MODULE-1

INTRODUCTION TO DIGITAL DESIGN

1.9 BINARY LOGIC

Binary logic deals with variables that take on two discrete values and with operations that assume **logical meaning**. The two values the variables assume may be called by different names (***true* and *false, yes* and *no,*** etc.), but for our purpose, it is convenient to think in terms of bits and assign the **values 1 and 0.**

Definition of Binary Logic

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

1. **AND:** This operation is represented by a **dot** or by the absence of an operator. For example, x.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.

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** 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,

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.**

Table 1.8
Truth Tables of Logical Operations

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

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**.
- In practice, each voltage level has an acceptable range, as shown in **Fig. 1.3.** The input terminals of digital circuits accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within the specified range. The intermediate region between the allowed regions is crossed only during a **state transition**.

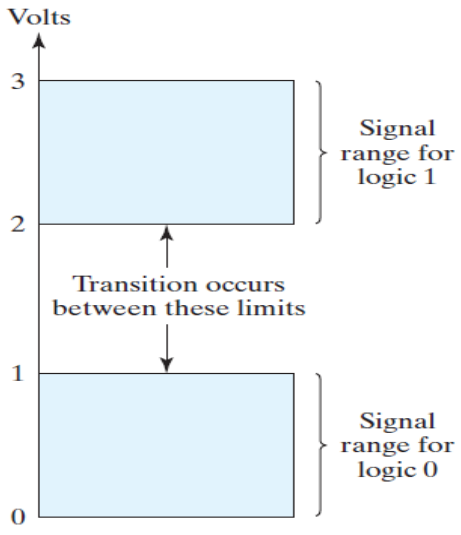
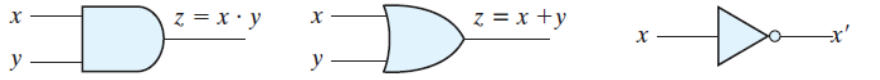


FIGURE 1.3
Signal levels for binary logic values



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

FIGURE 1.4
Symbols for digital logic circuits

The **graphic symbols** used to designate the three types of gates are shown in **Fig. 1.4** The gates are blocks of hardware that produce the equivalent of logic-1 or logic-0 output signals if input logic requirements are satisfied.

The input signals x and y in the AND and OR gates may exist in one of four possible states: 00, 10, 11, or 01.

These input signals are shown in **Fig. 1.5** together with the corresponding output signal for each gate. The **timing diagrams** illustrate the idealized response of each gate to the four input signal combinations. The horizontal axis of the timing diagram represents the time, and the vertical axis shows the signal as it changes between the two possible voltage levels.

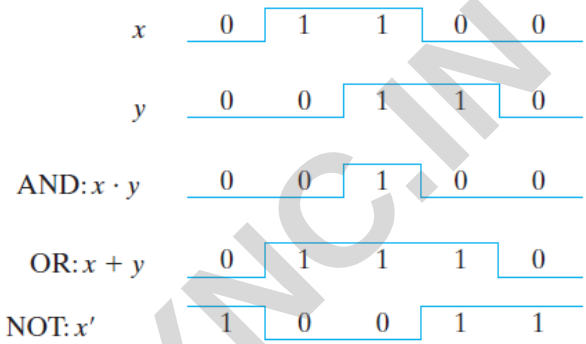
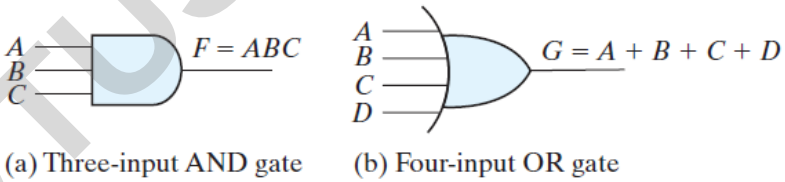


FIGURE 1.5
Input–output signals for gates



(a) Three-input AND gate (b) Four-input OR gate

FIGURE 1.6
Gates with multiple inputs

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

2.4 BASIC THEOREMS AND PROPERTIES OF BOOLEAN ALGEBRA

Duality

One part may be obtained from the other if the binary operators and the identity elements are interchanged. This important property of Boolean algebra is called the ***duality principle*** and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.

In a two-valued Boolean algebra, the identity elements and the elements of the set *B* are the same: 1 and 0. The duality principle has many applications. If the *dual* of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Basic Theorems

Table 2.1 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.

Table 2.1
Postulates and Theorems of Boolean Algebra

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

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 $(x')'$.

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**, 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.

BOOLEAN FUNCTIONS

Boolean algebra is an algebra that deals with binary variables and logic operations. For a given value of the binary variables, the function can be equal to either 1 or 0. As an example, consider the Boolean function

$$F_1 = x + y'z$$

The function $F1$ 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 2.2 shows the truth table for the function F1. There are eight possible binary combinations for assigning bits to the three variables x, y, and z. The column labelled F1 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 y z = 0 1 and is equal to 0 otherwise.

Table 2.2
Truth Tables for F_1 and F_2

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

The logic-circuit diagram (also called a schematic) for F1 is shown in **Fig. 2.1**

There is an inverter for input y to generate its complement. There is an AND gate for the term y'z and an OR gate that combines x with y'z. In logic-circuit diagrams, the variables of the function are taken as the inputs of the circuit and the binary variable F1 is taken as the output of the circuit.

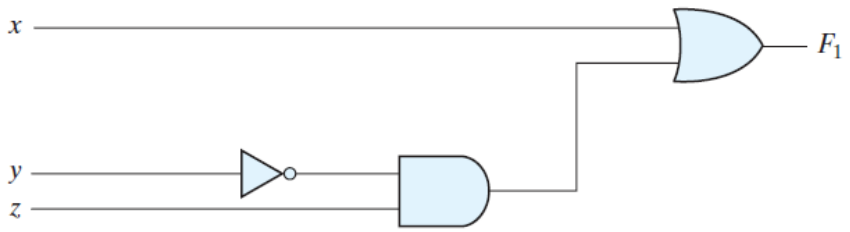


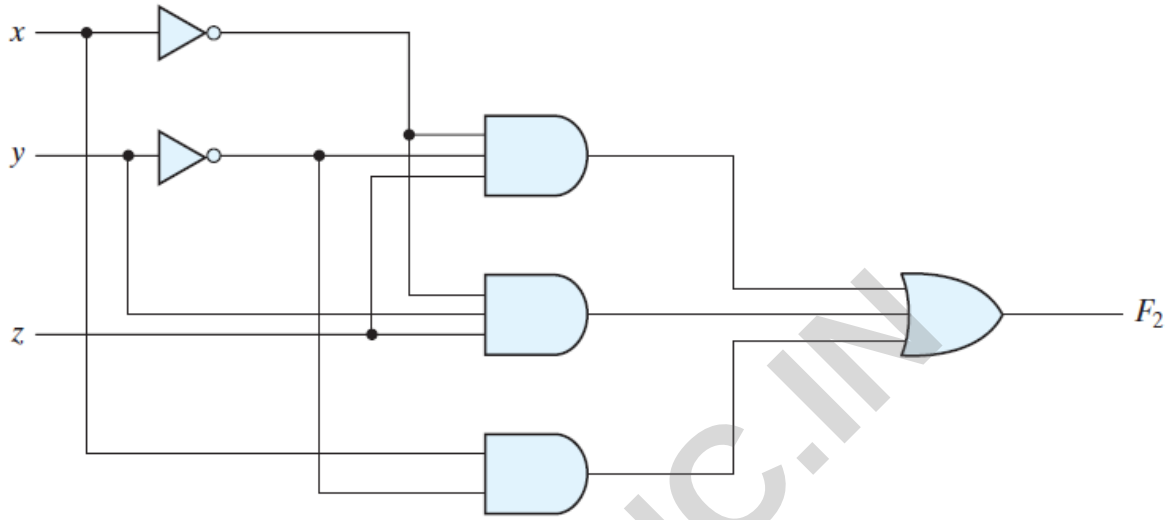
FIGURE 2.1
Gate implementation of $F_1 = x + y'z$

By manipulating a Boolean expression according to the rules of Boolean algebra, Designers are motivated to reduce the complexity and number of gates.

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).



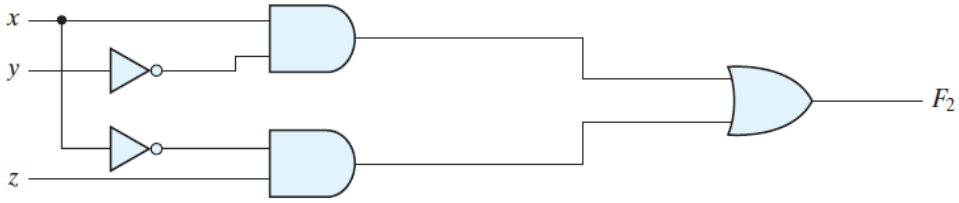
(a) $F_2 = x'y'z + x'yz + xy'$

The truth table for F2 is listed in Table 2.2 . The function is equal to 1 when xyz = 001 or 011 or when xy = 10 (irrespective of the value of z) and is equal to 0 otherwise. This set of conditions produces four 1's and four 0's for 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. 2.2 (b).



(b) $F_2 = xy' + x'z$

FIGURE 2.2
Implementation of Boolean function F_2 with gates

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.

Algebraic Manipulation

When a **Boolean expression** is implemented with **logic gates**, each term requires a gate and each variable within the term designates an input to the gate. We define **a literal** to be a **single variable** within a term, in complemented or uncomplemented form. The function of **Fig. 2.2 (a)** has three terms and eight literals, and the one in **Fig. 2.2 (b)** has two terms and four literals. By **reducing the number of terms**, the number of literals, or both in a Boolean expression, it is often possible to obtain a **simpler circuit**.

EXAMPLE:2.1

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

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

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

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

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.

1)P5B **2)**P4B **3)**P4B **4)**T2A **5)** From (4)

Complement of a Function

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

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

EXAMPLE:2.2

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

EXAMPLE:2.3

Find the complement of the functions F_1 and F_2 of Example 2.2 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: $x' + (y + z)(y' + z') = F_2'$.

2.8 DIGITAL LOGIC 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 **triangle symbol** by itself designates a **buffer circuit**. A buffer produces the transfer function, but does not produce a logic operation, since the binary value of the output is equal to the binary value of the input. This circuit is used for power amplification of the signal and is equivalent to two inverters connected in cascade.

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.

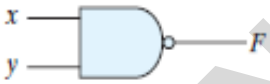
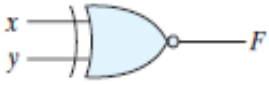
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	$x\ y \mid F$: 0 0 | 0; 0 1 | 0; 1 0 | 0; 1 1 | 1
OR		$F = x + y$	$x\ y \mid F$: 0 0 | 0; 0 1 | 1; 1 0 | 1; 1 1 | 1
Inverter		$F = x'$	$x \mid F$: 0 | 1; 1 | 0
Buffer		$F = x$	$x \mid F$: 0 | 0; 1 | 1
NAND		$F = (xy)'$	$x\ y \mid F$: 0 0 | 1; 0 1 | 1; 1 0 | 1; 1 1 | 0
NOR		$F = (x + y)'$	$x\ y \mid F$: 0 0 | 1; 0 1 | 0; 1 0 | 0; 1 1 | 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	$x\ y \mid F$: 0 0 | 0; 0 1 | 1; 1 0 | 1; 1 1 | 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	$x\ y \mid F$: 0 0 | 1; 0 1 | 0; 1 0 | 0; 1 1 | 1

FIGURE 2.5
Digital logic gates

Extension to Multiple Inputs

The gates shown in **Fig. 2.5** except for the inverter and buffer can be extended to have **more than two inputs**.

A gate can be extended to have multiple inputs if the binary operation it represents is

commutative and associative. The AND and OR operations, defined in Boolean algebra, possess these two properties.

For the OR function, we have

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

and

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

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**, The difficulty is that the NAND and NOR operators are **not associative.**

(i.e., $(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$), as shown in Fig. 2.6 and the following equations:

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

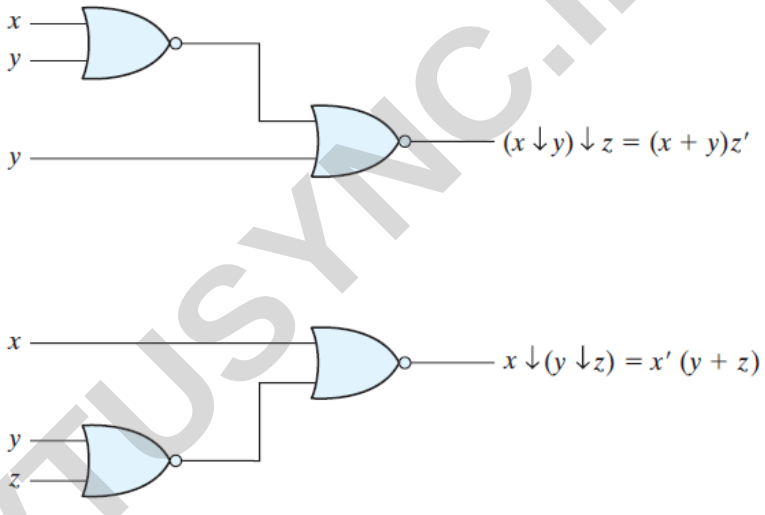


FIGURE 2.6
Demonstrating the nonassociativity of the NOR operator: $(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$

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

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

The graphic symbols for the **three-input gates** are shown in **Fig. 2.7.** To demonstrate this principle, consider the circuit of Fig. **2.7 (c).** The Boolean function for the circuit must be written as

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

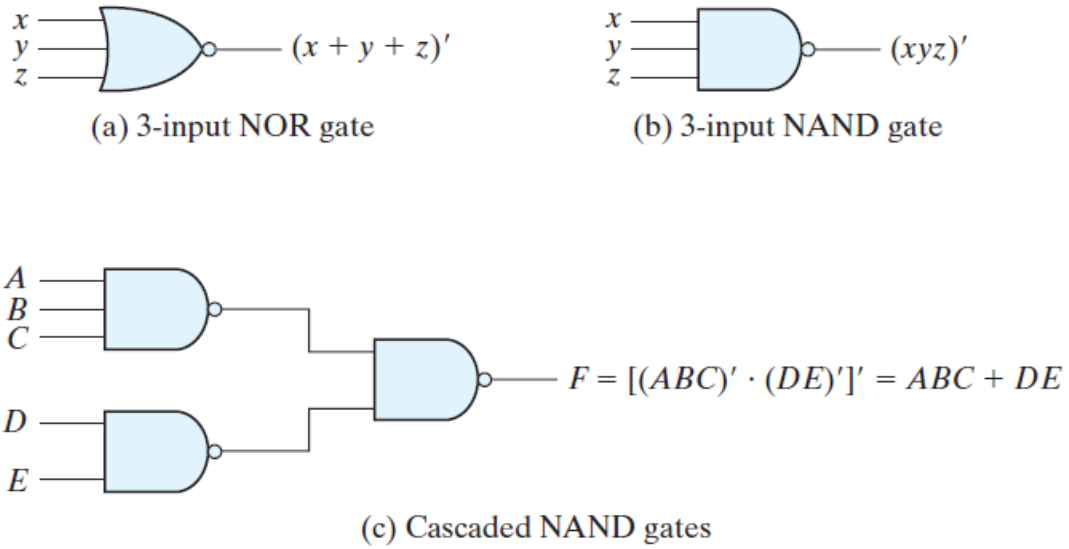
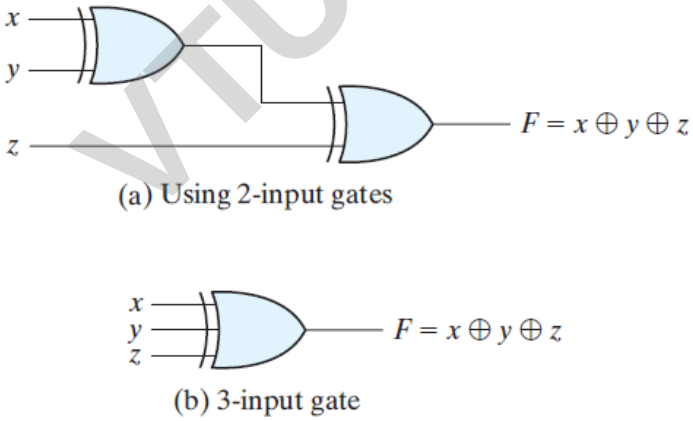


FIGURE 2.7
Multiple-input and cascaded NOR and NAND gates

The exclusive-OR and equivalence gates are **both commutative and associative** and can be extended to more than two inputs. 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 of a three-input exclusive-OR function** is shown in **Fig. 2.8** . 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.



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

(c) Truth table

FIGURE 2.8
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. 2.9 .**

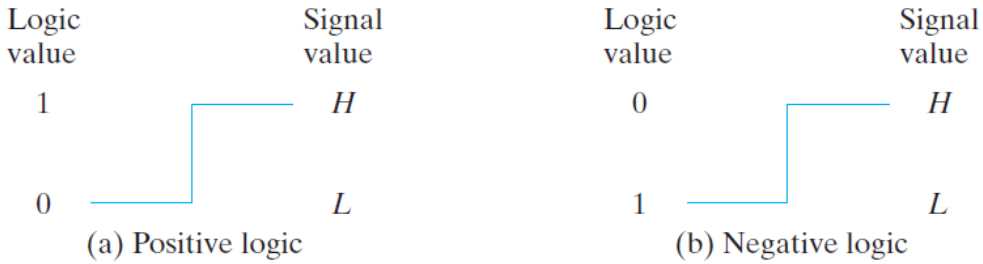


FIGURE 2.9
Signal assignment and logic polarity

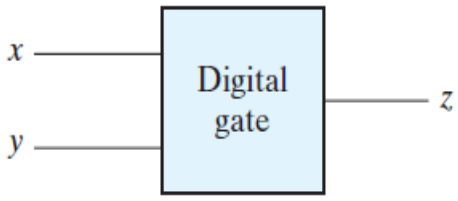
The higher signal level is designated by H and the lower signal level by L. Choosing the **high-level H to represent logic 1** defines **a positive logic system.** Choosing the **low-level L to represent logic 1** defines **a negative logic system.**

Consider, for **example**, the electronic gate shown in **Fig. 2.10 (b).** The truth table for this gate is listed in **Fig. 2.10 (a).** It specifies the physical behaviour of the gate when H is 3 V and L is 0 V. The truth table of **Fig. 2.10 (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. 2.10 (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. 2.10 (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. 2.10 (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.

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

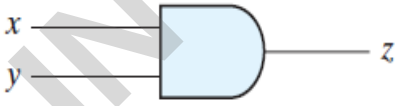
(a) Truth table with H and L



(b) Gate block diagram

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

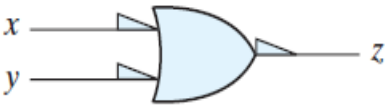
(c) Truth table for positive logic



(d) Positive logic AND gate

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

(e) Truth table for negative logic



(f) Negative logic OR gate

FIGURE 2.10
Demonstration of positive and negative logic

Gate-Level Minimization

3.1 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.

3.2 THE MAP METHOD

The complexity of the digital logic gates that implement a **Boolean function** is directly related to the complexity of the **algebraic expression** from which the function is implemented.

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 **min term** of the function that is to be minimized. Since any Boolean function can be expressed as a sum of minterms.

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

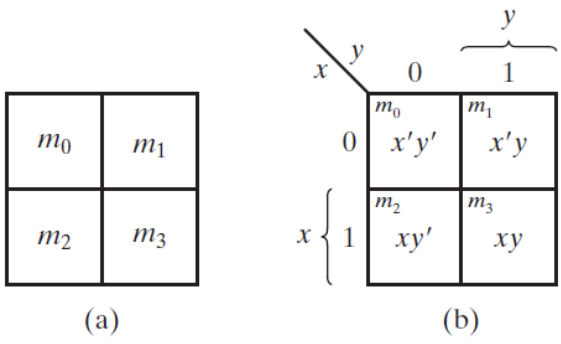


FIGURE 3.1
Two-variable K-map

If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables. As an example, the function xy is shown in **Fig. 3.2 (a).** Since **xy** is equal to m3, a 1 is placed inside the square that belongs to m3. Similarly, the function **x + y** is represented in the map of **Fig. 3.2 (b)** by three squares marked with 1's. These squares are found from the minterms of the function:

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

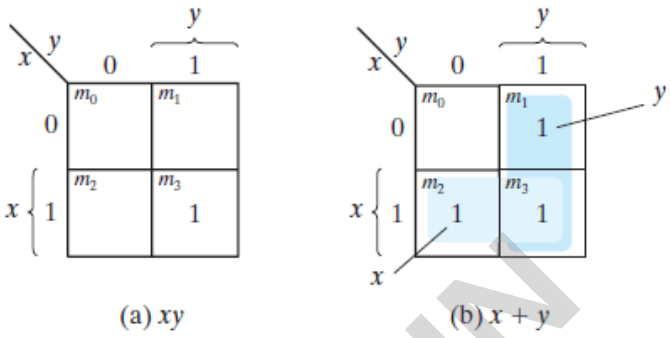


FIGURE 3.2
Representation of functions in the map

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. 3.3 .There are **eight minterms** for three **binary variables**; therefore, the map consists of **eight** squares. Note that the minterms are arranged, not in a binary sequence, but in a sequence similar to the Gray code (Table 1.6).

The characteristic of this sequence is that **only one-bit changes** in value from one adjacent column to the next. The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables.

For example, the square assigned to m_5 corresponds to row 1 and column 01. When these two numbers are concatenated, they give the binary number 101, whose decimal equivalent is 5. Each cell of the map corresponds to a unique minterm, so another way of looking at square m_5 = xy'z is to consider it to be in the row marked x and the column belonging to y'z (column 01).

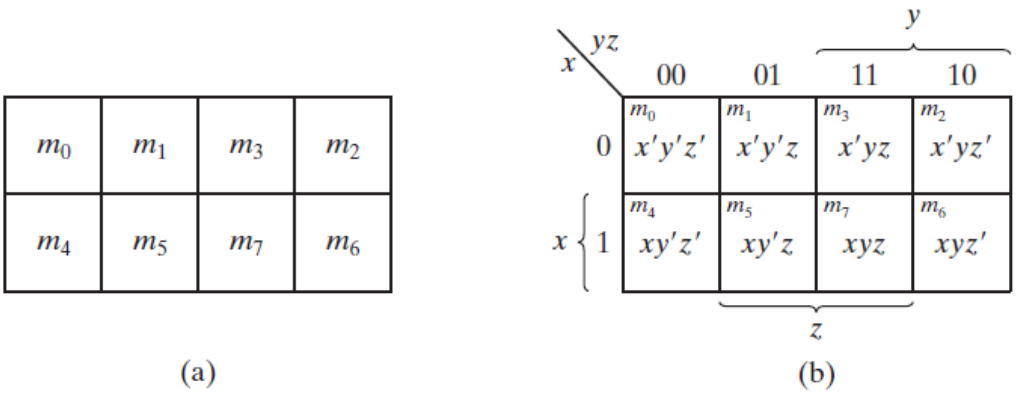


FIGURE 3.3
Three-variable K-map

Any two adjacent squares in the map differ by only one variable,For example, m5 and m7 lie in two adjacent squares. Variable y is primed in m5 and unprimed in m7, 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 m5 and m7:

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

EXAMPLE 3.1

Simplify the Boolean function

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

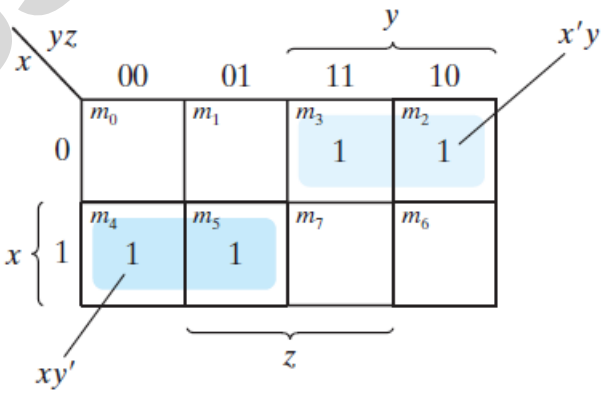


FIGURE 3.4
Map for Example 3.1, $F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$

EXAMPLE 3.2

Simplify the Boolean function

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

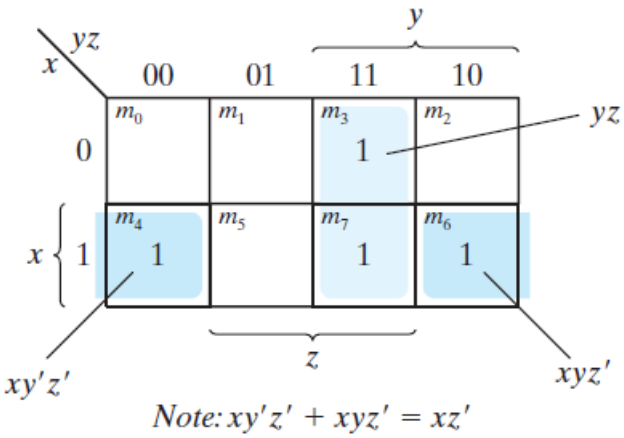


FIGURE 3.5
Map for Example 3.2, $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

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

- One square represents one minterm, giving a term with three literals.
- Two adjacent squares represent a term with two literals.
- Four adjacent squares represent a term with one literal.
- Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.

EXAMPLE 3.3

Simplify the Boolean function

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

$$F = z' + xy'$$

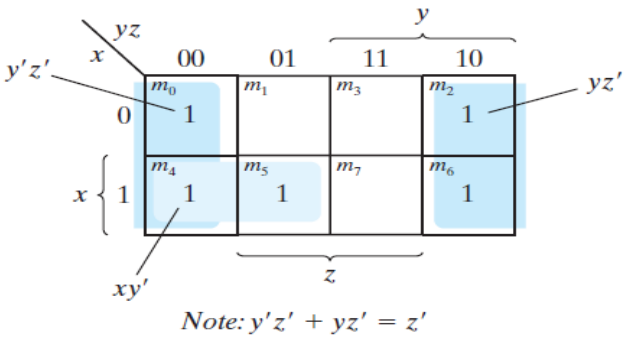


FIGURE 3.6
Map for Example 3.3, $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$

EXAMPLE 3.4

For the Boolean function

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

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

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

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:

$$F = C + A'B$$

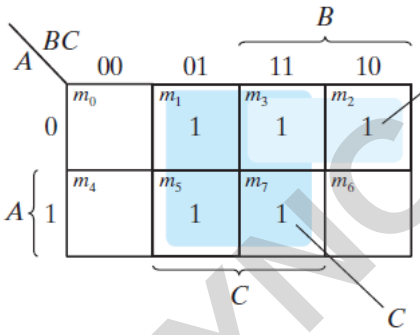


FIGURE 3.7
Map of Example 3.4, $A'C + A'B + AB'C + BC = C + A'B$

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, m0 and m2 form adjacent squares, as do m3 and m11. 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.

3.3 Four-Variable K-Map

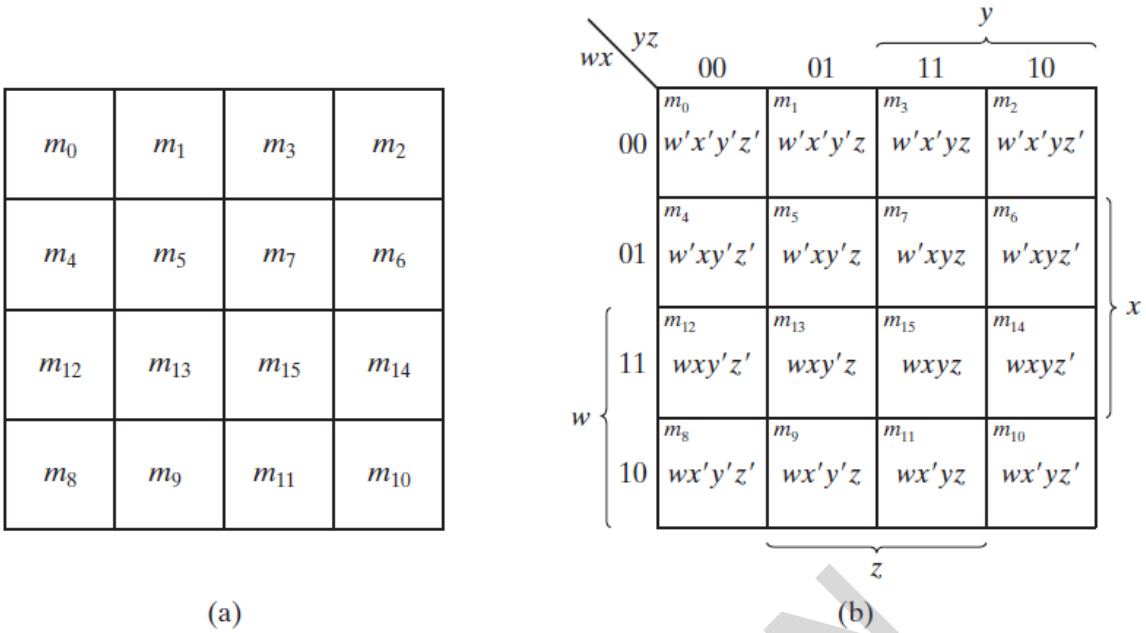


FIGURE 3.8
Four-variable map

EXAMPLE 3.5

Simplify the Boolean function

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

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

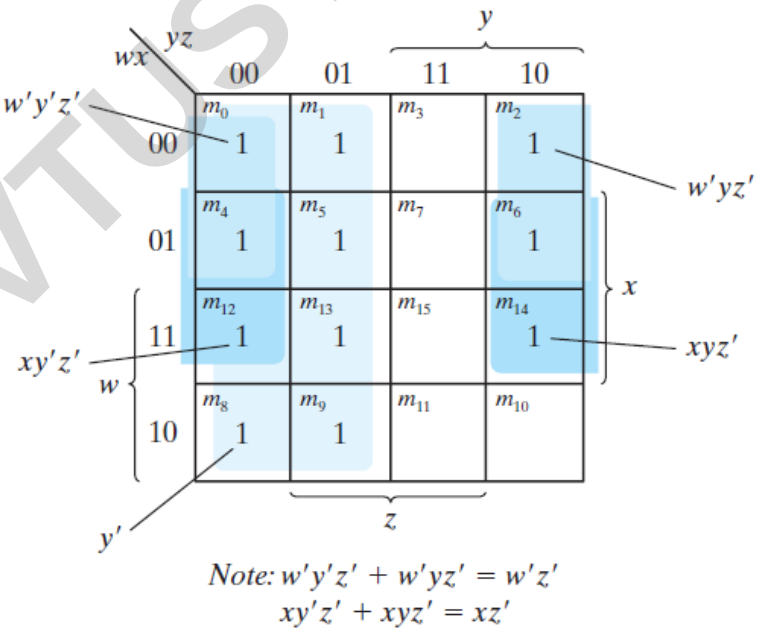


FIGURE 3.9
Map for Example 3.5, $F(w, x, y, z) = \Sigma(0,1, 2, 4, 5, 6, 8, 9, 12, 13, 14) = y' + w'z' + xz'$

EXAMPLE 3.6

Simplify the Boolean function

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

The area in the map covered by this function consists of the squares marked with 1's in **Fig. 3.10** . The function has four variables and, as expressed, consists of three terms

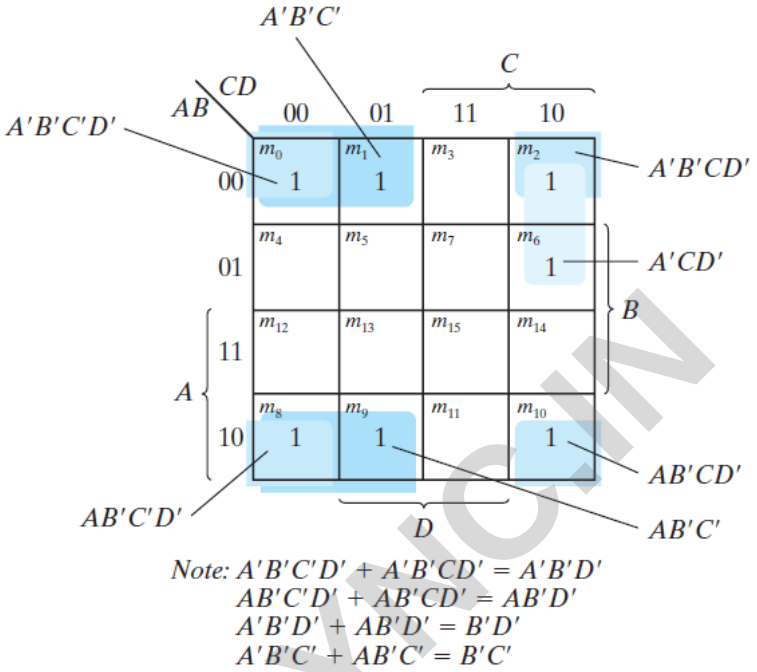


FIGURE 3.10
Map for Example 3.6, $A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$

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

(1) all the minterms of the function are covered when we combine the squares,

(2) the number of terms in the expression is minimized, and

(3) there are no redundant terms (i.e., minterms already covered by other terms).

Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of terms. **A *prime implicant* is a product term obtained by combining the maximum possible number of adjacent squares in the map.** If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be *essential.*

The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares. This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's form a prime implicant, provided that they are not within a group of four adjacent squares. Four adjacent 1's form a prime implicant if they are not within a group of eight adjacent squares, and so on. The prime implicant is essential if it is the only prime implicant that covers the minterm.Consider the following four-variable Boolean function:

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

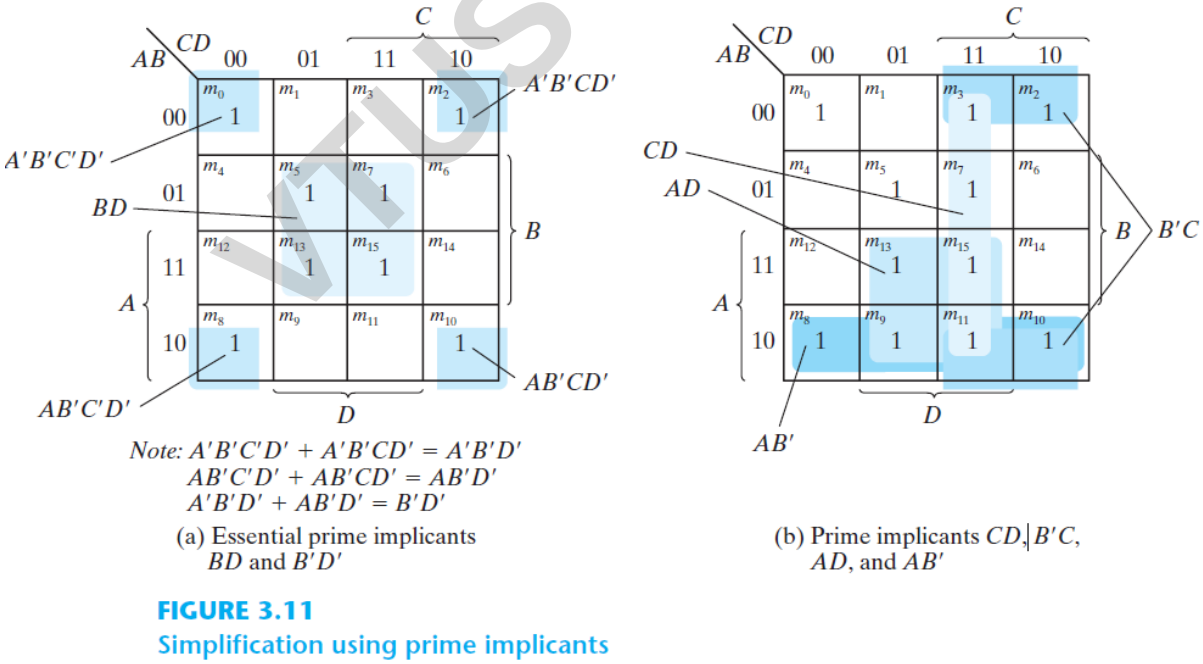


FIGURE 3.11
Simplification using prime implicants

The minterms of the function are marked with 1's in the maps of Fig. 3.11 . The partial map (Fig. 3.11(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 m5 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 **(m3, m9, and m11**) must be considered next.

Figure 3.11 (b) shows all possible ways that the three minterms can be covered with prime implicants. Minterm *m*3 can be covered with either prime implicant *CD* or prime implicant *B*'*C*. Minterm *m*9 can be covered with either *AD* or *AB*' .Minterm *m*11 is covered with any one of the four prime implicants.

The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants that cover minterms m3, m9, and m11. There are four possible ways that the function can be expressed with four product terms of two literals each:

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

3.5 DON'T-CARE CONDITIONS

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms.

This pair of conditions assumes that all the combinations of the values for the variables of the function are valid. In practice, in some applications the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified.

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

A don't-care minterm is a combination of variables whose logical value is not specified. Such a minterm cannot be marked with a 1 in the map, because it would require

that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the particular minterm.

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

EXAMPLE 3.8

Simplify the Boolean function

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

which has the don't-care conditions

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

The minterms of F are the variable combinations that make the function equal to 1. The minterms of d are the don't-care minterms that may be assigned either 0 or 1. The map simplification is shown in **Fig. 3.15** .

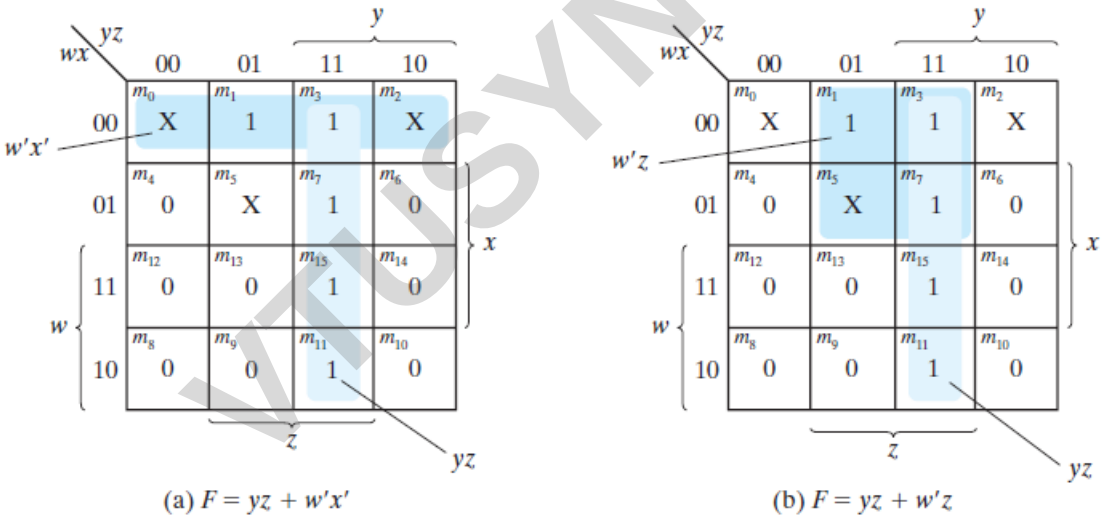


FIGURE 3.15
Example with don't-care conditions

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, 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. 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

$$F = yz + w'z$$

Consider the two simplified expressions obtained in **Example 3.8 :**

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

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.

NAND AND NOR IMPLEMENTATION