5

Boolean Algebra and Logic Gates

Learning Objectives

In this chapter, we will learn Boolean algebra and understand Boolean operations using Venn diagrams. For more clarity and better understanding of Boolean logic concepts in computer science, we will represent Boolean functions as expressions, truth tables, and circuit diagrams. In order to minimize the need for digital components, we will also learn to simplify Boolean expressions. The chapter will also discuss vital digital circuits such as logic gates, adder circuits, and flip-flops.

5.1 BOOLEAN ALGEBRA

As already discussed in Chapter 4, computers understand only binary numbers, in which only the digits 0 and 1 are used. These binary computer systems work on propositional logic, wherein a proposition may be *true* or *false* and may also be stated as functions of other propositions that are connected by three basic logical connectives—AND, OR, and NOT. For example,

I will do post-graduation if I get admission in MSc or MCA.

This statement functionally connects the proposition 'I will do post-graduation' to two propositions—'if I get admission in MSc' and 'if I get admission in MCA'. This scenario can be represented as shown in Figure 5.1.

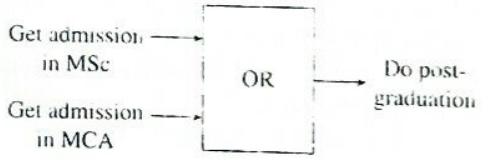


Fig. 5.1 Statement and simple propositions

From this discussion, we can conclude that the meaning of the OR connective is that the corresponding output is true if either one of the inputs is true; otherwise, it is false.

We can make this proposition a little more complex by saying 'I will do post-graduation if I do not get a job and if I get admission either in MSc or in MCA'. Before preparing the block diagram for this preposition, we must first state it in a well-structured way to understand how it is composed. The proposition can be given as follows:

Do post-graduation = (NOT (Job)) AND ((Get admission in MSc) OR (Get admission in MCA))

The proposition can be represented diagrammatically as shown in Figure 5.2.

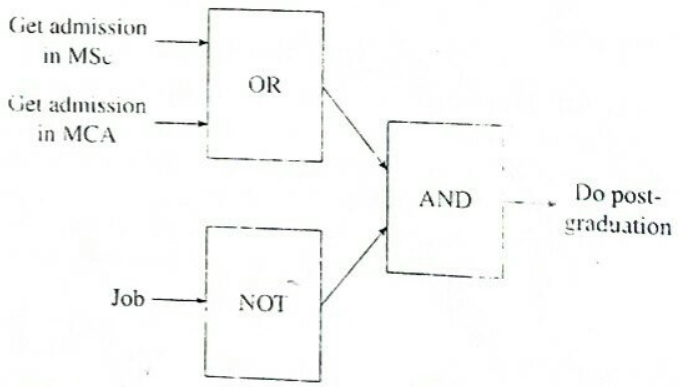


Fig. 5.2 Statement and complex propositions

Thus, we see that a complex proposition can be easily stated in terms of binary (two) variables and binary operators (OR, AND, and NOT). It is for this purpose that the mathematician George Boole developed Boolean algebra in 1854.

In Boolean algebra, a Boolean system has either of two states—*true* (T) or *false* (F). The following are the salient points of Boolean algebra:

- Value '1' means true.
- Value '0' means false.
- A *Boolean* expression is a combination of Boolean variables and Boolean operators. Boolean expressions that are logically equivalent to one another are called equivalent expressions.
- A *Boolean function* has one or more input variables and generates an output based on these input values. The result may be 0 or 1.
- Boolean operators take certain inputs and produce an output based on a predetermined table of results (also known as the *truth table*).
- The Boolean operator AND (conjunction) is used as a '.' and in some texts as '$\cap$'. For example, $A \cdot B$ or $A \cap B$ means A and B. The AND operator takes two (or more) inputs and returns a 1 only when both (or all) inputs are 1.
- The Boolean operator OR (disjunction) is used as a '+' and in some texts as '$\cup$'. For example, $A + B$ or $A \cup B$ means A or B. The OR operator takes two (or more) inputs and returns a 1 when any or all of the inputs are 1.
- The Boolean operator NOT (negation) is used as ''' and in some texts as '~'. It simply negates the value of the operator. For example, A' means NOT A. If the input is 1, the output will be 0 and vice versa.

Boolean algebra forms the basis of current digital systems. It is extensively used to design digital circuits and is applied in digital logic, computer programming, set theory, and statistics.

5.2 VENN DIAGRAMS

A Venn diagram can be used to represent Boolean operations using shaded overlapping regions. In the Venn diagrams that are presented in this section, there is one circular region for each variable. The interior and exterior of a region of a particular variable corresponds to the values 1 (true) and 0 (false), respectively. The Venn diagrams corresponding to AND, OR, and NOT operations are shown in Figure 5.3.

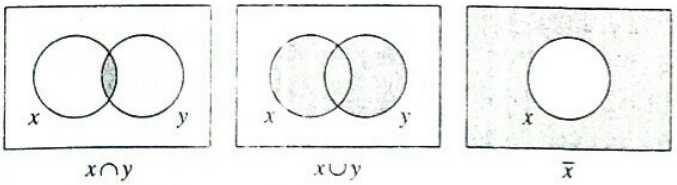


Fig. 5.3 Venn diagrams for conjunction, disjunction, and complement

Here, the shaded area indicates the value 1 of the operation and the non-shaded area denotes the value 0.

The following points can be observed from Figure 5.3:

- For the AND operation, the region common to both circles is shaded to indicate that $x \cap y$ is 1 when both variables are 1. The non-shaded area indicates that $x \cap y$ is 0 for the other three combinations of variables x and y.
- For the OR operation, the regions that lie inside either or both the circles are shaded.
- For the NOT operation, the Venn diagram represents complement x' by shading the region *not* inside the circle.

Venn diagrams are useful to understand and visualize the laws of Boolean algebra.

Absorption Law: $x \cap (x \cup y) = x$

Shade the portion for $x \cup y$. Next, shade the portion for x. Then, shade the area that is common in both the diagrams. The resultant shaded area will be the whole of the circle x (Figure 5.4).

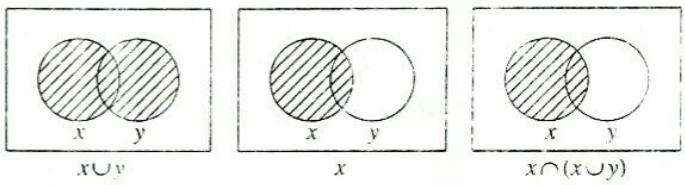


Fig. 5.4 Venn diagrams for absorption law

Double Negation Law: $(x')' = x$

First, shade the circle x. Then, shade the area not in x. This will give the region for x'. Next shade the area that is not in x', which will be the whole of circle x (Figure 5.5).

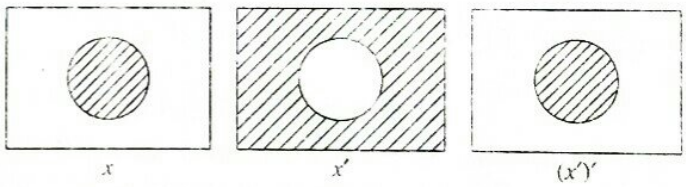


Fig. 5.5 Venn diagrams for double negation law

De Morgan's Law: $x' \cap y' = (x \cup y)'$

To visualize De Morgan's law, for the left-hand side (LHS), shade the area that is neither in x nor in y. Then, for the right-hand side (RHS), shade the area corresponding to $x \cup y$, which is the area in either or both the circles. Finally, shade the area that is not in the region of $x \cup y$. It can be seen from Figure 5.6 that it is the same area that we had shaded for the LHS.

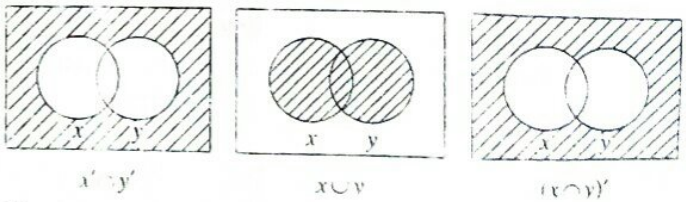


Fig. 5.6 Venn diagrams for De Morgan's law

5.3 TRUTH TABLES

A truth table is used to describe a Boolean function of n input variables. It lists all possible values of input combinations of the function and the output that will be produced based on these input combinations. Such a table provides a useful visual tool for defining the input–output relationship of binary variables in a Boolean function.

A Boolean function of n variables has 2^n rows of possible input combinations. Each row specifies the output of the Boolean function for a different combination. The truth tables for the Boolean operators are shown in Table 5.1.

Table 5.1 Truth tables for Boolean operators

AND		
A	B	Z = A · B
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	Z = A + B
0	0	0
0	1	1
1	0	1
1	1	1

NOT	
A	Z = A′
0	1
1	0

A truth table can also be used to represent one or more Boolean functions. For example, Table 5.2 represents the truth table for the Boolean function $Z = A \cup (B \cap C)$.

Table 5.2 Truth table for Boolean function $Z = A \cup (B \cap C)$

A	B	C	B ∩ C	Z = A ∪ (B ∩ C)
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

5.4 BASIC LAWS OF BOOLEAN ALGEBRA

The laws of Boolean algebra can be divided into two categories—laws applicable on a single variable and laws applicable on multiple variables (Figure 5.7).

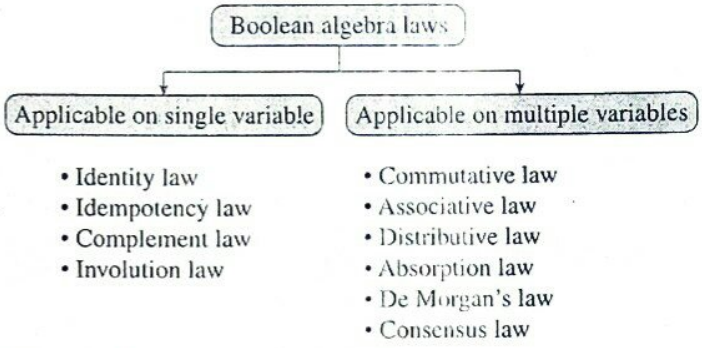


Fig. 5.7 Classification of laws of Boolean algebra

1. $A + 0 = A$
   If $A = 0$, then $0 + 0 = 0$, which is equal to A.
   If $A = 1$, then $1 + 0 = 1$, which is again equal to A.
2. $A + 1 = 1$
   If $A = 0$, then $0 + 1 = 1$.
   If $A = 1$, then $1 + 1 = 1$.
3. $A \cdot 0 = 0$
   If $A = 0$, then $0 \cdot 0 = 0$.
   If $A = 1$, then $1 \cdot 0 = 0$.
4. $A \cdot 1 = A$
   If $A = 0$, then $0 \cdot 1 = 0$, which is equal to A.
   If $A = 1$, then $1 \cdot 1 = 1$, which is again equal to A.

5.4.1 Identity Law

The identity law states that a term OR'ed with a 0 or AND'ed with a 1 will always equal that term. Similarly, a term OR'ed with a 1 is always 1 and a term AND'ed with a 0 will always result in 0.

5.4.2 Idempotency Law

The idempotency law states that a term AND'ed with itself or OR'ed with itself is equal to that term.

1. $A + A = A$
   If $A = 0$, then $0 + 0 = 0$, which is equal to A.
   If $A = 1$, then $1 + 1 = 1$, which is again equal to A (here $A + B$ means a true value, that is, a non-zero value).
2. $A \cdot A = A$
   If $A = 0$, then $0 \cdot 0 = 0$, which is equal to A.
   If $A = 1$, then $1 \cdot 1 = 1$, which is again equal to A.

5.4.3 Complement Law

The complement law states that a term AND'ed with its complement always results in 0 and a term OR'ed with its complement always results in 1.

1. $A + A' = 1$
   If $A = 0$, then $0 + 1 = 1$.
   If $A = 1$, then $1 + 0 = 1$.
2. $A \cdot A' = 0$
   If $A = 0$, then $0 \cdot 1 = 0$, which is equal to 0.
   If $A = 1$, then $1 \cdot 0 = 0$, which is again equal to 0.

5.4.4 Involution Law

The involution law, also known as the double negation law states that a term that is inverted twice is equal to the original term.

$(A')' = A$

If $A = 0$, then $(0')'$ can be solved as $1' = 0$, which is equal to A.

If $A = 1$, then $(1')'$ can be solved as $0' = 1$, which is equal to A.

5.4.5 Commutative Law

The commutative law states that the order of application of two separate terms is not important.

1. $A + B = B + A$

   The truth table of the Boolean functions proves this law, as shown in Table 5.3.

Table 5.3 Truth table to prove the commutative law of the OR operator

A	B	A + B	B + A
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

2. $A \cdot B = B \cdot A$

   Table 5.4 shows the truth table of the Boolean functions, which proves this law.

Table 5.4 Truth table to prove the commutative law of the AND operator

A	B	A · B	B · A
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

5.4.6 Associative Law

The associative law allows the removal of brackets from an expression and regrouping of the variables.

1. $A + (B + C) = (A + B) + C$

   The truth table of the Boolean functions, shown in Table 5.5, proves this law.

Table 5.5 Truth table to prove the associative law of the OR operator

A	B	C	A + (B + C)	(A + B) + C
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

2. $A \cdot (B \cdot C) = (A \cdot B) \cdot C$

   The truth table of the Boolean functions, shown in Table 5.6, proves this law.

Table 5.6 Truth table to prove the associative law of the AND operator

A	B	C	A · (B · C)	(A · B) · C
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

5.4.7 Distributive Law

The distributive law enables multiplication or factoring out of an expression.

1. $A(B + C) = A \cdot B + A \cdot C$

   The truth table of the Boolean functions, shown in Table 5.7, proves this law.

Table 5.7 Truth table to prove the first distributive law

A	B	C	B + C	A · (B + C)	A · B	A · C	A · B + A · C
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

2. $A + (B \cdot C) = (A + B) \cdot (A + C)$

   RHS $= (A + B) \cdot (A + C)$

   $= A \cdot A + A \cdot B + A \cdot C + B \cdot C$ (Opening the brackets and multiplying)

   $= A + A \cdot B + A \cdot C + B \cdot C$ (Since $A \cdot A = A$)

   $= A \cdot (1 + B) + A \cdot C + B \cdot C$ (Take A common from the first two terms)

   $= A + A \cdot C + B \cdot C$ (Since $1 + B = 1$ and $A \cdot 1 = A$)

   $= A(1 + C) + B \cdot C$

   $= A + B \cdot C = \text{LHS}$ (Since $1 + C = 1$ and $A \cdot 1 = A$)

Let us prove this law using the truth table of the Boolean functions (Table 5.8).

Table 5.8 Truth table to prove the second distributive law

A	B	C	B · C	A + B · C	A + B	A + C	(A + B) · (A + C)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

5.4.8 Absorption Law

The absorption law permits the reduction of a complicated expression into a simpler one by absorbing like terms.

1. $A + A \cdot B = A$

   LHS $= A + A \cdot B$
   $= A \cdot 1 + A \cdot B$
   $= A(1 + B)$ (Since $A \cdot 1 = A$)
   $= A \cdot 1$ (Since $1 + B = 1$)
   $= A$ = RHS

2. $A \cdot (A + B) = A$

   LHS $= A \cdot (A + B)$
   $= A \cdot A + A \cdot B$
   $= A + A \cdot B$ (Since $A \cdot A = A$)
   $= A \cdot 1 + A \cdot B$
   $= A \cdot (1 + B)$ (Since $A \cdot 1 = A$)
   $= A \cdot 1$ (Since $1 + B = 1$)
   $= A$ = RHS (Since $A \cdot 1 = A$)

3. $A + A' \cdot B = A + B$

   LHS $= A + A' \cdot B$
   $= (A + A') \cdot (A + B)$ (Apply distributive law in which $A + B \cdot C = (A + B) \cdot (A + C)$)
   $= 1 \cdot (A + B)$ (Since $A + A' = 1$)
   $= A + B$ = RHS

4. $A \cdot (A' + B) = A \cdot B$

   LHS $= A \cdot (A' + B)$
   $= A \cdot A' + A \cdot B$
   $= 0 + A \cdot B$ (Since $A \cdot A' = 0$)
   $= A \cdot B$ = RHS (Since $A + 0 = A$)

5.4.9 Consensus Law

The consensus law is the conjunction of all the unique literals of the terms, excluding the literal that is not negated in one term but negated in the other.

1. $A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C$

   LHS $= A \cdot B + A' \cdot C + B \cdot C$
   $= A \cdot B + A' \cdot C + B \cdot C \cdot 1$ (Since $A \cdot 1 = A$)
   $= A \cdot B + A' \cdot C + B \cdot C (A + A')$
   $= A \cdot B + A' \cdot C + A \cdot B \cdot C + A' \cdot B \cdot C$
   $= A \cdot B + A \cdot B \cdot C + A' \cdot C + A' \cdot B \cdot C$
   $= A \cdot B(1 + C) + A' \cdot C(1 + B)$
   $= A \cdot B + A'C$ = RHS (Since $1 + C = 1$ and $1 + B = 1$)

2. $(A + B)(A' + C)(B + C) = (A + B)(A' + C)$

   LHS $= (A + B)(A' + C)(B + C)$
   $= (A \cdot A' + A' \cdot B + A \cdot C + B \cdot C)(B + C)$ (Opening the brackets)
   $= (A' \cdot B + A \cdot C + B \cdot C)(B + C)$ (Since $A \cdot A' = 0$)
   $= A' \cdot B \cdot B + A \cdot B \cdot C + B \cdot B \cdot C + A' \cdot B \cdot C + A \cdot C \cdot C + B \cdot C \cdot C$ (Opening the brackets)
   $= A' \cdot B + A \cdot B \cdot C + B \cdot C + A' \cdot B \cdot C + A \cdot C + B \cdot C$ (Since $A \cdot A = A$)
   $= A' \cdot B + B \cdot C(A + A') + B \cdot C + A \cdot C$ (Since $A + A = A$)
   $= A' \cdot B + B \cdot C + A \cdot C$ (Since $A + A' = 1$)

   RHS $= (A + B)(A' + C)$
   $= A \cdot A' + A' \cdot B + A \cdot C + B \cdot C$
   $= A' \cdot B + A \cdot C + B \cdot C$

Therefore, LHS = RHS.

5.4.10 De Morgan's Laws

The two laws of De Morgan are as follows:

1. $(A + B)' = A' \cdot B'$

   The first law states that two separate terms NAND'ed together gives the same result as when these two terms are complemented and then OR'ed together.

   Let us prove this law using the truth table of the Boolean functions (Table 5.9).

Table 5.9 Truth table to prove the first De Morgan's law

A	B	A + B	(A + B)'	A'	B'	A' · B'
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

2. $(A \cdot B)' = A' + B'$

   De Morgan's second law states that two separate terms NOR'ed together gives the same result as when these two terms are complemented and then AND'ed together.

Let us prove this law using the truth table of the Boolean functions (Table 5.10).

Table 5.10 Truth table to prove the second De Morgan's law

A	B	A · B	(A · B)′	A′	B′	A′ + B′
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

5.5 REPRESENTATIONS OF BOOLEAN FUNCTIONS

As shown in Figure 5.8, a Boolean function can be represented using any one of the following methods:

- A Boolean expression (canonical form)
- A truth table (shown in Section 5.5)
- A circuit diagram (refer to Section 5.7)

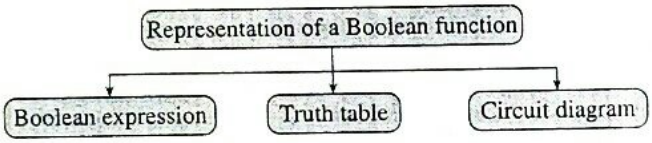


Fig. 5.8 Different ways of representing a Boolean function

In Boolean algebra, a variable or a complemented variable is called a *literal*. Designers of logical circuits prefer to use a standardized form of literals known as the *canonical form* to represent a Boolean function. The key advantage of using a canonical form is that it helps to reduce the number of logic gates and thus interconnections between the components, thereby helping in the simplification and minimization of digital circuits. A canonical form can be either a sum of *minterms* or a product of *maxterms*.

5.5.1 Minterm

A *minterm* is a canonical product (or a canonical sum of products—SOP) term and has the following features:

- It includes all variables of a function.
- Each variable is in either the un-complemented or complemented (inversed) form.
- Each variable appears exactly once.

Note: A minterm is a product term, but a product term may or may not be a minterm.

Let us look at some examples of product terms and minterms for a function of three variables A, B, and C:

Product terms $A, B \cdot C, A' \cdot C, A \cdot B \cdot C, A \cdot B' \cdot C, A' \cdot B' \cdot C'$

Minterms $A \cdot B \cdot C', A \cdot B \cdot C, A' \cdot B \cdot C, A \cdot B' \cdot C'$

Basically, a minterm is represented by the symbol m_j, where the subscript j is the decimal equivalent of the minterm. For example, consider the truth table of the XOR operation given in Table 5.11 and note the corresponding minterms.

Table 5.11 Truth table of XOR operation

A	B	A XOR B	Minterm
0	0	0	–
0	1	1	$m_1 = A'B$
1	0	1	$m_2 = AB'$
1	1	0	–

By looking at the truth table, we can say that the Exclusive OR function can be represented as a sum of the minterms or the canonical SOP form as follows:

$$F(A, B) = A'B + AB'$$

The shorthand notation is as follows:

$$F(A, B) = m_1 + m_2 \text{ or } F(A, B) = \Sigma m(1, 2)$$

A Boolean function with n variables has 2^n minterms, as each variable can be in either the complemented or uncomplemented form. Thus, a three-variable function, $F(A, B, C)$, has $2^3 = 8$ minterms as shown in Table 5.12.

Table 5.12 Minterms of a three-variable function

A	B	C	Minterm
0	0	0	$m_0 = A'B'C'$
0	0	1	$m_1 = A'B'C$
0	1	0	$m_2 = A'BC'$
0	1	1	$m_3 = A'BC$
1	0	0	$m_4 = AB'C'$
1	0	1	$m_5 = AB'C$
1	1	0	$m_6 = ABC'$
1	1	1	$m_7 = ABC$

Only those minterms for which the function produces 1 as the output are used to represent the Boolean functions.

Note: A Boolean function can be represented as a *sum of minterms*; $f = \Sigma$(minterms).

Note that the sum of minterms form is unique for any function. We can convert any SOP expression into the canonical SOP form using the following steps:

- Determine the product terms of the expression.
- Ensure that each product term has all the variables used in the Boolean expression.
- If there is a product term in which one or more variables are missing, then multiply that term with the sum of the missing variables and their complement.

- Expand the Boolean expression and delete the repeated terms from the expression.

Example 5.1 Convert $F(A, B, C) = AB' + BC' + AC'$ to the canonical SOP form.

Solution

Multiply each term with the sum of the missing variable and its complement. Therefore,

$$\begin{aligned} F(A, B, C) &= AB'(C + C') + BC'(A + A') + AC'(B + B') \\ &= AB'C + AB'C' + ABC' + A'BC' + ABC' + AB'C' \\ &= AB'C + AB'C' + ABC' + A'BC' \quad \text{(Because } A + A = A\text{)} \\ &= 101 + 100 + 110 + 010 \\ &= m_5 + m_4 + m_6 + m_2 \end{aligned}$$

Therefore, the canonical SOP form of the given Boolean expression can be given as

$$F(A, B, C) = m_5 + m_4 + m_6 + m_2 \text{ or } F(A, B, C) = \Sigma m(2, 4, 5, 6)$$

5.5.2 Maxterm

A *maxterm* is a canonical sum (or a canonical product of sums—POS) term and has the following features:

- It includes all variables of a function.
- Each variable is in either the un-complemented or complemented (inversed) form.
- Each variable appears exactly once.
- A function with n variables has 2^n maxterms.
- Each maxterm is false for exactly one combination of input variables.

Note A maxterm is a sum term, but a sum term may or may not be a maxterm.

Let us consider some examples of sum terms and maxterms for a function of three variables A, B, and C:

Sum terms $A, A + B, B + C, A' + B, A' + B'$

Maxterms $A + B + C, A + B' + C, A' + B + C, A' + B' + C'$

Basically, a maxterm is represented by the symbol M_j, where the subscript j is the decimal equivalent of the maxterm. For example, consider the truth table of the XOR operation given in Table 5.13 and note the corresponding maxterms.

Table 5.13 Truth table of XOR operation

A	B	A XOR B	Maxterms
0	0	0	$M_0 = A + B$
0	1	1	–
1	0	1	–
1	1	0	$M_3 = A' + B'$

Only those input combinations for which the function produces 0 as the output are considered to find the product of maxterms.

It should be noted that for M_0 we have written $M_0 = A + B$ and not $A' + B'$. This is because any minterm m_i is the complement of the corresponding maxterm M_i, that is, $m_i' = m_i$. It is possible to check this. $(A' \cdot B')' = A + B$. Therefore, in a maxterm, 0 will be represented in the un-complemented form and 1 will be represented in the complemented form.

By looking at the truth table, we can say that the Exclusive OR function can be represented as the product of the maxterms or the canonical POS form as follows:

$$F(A, B) = (A + B) \cdot (A' + B')$$

The shorthand notation is as follows:

$$F(A, B) = M_0 \cdot M_3 \text{ or } F(A, B) = \Pi M(0, 3)$$

Therefore, $A' \cdot B + A \cdot B'$ and $(A + B) \cdot (A' + B')$ are equivalent expressions for the XOR function.

Note A Boolean function can be represented as a product of maxterms; $f = \Pi M(\text{maxterms})$

We can convert any POS expression into the canonical POS form using the following steps:

- Determine the sum terms of the expression.
- Ensure that each sum term has all the variables used in the Boolean expression.
- If there is a sum term in which one or more variables are missing, then add that term with the product of the missing variables and their complement.
- Expand the Boolean expression and delete the repeated terms from the expression.

Example 5.2 Convert $F(A, B, C) = (A + B')(B + C')$ to the canonical POS form.

Solution

Add each term with the product of the missing variable and its complement. Therefore,

$$\begin{aligned} F(A, B, C) &= [(A + B') + (C \cdot C')][(B + C') + (A \cdot A')] \\ &= (A + B' + C \cdot C')(B + C' + A \cdot A') \end{aligned}$$

Using distributive law, we can write

$$\begin{aligned} F(A, B, C) &= (A + B + C)(A + B' + C')(B + C' + A)(B + C' + A') \\ &= (A + B + C)(A + B' + C')(A + B + C')(A' + B + C') \\ &= (000)(011)(001)(101) \\ &= (M_0)(M_3)(M_2)(M_5) \end{aligned}$$

Therefore, the canonical SOP form of the given Boolean expression can be given as

$$F(A, B, C) = M_0 \cdot M_3 \cdot M_2 \cdot M_5 \text{ or } F(A, B, C) = \Pi M(0, 2, 3, 5)$$

5.6 LOGIC GATES

We have seen that computers work on binary digits, which form the heart of digital electronics. In digital electronics, we are not concerned about electricity; rather, we just need to understand the difference between *on* (1) and *off* (0). Moreover, in digital electronics, operations are performed using logic gates, which are also known as Boolean gates.

These gates work only on the logic that comprises either a 0 or 1. There are different types of Boolean gates, each of which follows a different set of rules. In this section, we will discuss Boolean gates and see how Boolean expressions can be represented using these gates in digital electronics.

AND gate The AND gate is represented with the symbol. This gate accepts two inputs and gives an output. The output will be 1 if and only if both the inputs are 1, and will be a 0 otherwise. The truth table of the AND gate is given in Table 5.14. The expression involving an AND operation can be written as $Input_1 \cap Input_2$, or $Input_1 \cdot Input_2$.

Table 5.14 Truth table of AND gate

$Input_1$	$Input_2$	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR gate The OR gate is represented with the symbol. This gate accepts two inputs and gives an output. The output will be 1 if any of the two inputs is 1. That is, it gives a 0 output only when both the inputs are 0. The truth table of the OR gate is given Table 5.15. The expression involving an OR operation can be written as $Input_1 \cup Input_2$, or $Input_1 + Input_2$.

Table 5.15 Truth table of OR gate

$Input_1$	$Input_2$	Output
0	0	0
0	1	1
1	0	1
1	1	1

NOT gate The NOT gate is represented by the symbol. Unlike the AND and OR gates, the NOT gate accepts just one input. The output produced by a NOT gate is just the opposite of the input given. For example, if the input is 1 the output is 0, and vice versa. The expression involving a NOT operation can be written as NOT Input, or as ~Input, or as Input. Table 5.16 shows the truth table of this gate.

Table 5.16 Truth table of NOT gate

Input	Output
0	1
1	0

NAND gate The NAND gate is represented using the symbol. In addition, it is known as the negated AND or NO AND gate. It is just the opposite of the AND gate. The NAND gate accepts two inputs and returns 0 if both the inputs are 1, else returns 1. The truth table of the NAND gate is given in Table 5.17. The expression involving a NAND operation can be written as $Input_1 \mid Input_2$, or $\overline{Input_1 \cdot Input_2}$.

Table 5.17 Truth table of NAND gate

$Input_1$	$Input_2$	Output
0	0	1
0	1	1
1	0	1
1	1	0

NOR gate The NOR gate is represented using the symbol. In addition, it is known as the negated OR or NO OR gate. It is just the opposite of the OR gate. The NOR gate accepts two inputs and returns 1 only if both the inputs are low (0). The truth table of the NOR gate is given in Table 5.18. The expression involving a NOR operation can be written as $Input_1 \text{–} Input_2$, or $\overline{Input_1 + Input_2}$.

Table 5.18 Truth table of NOR gate

$Input_1$	$Input_2$	Output
0	0	1
0	1	0
1	0	0
1	1	0

XOR gate The XOR gate is represented using the symbol. The X in the XOR gate stands for *exclusive*. This means that the output from this gate will be 1 if only one of the inputs is a 1, but not both. Notice in Table 5.19 that the output is a 0 if both the inputs are 1 or 0. In general, the output of the XOR gate is 1 when the number of its high inputs is odd. The expression involving an XOR operation can be written as $Input_1 + Input_2$. This also means that the output will be 1 if the inputs are different.

Table 5.19 Truth table of XOX gate

$Input_1$	$Input_2$	Output
0	0	0
0	1	1
1	0	1
1	1	0

XNOR gate The XNOR gate is represented using the symbol. In addition, it is known as the negated XOR or NO

These gates work only on the logic that comprises either a 0 or 1. There are different types of Boolean gates, each of which follows a different set of rules. In this section, we will discuss Boolean gates and see how Boolean expressions can be represented using these gates in digital electronics.

AND gate The AND gate is represented with the  symbol. This gate accepts two inputs and gives an output. The output will be 1 if and only if both the inputs are 1, and will be a 0 otherwise. The truth table of the AND gate is given in Table 5.14. The expression involving an AND operation can be written as $Input_1 \cap Input_2$, or $Input_1 \cdot Input_2$.

Table 5.14 Truth table of AND gate

$Input_1$	$Input_2$	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR gate The OR gate is represented with the  symbol. This gate accepts two inputs and gives an output. The output will be 1 if any of the two inputs is 1. That is, it gives a 0 output only when both the inputs are 0. The truth table of the OR gate is given Table 5.15. The expression involving an OR operation can be written as $Input_1 \cup Input_2$, or $Input_1 + Input_2$.

Table 5.15 Truth table of OR gate

$Input_1$	$Input_2$	Output
0	0	0
0	1	1
1	0	1
1	1	1

NOT gate The NOT gate is represented by the  symbol. Unlike the AND and OR gates, the NOT gate accepts just one input. The output produced by a NOT gate is just the opposite of the input given. For example, if the input is 1 the output is 0, and vice versa. The expression involving a NOT operation can be written as NOT Input, or as ~Input, or as Input. Table 5.16 shows the truth table of this gate.

Table 5.16 Truth table of NOT gate

Input	Output
0	1
1	0

NAND gate The NAND gate is represented using the  symbol. In addition, it is known as the negated AND or NOT AND gate. It is just the opposite of the AND gate. The NAND gate accepts two inputs and returns 0 if both the inputs are 1, else returns 1. The truth table of the NAND gate is given in Table 5.17. The expression involving a NAND operation can be written as $Input_1 \mid Input_2$, or $\overline{Input_1 \cdot Input_2}$.

Table 5.17 Truth table of NAND gate

$Input_1$	$Input_2$	Output
0	0	1
0	1	1
1	0	1
1	1	0

NOR gate The NOR gate is represented using the  symbol. In addition, it is known as the negated OR or NOT OR gate. It is just the opposite of the OR gate. The NOR gate accepts two inputs and returns 1 only if both the inputs are low (0). The truth table of the NOR gate is given in Table 5.18. The expression involving a NOR operation can be written as $Input_1 \downarrow Input_2$, or $\overline{Input_1 + Input_2}$.

Table 5.18 Truth table of NOR gate

$Input_1$	$Input_2$	Output
0	0	1
0	1	0
1	0	0
1	1	0

XOR gate The XOR gate is represented using the symbol. The X in the XOR gate stands for *exclusive*. This means that the output from this gate will be 1 if only one of the inputs is a 1, but not both. Notice in Table 5.19 that the output is a 0 if both the inputs are 1 or 0. In general, the output of the XOR gate is 1 when the number of its high inputs is odd. The expression involving an XOR operation can be written as $Input_1 + Input_2$. This also means that the output will be 1 if the inputs are different.

Table 5.19 Truth table of XOX gate

$Input_1$	$Input_2$	Output
0	0	0
0	1	1
1	0	1
1	1	0

XNOR gate The XNOR gate is represented using the symbol. In addition, it is known as the negated XOR or NO

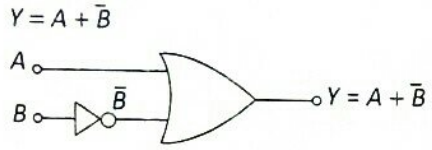
XOR gate and is just the opposite of the XOR gate. The XNOR gate accepts two inputs and returns 1 only if both the inputs are same and a 0 otherwise. The expression involving an XNOR operation can be written as $\overline{\text{Input}_1 \oplus \text{Input}_2}$. The truth table of the XNOR gate is given in Table 5.20.

Table 5.20 Truth table of XNOR gate

Input_1	Input_2	Output
0	0	1
0	1	0
1	0	0
1	1	1

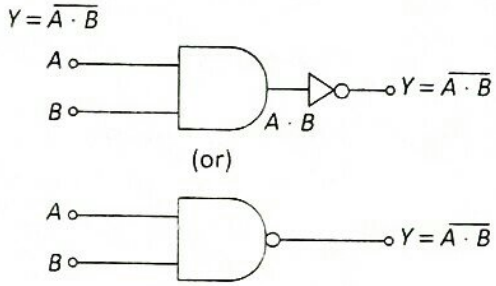
Example 5.3 Represent $Y = A + \bar{B}$ using logic gates.

Solution



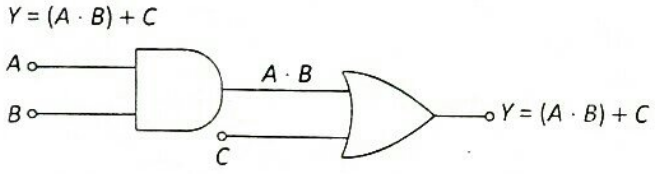
Example 5.4 Represent $Y = \overline{A \cdot B}$ using logic gates.

Solution



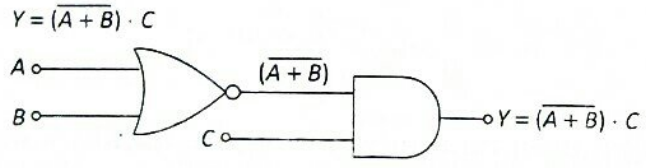
Example 5.5 Represent $Y = (A \cdot B) + C$ using logic gates.

Solution



Example 5.6 Represent $Y = (\overline{A + B}) \cdot C$ using logic gates.

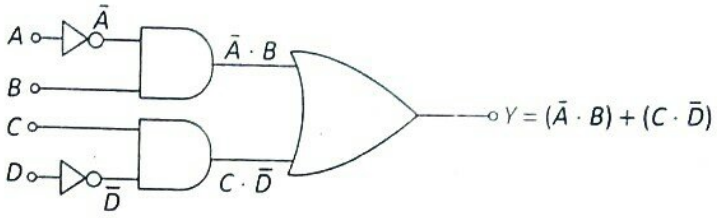
Solution



Example 5.7 Represent $Y = (\bar{A} \ B) + (C \cdot \bar{D})$ using logic gates.

Solution

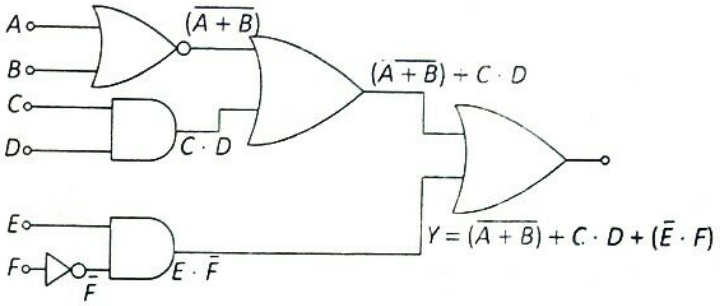
$Y = (\bar{A} \cdot B) + (C \cdot \bar{D})$



Example 5.8 Represent $Y = (\overline{A + B}) + C \cdot D + (E \cdot \bar{F})$ using logic gates.

Solution

$Y = (\overline{A + B}) + C \cdot D + (\bar{E} \cdot F)$



5.7 LOGIC DIAGRAMS AND BOOLEAN EXPRESSIONS

A logic diagram is a pictorial representation of a combination of logic gates to specify a Boolean expression. Therefore, we can directly obtain the Boolean expression by analysing the circuit diagram or vice versa.

Let us first convert a logic circuit diagram to a Boolean expression. In order to do this, first list the inputs at the correct place and then process the inputs through the gates by considering one gate at a time. Write the output of each gate. This will give you the resulting Boolean expression of each of the gates. For example, consider the logic circuit diagram given in Figure 5.9. Let us use this diagram to obtain its corresponding Boolean expression.

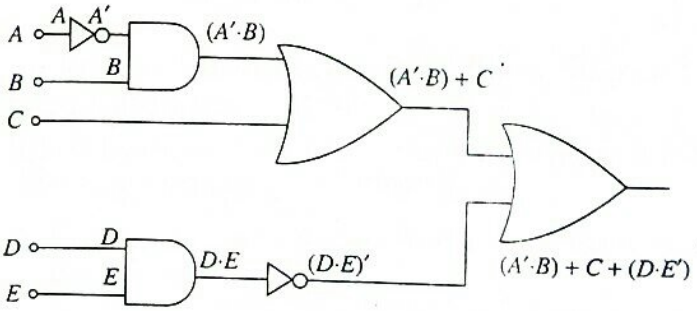


Fig. 5.9 Conversion of logic circuit diagram to Boolean expression

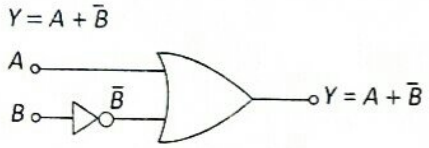
XOR gate and is just the opposite of the XOR gate. The XNOR gate accepts two inputs and returns 1 only if both the inputs are same and a 0 otherwise. The expression involving an XNOR operation can be written as $\overline{\text{Input}_1 \oplus \text{Input}_2}$. The truth table of the XNOR gate is given in Table 5.20.

Table 5.20 Truth table of XNOR gate

Input_1	Input_2	Output
0	0	1
0	1	0
1	0	0
1	1	1

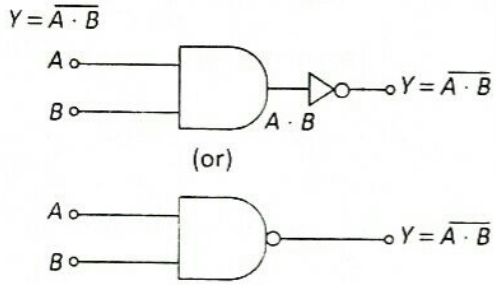
Example 5.3 Represent $Y = A + \bar{B}$ using logic gates.

Solution



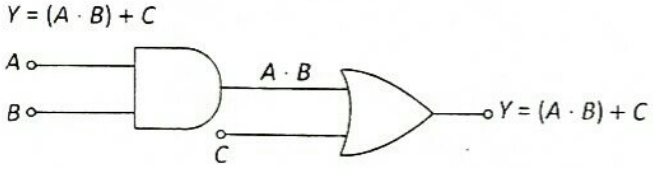
Example 5.4 Represent $Y = \overline{A \cdot B}$ using logic gates.

Solution



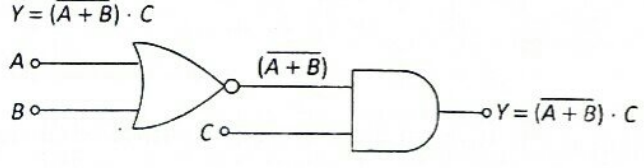
Example 5.5 Represent $Y = (A \cdot B) + C$ using logic gates.

Solution



Example 5.6 Represent $Y = (\overline{A + B}) \cdot C$ using logic gates.

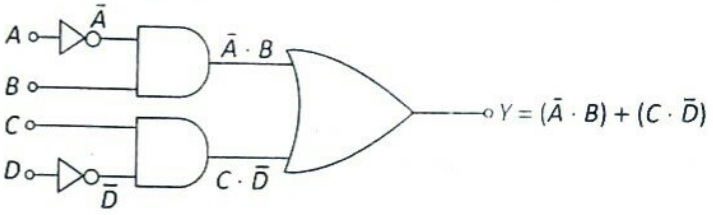
Solution



Example 5.7 Represent $Y = (\bar{A} \quad B) + (C \cdot \bar{D})$ using logic gates.

Solution

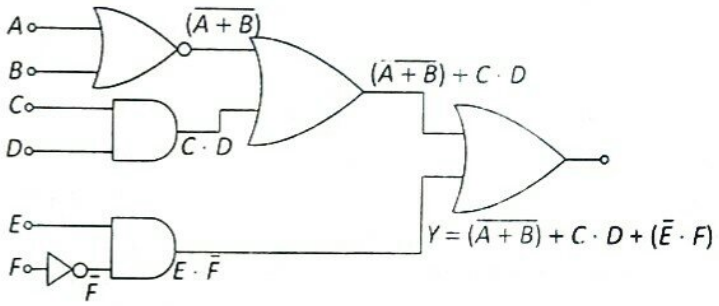
$Y = (\bar{A} \cdot B) + (C \cdot \bar{D})$



Example 5.8 Represent $Y = (\overline{A + B}) + C \cdot D + (E \cdot \bar{F})$ using logic gates.

Solution

$Y = (\overline{A + B}) + C \cdot D + (\bar{E} \cdot F)$



5.7 LOGIC DIAGRAMS AND BOOLEAN EXPRESSIONS

A logic diagram is a pictorial representation of a combination of logic gates to specify a Boolean expression. Therefore, we can directly obtain the Boolean expression by analysing the circuit diagram or vice versa.

Let us first convert a logic circuit diagram to a Boolean expression. In order to do this, first list the inputs at the correct place and then process the inputs through the gates by considering one gate at a time. Write the output of each gate. This will give you the resulting Boolean expression of each of the gates. For example, consider the logic circuit diagram given in Figure 5.9. Let us use this diagram to obtain its corresponding Boolean expression.

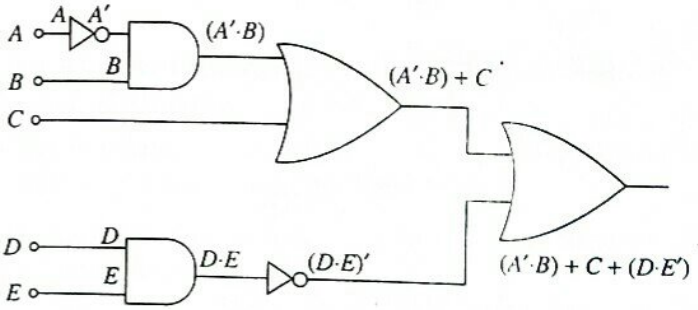


Fig. 5.9 Conversion of logic circuit diagram to Boolean expression

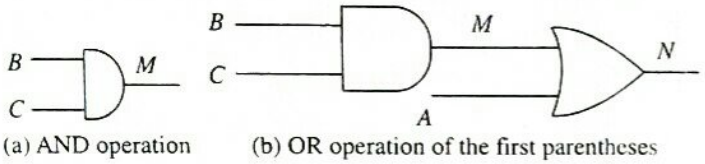
Converting a Boolean expression into its equivalent logic diagram is a little difficult, as it requires a good understanding of the order of operations, which is as follows:

- Parentheses
- NOT
- AND
- OR

Now, let us draw the circuit diagram for the Boolean expression $Y = (A + B \cdot C) + (B' + C)$ (Figure 5.10). First, consider the parentheses $(A + B \cdot C)$. Within this parentheses, we will first perform the AND operation followed by the OR operation.

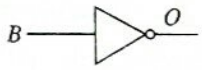
Now, consider the second parentheses $(B' + C)$. Within this parentheses, we will first perform the NOT operation followed by the OR operation.

Finally, the OR operation between the two parentheses is performed in Figure 5.10(e). Note that in Figure 5.10(f), the same figure has been redrawn with single inputs of each input variable.

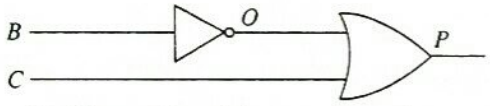


(a) AND operation

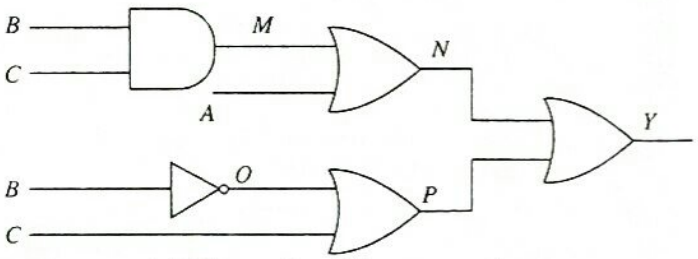
(b) OR operation of the first parentheses



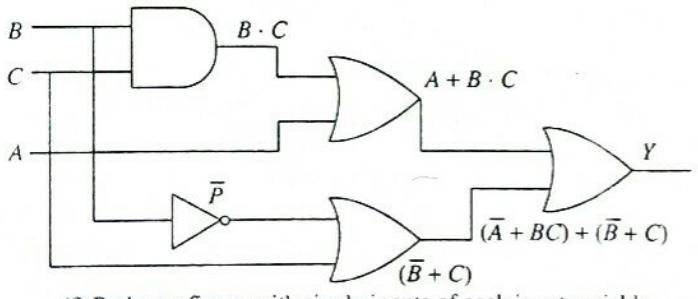
(c) NOT operation



(d) OR operation of the second parentheses



(e) OR operation of the two parentheses



(f) Redrawn figure with single inputs of each input variable

Fig. 5.10 Conversion of Boolean expression into logic diagram

5.8 UNIVERSAL GATES

Universal gates are those gates that can be used for implementing any gate such as AND, OR, and NOT. The two important universal gates are the NAND and NOR gates.

5.8.1 NAND Universal Gate

Let us first see how the basic gates can be represented using a NAND gate.

NOT Gate

A NOT gate is made by joining the inputs of a NAND gate (Figure 5.11).

Fig. 5.11 Representation of a NOT gate using a NAND gate

AND Gate

An AND gate can be implemented by following a NAND gate with a NOT gate to get a NOT NAND, that is, AND output (Figure 5.12).

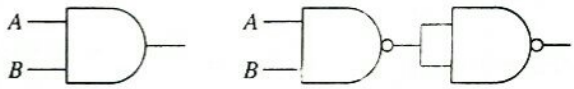


Fig. 5.12 Representation of an AND gate using a NAND gate

OR Gate

An OR gate can be implemented using a NAND gate as shown in Figure 5.13.

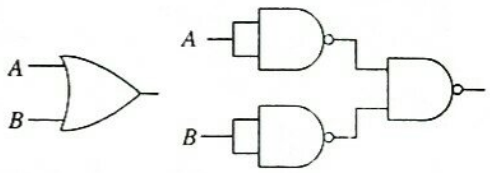


Fig. 5.13 Representation of an OR gate using a NAND gate

Conversion of Boolean Expression into Circuit Diagram using NAND Gates

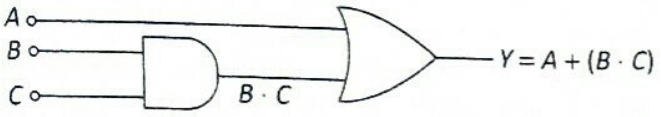
The following are the steps to convert a given expression into a circuit diagram using only NAND gates:

- Read the expression carefully and draw its corresponding circuit diagram.
- Replace every AND, OR, and NOT gate with its equivalent NAND gate.
- Redraw the circuit.

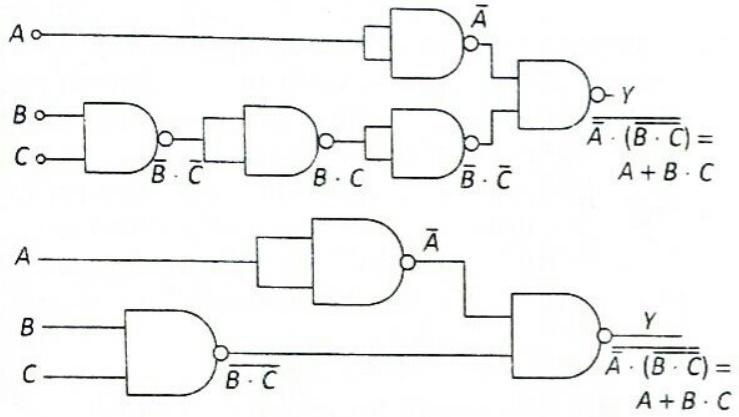
- Identify and remove double inversions (i.e., back-to-back inverters, if any).
- Redraw the final circuit.

Example 5.9 $Y = A + (B \cdot C)$ has been represented using AND and OR gates. Represent it using NAND gates.

$Y = A + (B \cdot C)$



Solution



5.8.2 NOR Universal Gate

Let us see now how the basic gates can be represented using a NOR gate.

NOT Gate

The NOT gate is implemented by joining the inputs of a NOR gate. This is because a NOR gate is equivalent to an OR gate leading to a NOT gate (Figure 5.14).

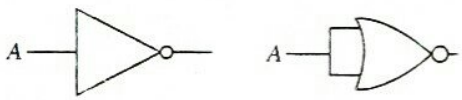


Fig. 5.14 Representation of a NOT gate using a NOR gate

OR Gate

The OR gate is a NOR gate followed by a NOT gate (Figure 5.15).

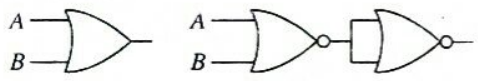


Fig. 5.15 Representation of an OR gate using a NOR gate

AND Gate

From the truth tables, we can observe that an AND gate gives the output as 1 when both inputs are 1, whereas a NOR gate outputs 1 only when both inputs are 0. This means that an AND gate can be implemented by inverting the inputs to a NOR gate (Figure 5.16).

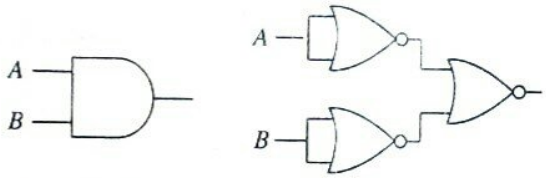


Fig. 5.16 Representation of an AND gate using a NOR gate

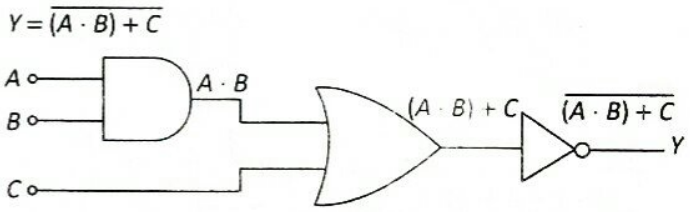
Conversion of Boolean Expression into Circuit Diagram using NOR Gates

The following are the steps to convert a given expression into a circuit diagram using only NOR gates:

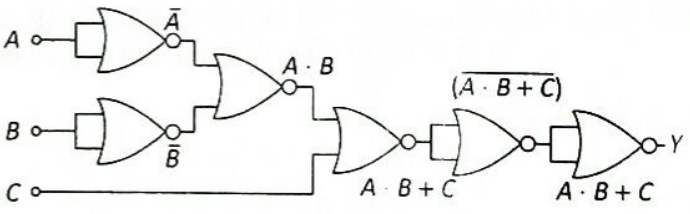
- Read the expression carefully and draw its corresponding circuit diagram.
- Replace every AND, OR, and NOT gate with its equivalent NOR gate.
- Redraw the circuit.
- Identify and remove double inversions (i.e., back-to-back inverters, if any).
- Redraw the final circuit.

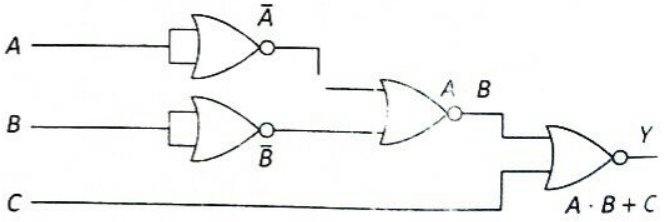
Example 5.10 $Y = (A \cdot B) + C$ has been represented using AND, OR, and NOT gates. Represent it using the NOR universal gate.

$Y = \overline{(A \cdot B) + C}$



Solution





5.9 SIMPLIFICATION OF BOOLEAN EXPRESSIONS USING KARNAUGH MAP

Karnaugh map (K-map) is a tool to represent Boolean functions of up to six variables. It is a pictorial method to minimize Boolean expressions without using Boolean algebra theorems and equations, thereby making the process of minimization simpler, faster, and efficient.

An n-variable K-map has 2^n cells where each cell corresponds to an n-variable truth table value. The cells in a K-map are arranged in such a way that adjacent cells correspond to truth rows that differ in position by only one bit (logical adjacency).

K-maps can be easily used to represent expressions that involve two to four variables, expressions with five to six variables are comparatively difficult but achievable, but expressions with seven or more variables are extremely difficult (if not impossible) to minimize using a K-map.

Consider the truth table of two variables and observe the k-map for this truth table (Table 5.21). A cell of the K-map contains a 1 if the output value for A and B in the truth table has a 1.

Let us consider the truth tables of three (Table 5.22) and four (Table 5.23) variables and observe the K-map for these tables.

Table 5.21 Truth table and K-map for two variables

A	B	C	Output Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0

A \ B	0	1
0	1	1
1	0	0

Table 5.22 Truth table and K-map for three variables

A	B	C	Output Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

A \ BC	00	01	11	10
0	1	1	0	0
1	0	1	0	1

Table 5.23 Truth table and K-map for four variables

A	B	C	D	Output Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

AB \ CD	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	0	1	0	0
10	0	1	0	0

Rules for Minimization

The following are the rules for minimizing an expression using a K-map.

Rule I Form groups of cells in such a way that a group should not include a cell with a zero value.

A \ B	0	1
0	1	1
1	0	0

A \ B	0	1
0	1	1
1	0	0

Wrong. This is not a group.

Rule II Groups may be formed either horizontally or vertically. They cannot be formed diagonally.

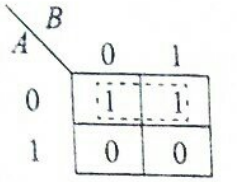
A \ B	0	1
0	1	1
1	0	0

A \ B	0	1
0	0	1
1	0	1

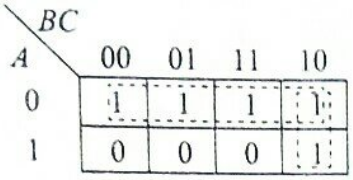
A \ B	0	1
0	0	1
1	1	0

Wrong: This is not a group.

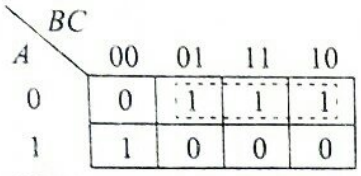
Rule III Groups should contain one, two, four, eight, or generally 2^n cells.



Group of two 1s

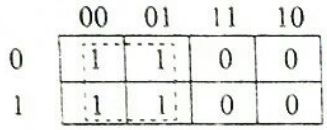


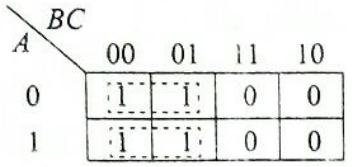
Group of four 1s



Group of three 1s is wrong.

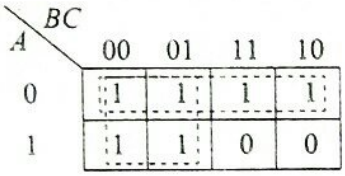
Rule IV Groups must be as large as possible.



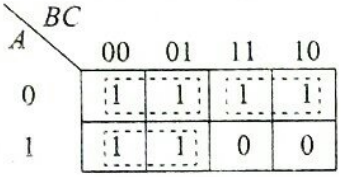


Wrong: Groups should be as large as possible

Rule V Groups may overlap.

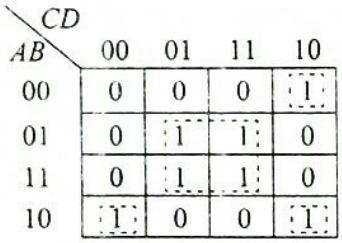


Overlapping groups



Wrong: Non-overlapping groups

Rule VI Groups can be wrapped around the table. For example, the leftmost cell in a row can be grouped with the rightmost cell. Similarly, the topmost cell in the table may be grouped with the bottommost cell in the group.



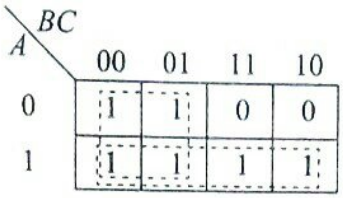
Topmost cell

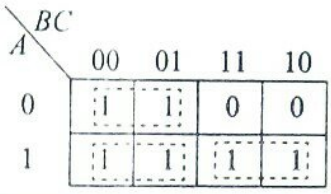
Rightmost cell

Leftmost cell

Bottommost cell

Rule VII The number of groups should be as small as possible (provided any of the earlier-stated rules are not violated).

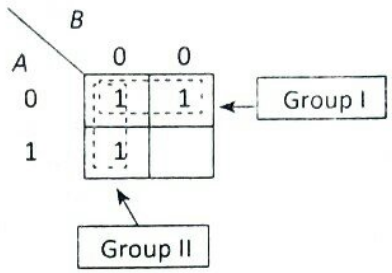




Wrong: Too many groups.

Example 5.11 Consider the expression $Y = A' \cdot B' + A \cdot B' + A' \cdot B$. Plot a K-map to minimize the expression.

Solution



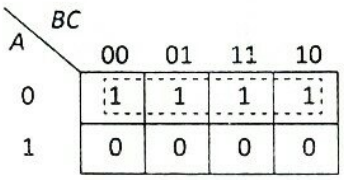
In group I, there is an entry 1 when $A = 0$ and $B = 0$ and when $A = 0$ and $B = 1$, which means that the output is 1 when $A = 0$ or $A' = 1$. That is, the output is independent of the input value of B.

Similarly, in group II, there is a 1 when $A = 0$ and $B = 0$ and when $A = 1$ and $B = 0$, which means that the output is 1 when $B = 0$ or $B' = 1$. Thus, the output is independent of the input value of A. Hence, the minimized form of this expression can be given as $A' + B'$.

Therefore, $Y = A' + B'$.

Example 5.12 Consider the expression $Y = A' \cdot B' \cdot C' + A' \cdot B' \cdot C + A' \cdot B \cdot C + A' \cdot B \cdot C'$ and minimize it using a K-map.

Solution

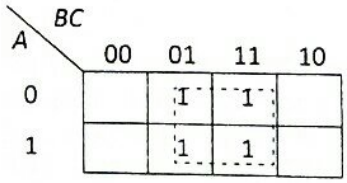


In the group of 1s, we see that the output is independent of the values of B and C. This is because there is a 1 when $B = 0$ as well as when $B = 1$. Similarly, there is a 1 when $C = 0$ as well as when $C = 1$. However, the output is 1 only when $A = 0$ or $A' = 1$. Therefore, $Y = A'$.

Example 5.13 Consider the expression $Y = A' \cdot B' \cdot C + A' \cdot B \cdot C + A \cdot B' \cdot C + A \cdot B \cdot C$ and minimize it using a K-map.

Solution

The group of 1s is independent of the value of A (as output is 1 when $A = 0$ or $A = 1$). Similarly, the output is also independent of the value of B. However, the output is 1 when the value of C is 1. Hence, the result is $Y = C$.



Example 5.14 Consider the expression $Y = A' \cdot B' \cdot C' + A' \cdot B + A \cdot B \cdot C' + A \cdot C$ and minimize it using K-map.

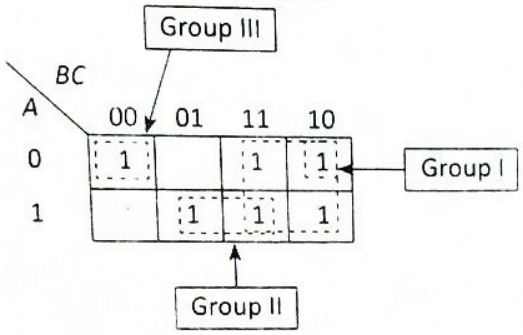
Solution

$A' \cdot B' \cdot C' + A' \cdot B + A \cdot B \cdot C' + A \cdot C$ can be expanded as,

$A' \cdot B' \cdot C' + A' \cdot B \cdot (C + C') + A \cdot B \cdot C' + A \cdot C \cdot (B + B')$

$A' \cdot B' \cdot C' + A' \cdot B \cdot C + A' \cdot B \cdot C' + A \cdot B \cdot C' + A \cdot C \cdot B + A \cdot C \cdot B'$.

Hence, the K-map can be drawn as



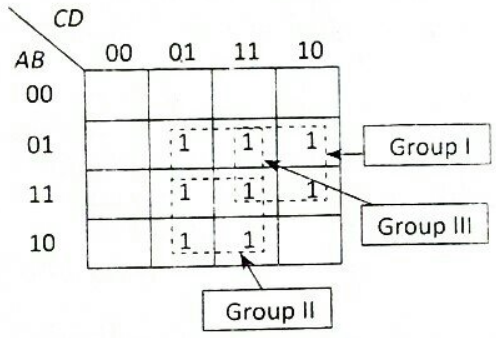
In group I, the output is independent of C and A as whether the value of A or C is 0 or 1 the output is always 1. So, the first group minimizes to B.

In group II, the output is independent of B. Rather, the output is 1 when $A = 1$ and $C = 1$. Therefore, group II minimizes to AC.

In group III, the output is independent of B. The output is a 1 when $A = 0$ ($\bar{A} = 1$) and $C = 0$ when $\bar{C} = 1$. Therefore, group III minimizes to $\bar{A}\bar{C}$. Hence, $Y = B + AC + \bar{A}\bar{C}$.

Example 5.15 Consider the expression $Y = \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + AB\bar{C}\bar{D} + ABCD + ABC\bar{D} + A\bar{B}\bar{C}D + A\bar{B}CD + \bar{A}BCD$ and minimize it using K-map.

Solution



In group I, the output is independent of A and D. The output is 1 when both B and C are equal to 1. Hence, group I minimizes to BC.

In group II, the output is independent of B and C. The output is 1 when $A = 1$ and $D = 1$. Hence, group II minimizes to AD.

In group III, the output is independent of C. The output is 1, when $A = 0$, $B = 1$ and $D = 1$. Therefore, group III minimizes to $\bar{A}BD$.

Hence, $Y = BC + AD + \bar{A}BD$

5.10 ADDER CIRCUITS

An adder or summer is a combinational digital circuit that is used to perform addition of numbers. A typical adder circuit generates a sum bit and a carry bit as the output.

In many computers and other types of processors, adders are present not only in the arithmetic and logic unit (ALU) but also in other parts to calculate addresses and table indices and to perform similar operations.

While adders can be constructed for decimal numbers, binary coded decimal (BCD) numbers, or excess 3 codes, they are most commonly used with binary numbers. In this section, we will discuss three types of adder circuits—half adder, full adder, and multi-bit (or parallel) adder.

5.10.1 Half Adder

A half adder is a simple, combinational arithmetic circuit constructed with just two logic gates. It is used to add two 1-bit numbers and produces S and C as the output.

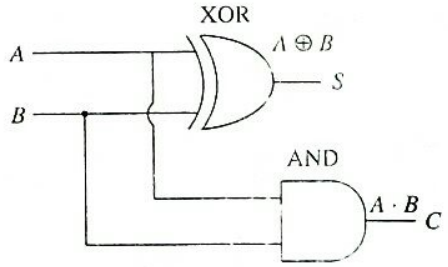
A half adder can be easily constructed with two logic gates—one AND gate and one XOR gate. Given input bits A and B, the sum bit (S) is the XOR of A and B, and the carry bit (C) is the AND of A and B. This is evident from the truth table of the half adder. The truth table, schematic representation, and the circuit diagram for a half adder are given in Figure 5.17.

Although the half adder is the simplest of all adder circuits, a major disadvantage is that it is able to add only two input bits (A and B) at a time. A half adder circuit ignores the carry if there is any in the input and adds only the A and B bits. This means that the binary addition process is not complete, which is why it is called a half adder.

Input		Output	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

(a) Truth table

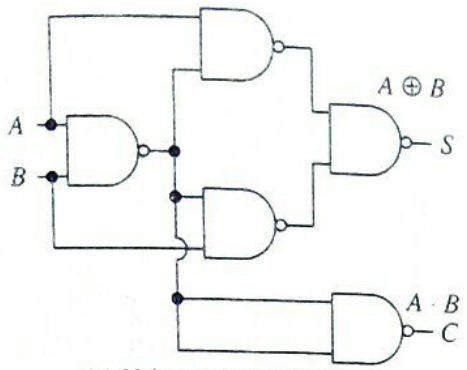
(b) Schematic representation



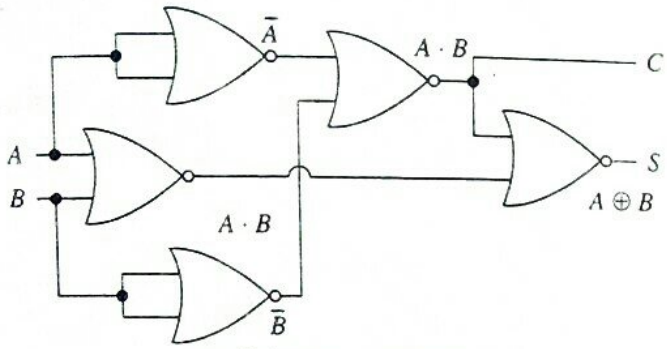
(c) Circuit diagram

Fig. 5.17 Half adder

We have learnt that NAND gates and NOR gates are universal gates that can be used to design any circuit. Figure 5.18 shows the implementation of a half adder circuit using only NAND gates and only NOR gates.



(a) Using only NAND logic



(b) Using only NOR logic

Fig. 5.18 Half adder

5.10.2 Full Adder

A full adder is slightly more complicated to implement than a half adder. The half adder has two inputs and two outputs; on the other hand, the full adder has three inputs (A, B, and an input carry C_{in}) and two outputs (S and output carry C_{out}). The truth table of a full adder is given in Table 5.24.

A full adder circuit is usually implemented with the help of two half adder circuits. The first half adder circuit will add A and B to produce a partial sum. The second half adder circuit will add C_{in} to the partial sum (produced by the first half adder) to get the final sum S. If any of the half adder circuit produces a carry, a C_{out} will then be generated. Thus, C_{out} can be implemented as an OR function of the half adder carry outputs. Figure 5.19 shows the implementation of a full adder circuit.

The schematic representation or the symbol of a full adder is shown in Figure 5.20.

Table 5.24 Truth table of a full adder

Input			Output	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 5.21 shows the implementation of a full adder circuit using only NAND gates and only NOR gates.

5.10.3 Ripple Carry Adder

The concept of a single-bit full adder can be extended to add two multi-bit numbers. This can be done easily when multiple full adder circuits are cascaded in parallel to add an N-bit number. Basically, for an N-bit parallel adder, N numbers of full adder circuits must be connected. Each bit must be represented by a full adder and must be added simultaneously. Therefore, to add two 8-bit numbers, we need eight full adders.

A ripple carry adder is a logic circuit in which the carry out of each full adder becomes the carry in of the next (consecutive) full adder. The adder got its name because each carry bit gets rippled into the next stage.

Moreover, in a ripple carry adder, the sum and carry out bits of a half adder circuit are not considered valid until the carry in of that stage occurs. This is due to propagation delay occurring inside the logic circuitry. Propagation delay is calculated as the time elapsed between the application of an input and the occurrence of the corresponding output. In case of ripple carry adders, it is calculated as the time elapsed between the application of the carry in and the occurrence of the carry out.

In a real circuit, propagation delays are common, and logic gates take time to switch states. Though it takes only

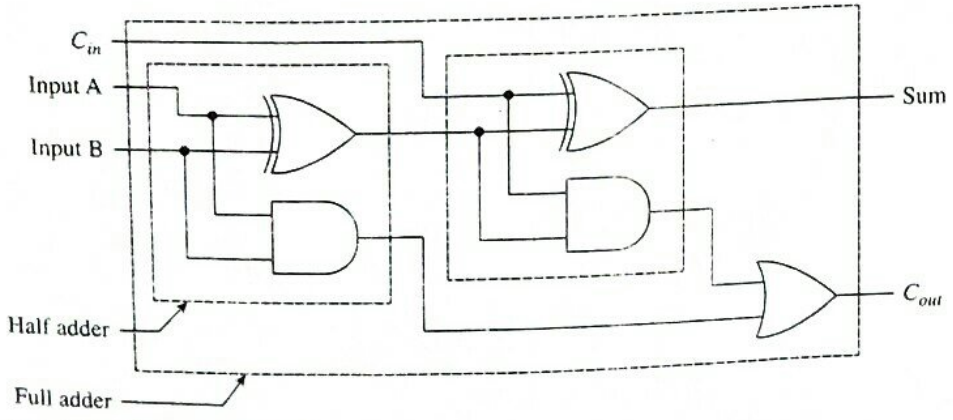


Fig. 5.19 Full adder circuit

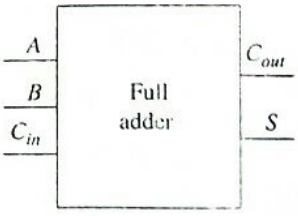
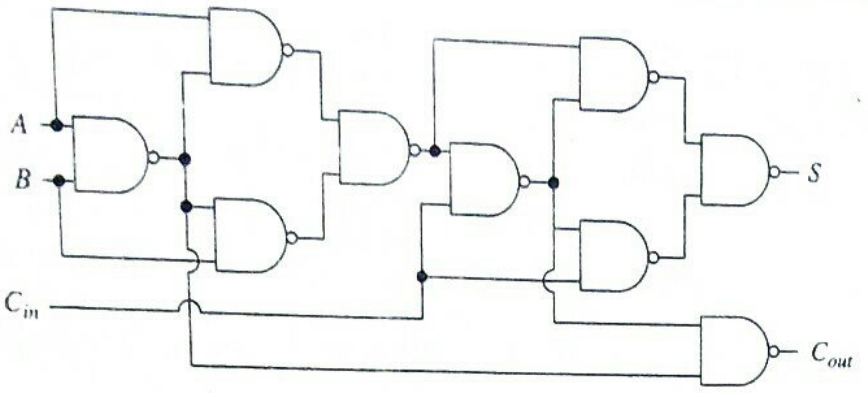
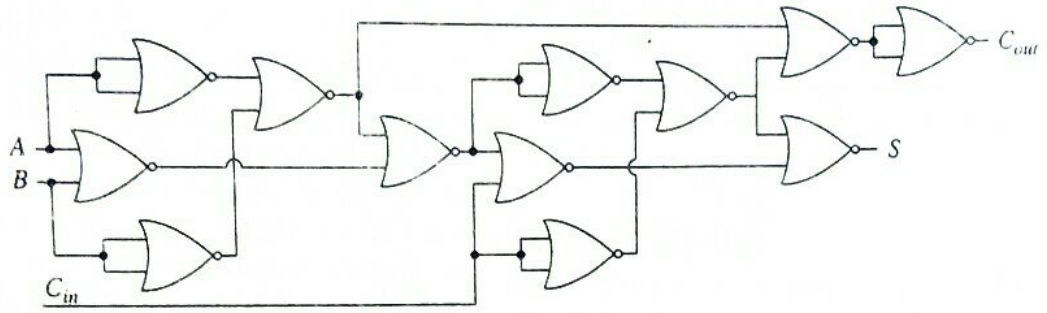


Fig. 5.20 Schematic representation of a full adder



(a) Using only NAND logic



(b) Using only NOR gate

Fig. 5.21 Full adder circuit

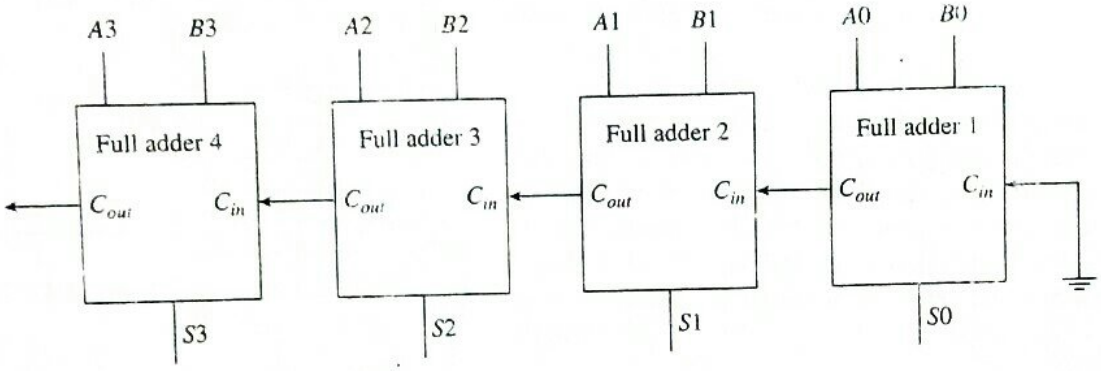


Fig. 5.22 4-bit ripple carry adder

a few nanoseconds to switch states, in high-speed computers even nanoseconds matter. Since 32-bit or 64-bit ripple carry adders usually take 100–200 nanoseconds to generate the final sum because of carry ripple, more advanced adders called carry-lookahead adders have been developed. Such adders require more numbers of gates for implementation but the settling time is much better. The circuit diagram of a 4-bit ripple carry adder is depicted in Figure 5.22.

In this figure, sum out *S*0 and carry out C_{out} from full adder 1 is valid only after the propagation delay of full adder 1.

5.11 FLIP-FLOPS

A flip-flop or latch forms the fundamental building block of digital electronics systems used in computers, communications, and many other types of systems. Flip-flops, also called bistable gates, are digital logic circuits that have two stable states. They can be in one of the two states to store d elements. Flip-flops maintain their state until an input pu called a *trigger* is received. Once the trigger is received, flip-flop outputs may change state and remain in those sta until another trigger is received.

A basic flip-flop may have zero, one, or two input sign as well as a clock signal and an output signal. Some flip-flo also include a clear input signal to reset the current outp There are two types of flip-flops (Figure 5.23).

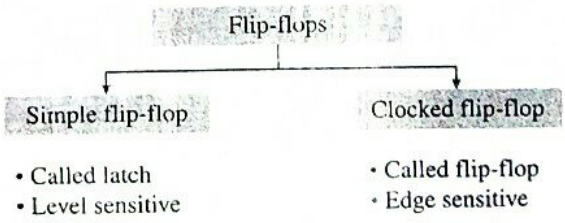


Fig. 5.23 Classification of flip-flops

Simple flip-flops, commonly known as latches, are used to store data. A latch is level sensitive; that is, when a latch is enabled, it becomes transparent.

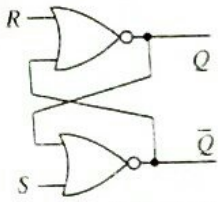
Clocked, synchronous, or edge-triggered flip-flops are commonly known as flip-flops. They are edge sensitive, which means that the output changes only on a single type (positive going or negative going) of clock edge.

In this section, we will look at different types of flip-flop circuits such as T (toggle), SR (set–reset), JK (Jack Kilby), and D (delay).

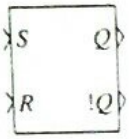
5.11.1 SR Flip-flop

The SR flip-flop, also known as the SR latch, is one of the most basic sequential logic circuits. It is an arrangement of logic gates that maintains a stable output even after the inputs are turned off.

The SR flip-flop is a 1-bit memory bistable device that has two inputs—S (set) to set the flip-flop giving an output 1 and R (reset) to reset the flip-flop back to its original state giving an output 0. The SR flip-flop has two outputs—Q and $!Q$ (as shown in the truth table in Figure 5.24) where Q_{n-1} is the output at the previous time step. The circuit diagram (constructed using NOR gates), symbol, and truth table of an SR flip-flop are shown in Figure 5.24.



(a) Circuit diagram

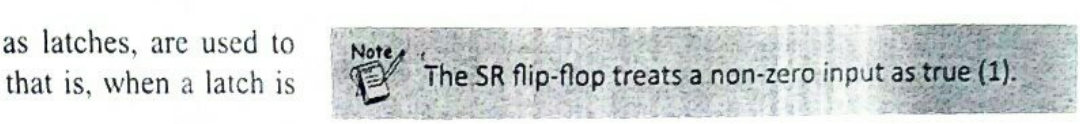


(b) Symbol

S	R	Q_n	$!Q_n$
0	0	Q_{n-1}	$!Q_{n-1}$
0	1	0	1
1	0	1	0
1	1	0	0

(c) Truth table

Fig. 5.24 SR flip-flop



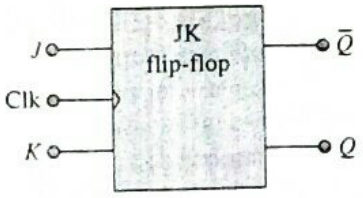
The following can be observed from the truth table of the SR flip-flop:

- When input $S = 1$ and $R = 0$, the flip-flop goes to the set state (Q_n is 1).
- When input $R = 1$ and $S = 0$, the flip-flop goes to the reset state (Q_n is 0).
- When $S = R = 0$, the flip-flop stays in the previous state (Q_n is Q_{n-1}).
- When $S = R = 1$, the state is undefined and must be avoided. It is clear from the output that Q can never be equal to its complement.

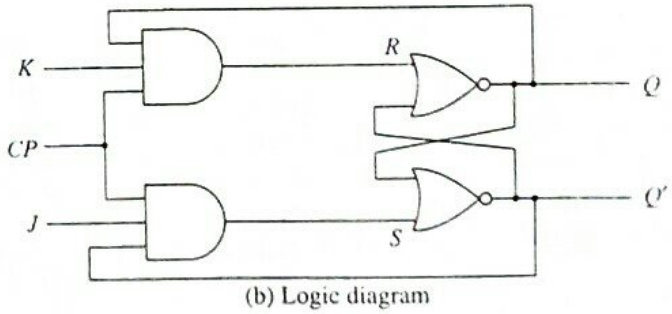
5.11.2 JK Flip-flop

The JK flip-flop is the most widely used flip-flop and is considered a versatile universal flip-flop circuit. The operation of the JK flip-flop is identical to that of the SR flip-flop. The only difference is the JK flip-flop has no invalid or forbidden input states of the SR flip-flop (when $S = R = 1$). In this flip-flop, the intermediate state is more refined and precise than in the SR flip-flop.

Hence, we can say that the JK flip-flop is a refinement of the SR flip-flop. Similar to the SR flip-flop, the JK flip-flop also has two inputs—J (set) and K (reset or clear). In addition to these inputs, it has a clock input to prevent the illegal or invalid output condition that can occur when both inputs are equal to 1. The symbol, logic diagram, characteristic table, and characteristic equation of a JK flip-flop are shown in Figure 5.25.



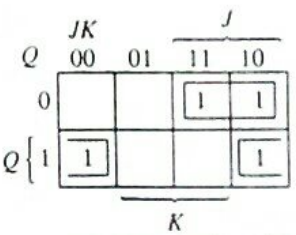
(a) Symbol



(b) Logic diagram

Q	J	K	$Q(t+1)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(c) Characteristic table



$Q(t+1) = JQ' + K'Q$

(d) Characteristic equation

Fig. 5.25 JK flip-flop

From the circuit diagram, we can observe the following points:

- A JK flip-flop is an SR flip-flop with two 3-input AND gates.
- The output Q of the flip-flop is returned as a feedback to the input of the first AND gate along with other inputs (such as K and clock pulse CP). The output Q is AND'ed with the input K and the clock pulse CP. If $CP = 1$ and $Q = 1$, then the flip-flop gets a CLEAR signal.
- The output Q' of the flip-flop is given as a feedback to the input of the second AND along with other inputs such as J and clock pulse CP. Q' is AND'ed with the input J and the clock pulse CP. If $CP = 1$ and $Q' = 1$, the flip-flop gets a SET signal.

From the characteristic table of the JK flip-flop, we can observe the following:

- When $J = K = 1$, the flip-flop switches to the complement state. That is, if $Q = 1$, it switches to $Q = 0$, and if $Q = 0$, it switches to $Q = 1$.
- When $J = K = 0$, there will be no change in the output of the flip-flop. It will remain the same as its previous value and even CP will have no effect over it. This is because the output of the two AND gates will be 0.
- When $J = 0$ and $K = 1$, the flip-flop will be reset, as the output of the AND gate corresponding to the input J becomes 0 (same as when $S = 0$ and $R = 1$).
- When $J = 1$ and $K = 0$, the flip-flop will be set, as the output of the AND gate corresponding to K becomes 0 (same as when $S = 1$ and $R = 0$).

Note When $CP = J = K = 1$, the output will be complemented repeatedly until the clock pulse goes back to 0. Since this condition is undesirable, it must be eliminated by using an edge triggering of JK flip-flop or by using master–slave JK flip-flops.

5.11.3 T Flip-flop

A T flip-flop is an edge-triggered device that toggles state every time it is triggered if the T input is asserted; otherwise, it holds the current output. The characteristic table, symbol, and circuit of a T flip-flop are shown in Figure 5.26.

The T flip-flop characteristic table has three columns; the first column is the value of the control input T, the second column is the *current state* (current value being output by Q), and the third column is the *next state*, that is, the value of Q at the next positive edge.

From the truth table, the characteristic equation of a T flip-flop can be derived as

$$Q_{next} = T \oplus Q = T\bar{Q} + \bar{T}Q$$

The T flip-flop has one input T in addition to the clock pulse CP. When T is 0, the next output will be the same as the previous output. This means at $T = 0$, the flip-flop does a hold. However, when $T = 1$, the next output state will be the inverse of the present state, thereby indicating that at $T = 1$, the flip-flop does a toggle. The next output is available at the next positive edge of the clock.

Thus, in a T flip-flop, either the current state's value can be maintained for another cycle or the value can be toggled (negated) at the next clock edge

From the circuit diagram of the T flip-flop, we can see that it is a single-input version of the JK flip-flop. The T flip-flop can be obtained from the JK type if both inputs are tied together.

When the control input T is 1, the T flip-flop divides the clock frequency by two. That is, if the clock frequency is 4 MHz, the output frequency from the flip-flop will be 2 MHz. This 'divide-by' feature makes the T flip-flop a good candidate for constructing various types of digital counters, frequency dividers, and general binary addition devices.

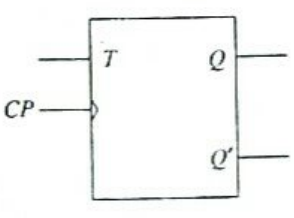
5.11.4 D Flip-flop

In the D flip-flop, the term D stands for *data*. The flip-flop is called so because it is used to store the value on the data line. It can be thought of as a basic memory cell. D flip-flops are also used for making shift registers, which are an essential part of many electronic devices.

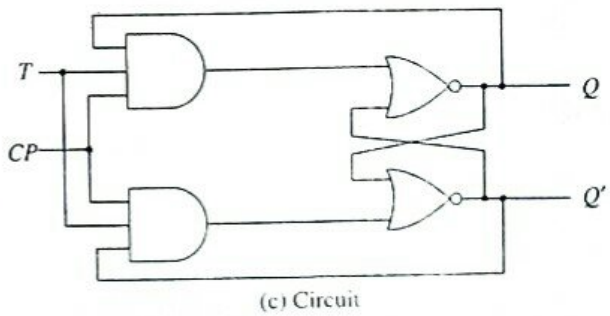
A D flip-flop can be constructed using an SR flip-flop by connecting the input S and R through an inverter to allow for a single D (data) input. The circuit of the clocked D flip-flop ensures that the inputs S and R are never equal to 1 at the same time. The D input is used for the set (S) input and the inverter is used to generate the reset (R) input. Connecting an inverter to the SR flip-flop makes the two inputs as complements of each other and thus helps avoid the ambiguity inherent in the SR flip-flop when both inputs are LOW. In the

T	Previous output Q	Next output Q_{next}	Operation
0	0	0	Hold
0	1	1	Hold
1	0	1	Toggle
1	1	0	Toggle

(a) Characteristic table



(b) Symbol



(c) Circuit

Fig. 5.26 T flip-flop

D flip-flop, unlike the SR flip-flop, both the inputs can neither be 1 nor be 0 simultaneously.

The other names of the D flip-flop are data latch, delay, or D type bistable flip-flop. The characteristic table, graphical symbol, and logic diagram of the D flip-flop are shown in Figure 5.27. According to the characteristic table, the flip-flop captures the value of the input D at the rising edge of the clock and that captured value becomes the output Q of the flip-flop. At the non-rising edge, there is no change to the output Q.

Note In the absence of the clock input, the D flip-flop's output will change on every data input, which is highly undesirable as the D flip-flop is used to retain data.

In addition to this, the D flip-flop has two possible values. When $D = 0$, the flip-flop does a reset, thereby giving an output 0. When $D = 1$, the flip-flop does a set setting the output Q to 1.

A D flip-flop is also known as a *delay* flip-flop, as there is a small amount of delay in reading the value of the control signal D. The value of D is not read immediately but only at the next positive clock edge.

5.12 APPLICATIONS OF FLIP-FLOPS

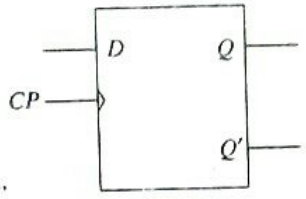
The following are some of the applications of flip-flops:

- CPU registers, which are temporary storage of data, are made of a group of flip-flops to store multiple bits of information. Each flip-flop used in the register can store a single bit of information.
- In a sequential logic, flip-flops are used to store information about the state.
- In a finite-state machine, the output and next state depend on the current input as well as on the previous input. Hence, they can be used to preserve the state of the machine (to get the previous input).
- Flip-flops can be used as a counter for counting pulses.
- They can be used for synchronizing variably timed input signals to some reference timing signal.
- Flip-flop circuits are interconnected to form the logic gates for digital integrated circuits (ICs), which are extensively used in the design of memory chips and microprocessors.

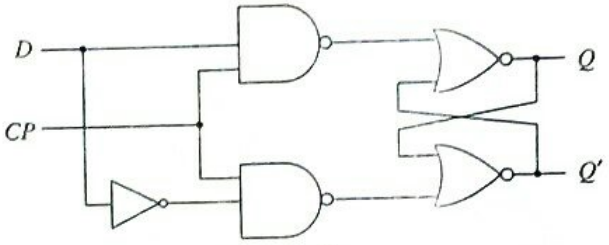
Clock	D	Q
Rising edge	0	0
Rising edge	1	1
Non-rising	X	Previous Q

Here, 'X' means a *Don't care* condition, indicating the value is insignificant.

(a) Characteristic table



(b) Graphical symbol



(c) Logic diagram

Fig. 5.27 D flip-flop

GENERATING TRIGGERS

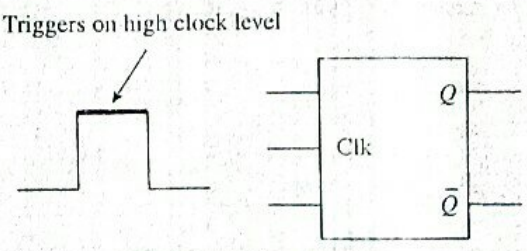
We have seen that the D flip-flop gives an output only when the flip-flop is triggered. The triggering is done when there is a small amount of change in the input, which is provided with the help of a clock pulse also known as a *trigger pulse.*

A number of flip-flops are connected to each other as sequential circuits for the design of multi-bit counters or registers. These circuits thus formed require a number of trigger pulses. A single trigger when applied to a multi-bit register makes the bit move one position.

If a clock pulse is applied to the input of the flip-flop at the same time that its output is changing, it may cause instability to the circuit because the feedback is given from the output combinational circuit to the memory elements. This problem can be solved to a certain level by making the flip-flop more sensitive to pulse transition than to pulse duration.

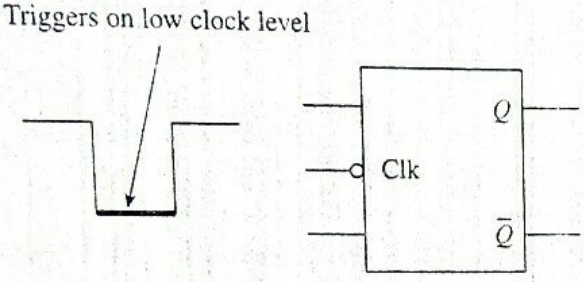
The following are the four types of pulse-triggering methods. They differ in the way the electronic circuits respond to the pulse.

High-level triggering This is done when the flip-flop is required to respond at its HIGH state. It is represented in the diagram by a straight lead from the clock input.



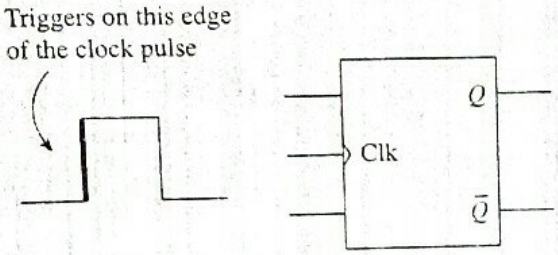
High-level triggering

Low-level triggering This is done when the flip-flop is required to respond at its LOW state. It is represented in the diagram by a clock input lead along with a low-state indicator bubble.



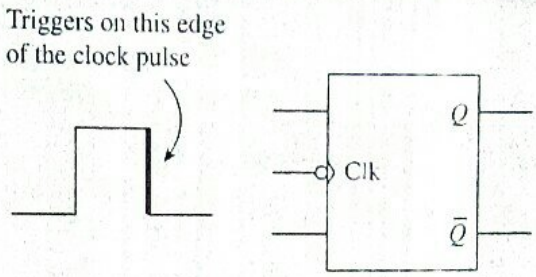
Low-level triggering

Positive edge triggering This is done when the flip-flop is required to respond at a LOW to HIGH transition state. It is represented in the diagram by a clock input lead along with a triangle.



Positive edge triggering

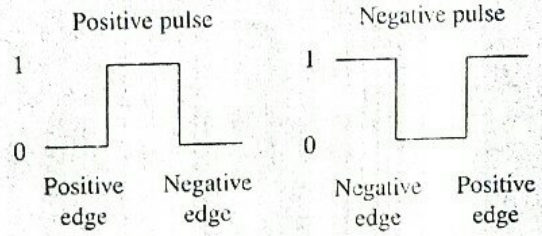
Negative edge triggering This is done when the flip-flop is required to respond during a HIGH to LOW transition state. It is represented in the diagram by a clock input lead along with a low state indicator and a triangle.



Negative edge triggering

Clock Pulse Transition

The movement of a trigger pulse is always from 0 to 1 and then 1 to 0 of a signal. This implies that there are two transitions in a single signal. When the trigger pulse moves from 0 to 1 it is called a positive transition, and when it moves from 1 to 0 it is called a negative transition. This is shown in the following figure:



Definition of clock pulse transition

In general, clocked flip-flops are triggered during the 0 to 1 transition of the trigger pulse, and the state transition starts as soon as the pulse reaches the HIGH level.

SUMMARY

- Computers understand only binary numbers, in which only the digits 0 and 1 are used. A complex proposition can be easily stated in terms of binary variables (two variables) and binary operators (OR, AND, and NOT). It is for this purpose that George Boole developed Boolean algebra in 1854.
- Boolean operators take certain inputs and produce an output based on a predetermined table of results (also known as the *truth table*). A truth table lists all possible values of input combinations of the function and the output that will be produced based on these input combinations.
- A Venn diagram can be used to represent Boolean operations using shaded overlapping regions.
- A variable or a complemented variable is called a *literal*. Designers of logical circuits prefer to use a standardized form of literals called the *canonical form* to represent a Boolean function. This helps them to reduce the number of logic gates for simplification and minimization of digital circuits. A canonical form can be either a sum of *minterms* or a product of *maxterms*.
- An *n*-variable K-map has 2^n cells where each cell corresponds to an *n*-variable truth table value.
- A half adder is a simple, combinational arithmetic circuit constructed with just two logic gates. It is used to add two 1-bit numbers and produces a sum bit (S) and a carry bit (C) as the output.
- A full adder has three inputs (A, B, and an input carry C_{in}) and two outputs (S and C_{out}).The concept of a single-bit full adder can be extended to add two multi-bit numbers. This can be done easily when multiple full adder circuits are cascaded in parallel to add an N-bit number.
- Simple flip-flops (latches) are used to store data and are level sensitive, whereas clocked, synchronous, or edge-triggered flip-flops are edge sensitive.

) istributive Law ①

$A + (B \cdot C) = (A+B) \cdot (A+C)$

RHS -

$(A+B) \cdot (A+C)$ Multiplying the brackets.

$A \cdot A + A \cdot B + A \cdot C + B \cdot C$ $(A \cdot A = A)$

$A + A \cdot B + A \cdot C + B \cdot C$

$A \cdot (1+B) + A \cdot C + B \cdot C$ $(1+B = 1$ or $A \cdot 1 = A)$

$A + A \cdot C + B \cdot C$

$A(1+C) + B \cdot C$ $(1+C = 1$ or $A \cdot 1 = A)$

LHS - $A + (B \cdot C)$ ✗

Absorption Law

(Reduction of Complicated expression)

(1) $A + A \cdot B = A$

LHS $A + A \cdot B$ $(A = A \cdot 1)$

$A \cdot 1 + A \cdot B$

$A(1+B)$ $(1+B = 1)$

$A \cdot 1 = A = RHS$ ✓

(2) $A + A' \cdot B = A + B$

$(A + A') \cdot (A + B)$ Distributive Law: $A + B \cdot C = (A+B)(A+C)$

$1 \cdot A + B$ $(A + A' = 1)$

$A + B$ RHS ✓