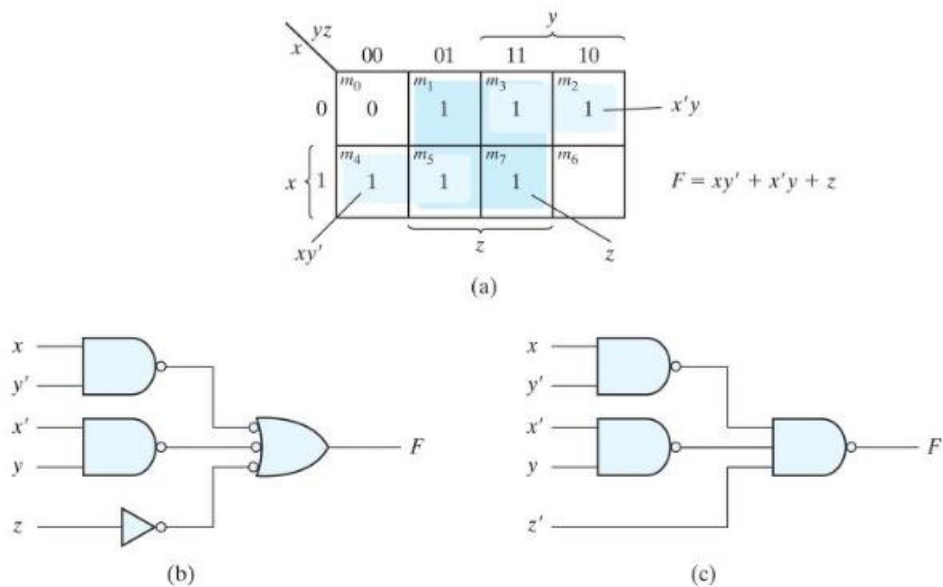


FIGURE 3.19
Solution to Example 3.9

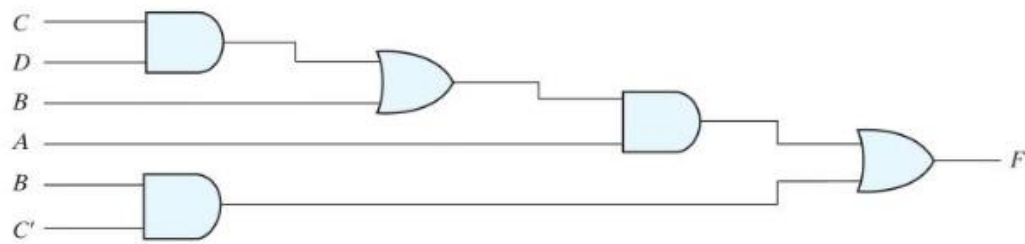
The procedure described in the previous example indicates that a Boolean function can be implemented with two levels of NAND gates. The procedure for obtaining the logic diagram from a Boolean function is as follows:

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.

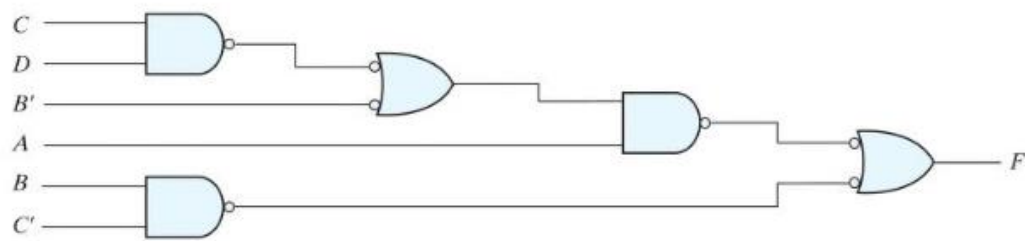
Multilevel NAND Circuits:

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

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



(a) AND-OR gates



(b) NAND gates

FIGURE 3.20

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

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

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

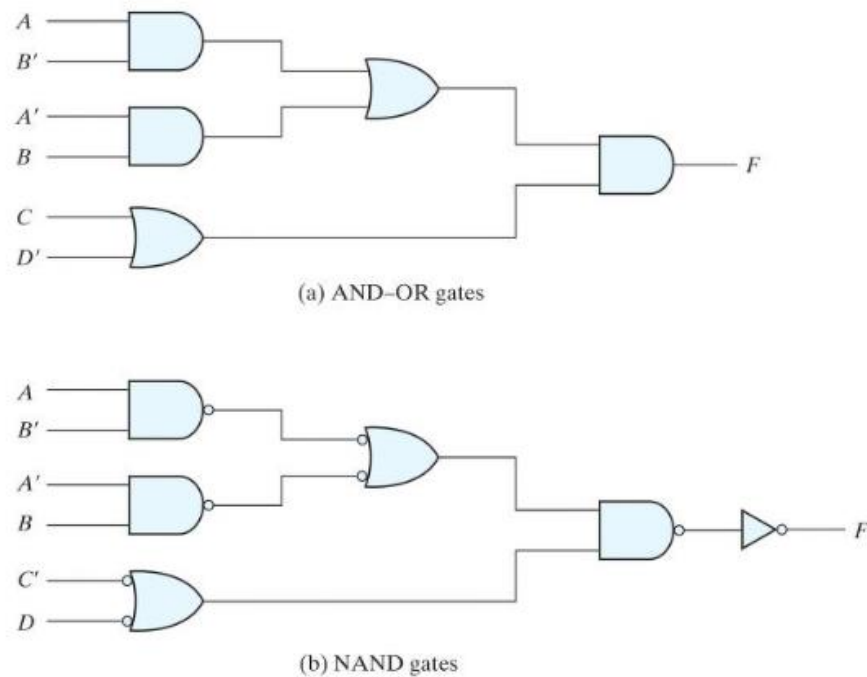


FIGURE 3.21

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. 3.22. 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. 3.23. The ORinvert 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.

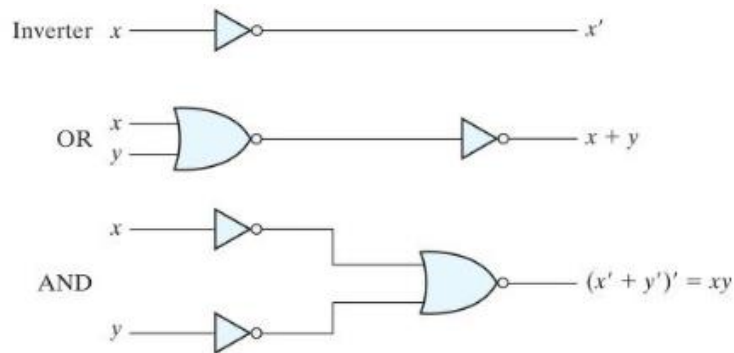


FIGURE 3.22
Logic operations with NOR gates

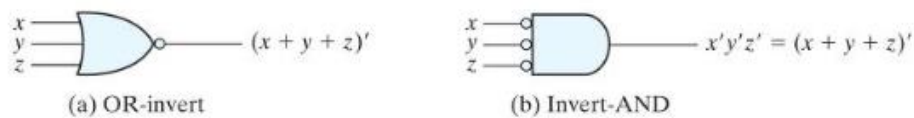


FIGURE 3.23
The OR-invert and Invert-AND gates

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. Figure 3.24 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. 3.21(a) into a NOR diagram is shown in Fig. 3.25. The Boolean function for this circuit is

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

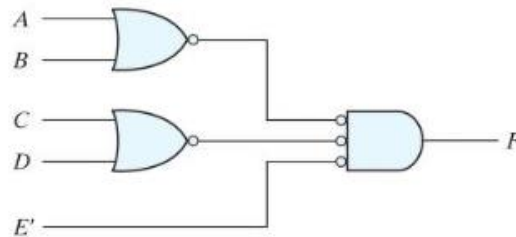


FIGURE 3.24
Implementing $F = (A + B)(C + D)E$

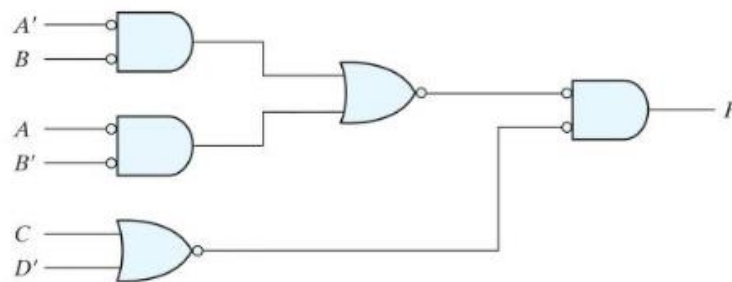
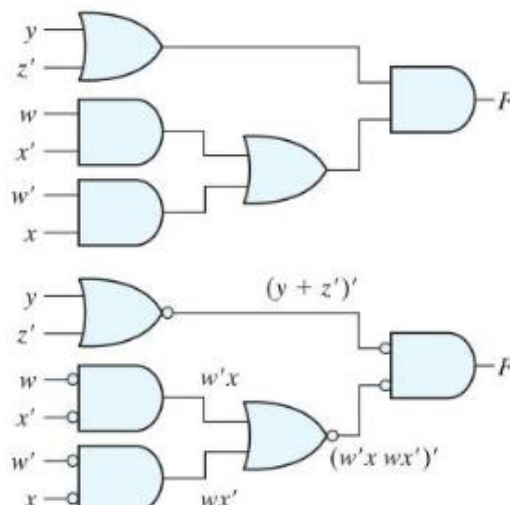


FIGURE 3.25
Implementing $F = (AB' + A'B)(C + D')$ with NOR gates

Implement the Boolean function $F(w, x, y, z) = (y + z')(wx' + w'x)$ with NOR gates.

Answer:



HARDWARE DESCRIPTION LANGUAGES (HDLs):

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 to reduce costs and minimize the risk of creating 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. Before the advent of HDLs, designers relied on schematics or block diagrams and logic gates to represent and specify a circuit. That methodology is prone to error and its results are costly to edit, especially for complex circuits. In contrast, today's HDL-based design tools create an HDL description, then derive a schematic automatically and correctly, as a by-product of the design methodology. Revisions of the HDL description simplify the creation and revision of a schematic.

An HDL is a modeling language rather than a computational language. An HDL 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. Those features distinguish an HDL from other types of languages, many of which are used to perform computations on numerical data. 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 the 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 net list 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 shows how the hardware will behave before it is actually fabricated. Simulation detects 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 Design (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. That approach is beyond the level of this text. We will focus exclusively on simulation.

Logic synthesis derives an optimized 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 produces a database describing the elements and structure of a circuit. It specifies how to fabricate a physical integrated circuit that implements in silicon the functionality described by statements made in an HDL. Logic synthesis (1) is based on formal procedures that implement digital circuits, and (2) performs logic minimization on those parts of a digital design process, which can be automated with computer software. The design of today's large, complex circuits is made possible by logic

synthesis software. It is essential that users of an HDL realize that not all constructs of the language are synthesizable.

Timing verification confirms that a synthesized and 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 implementation 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.

Module Declaration

The language reference manual for the Verilog HDL presents the syntax that describes precisely the constructs of the language. 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 emphasis and clarity, keywords will be identified in the text by displaying them in boldface in all examples of code and wherever it is helpful to call attention to their use. Lines of text terminate with a semicolon (;), and any text between two forward slashes (//) and the end of the line is interpreted as a comment. A comment has no effect on a simulation using the model. Multiline comments begin with /* and terminate with */. They may not be nested. 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 by the keyword pair module . . . endmodule. A module is the fundamental descriptive (design) unit in the Verilog language. It is declared by the keyword module and must always be terminated by the keyword endmodule.

Previous sections of the text have demonstrated that combinational logic can be described by a set of Boolean equations, by a schematic connection of gates, or by a truth table. Now we'll consider how HDLs implement these descriptions of combinational logic.

Verilog Example

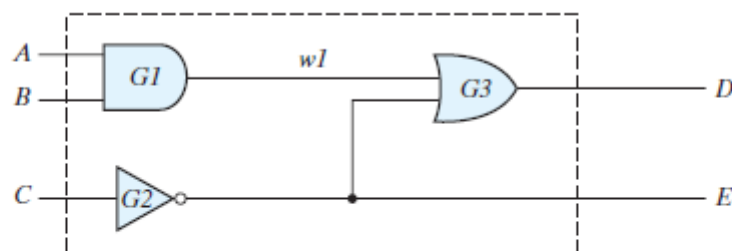


FIGURE 3.35
Circuit to demonstrate an HDL

HDL Example 3.1 (Combinational Logic Modeled with Primitives)

// Verilog model of circuit of Figure 3.35. 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 inputs are 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 within 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**.

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 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 1ns ,Our examples will use only the default time unit.

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
```

HDL Example 3.3 (Test Bench)

// 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
```

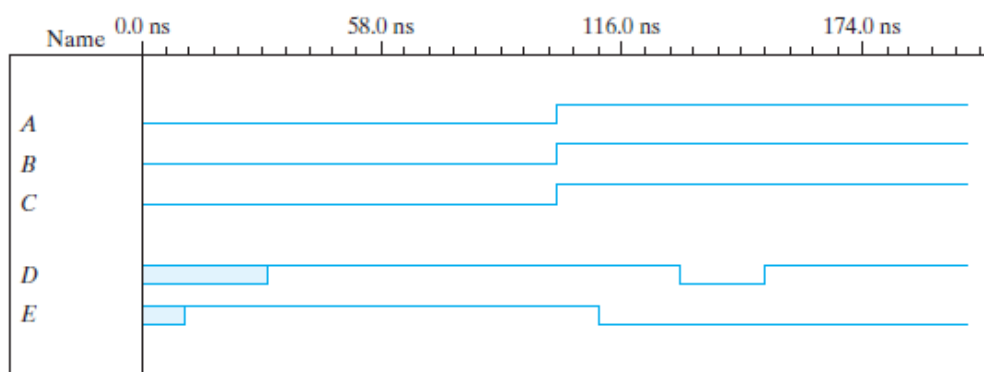


FIGURE 3.36
Simulation output of HDL Example 3.3

Boolean Expressions

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
```

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**.

HDL Example 3.5 (User-Defined Primitive)

// Verilog model: User-defined Primitive

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

output D;

input A, B, C;

// Truth table for $D = f(A, B, C) = \sum(0, 2, 4, 6, 7)$;

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

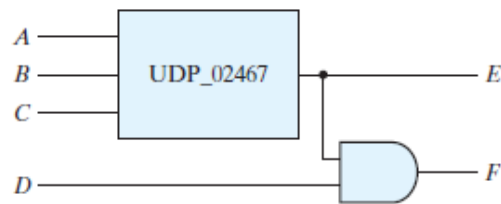


FIGURE 3.37
Schematic for *Circuit with_UDP_02467*

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 3.37.

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 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 organized 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.