

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.

Text book 1: 1.9, 2.4, 2.5, 2.8, 3.1, 3.2, 3.3, 3.5, 3.6, 3.9

1.1 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, it is convenient to think in terms of bits and assign the values 1 and 0.
- The binary logic introduced is equivalent to an algebra called Boolean algebra.

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 have 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 $z = 0$.

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, i.e., the result of complementing 1 is 0, and vice versa.

Binary logic resembles binary arithmetic, and the operations AND and OR have similarities to multiplication and addition, respectively.

In fact, the symbols used for AND and OR are the same as those used for multiplication and addition.

However, **binary logic should not be confused with binary arithmetic**. One should realize that an arithmetic variable designates a number that may consist of many digits.

A logic variable is always either 1 or 0.

For example, in binary arithmetic, we have $1 + 1 = 10$ (read “one plus one is equal to 2”), whereas in binary logic, we have $1 + 1 = 1$.

- 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 Fig 1.1.

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		

FIGURE 1.1 Truth table

Logic Gates

- Logic gates are electronic circuits that operate on one or more 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 to 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. 1.2.

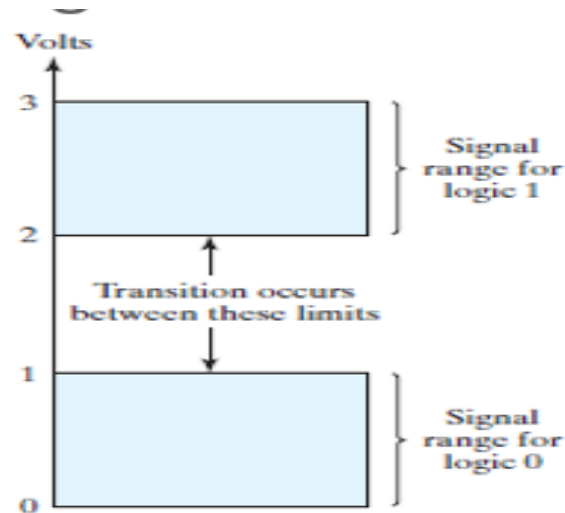


FIGURE 1.2 Signal levels for binary logic values

- 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. The graphic symbols used to designate the three types of gates are shown in Fig. 1.3.

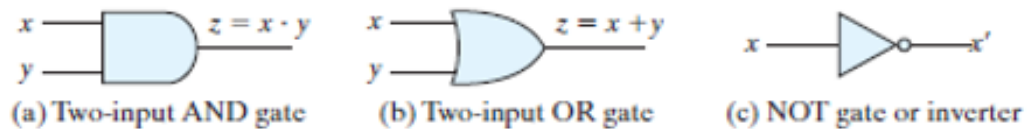


FIGURE 1.3 Symbols for digital logic circuits

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. 1.4 together with the corresponding output signal f each gate.

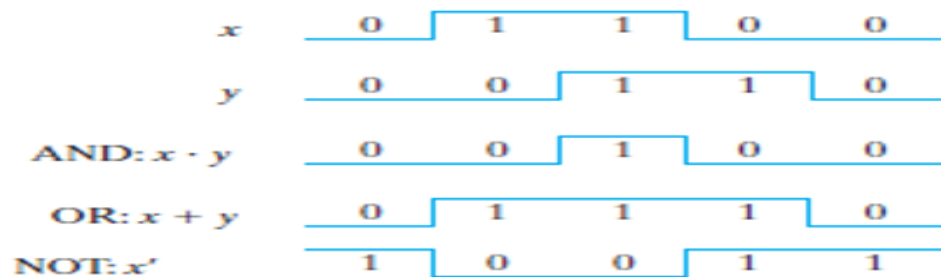


FIGURE 1.4 Input-output signals for gates

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, 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. 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. 1.5.

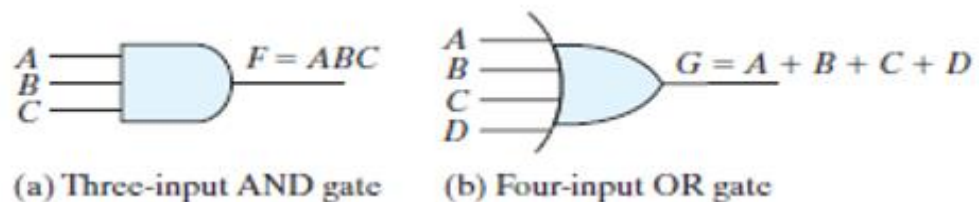


FIGURE 1.5 Gates with multiple inputs

- 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.

1.2 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 1.1 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.

TABLE 1.1

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)	$x(x + y) = x$

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

The algebraic proofs of the associative law and DeMorgan's theorem are long and will not be shown

here. However, their validity is easily shown with truth tables. For example, the truth table for the first DeMorgan's theorem, $(x + y)' = x'y'$, is as follows:

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 DeMorgan'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.

1.3 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 $F1 = x + y'z$. The function $F1$ is equal to 1 if x is equal to 1 or if both y' and z are equal to 1. $F1$ is equal to 0 otherwise.
- The complement operation dictates that when $y' = 1$, $y = 0$. Therefore, $F1 = 1$ 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 1.2

<i>Truth Tables for F_1 and F_2</i>				
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

- Table 1.2 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 is equal to 0 otherwise.
- A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure.
- The logic-circuit diagram (also called a schematic) for F_1 is shown in Fig.1.6 . 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 $y'z$.

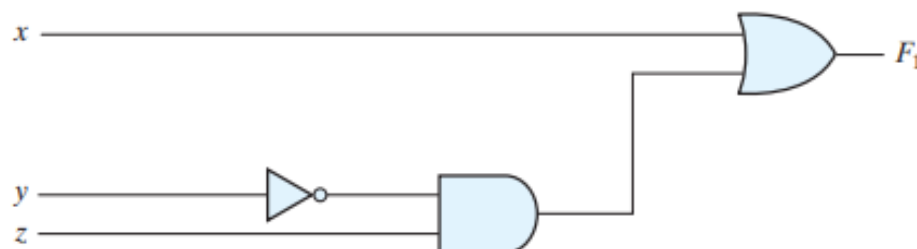


FIGURE 1.6 Gate Implementation of $F_1 = x + y'z$

- In logic-circuit diagrams, the variables of the function are taken as the inputs of the circuit and the binary variable F_1 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.
- 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.
- 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. 1.7 (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 F₂ is listed in Table 1.2 .
- 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 F₂.

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. 1.7 (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 produce the same truth table, they are equivalent.

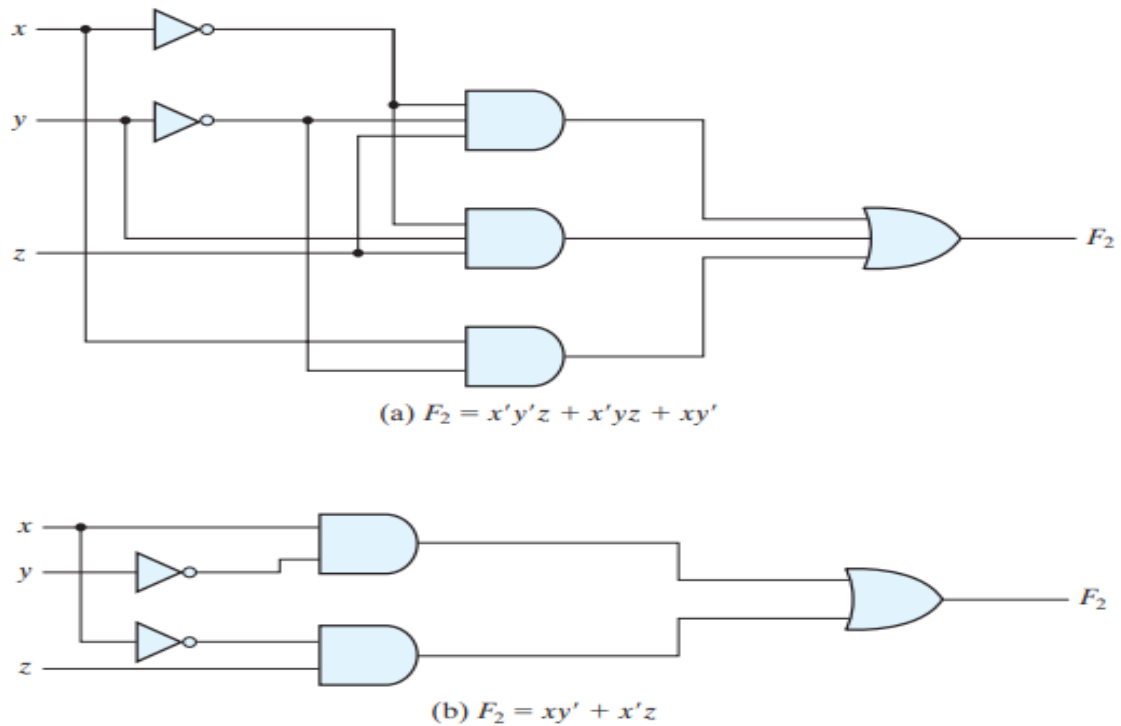


FIGURE 1.7 Implementation of Boolean function F_2 with gates

- 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. 1.9 (a) has three terms and eight literals, and the one in Fig. 1.9 (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.

- 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. The only manual method available is a cut-and-try procedure employing the basic relations and other manipulation techniques that become familiar with use, but remain, nevertheless, subject to human error.
- The examples that follow illustrate the algebraic manipulation of Boolean algebra to acquaint the reader with this important design task.

Example 1.1: Simplify the following Boolean functions to a minimum number of literal

1. $x(x' + y) = xx' + xy = 0 + xy = xy.$
2. $x + x'y = (x + x')(x + y) = 1(x + y) = x + y.$
3. $(x + y)(x + y') = x + xy + xy' + yy' = x(1 + y + y') = x.$
4. $xy + x'z + yz = xy + x'z + yz(x + x')$
 $= xy + x'z + xyz + x'yz$
 $= xy(1 + z) + x'z(1 + y)$
 $= xy + x'z.$
5. $(x + y)(x' + z)(y + z) = (x + y)(x' + z),$ by duality from function 4.

- Functions 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 1.1 :

$$(x + y)(x + y') = x + yy' = x.$$
- The fourth function illustrates the fact that an increase in the number of literals sometimes leads to a simpler final expression.
- Function 5 is not minimized directly, but can be derived from the dual of the steps used to derive function 4. Functions 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 1.1 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 1.1 :

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

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:

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

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.

EXAMPLE1.2: 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:

$$\begin{aligned}
 F_1' &= (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z') \\
 F_2' &= [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')'(yz)' \\
 &= x' + (y + z)(y' + z') \\
 &= x' + yz' + y'z
 \end{aligned}$$

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.

EXAMPLE 1.3: Find the complement of the functions F_1 and F_2 of Example 1.2 by taking their duals and complementing each literal.

1. $F_1 = x'yz' + x'y'z.$

The dual of F_1 is $(x' + y + z')(x' + y' + z).$

Complement each literal: $(x + y' + z)(x + y + z') = F_1'.$

2. $F_2 = x(y'z' + yz).$

The dual of F_2 is $x + (y' + z')(y + z).$

Complement each literal: $x' + (y + z)(y' + z') = F_2'.$

Boolean algebra has two binary operators, which we have called AND and OR, and a unary operator, NOT (complement). From the definitions, we have deduced a number of properties of these operators and now have defined other binary operators in terms of them. There is nothing unique about this procedure. We could have just as well started with the operator NOR (\downarrow), for example, and later defined AND, OR, and NOT in terms of it.

1.4 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 types of gates. 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.



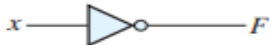





Name	Graphic symbol	Algebraic function	Truth table															
AND		$F = x \cdot y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	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																

FIGURE 1.8 Digital logic gates

- 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 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.

- The NAND function is the complement of the AND function, as indicated by a graphic symbol that consists of an AND graphic symbol followed by a small circle.
- The NOR function is the complement of the OR function and uses an OR graphic symbol followed by a small circle. NAND and NOR gates are used extensively as standard logic gates and are in fact far more popular than the AND and OR gates. This is because NAND and NOR gates are easily constructed with transistor circuits and because digital circuits can be easily implemented with them.
- The exclusive-OR gate has a graphic symbol similar to that of the OR gate, except for the additional curved line on the input side.
- The exclusive-NOR, gate is the complement of the exclusive-OR, as indicated by the small circle on the output side of the graphic symbol.

Extension to Multiple Inputs

- The gates shown in Fig. 1.8 —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

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

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

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. 1.9 and the following equations:

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

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. 1.10 . In writing cascaded NOR and NAND operations, one must use the correct parentheses to signify the proper sequence of the gates. To demonstrate this principle, consider the circuit of Fig. 1.10 (c). The Boolean function for the circuit must be written as

$$F = [(ABC)'(DE)']' = ABC + DE$$

The second expression is obtained from one of DeMorgan's theorems. It also shows that an expression in sum-of-products form can be implemented with NAND gates.

The exclusive-OR and equivalence gates are both commutative and associative and can be extended to more than two inputs. However, multiple-input exclusive-OR gates are uncommon from the hardware standpoint.

In fact, even a two-input function is usually constructed with other types of gates. Moreover, the definition of the function must be modified when extended to more than two variables. Exclusive-OR is an odd function (i.e., it is equal to 1 if the input variables have an odd number of 1's). The construction

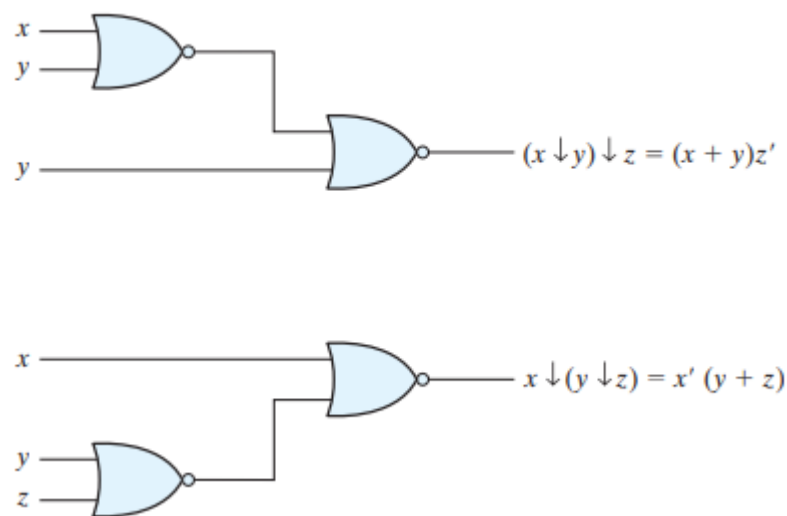


FIGURE 1.9 Demonstrating the nonassociativity of the NOR operator:

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

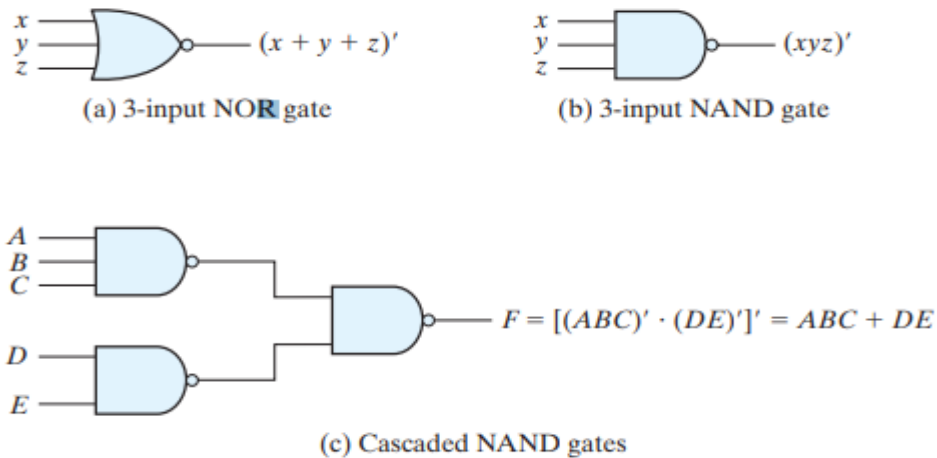


FIGURE 1.10 Multiple-input and cascaded NOR and NAND gates

of a three-input exclusive-OR function is shown in Fig. 1.11 . This function is normally implemented by cascading two-input gates, as shown in (a). Graphically, it can be represented with a single three-input gate, as shown in (b). The truth table in (c) clearly indicates that the output F is equal to 1 if only one input is equal to 1 or if all three inputs are equal to 1 (i.e., when the total number of 1's in the input variables is odd).

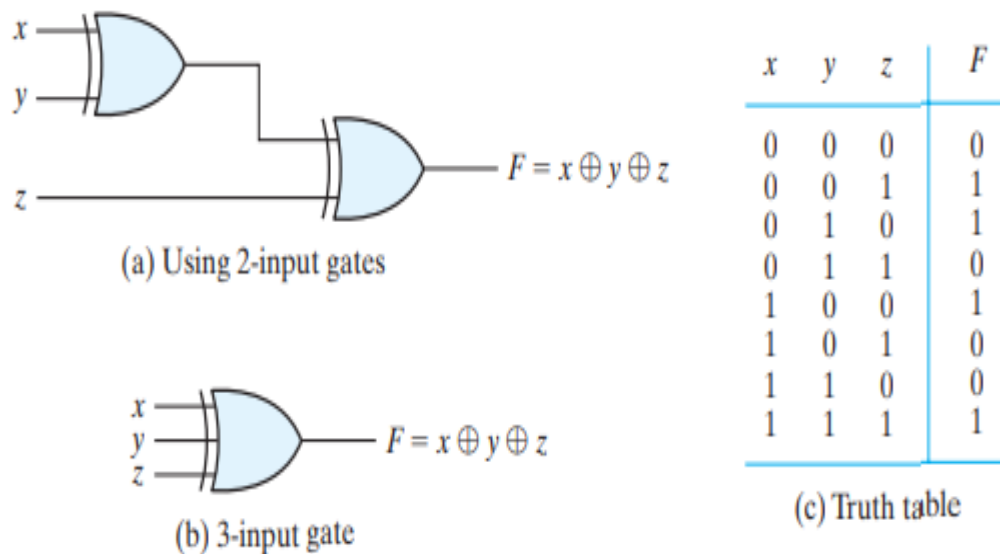


FIGURE 1.11 Three-input exclusive-OR gate

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. 1.12 .

- 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.

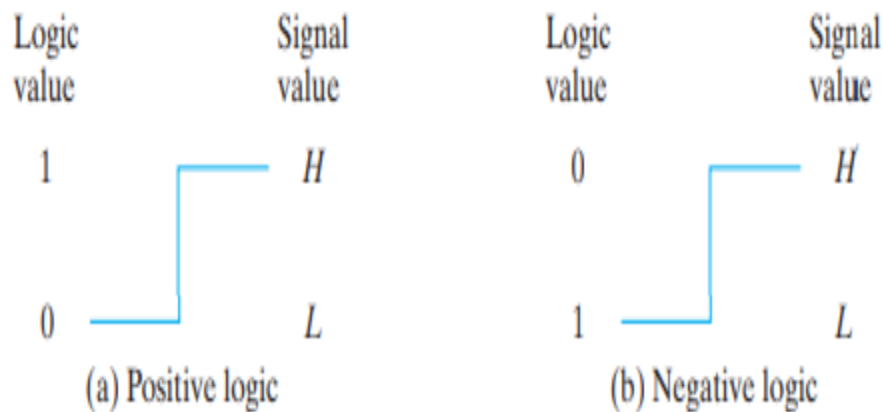


FIGURE 1.12 Signal assignment and logic polarity

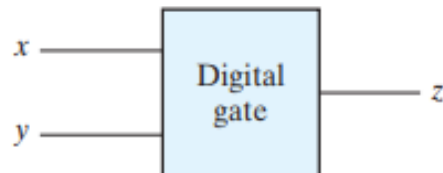
Hardware digital gates are defined in terms of signal values such as H and L. It is up to the user to decide on a positive or negative logic polarity.

- Consider, for example, the electronic gate shown in Fig. 1.13 (b). The truth table for this gate is listed in Fig. 1.13 (a). It specifies the physical behavior of the gate when H is 3 V and L is 0 V. The truth table of Fig. 1.13 (c) assumes a positive logic assignment, with H = 1 and L = 0. This truth table is the same as the one for the AND operation. The graphic symbol for a positive logic AND gate is shown in Fig. 1.13 (d).
- Now consider the negative logic assignment for the same physical gate with L = 1 and H = 0. The result is the truth table of Fig. 1.13 (e). This table represents the OR operation, even though the entries are reversed. The graphic symbol for the negative-logic OR gate is shown in Fig. 1.13 (f).
- The small triangles in the inputs and output designate a polarity indicator, the presence of which along a terminal signifies that negative logic is assumed for the signal.
- Thus, the same physical gate can operate either as a positive-logic AND gate or as a negative-logic OR gate.

- The conversion from positive logic to negative logic and vice versa is essentially an operation that changes 1's to 0's and 0's to 1's in both the inputs and the output of a gate. Since this operation produces the dual of a function, the change of all terminals from one polarity to the other results in taking the dual of the function.

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 1.13 Demonstration of positive and negative logic

1.5 INTEGRATED CIRCUITS

An integrated circuit (IC) is fabricated on a die of a silicon semiconductor crystal, called a *chip*, containing the electronic components for constructing digital gates.

The various gates are interconnected inside the chip to form the required circuit. The chip is mounted in a ceramic or plastic container, and connections are welded to external pins to form the integrated circuit. The number of pins may range from 14 on a small IC package to several thousand on a larger package. Each IC has a numeric designation printed on the surface of the package for identification.

Levels of Integration

- Digital ICs are often categorized according to the complexity of their circuits, as measured by the number of logic gates in a single package. The differentiation between those chips which have a few internal gates and those having hundreds of thousands of gates is made by customary reference to a package as being either a small-, medium-, large-, or very large-scale integration device.
- **Small-scale integration (SSI)** devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually fewer than 10 and is limited by the number of pins available in the IC.
- **Medium-scale integration (MSI)** devices have a complexity of approximately 10 to 1,000 gates in a single package. They usually perform specific elementary digital operations
- **Large-scale integration (LSI)** devices contain thousands of gates in a single package. They include digital systems such as processors, memory chips, and programmable logic devices.
- **Very large-scale integration (VLSI)** devices now contain millions of gates within a single package. Examples are large memory arrays and complex microcomputer chips. Because of their small size and low cost, VLSI devices have revolutionized the computer system design technology, giving the designer the capability to create structures that were previously uneconomical to build.

Digital Logic Families

- Digital integrated circuits are classified not only by their complexity or logical operation, but also by the specific circuit technology to which they belong. The circuit technology is referred to as a *digital logic family*.
- Each logic family has its own basic electronic circuit upon which more complex digital circuits and components are developed. The basic circuit in each technology is a NAND, NOR, or inverter gate.

1.6 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, 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.

1.7 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. 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.

Two-Variable K-Map

The two-variable map is shown in Fig. 1.14 a)

(a). There are four minterms for two variables; hence, the map consists of four squares, one for each minterm.

(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.

- 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. 1.15 (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. 1.15 (b) by three squares marked with 1's. These squares are found from the minterms of the function:

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

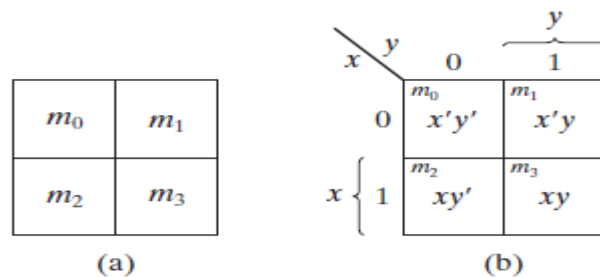


FIGURE 1.14 Two-variable K-map

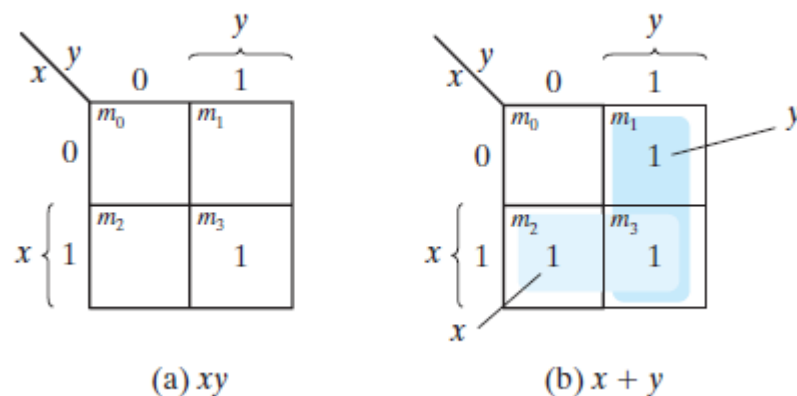


FIGURE 1.15 Representation of functions in the map

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.

Three-Variable K-Map

- A three-variable K-map is shown in Fig. 1.16 . 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. 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.

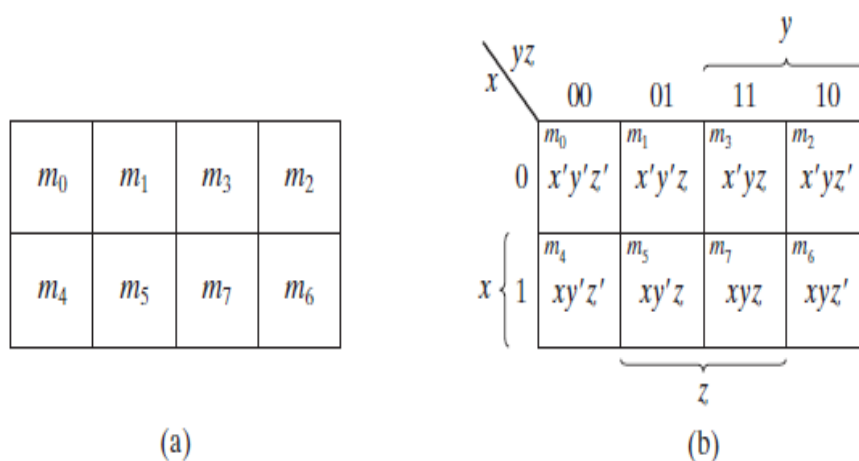


FIGURE 1.16 Three-variable K-map

- 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.
- 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:1.4 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. 1.17, 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 can be replaced by a sum of only two product terms.

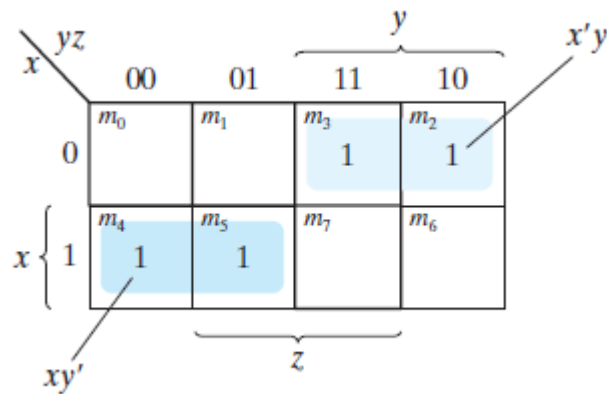


FIGURE 1.17 The logical sum of these two product terms gives the simplified Expression

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

In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other. In Fig. 1.16 (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:

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

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.

EXAMPL:1.5 Simplify the Boolean function

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

The map for this function is shown in Fig. 1.18 . 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

$$F = yz + xz'$$

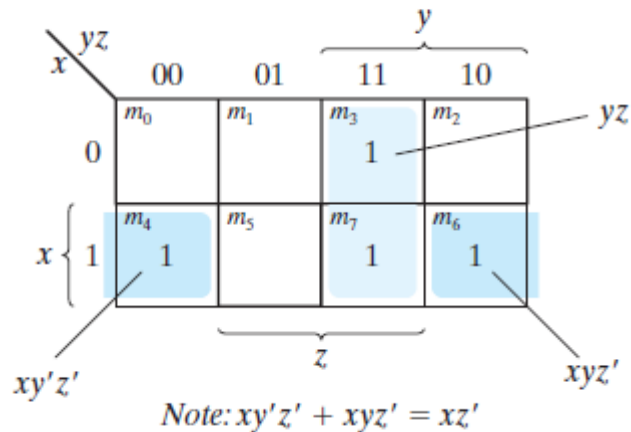


FIGURE 1.18 Map for Example $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

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:1.6 Simplify the Boolean function

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

- The map for F is shown in Fig. 1.19 . 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

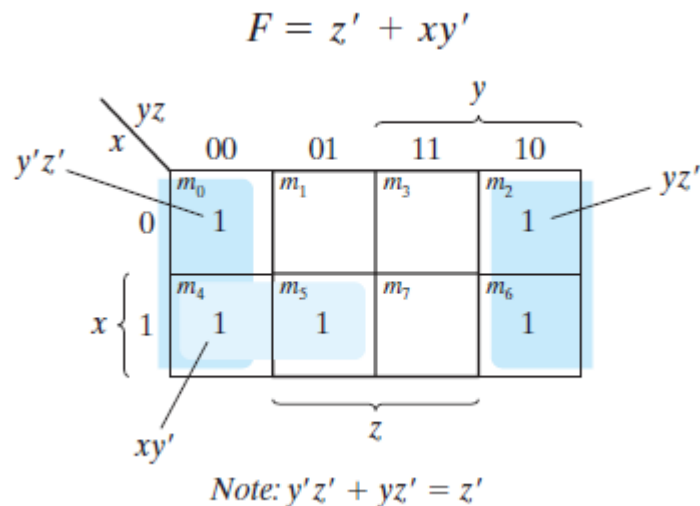


FIGURE 1.19 Map for Example 3 $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

- 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, more squares.
- The minterms of the function are then read directly from the map.

EXAMPLE :1.7 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. 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. 1.20 from the coincidence of A' (first row) and C (two middle columns) to give squares 001 and 011.

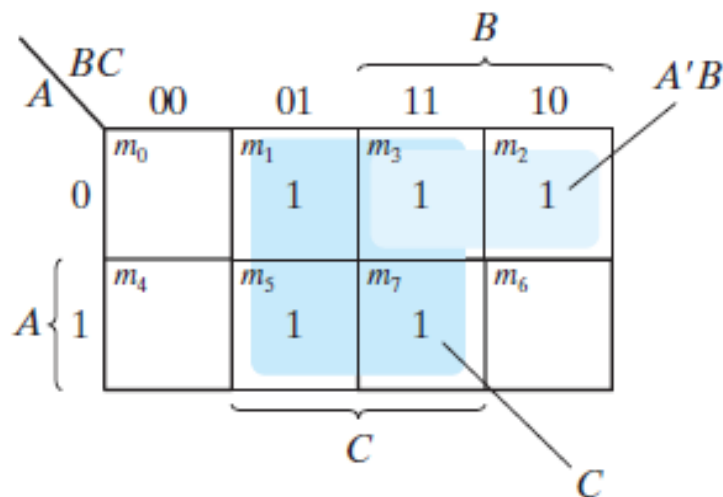
Note that, in marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term. This happens with the second term, $A'B$, which has 1's in squares 011 and 010.

- 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. 1.20. 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:

$$F = C + A'B$$



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

FIGURE 1.20 Map of Example

1.8 FOUR-VARIABLE K-MAP

- In Fig. 1.21(a) are listed the 16 minterms and the squares assigned to each. In Fig. 1.21(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, m_0 and m_2 form adjacent squares, as do m_3 and m_1 . 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.

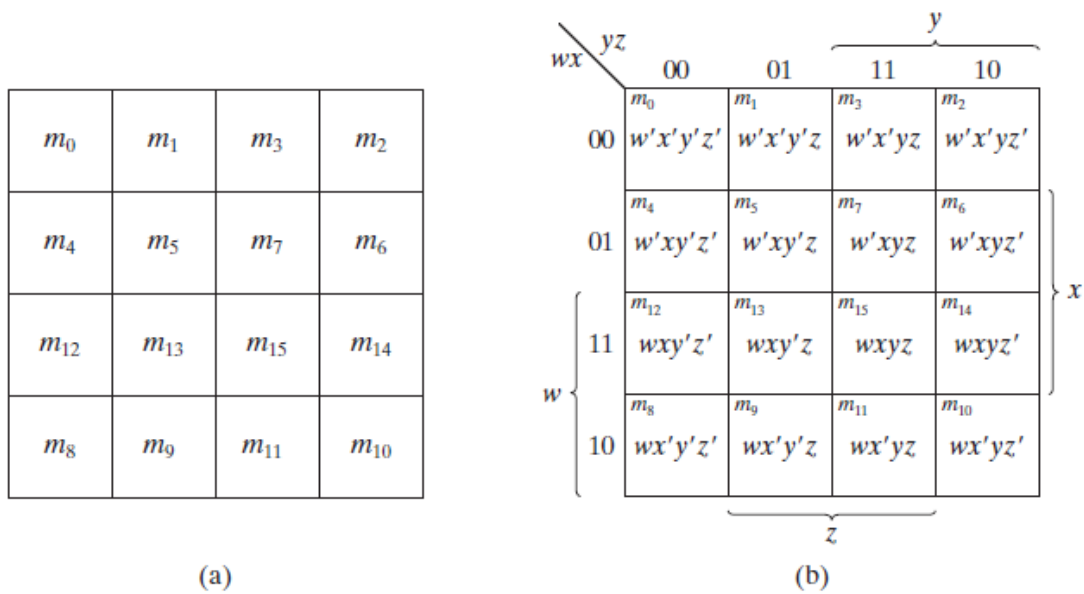


FIGURE 1.21 Four-variable map

EXAMPLE 1.8 :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. 1.22 . 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 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

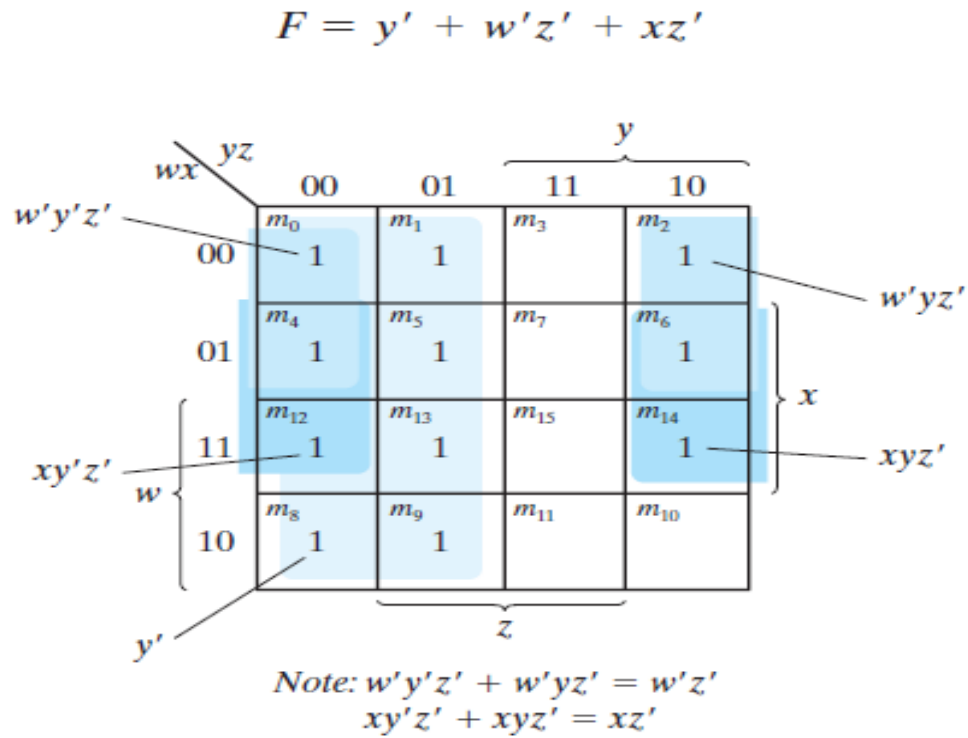


FIGURE 1.22 Map for Example

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

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

EXAMPLE 1.9 :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 Fig1.23 .

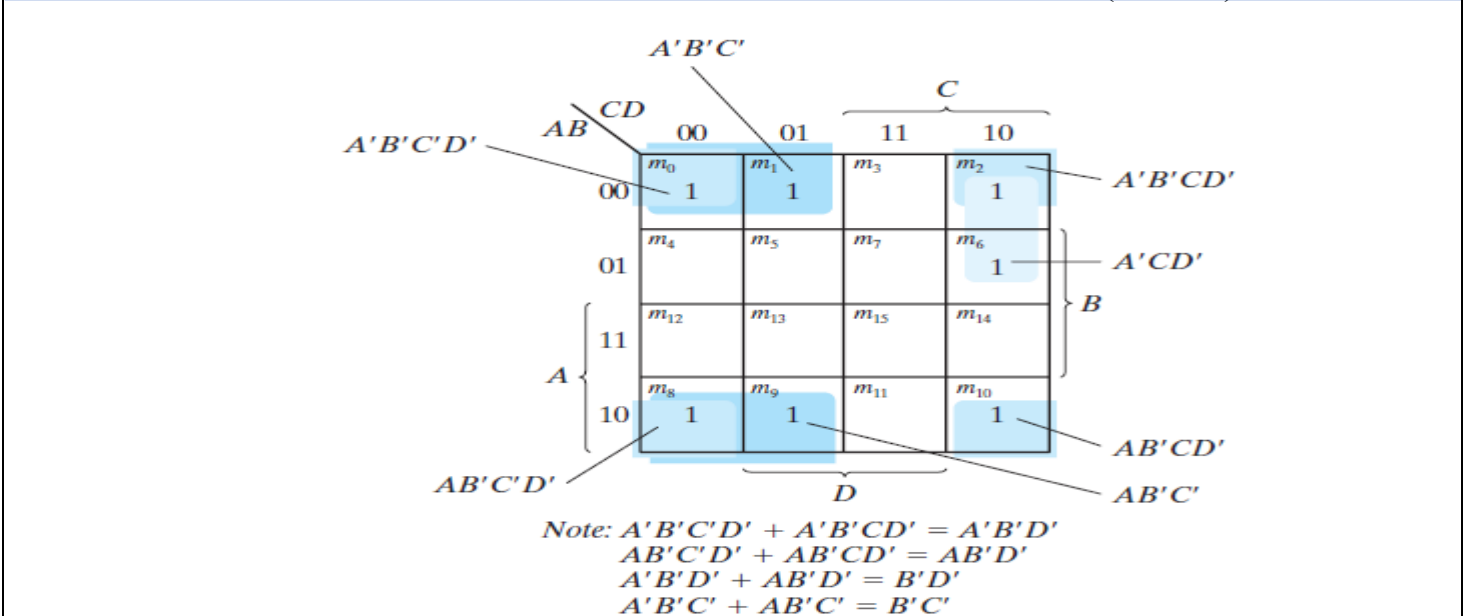


FIGURE 1.23 Map for Example 3.6, $A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$

- 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

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

Prime Implicants

- In choosing adjacent squares in a map, we must ensure that
 - all the minterms of the function are covered when we combine the squares,
 - the number of terms in the expression is minimized, and
 - there are no redundant 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. 1.24. The partial map (Fig. 1.24(a)) 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 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 1.24 (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} .

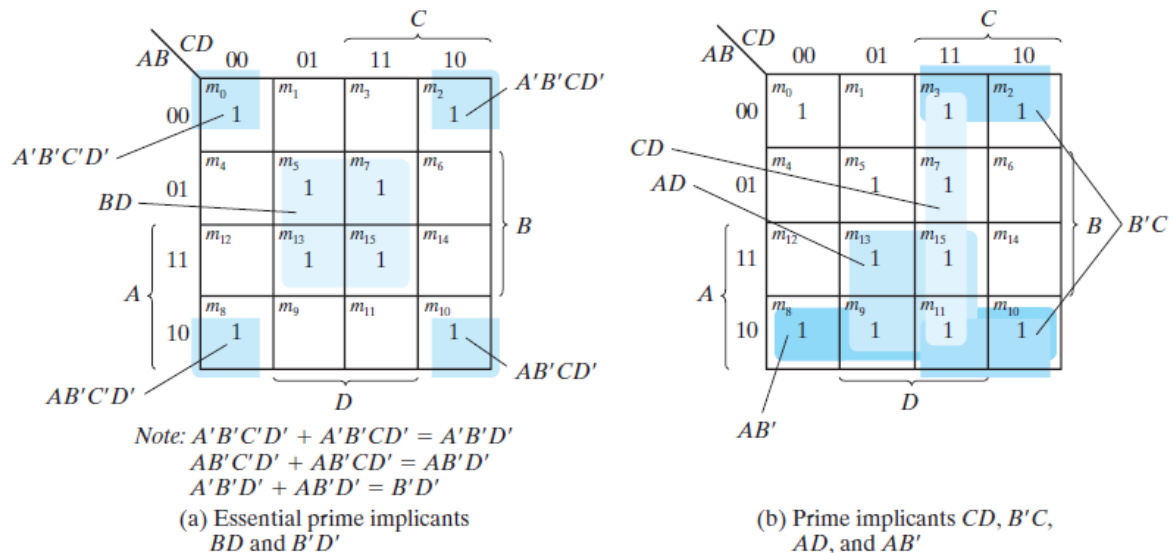


FIGURE1.24 Simplification using prime implicants

- 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.

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.
- Maps for more than four variables are difficult to use and will not be considered here.

1.9 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 1.10 :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. 1.25 . 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.

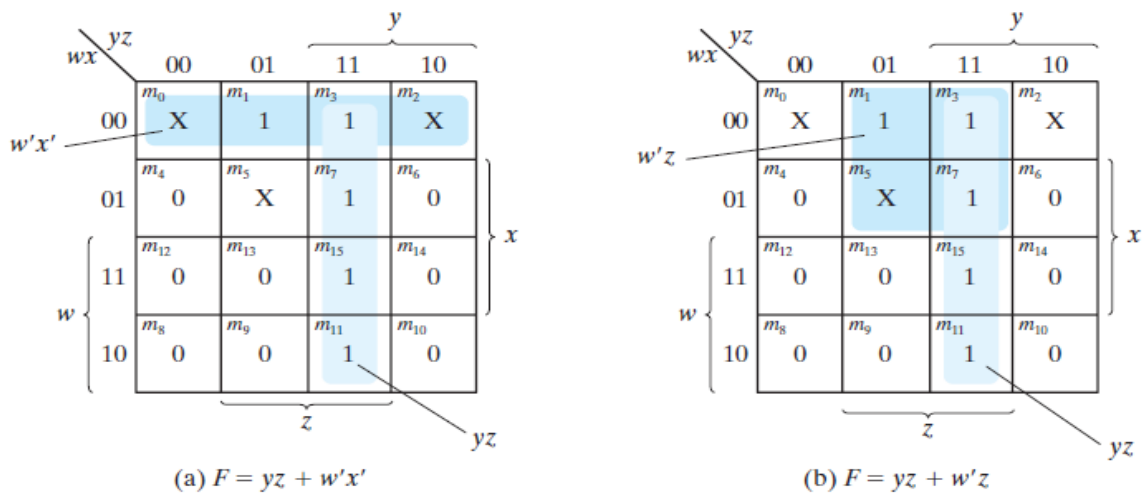


FIGURE 1.25 Example with don't-care conditions

- The remaining minterm, $m1$, can be combined with minterm $m3$ 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. 1.25(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. 1.25(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 all y 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 1.10 :

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

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

- 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. 1.25 . 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.

1.10 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.

NAND Circuits

- The NAND gate is said to be a **universal gate** because any logic circuit 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.
- **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. 1.27 .
- 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 (bubble) 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.

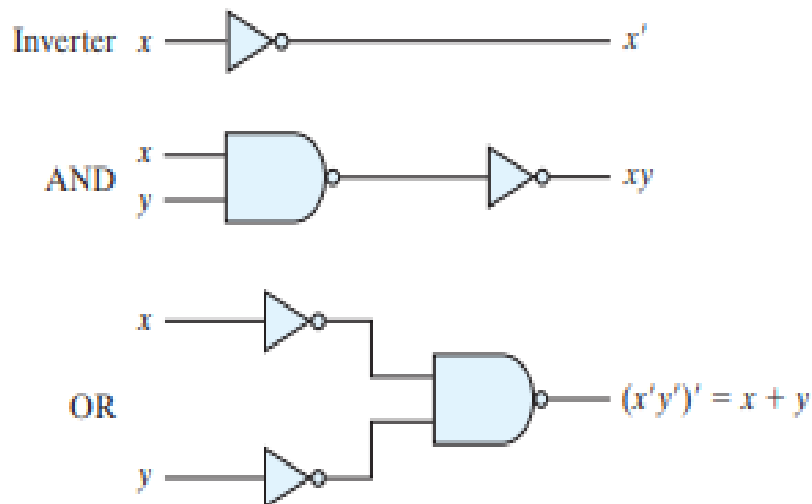


FIGURE 1.26 Logic operations with NAND gates

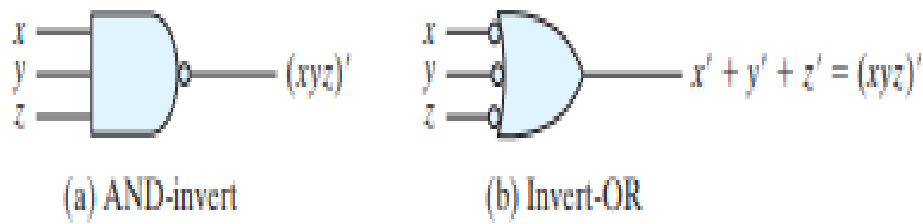


FIGURE 1.27 Two graphic symbols for a three-input NAND gate

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-products expression and its equivalent NAND implementation, consider the logic diagrams drawn in Fig. 1.28 .

- All three diagrams are equivalent and implement the function

$$F = AB + CD$$

- The function is implemented in Fig. 1.28(a) with AND and OR gates. In Fig. 1.28(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.

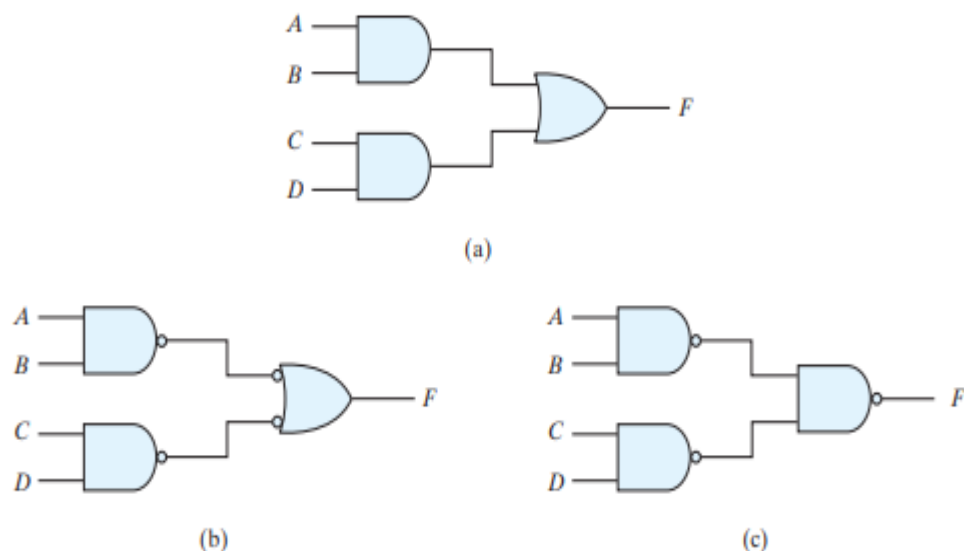


FIGURE 1.28 Three ways to implement $F = AB + CD$

- In Fig. 1.28 (c), the output NAND gate is redrawn with the AND-invert graphic symbol. In drawing NAND logic diagrams, the circuit shown in either Fig. 1.28(b) or (c) is acceptable. The one in Fig. 1.28(b) is in mixed notation and represents a more direct relationship to the Boolean expression it implements.
- The NAND implementation in Fig. 1.28 (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 1.11 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. 1.29 (a), from which the simplified function is obtained:

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

- The two-level NAND implementation is shown in Fig. 1.29 (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. 1.29 (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'.

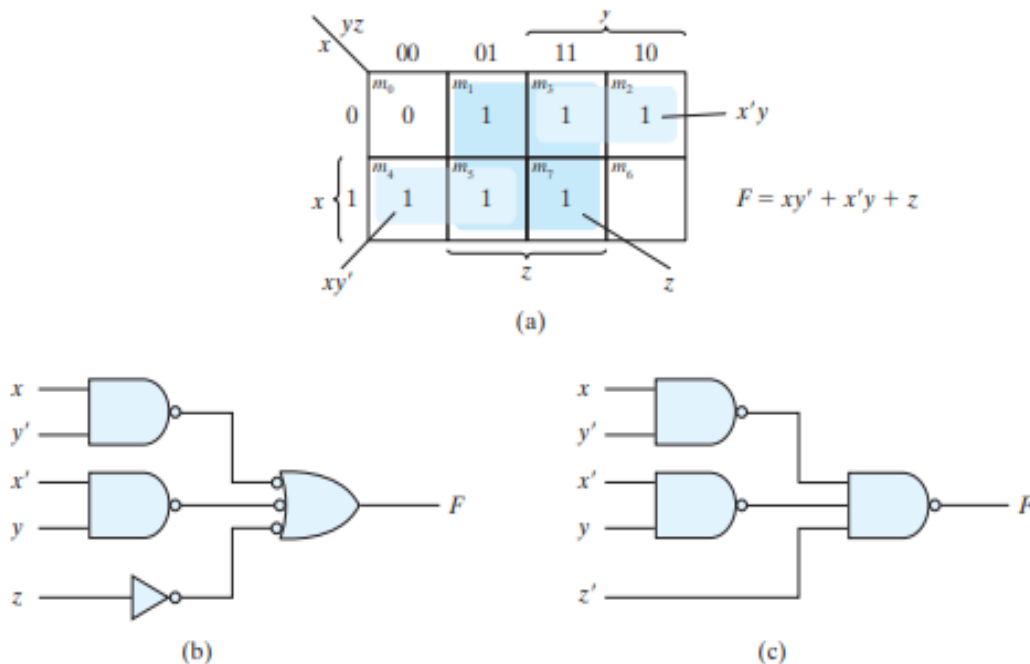


FIGURE 1.29 Solution to Example 1.9

- 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.
 4. 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.
 5. 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

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

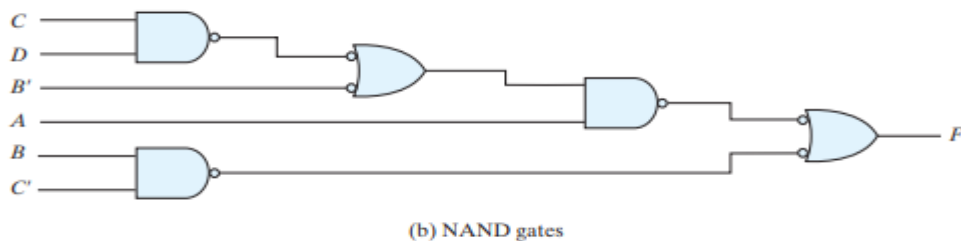
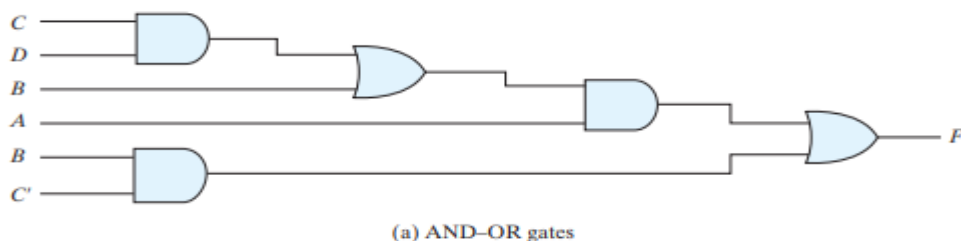


FIGURE 1.30 Implementing $F = A(CD + B) + BC'$

- 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. 1.30 (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. 1.30 (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 or complement the input literal.

As another example, consider the multilevel Boolean function

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

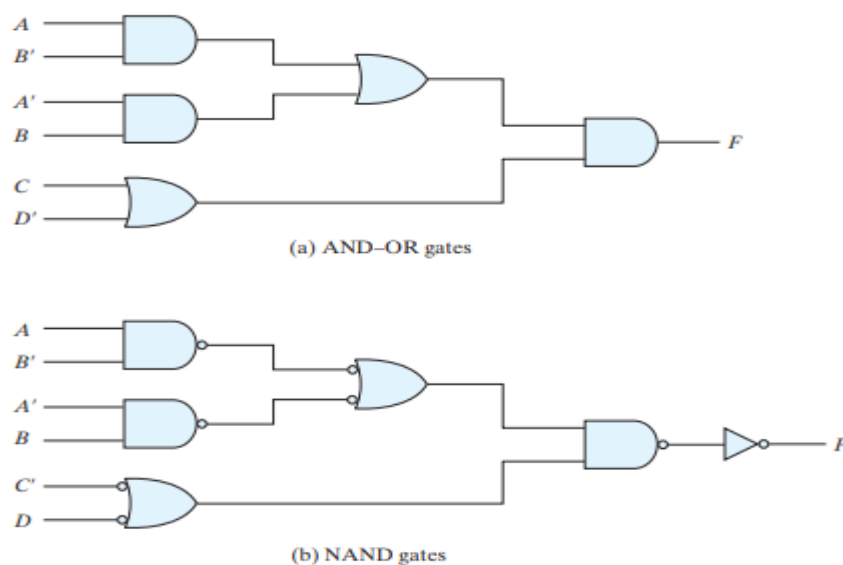


FIGURE 1.31 Implementing $F = (AB' + A'B)(C + D')$

- The AND–OR implementation of this function is shown in Fig. 1.31 (a) with three levels of gating. The conversion to NAND with mixed notation is presented in Fig. 1.31(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. 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. 1.32 .
- The complement operation is obtained from a oneinput 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. 1.33 .
- 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.

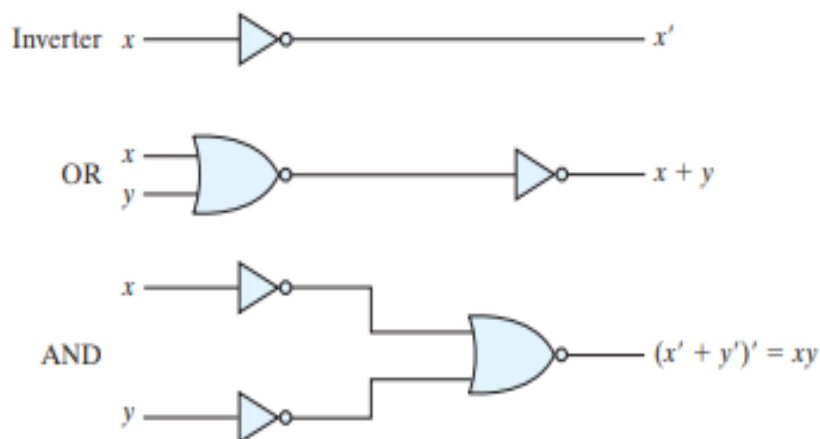


FIGURE 1.32 Logic operations with NOR gates

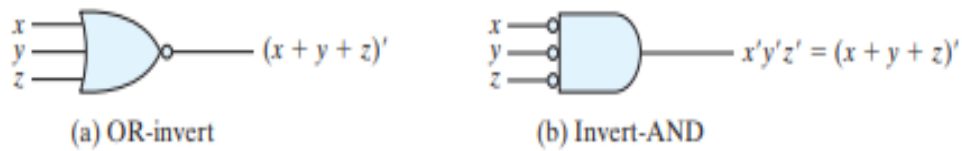


FIGURE 1.33 Two graphic symbols for the NOR gate

- 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. Figure 1.34, 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. 1.31 (a) into a NOR diagram is shown in Fig. 1.35 .
- The Boolean function for this circuit is

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

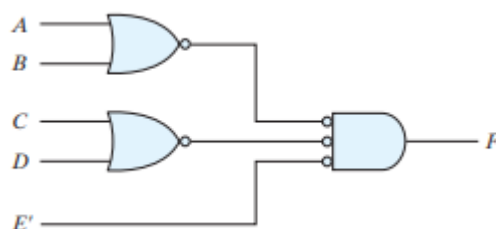


FIGURE 1.34 Implementing $F = (A + B)(C + D)E$

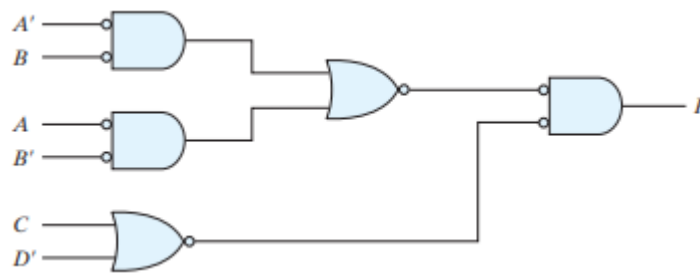


FIGURE 1.35 Implementing $F = (AB' + A,B)(C + D')$ with NOR gates

- 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.

1.11 HARDWARE DESCRIPTION LANGUAGE

- **Manual vs. Computer-Based Design:** Manual methods for designing logic circuits are practical only for small circuits. For larger, practical circuits, computer-based design tools are essential. These tools reduce the risk of errors and leverage the designer's creativity.
- **Hardware Description Language (HDL):** An HDL is a computer-based language used to describe the hardware of digital systems in textual form. It's specialized for representing hardware structures and logic circuit behavior. HDLs enable the representation of logic diagrams, truth tables, Boolean expressions, and complex system behaviors.
- **Documentation and Exchange:** HDLs serve as documentation languages, allowing both humans and computers to read, edit, store, and transmit digital system descriptions efficiently. They facilitate communication between designers.
- **HDL in Design Flow:** HDLs are used in various stages of integrated circuit design, including design entry, functional simulation, logic synthesis, timing verification, and fault simulation.
- **Design Entry:** Designers use HDLs to describe the functionality they want to implement in hardware. This can take various forms, including Boolean logic equations, truth tables, netlists of interconnected gates, or abstract behavioral models.
- **Functional Simulation:** HDLs are used with simulators to predict how a digital system will behave before it's physically built. Test benches are created to test the design's functionality and detect errors.

- **Logic Synthesis:** Logic synthesis translates the HDL description into a netlist, specifying the physical components and their interconnections. It's akin to compiling a high-level program, but it produces a database for circuit fabrication.
- **Timing Verification:** Timing verification checks signal paths to ensure they are not compromised by propagation delays, confirming that the circuit will operate at the specified speed.
- **Fault Simulation:** In VLSI design, fault simulation identifies differences between ideal and flawed circuits caused by manufacturing issues. It generates test patterns to ensure only fault-free devices are shipped to customers.
- **HDL Standards:** Two widely used HDLs supported by the IEEE are VHDL and Verilog. VHDL was mandated by the Department of Defense, while Verilog is more widely used due to its ease of learning and use.
- **Choice of Verilog:** The content of the book focuses on Verilog as it's considered easier to learn and use compared to VHDL. It emphasizes computer-aided modeling of digital systems using Verilog for modeling, verification, and synthesis.
- **Evolution of Verilog:** The Verilog HDL has evolved over the years and was initially approved as a standard in 1995, with revisions and enhancements approved in 2001 and 2005. The book covers features of Verilog that support HDL-based design methodology for integrated circuits.

Module Declaration

- **Keywords:** Verilog uses keywords, which are predefined lowercase identifiers that define language constructs. Examples of keywords include module, endmodule, input, output, wire, and, or, and not. Keywords are displayed in boldface in code examples.
- **Comments:** Comments in Verilog are indicated by double forward slashes (//) and extend to the end of the line. Comments do not affect the simulation.
- **Multiline Comments:** Multiline comments are enclosed between /* and */.
- **Case Sensitivity:** Verilog is case-sensitive, meaning uppercase and lowercase letters are distinct. For example, not is not the same as NOT.
- **Modules:** A Verilog model is composed of one or more modules. A module is declared using the module keyword and terminated with endmodule. Modules are the fundamental descriptive units in Verilog.

- **Module Name:** A module declaration includes a name, which is an identifier. Identifiers are composed of alphanumeric characters and underscores (_). They must start with an alphabetic character or underscore but cannot start with a number.
- **Port Lists:** After the module name, a list of ports is specified. Ports define the inputs and outputs of the module.
- **Combinational Logic:** Verilog can describe combinational logic using various methods, including schematic connections of gates, Boolean equations, or truth tables.
- **Example Circuit:** The text provides an example circuit described in Verilog HDL (HDL Example p1.1) to illustrate the language's usage. Shown in figure 1.36

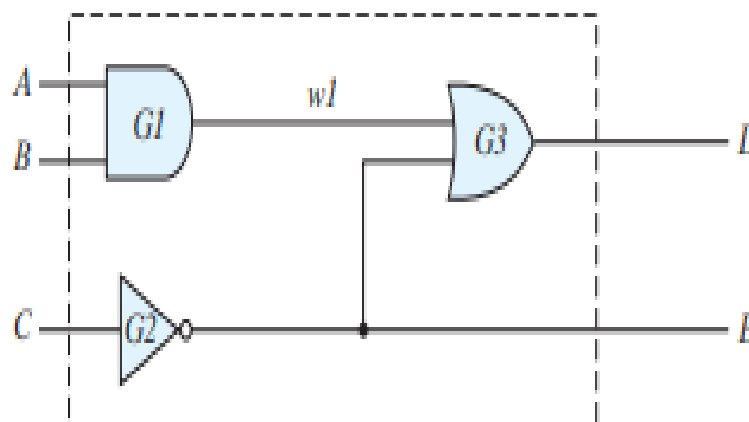


FIGURE 1.36 Circuit to demonstrate an HDL

HDL Example p1.1 (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
  
```

- **Port List:** The port list of a Verilog module defines the interface between the module and its environment, specifying inputs and outputs. It is enclosed in parentheses, separated by commas, and terminated with a semicolon.

- **Input and Output Ports:** The keywords input and output are used to specify which ports are inputs and which are outputs. Internal connections within the module are declared as wires.
- **Internal Connections:** Internal connections within the module are declared using the keyword wire.
- **Primitive Gates:** The structure of the circuit is specified using predefined primitive gates (e.g., and, or, not). Each gate is identified by a descriptive keyword, and gate instances are created with optional names, followed by the gate's output and inputs in parentheses.
- **Gate Instantiation:** Each gate instantiation consists of an optional name, the gate's output, and its inputs, separated by commas within parentheses. The output is always listed first.
- **Declaration vs. Instantiation:** Modules are declared to specify their input-output behavior, while predefined primitives (gates) are instantiated to populate the design. Primitives are not declared since their definition is predefined in the language.
- **Descriptive Model:** Verilog HDL is not a computational model like regular programming languages. The order of statements in the model does not imply a sequence of computations. It is a descriptive model that defines what primitives make up a circuit and how they are connected.
- **Behavior Specification:** The input-output behavior of the circuit is implicitly specified within the model because the behavior of each logic gate is predefined. This allows for simulating the represented circuit using the HDL-based model.

Table 1.3

Output of Gates after Delay

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

Gate Delays

- In Verilog, propagation delays in physical circuits are specified using time units and the '#' symbol. The time units are dimensionless, and the timescale is established with a compiler directive. For example, "timescale 1ns/100ps" sets the time unit to 1 nanosecond (ns) and the precision to 0.1 picoseconds (ps).
- Let's consider an example circuit described in HDL (Hardware Description Language) with specified gate delays. The circuit has AND, OR, and NOT gates with delays of 30 ns, 20 ns, and 10 ns, respectively.
- When simulating this circuit and transitioning its inputs from A, B, C = 0 to A, B, C = 1, the outputs change as follows (using the specified gate delays and assuming the default time unit is 1 ns):
- The output of the inverter at E changes from 1 to 0 after a 10-ns delay due to the NOT gate's delay.
- The output of the AND gate at w1 changes from 0 to 1 after a 30-ns delay due to the AND gate's delay.
- The output of the OR gate at D changes from 1 to 0 at t = 30 ns due to the OR gate's delay and then changes back to 1 at t = 50 ns. This change in the OR gate's output is a result of a change in its inputs 20 ns earlier.
- This behavior reveals that the gate delays in the circuit introduce a negative spike in the output waveform. Specifically, for output D, there is a 20-ns period during which the output is 0 before it returns to its final value of 1.
- This spike is a consequence of the delay in the gates and is crucial to consider when designing and analyzing digital circuits, especially in timing-critical applications.

HDL Example p1. 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.
- HDL Example p1.3 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 "initial" keyword is employed in the test bench to define the initial conditions. In this case, the initial statements indicate that A, B, and C are initially set to "1!b0," representing one binary digit with a value of 0. After 100 ns, these inputs transition to A, B, C = 1.

HDL Example p1.3 (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
```

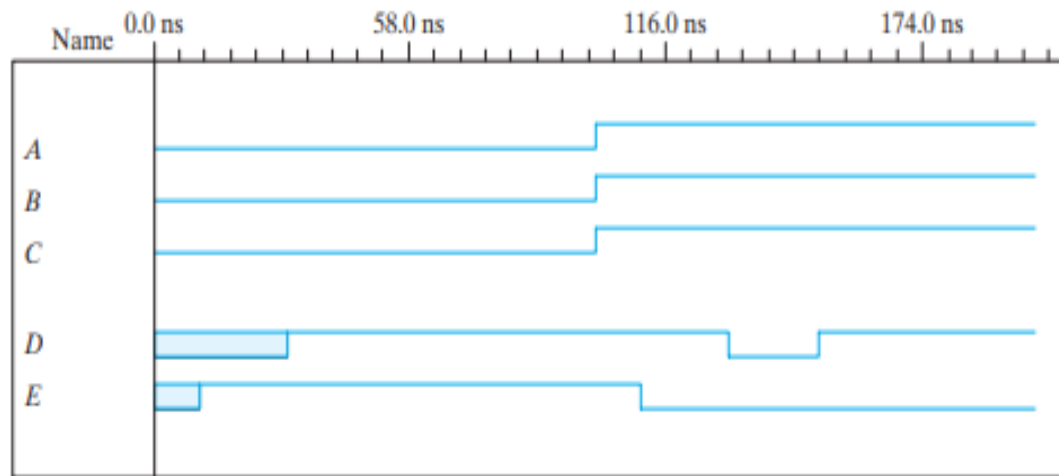


FIGURE 1.37 Simulation output of HDL Example 1.3

- The simulation runs for a total of 200 ns, and a second "initial" statement uses the "\$finish" system task to specify the termination of the simulation. Delay values, such as "#100," can be used to schedule statements to execute after a specified time delay.
- The resulting timing diagram (Figure 1.37) shows the waveforms over the 200 ns interval. Initially, outputs E and D are unknown for the first 10 ns and 30 ns, respectively, as denoted by shading. Output E transitions from 1 to 0 at 110 ns, and output D transitions from 1 to 0 at 130 ns, returning to 1 at 150 ns, as predicted in Table 1.3.
- Overall, the simulation process involves abstractly modeling input signals, specifying initial conditions, and executing the simulation to verify the behavior of the HDL model over a defined time interval.

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. 1.36 with a Boolean expression, we use the statement

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

- HDL Example p1.4 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 p1.4 (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 .

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. 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 p1.5 defines a UDP with a truth table. It proceeds according to the following general rules:
 - a. It is declared with the keyword `primitive` , followed by a name and port list.
 - b. There can be only one output, and it must be listed first in the port list and declared with keyword `output` .
 - c. 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.
 - d. The truth table is enclosed within the keywords `table` and `endtable`.
 - e. 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 (;).
 - f. The declaration of a UDP ends with the keyword `endprimitive`.

HDL Example p1.5 (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 5 f (A, B, C) 5 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
```

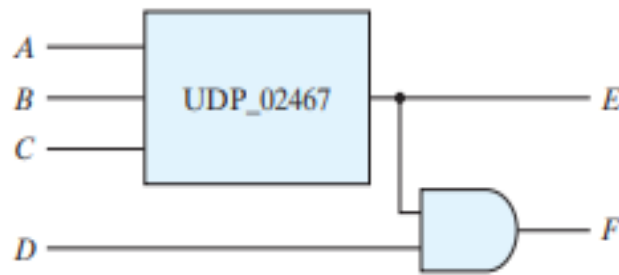


FIGURE 1.38 Schematic for Circuit with_UDP_02467

- 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 1.38 .
- 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.