

## MODULE-1

### Introduction to Digital Design

#### Syllabus

**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

Text book 1: **M. Morris Mano** & Michael D. Ciletti, **Digital Design With an Introduction to Verilog Design**, 5e, Pearson Education.

#### BINARY LOGIC

##### Definition of Binary Logic

- **Binary logic** consists of **binary variables** and a **set of logical operations**. The variables are designated by letters of the alphabet, such as A, B, C, x, y, z, etc., with each variable having **two 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**.

Example,  $x \cdot y = z$  or  $xy = z$  is read “x AND y is equal to z.”

**If  $z = 1$  if and only if  $x = 1$  and  $y = 1$ ; otherwise  $z = 0$ .** The result of the operation  $x \cdot y$  is z.

x, y, and z are binary variables and can be equal either to 1 or 0

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 (or by an **overbar**).

For example,  $x' = z$  (or  $\overline{x} = z$ ) is read “not x is equal to z.”

**If  $x = 1$ , then  $z = 0$ , and 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.

#### Truth table

- 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.

*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		

### Gates:

“A digital circuit having one or more input signals but only one output signal is called a gate”.

### Logic Gates:

“The gates which perform logical operation is called logic gates”.

- It take binary input and gives binary outputs.
- The output of the logic gates can be understood using truth table, which contains inputs, outputs of logic circuits.

Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.

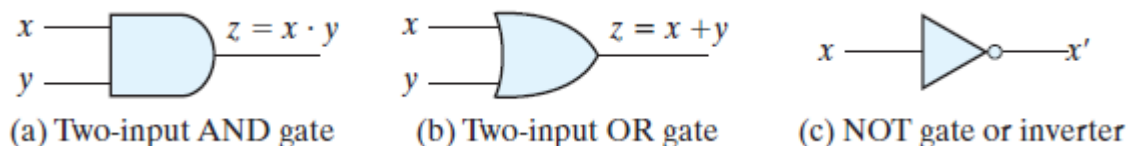


Fig: Symbols for digital logic circuits

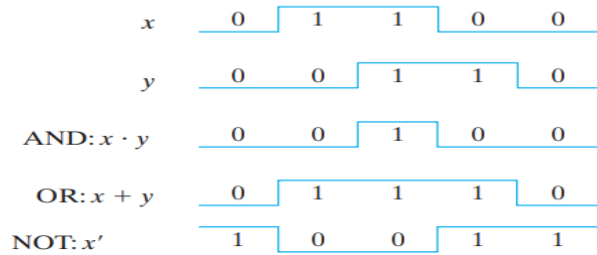
The **timing diagrams** illustrate the idealized response of each gate to the four input signal combinations. The **horizontal axis** of the timing diagram represents the **time**, and the **vertical axis** shows the **signal** as it changes between the two possible voltage levels.

The low level represents logic 0, the high level logic 1.

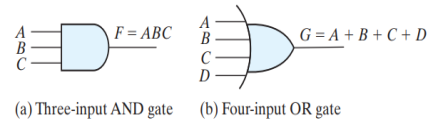
The **AND gate** responds with a logic 1 output signal when both input signals are logic 1.

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. The output signal inverts the logic sense of the input signal.



Input-Output Signals for gates



Gates with multiple inputs

Timing diagram

## 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 lists out six theorems of Boolean algebra and four of its postulates. 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.

**Table 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)..

**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.

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

Statement      Justification

$$\begin{aligned}
 x + xy &= x \cdot 1 + xy && \text{postulate 2(b)} \\
 &= x(1 + y) && 4(a) \\
 &= x(y + 1) && 3(a) \\
 &= x \cdot 1 && 2(a) \\
 &= x && 2(b)
 \end{aligned}$$

**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 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

**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.

Consider the Boolean function  $F_1 = x + y'z$ . The function  $F_1$  is equal to 1 if  $x$  is equal to 1 or if both  $y'$  and  $z$  are equal to 1.  $F_1$  is equal to 0 otherwise. Therefore,  $F_1 = 1$  if  $x = 1$  or if  $y = 0$  and  $z = 1$ .

- A Boolean function expresses the logical relationship between binary variables and is evaluated by determining the binary value of the expression for all possible values of the variables.
- A Boolean function can be represented in a truth table. The number of rows in the truth table is  $2^n$ , where  $n$  is the number of variables in the function.
- The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through  $2^n - 1$ .
- Table 1 shown below represents 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.

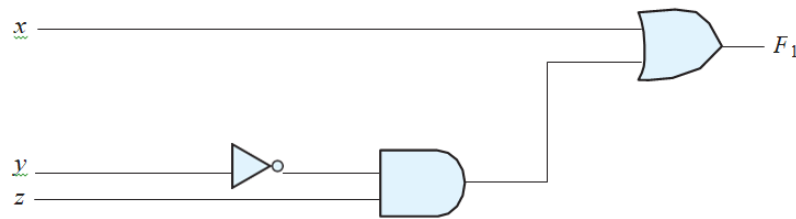
*Truth Tables for  $F_1$  and  $F_2$*

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

Table 1

- 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 for  $F_1$  is shown in Fig. 2.1. There is an inverter for input  $y$  to generate its complement. There is an AND gate

for the term  $y'z$  and an OR gate that combines  $x$  with  $y'z$ .



**FIGURE 2.1**

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 representation shows the relationship between the output of the circuit and its inputs.
- Rather than listing each combination of inputs and outputs, it indicates how to compute the logic value of each output from the logic values of the inputs.
- There is only one way that a Boolean function can be represented in a truth table but a function in algebraic form can be expressed in a variety of ways, all of which have equivalent logic.
- The particular expression used to represent the function will dictate the interconnection of gates in the logic-circuit diagram.
- By manipulating a Boolean expression according to the rules of Boolean algebra, it is sometimes possible to obtain a simpler expression for the same function and thus reduce the number of gates in the circuit and the number of inputs to the gate.
- Designers are motivated to reduce the complexity and number of gates because their effort can significantly reduce the cost of a circuit.

Consider, for example, the following Boolean function:  $F_2 = x'y'z + x'yz + xy'$

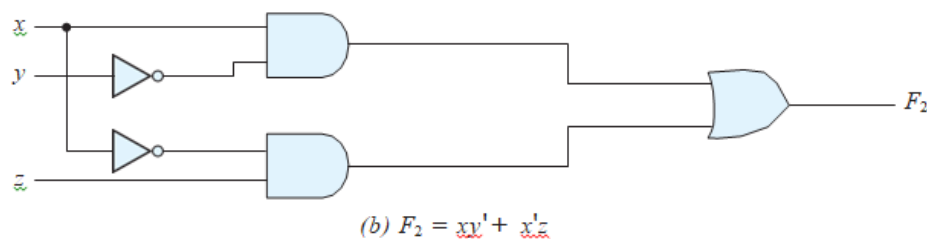
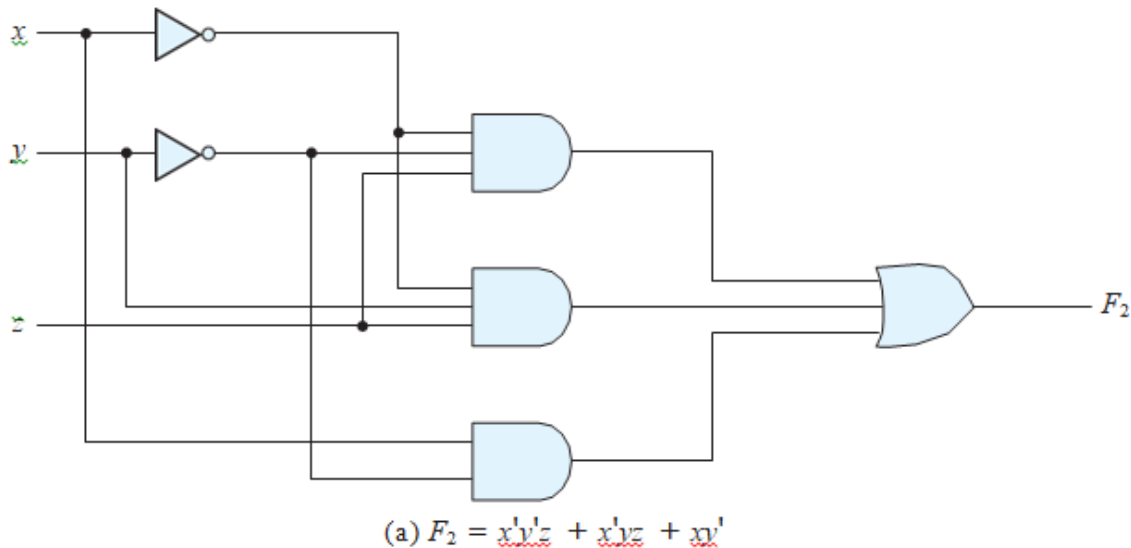
- A schematic of an implementation of this function with logic gates is shown in Fig. 2.2(a).
- Input variables  $x$  and  $y$  are complemented with inverters to obtain  $x'$  and  $y'$ . The three terms in the expression are implemented with three AND gates. The OR gate forms the logical OR of the three terms. The truth table for  $F_2$  is listed in Table 2.2.

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

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

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

- By means of a truth table, it is possible to verify that the two expressions are equivalent. The simplified expression is equal to 1 when  $xz = 01$  or when  $xy = 10$ .
- This produces the same four 1's in the truth table. Since both expressions produce the same truth table, they are equivalent.
- Therefore, the two circuits have the same outputs for all possible binary combinations of inputs of the three variables.
- Each circuit implements the same identical function, but the one with fewer gates and fewer inputs to gates is preferable because it requires fewer wires and components.
- In general, there are many equivalent representations of a logic function. Finding the most economic representation of the logic is an important design task.











**FIGURE 2.2**  
Implementation of Boolean function  $F_2$  with gates

### **DIGITAL LOGIC GATES**

- Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these type of gates.



- Factors to be weighed in considering the construction of other types of logic gates are (1) the feasibility and economy of producing the gate with physical components, (2) the possibility of extending the gate to more than two inputs, (3) the basic properties of the binary operator, such as commutativity and associativity, and (4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.
- The graphic symbols and truth tables of the eight gates are shown in Fig. 2.5. Each gate has one or two binary input variables, designated by  $x$  and  $y$ , and one binary output variable, designated by  $F$ .
- The 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 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 equivalence, or exclusive-NOR, gate is the complement of the exclusive-OR, as indicated by the small circle on the output side of the graphic symbol.

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 2.5**  
Digital logic gates

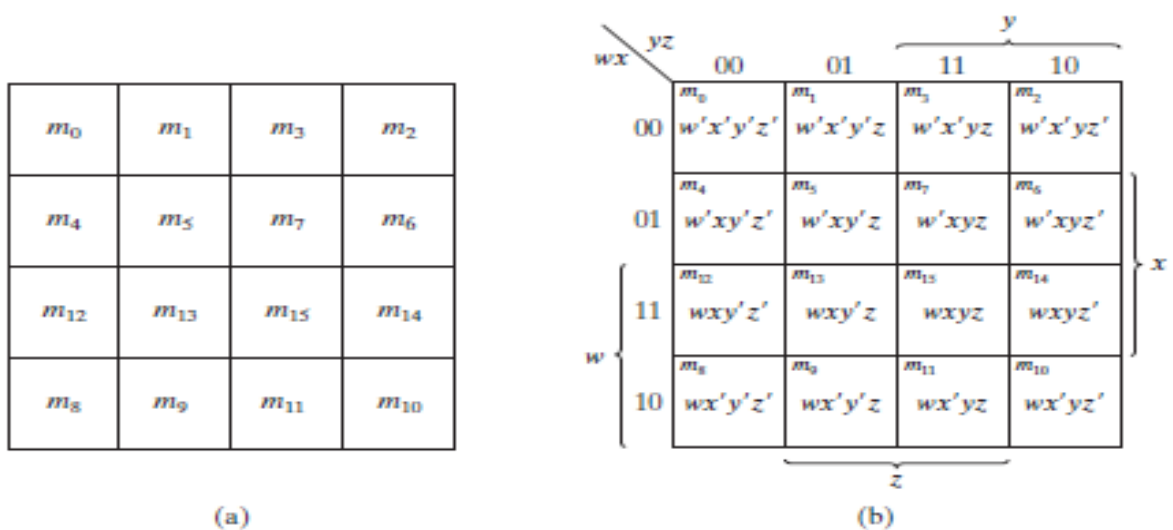
## THE MAP METHOD

- **The map method 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.
- Map represents 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.**
- 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.

#### FOUR-VARIABLE K-MAP

- The map for Boolean functions of four binary variables ( $w, x, y, z$ ) is shown in Fig. 3.8. In Fig. 3.8(a) are listed the 16 minterms and the squares assigned to each.
- In Fig. 3.8(b), the map is redrawn to show the relationship between the squares and the four variables. The rows and columns are numbered in a Gray code sequence, with only one digit changing value between two adjacent rows or columns.
- The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number. For example, the numbers of the third row (11) and the second column (01), when concatenated, give the binary number 1101, the binary equivalent of decimal 13. Thus, the square in the third row and second column represents minterm  $m_{13}$ .
- 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 No other combination of squares can simplify the function.



**FIGURE 3.8**  
Four-variable map

### 1)Simplify the Boolean function

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

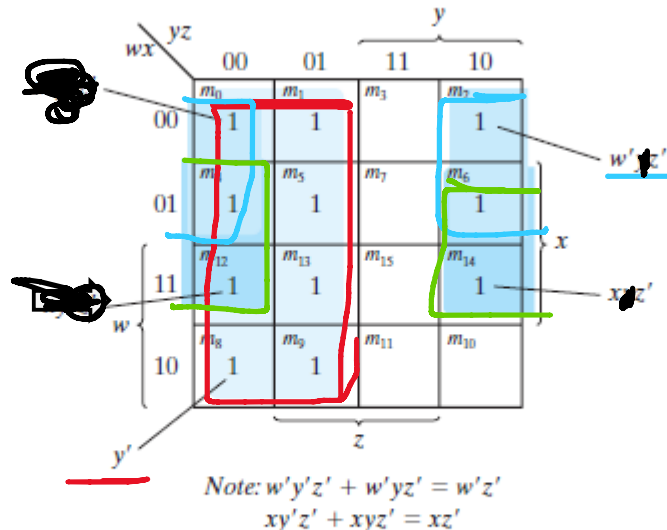
Since the function has four variables, a four-variable map must be used. The minterms listed in the sum are marked by 1's in the map of Fig. 3.9.

**Eight adjacent squares marked with 1's can be combined to form the one literal term  $y'$ .** The remaining three 1's on the right cannot be combined to give a simplified term; they must be combined as two or four adjacent squares.

**The larger the number of squares combined, the smaller is the number of literals in the term.** In this example, the top two 1's on the right are combined with the top two 1's on the left to give the term  $w'z'$ .

Note that it is permissible to use the same square more than once. We are 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 3.9**

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

## 2) Simplify the Boolean function

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

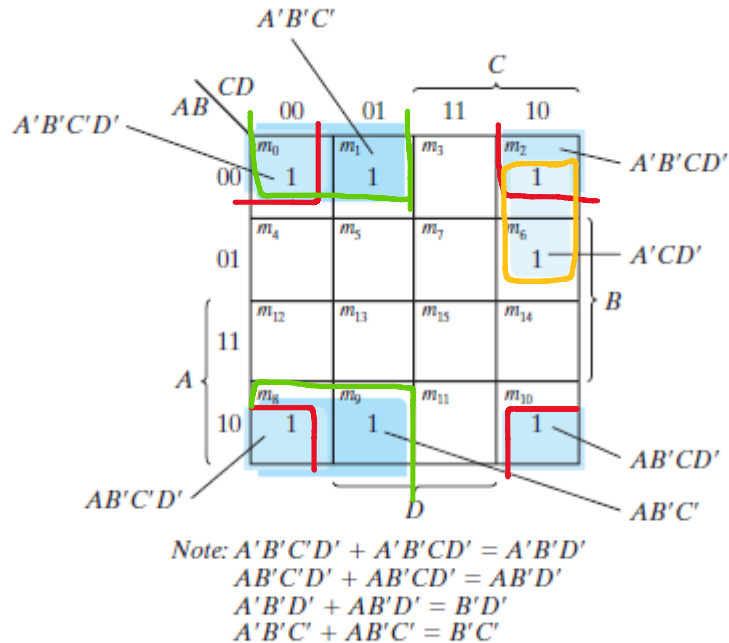


FIGURE 3.10

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

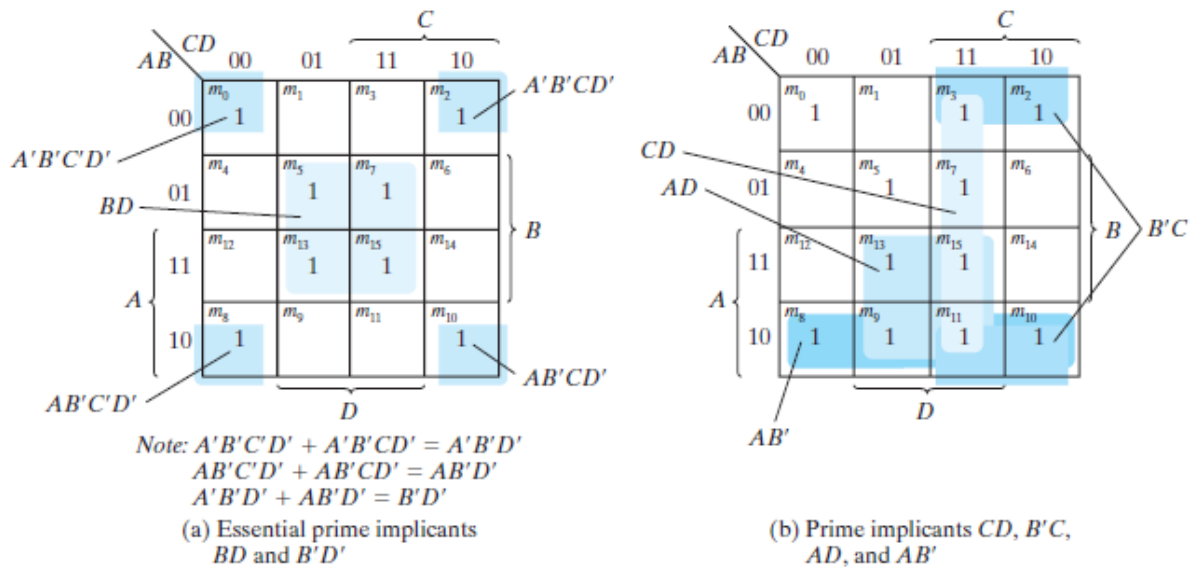
The simplified function is

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

## 3) Simplify following four-variable Boolean function:

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

The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants



**FIGURE 3.11**  
Simplification using prime implicants

that cover minterms  $m_3$ ,  $m_9$ , and  $m_{11}$ . There are four possible ways that the function can be expressed with four product terms of two literals each:

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

### **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 some applications the function is not specified for certain combinations of the variables.
- 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 min- terms 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.

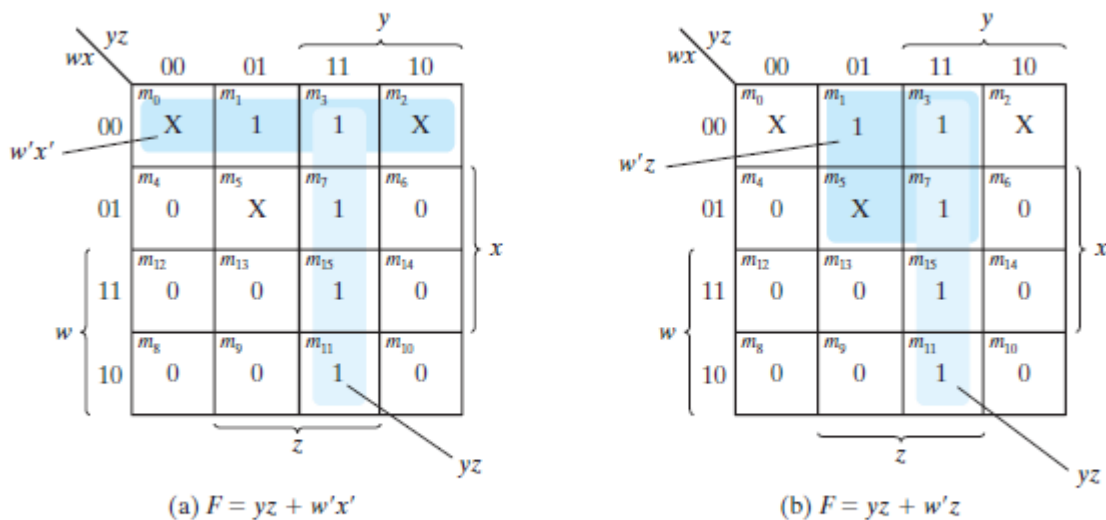
### **Simplify the Boolean function**

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

**which has the don't-care conditions**

$$\mathbf{d(w, x, y, z) = (0, 2, 5)}$$

The minterms of F are the variable combinations that make the function equal to 1. The minterms of d are the don't-care minterms that may be assigned either 0 or 1. The map simplification is shown in Fig. 3.15. The minterms of F are marked by 1's, those of d are marked by X's, and the remaining squares are filled with 0's. To get the simplified expression in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified. The term  $yz$  covers the four minterms in the third column. The remaining minterm,  $m_1$ , can be combined



**FIGURE 3.15**  
Example with don't-care conditions

with minterm  $m_3$  to give the three-literal term  $w'x'z$ . However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In Fig. 3.15(a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

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

In Fig. 3.15(b), don't-care minterm 5 is included with the 1's, and the simplified function is now

$$F = yz + w'z$$

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

## 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.

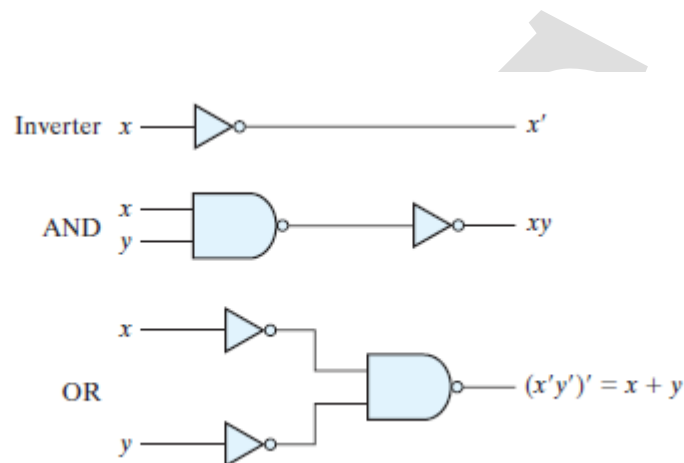
### NAND Circuits

- 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. This is indeed shown in Fig. 3.16.
- The complement operation is obtained from a one-input NAND gate that behaves exactly like an inverter. The AND operation requires two NAND gates. The first produces the NAND operation and the second inverts

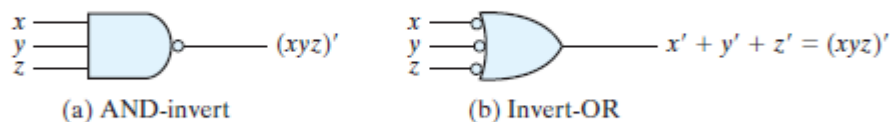


the logical sense of the signal. The OR operation is achieved through a NAND gate with additional inverters in each input.

- A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic.
- The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit manipulation techniques that change AND-OR diagrams to NAND diagrams.
- Two equivalent graphic symbols for the NAND gate are shown in Fig. 3.17. The AND-invert symbol has been defined previously and consists



**FIGURE 3.16**  
Logic operations with NAND gates



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

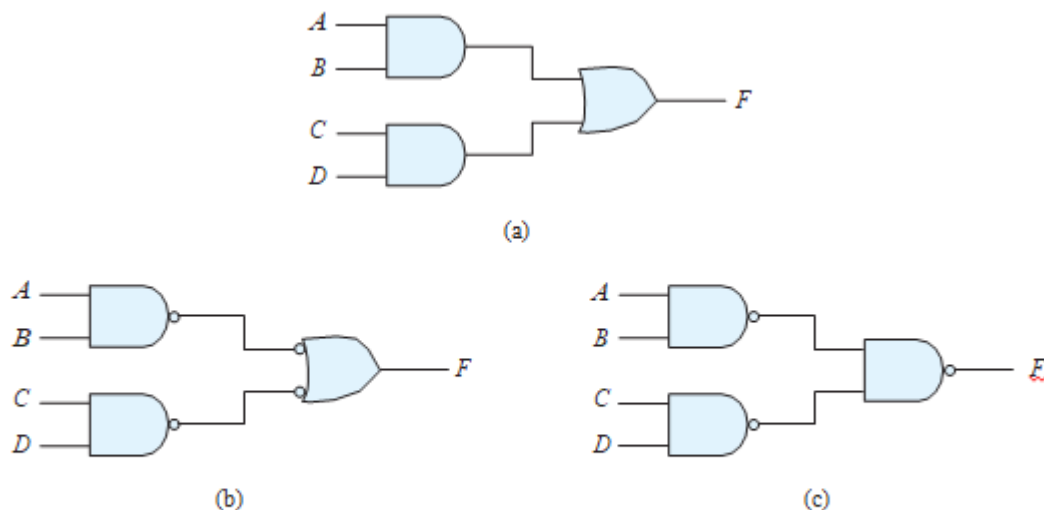
of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble.

- 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.

- 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. 3.18. All three diagrams are equivalent and implement the function

$$F = AB + CD$$

- The function is implemented in Fig. 3.18(a) with AND and OR gates. In Fig. 3.18(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 3.18**  
Three ways to implement  $F = AB + CD$

- In Fig. 3.18(c), the output NAND gate is redrawn with the AND-invert graphic symbol. In drawing NAND logic diagrams, the circuit shown in either Fig. 3.18(b) or (c) is acceptable.
- The one in Fig. 3.18(b) is in mixed notation and represents a more direct relationship to the Boolean expression it implements. The NAND implementation in Fig. 3.18(c) can be verified algebraically.
- The function it implements can easily be converted to sum-of-products form by DeMorgan's theorem:

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

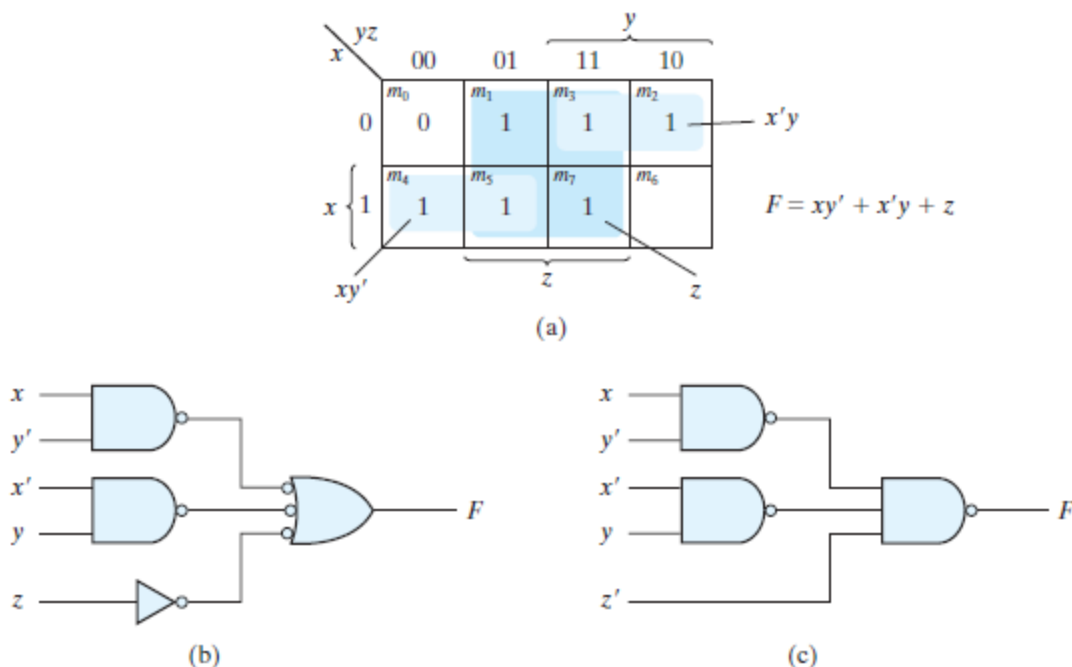
**Implement the following Boolean function with NAND gates:**

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

- The first step is to simplify the function into sum-of-products form. This is done by means of the map of Fig. 3.19(a), from which the simplified function is obtained:

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

- The two-level NAND implementation is shown in Fig. 3.19(b) in mixed notation. Note that input z must have a one-input NAND gate (an inverter) to compensate for the bubble in the second-level gate. An alternative way of drawing the logic diagram is given in Fig. 3.19(c).
- Here, all the NAND gates are drawn with the same graphic symbol. The inverter with input z has been removed, but the input variable is complemented and denoted by z'



**FIGURE 3.19**  
Solution to Example 3.9

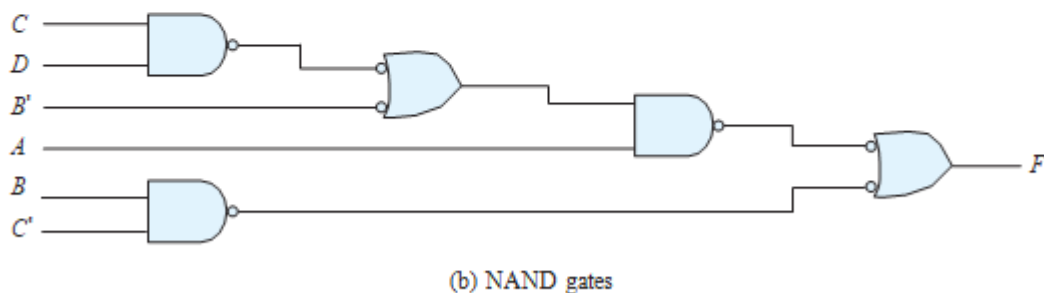
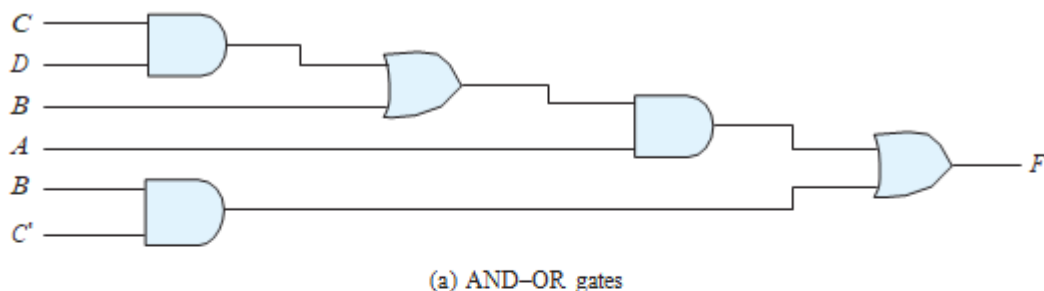
- The procedure described in the previous example indicates that a Boolean function can be implemented with two levels of NAND gates. The procedure for obtaining the logic diagram from a Boolean function is as follows:
1. Simplify the function and express it in sum-of-products form.
  2. Draw a NAND gate for each product term of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates.

3. Draw a single gate using the AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from outputs of first-level gates.
4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second-level NAND gate.

### Multilevel NAND Circuits

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

**Consider, for example, the Boolean function  $F = A(CD + B) + BC'$**



**FIGURE 3.20**  
Implementing  $F = A(CD + B) + BC'$

- The AND-OR implementation is shown in Fig. 3.20(a). There are four levels of gating in the circuit. The first level has two AND gates. The second level has an OR gate followed by an AND gate in the third level and an OR gate in the fourth level.
- A logic diagram with a pattern of alternating levels of AND and OR gates can easily be converted into a NAND circuit with the use of mixed notation, shown in Fig. 3.20(b).

- The procedure is to change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol. The NAND circuit performs the same logic as the AND-OR diagram as long as there are two bubbles along the same line.
- The bubble associated with input B causes an extra complementation, which must be compensated for by changing the input literal to B'.

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

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

As another example, consider the multilevel Boolean function

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

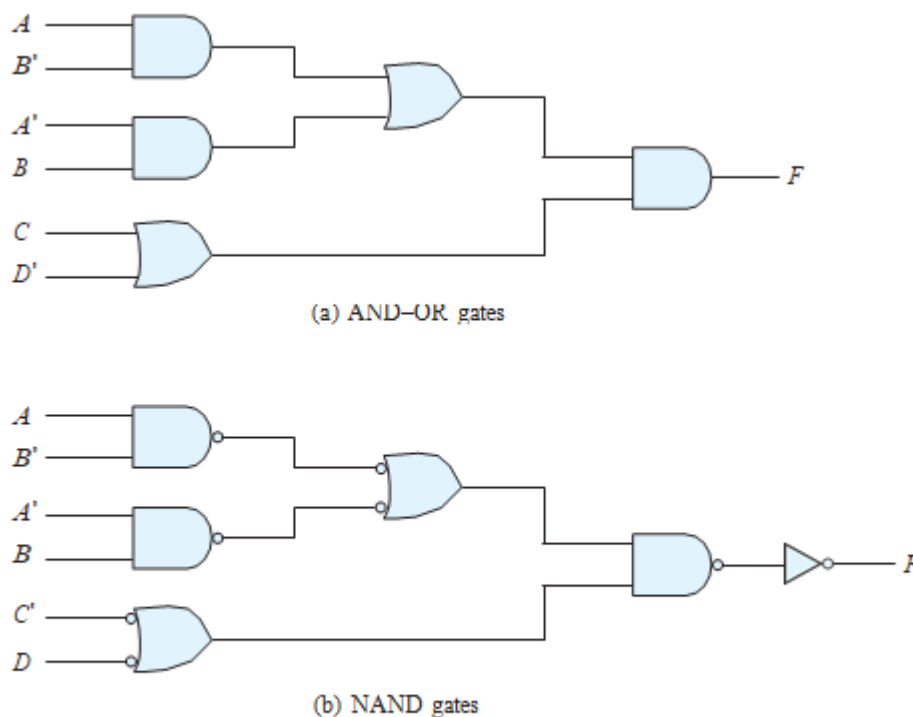


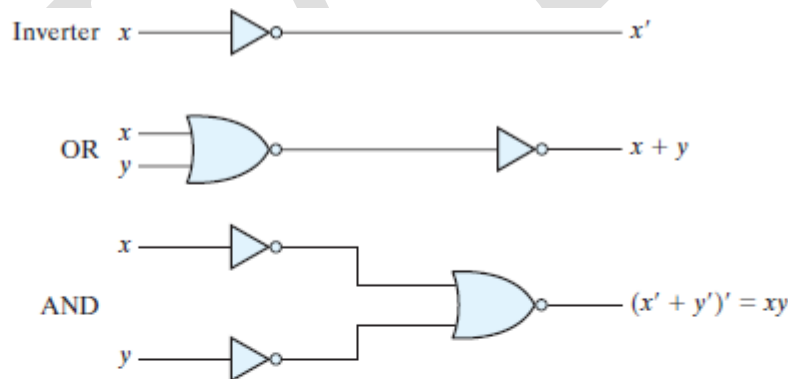
FIGURE 3.21  
Implementing  $F = (AB' + A'B)(C + D')$

- The AND-OR implementation of this function is shown in Fig. 3.21(a) with three levels of gating. The conversion to NAND with mixed notation is presented in Fig. 3.21(b) of the diagram.

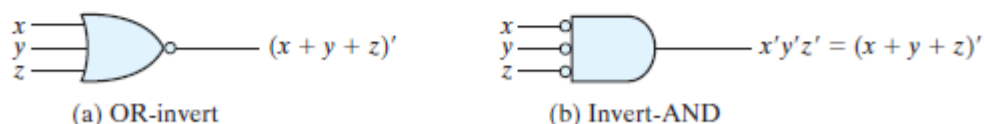
- The two additional bubbles associated with inputs C and D' cause these two literals to be complemented to C' and D.
- The bubble in the output NAND gate complements the output value, so we need to insert an inverter gate at the output in order to complement the signal again and get the original value back.

### NOR Implementation

- The NOR operation is the dual of the NAND operation. Therefore, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic.
- The NOR gate is another universal gate that can be used to implement any Boolean function. The implementation of the complement, OR, and AND operations with NOR gates is shown in Fig. 3.22.
- The complement operation is obtained from a one- input NOR gate that behaves exactly like an inverter. The OR operation requires two NOR gates, and the AND operation is obtained with a NOR gate that has inverters in each input.
- The two graphic symbols for the mixed notation are shown in Fig. 3.23. The 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 3.22**  
Logic operations with NOR gates



**FIGURE 3.23**  
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 3.24 shows the NOR implementation of a function expressed as a product of sums:  $F = (A + B)(C + D)E$

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

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

The transformation of the AND-OR diagram of Fig. 3.21(a) into a NOR diagram is shown in Fig. 3.25. The Boolean function for this circuit is

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

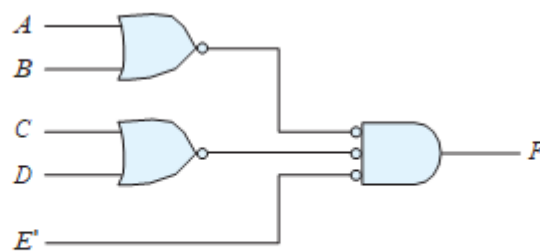


FIGURE 3.24  
Implementing  $F = (A + B)(C + D)E$

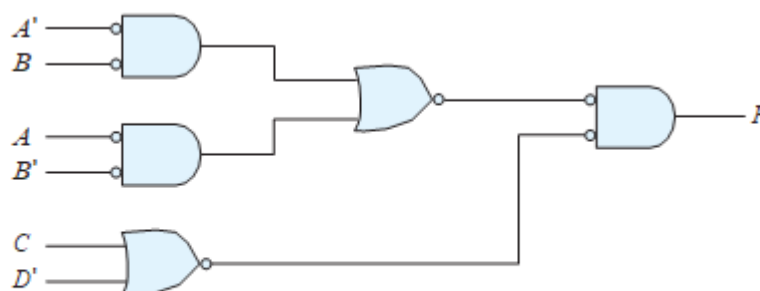


FIGURE 3.25  
Implementing  $F = (AB' + A'B)(C + D')$  with NOR gates

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.

SLIDE