**UNIT-2: BOOLEAN EXPRESSIONS AND COMBINATIONAL LOGIC CIRCUITS**

**STRUCTURE**

## 2.0 Objectives

At the end of the unit you will be able to

- Simplify Boolean expressions using algebraic method
- Describe sum of products and product of sums and convert them into canonical form
- Design karnaugh maps and use them to simplify Boolean expressions
- Implementing Boolean expressions using NAND and NOR gates
- Describe half adder, full adder, half subtractor, full subtractor, parallel binary adder and BCD adder
- Find, based on input conditions, the output of an encoder and decoder
- Determine the output of multiplexer and demultiplexer based on input conditions

## 2.1 Introduction

We have studied so far logic gates and Boolean algebra. Boolean algebra and theorems are used for the manipulations of logical expressions. It has also been seen that a logical expression can be realized by using the logic gates. The number of gates required and the number of input terminals for the implementation of a logical expression, in general, get reduced considerably if the expression can be simplified. Therefore, the simplification of logical expression is very important as it saves the hardware required to design a specific system

We know that logical expressions are implemented by connecting specific logic gates. These logic gates produce a specific output for certain specified combinations of input variables, with no storage involved. These circuits are commonly known as *combinational circuits*. In combinational circuits, the output level is always dependent on the combinations of the input levels.

The combinational circuits can be specified in one of the following ways:

- A set of statements
- Boolean expression, and
- Truth table.

In this section we will continue our study of combinational circuits and we will further study various methods of simplifications of logical circuits.

## 2.2 Simplification of Boolean Expressions:

Simplification of Boolean functions is mainly used to reduce the gate count of a design. Less number of gates means less power consumption, sometimes the circuit works faster and also when number of gates is reduced, cost also comes down. There are many ways to simplify a logic design; some of them are given below. We will be looking at each of these in detail in the next few pages.

- Algebraic Simplification.
  - Simplify symbolically using theorems/postulates.
  - Requires good skills
- Karnaugh Maps.
- Diagrammatic technique using 'Venn - diagram'.

- Limited to not more than 6 variables

Some of the examples are given here:

1. Simplify the Boolean expression

XY′Z′+XY′Z′W+XZ′

The above expression can be written as

XY′Z′ (1+W) +XZ′

=XY′Z′+XZ′ as 1+W=1

=XZ′ (Y′+1)

=XZ′ as Y′+1=1

2. Simplify the Boolean expression

X+X′Y+Y′+(X+Y′) X′Y

The above expression can be written as

X+X′Y+Y′+XX′Y+Y′X′Y

=X+X′Y+Y′ as XX′=0, and YY′=0

=X+Y+Y′ as X+X′Y=X+Y

=X+1 as Y+Y′=1

=1 as X + 1=1

3. Simplify the Boolean expression

Z(Y+Z) (X+Y+Z)

The above expression can be written as

(ZY+ZZ)(X+Y+Z)

= (ZY+Z) (X+Y+Z) as ZZ=Z

=Z(X+Y+Z) as Z+ZY=Z

=ZX+ZY+ZZ

=ZX+ZY+Z as ZZ=Z,

=ZX+Z as Z+ZY=Z

=Z as Z+ZX=Z

4. Simplify the Boolean expression

(X+Y)(X′+Z)(Y+Z)

The above expression can be written as

(XX′+XZ+YX′+YZ)(Y+Z)

=(XZ+YX′+YZ) (Y+Z) as XX′=0

=XZY+YYX′+YYZ+XZZ+YX′Z+YZZ

=XZY+YX′+YZ+XZ+YX′Z+YZ as YY=Y, ZZ=Z

Rearranging the terms we get

XZY+XZ+YX′+YX′Z+YZ as YZ+YZ=YZ

=XZ(Y+1) +YX′+YZ (X′+1) as Y+1=1, X′+1=1

=XZ+YX′+YZ

Now it seems that it cannot be reduced further. But apply the following trick:

The above expression can be written as

XZ+YX′+YZ(X+X′) as X+X′=1

=XZ+YX′+YZX+YZX′

Rearranging the terms we get

XZ+YXZ+Y X′+YX′Z

=XZ (1+Y) +YX′ (1+Z)

=XZ+YX′ as 1+Y=1, 1+Z=1

**2.2.1 Sum of Products:**

A sum of products expression consists of several product terms logically added. A product term is a logical product of several variables. The variables may or may not be complemented. The following are the examples of sum of products expressions.

1. XY+X'Y+XY'
2. AB+ABC+BC'
3. A+AB'+B'C
4. ABC+A'B+AB'C+A'BC'

Sometimes a product term may consist of a single variable.

### 2.2.2 Products of Sums:

A product of sums expression consists of several sum terms logically multiplied. A sum term is the logical addition of several variables. The variables may or may not be complemented. The following are examples of product of sums expressions:

A) (A+B) (A'+B')

B) A (B'+C') (B+C)

c) (X+Y') (X+Y+Z) (Y+Z)

Sometimes a sum term may consist of a single variable.

### 2.2.3 Canonical SOP and POS Forms:

When each term of a logic expression contains all variables, it's said to be in the canonical form. When a sum of products form of logic expression is in canonical form, each product term is called *minterm.* Each minterm contains all variables. The canonical form of a sum of products expression is also called *minterm* canonical form or standard sum of products. Similarly, when a product of sums form of logic expression is in canonical form, each sum term is called a *maxterm*. Each maxterm contains all variables. The canonical form of a product of sums expression is also called maxterm canonical form or standard product of sums.

When a logic expression is not in the canonical form, it can be converted into canonical form. In the canonical form there is uniformity in the expression, which facilitates minimization procedure

The following are examples of the canonical form of sum of products expressions (or minterm canonical form):

(i). $Z = XY + XY'$

(ii). $F = XYZ' + X'YZ + X'YZ' + XY'Z + XYZ$

In case of 2 variables, the maximum possible product terms are 4, for 3 variables, the possible product terms are 8, for 4 variables 16, and for *n* variables, $2^n$.

In the above examples the expression (ii) contains 5 out of 8 possible product terms. When the expression is in the canonical form all terms are mutually exclusive. It means that for a given set of values of the variables, when one of the terms is equal to 1, all others must be 0. Of course, it is possible that all terms may be 0.

The following are examples of canonical form of product of sums expressions (or maxterm canonical form).

(i). Z = (X + Y) (X + Y′)

(ii). F = (X′ + Y + Z′) (X′ + Y + Z) (X′ + Y′ + Z′)

The following table gives the minterms and maxterms for a three variable logical function where the number of minterms as well as maxterms is $2^3 = 8$. In general, for an n-variable logical function there are $2^n$ minterms and an equal number of maxterms.

| Variables | | | Minterms | Maxterms |
|---|---|---|---|---|
| **A** | **B** | **C** | **$m_i$** | **$M_i$** |
| 0 | 0 | 0 | A' B' C' = $m_0$ | A + B + C = $M_0$ |
| 0 | 0 | 1 | A' B' C = $m_1$ | A + B + C' = $M_1$ |
| 0 | 1 | 0 | A' B C' = $m_2$ | A + B' + C = $M_2$ |
| 0 | 1 | 1 | A' B C = $m_3$ | A + B' + C' = $M_3$ |
| 1 | 0 | 0 | A B' C' = $m_4$ | A' + B + C = $M_4$ |
| 1 | 0 | 1 | A B' C = $m_5$ | A' + B + C' = $M_5$ |
| 1 | 1 | 0 | A B C' = $m_6$ | A' + B' + C = $M_6$ |
| 1 | 1 | 1 | A B C = $m_7$ | A' + B' + C' = $M_7$ |

**Minterms and Maxterms for Three variables**

As shown in the above table each minterm is represented by $m_i$ and each maxterm is represented by $M_i$ where $i$ is the decimal number equivalent of the natural binary number. With these shorthand notations logical functions can be represented as follows:

1. Y = A' B' C’ + A’ B’ C + A’ B C + A B C’

   = $m_0 + m_1 + m_3 + m_6$

   = $\sum m( 0, 1, 3, 6 )$

2. Y = ( A + B + C’ ) ( A + B’ + C’ ) ( A’ + B’ + C )

   = $M_1 + M_3 + M_6$

   = $\pi M( 1, 3, 6 )$

Where $\sum$ denotes **sum of product** while $\pi$ denotes **product of sum**

**Conversion of Sum of Products Expressions into Canonical Form**:

The following examples will illustrate how logic expressions can be converted into canonical form.

**Example 1:** Convert the expression X + XY' into canonical form.

The expression has two variables. The first term has only one variable. So to make it of two variables it can be multiplied by (Y + Y'), as Y + Y' = 1. After multiplication the given logic expression can be written as

X(Y + Y′) + XY′, as Y + Y′ = 1

or XY + XY′ + XY′

or XY + XY′

**Conversion of Product of Sums Expression into Canonical Form:**

Before we proceed with such a conversion a few identities should be examined.

We can write A = (A + B) (A + B′)

This can be proved as follows:

A = A +A + 0

= A( B + B′ ) + A.A + B.B′, as B + B′ =1, AA=A, BB′=1

= AB + AB′ + AA + BB′

= A (A +B) + B′ (A + B)

= (A + B) (A + B′)

Similarly, we can write A + B = (A + B +C) (A + B + C′).

(A + B + C) (A + B + C′)

= AA + AB + AC′ + AB + BB + BC′ + AC + BC + CC′

Rearranging the terms we get

AA + BB + AC′ + BC′ + AC + BC + AB + AB, as CC′ = 0

= (A + B) + C′ (A + B) + C (A + B) + AB + AB [AA = A; BB = B]

= (A + B) + (A + B) (C + C′) + AB + AB

= (A + B) + (A + B) + AB + AB as C + C′ = 1

= A + B + AB + AB as (A + B) + (A + B) = (A + B)

= A + AB + B + AB

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

= A + B as 1 + B = 1, 1 + A =1

This technique can be extended to any number of variables such as

$(A + B' + C) = (A + B' + C + D)\ (A + B' + C + D')$

**Example 1:** Convert the following expression into canonical form:

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

To convert the above expression into canonical form the following identity can be used:

$X + Y = (X + Y + Z)\ (X + Y + Z')$

Applying the above identity, the given logic expression can be written as

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

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

### 2.2.4 Karnaugh Maps

Karnaugh maps provide a systematic method to obtain simplified sum-of-products (SOPs) Boolean expressions. This is a compact way of representing a truth table and is a technique that is used to simplify logic expressions. It is ideally suited for four or less variables, becoming cumbersome for five or more variables. Each square represents either a minterm or maxterm. A K-map of n variables will have 2 squares. For a Boolean expression, product terms are denoted by 1's, while sum terms are denoted by 0's.

A K-map consists of a grid of squares, each square representing one canonical minterm combination of the variables or their inverse. The map is arranged so that squares representing minterms which differ by only one variable are adjacent both vertically and horizontally. Therefore XY'Z' would be adjacent to X'Y'Z' and would also adjacent to XY'Z and XYZ'.

**Minimization Technique**

- Based on the Unifying Theorem: X + X' = 1
- The expression to be minimized should generally be in sum-of-products form (If necessary, the conversion process is applied to create the sum-of-products form).
- The function is mapped onto the K-map by marking a 1 in those squares corresponding to the terms in the expression to be simplified (The other squares may be filled with 0's).
- Pairs of 1's on the map which are adjacent are combined using the theorem Y(X+X') = Y where Y is any Boolean expression (If two pairs are also adjacent, then these can also be combined using the same theorem).

The minimization procedure consists of recognizing those pairs and multiple pairs

->These are circled indicating reduced terms.

- Groups which can be circled are those which have two ($2^1$) 1's, four ($2^2$) 1's, and eight ($2^3$) 1's.

->Note that because squares on one edge of the map are considered adjacent to those on the opposite edge, group can be formed with these squares.

->Groups are allowed to overlap.

The objective is to cover all the 1's on the map in the fewest number of groups and to create the largest groups to do this.

Once all possible groups have been formed, the corresponding terms are identified.

->A group of two 1's eliminates one variable from the original minterm.

->A group of four 1's eliminates two variables from the original minterm.

->A group of eight 1's eliminates three variables from the original minterm, and so on.

->The variables eliminated are those which are different in the original minterms of the group.

In any K-Map, each square represents a minterm. Adjacent squares always differ by just one literal (So that the unifying theorem may apply: X + X' = 1). For the 2-variable case (e.g.: variables X, Y), the map can be drawn as in Figure 2.2.4 (a). Two variable map is the one which has got only two variables as input.

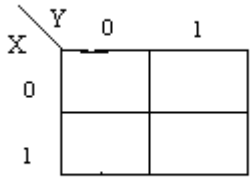



Figure 2.2.4 (a)

**Equivalent Labeling**

K-map need not follow the ordering as shown in the Figure 2.2.4(a). What this means is that we can change the positions of m0, m1, m2, m3 of the above figure as shown in the Figure 2.2.4 (b) and Figure 2.2.4(c).

Position assignment is the same as the default k-map positions. This is the one which we will be using throughout this unit.

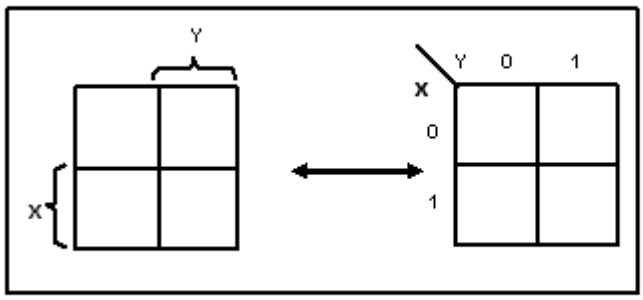



Figure 2.2.4 (b)

This figure is with changed positions of m0, m1, m2, m3.

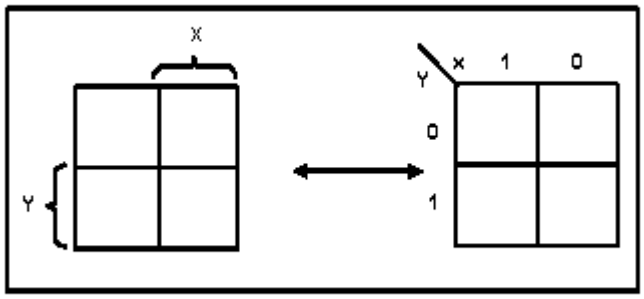



Figure 2.2.4(c)

The K-map for a function is specified by putting a '1' in the square corresponding to a minterm, a '0' otherwise.

**Grouping/Circling K-maps**

The power of K-maps is in minimizing the terms, K-maps can be minimized with the help of grouping the terms to form single terms as shown in Figure 2.2.4 (d). When forming groups of squares, observe/consider the following:

- Every square containing 1 must be considered at least once.
- A square containing 1 can be included in as many groups as desired

A group must be as large as possible.

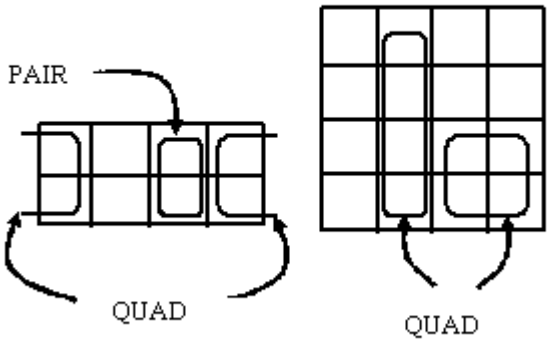



Figure 2.2.4 (d)

- If a square that is containing 1 which cannot be placed in a group, then leave it out to include in final expression.
- The number of squares in a group must be equal to 2(pair), 4(quad), 8(octet).

The map is considered to be folded or spherical; therefore squares at the end of a row or column are treated as adjacent squares.

The simplified logic expression obtained from a K-map is not always unique. Groupings can be made in different ways as shown in Figure 2.2.4(e).

Before drawing a K-map the logic expression must be in canonical form.

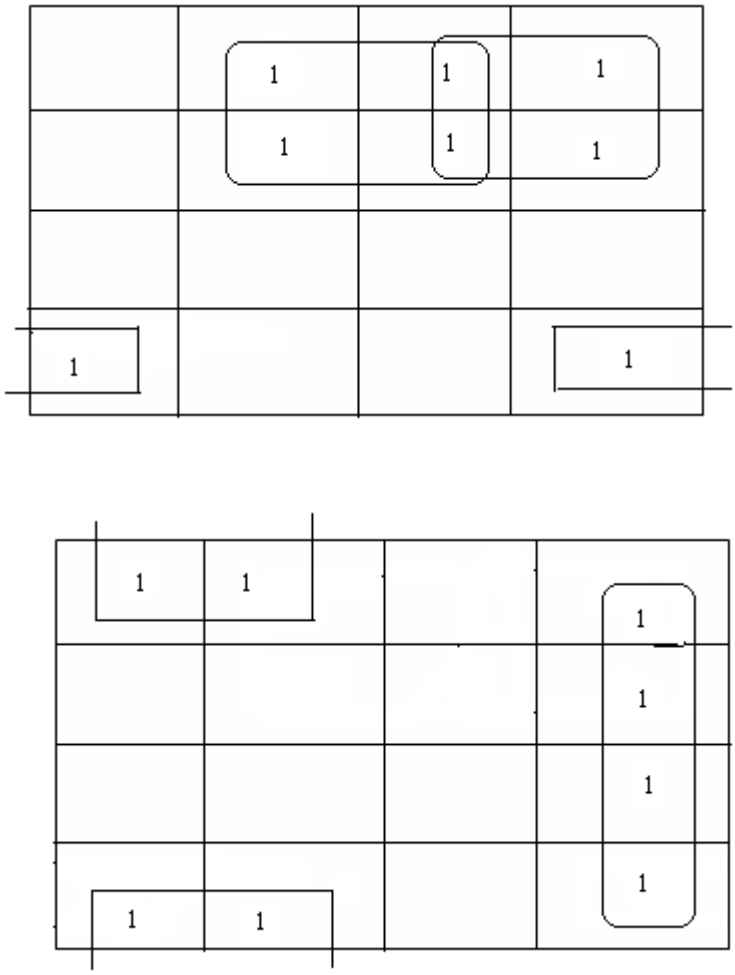



Figure 2.2.4 (e)

In the next few pages we will see some examples of grouping.

**2-Variable K-Map:**

**Example - F= X'Y+XY**

In this example we have the equation as input, and we have one output function. Draw the k-map for function F with marking 1 for X'Y and XY positions. Now combine two 1's as shown in Figure 2.2.4 (f) to form the single term. As you can see X and X' get canceled and only Y remains **F = Y**

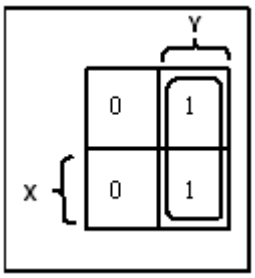



Figure 2.2.4 (f)

**Example - X'Y+XY+XY'**

In this example we have the equation as input, and we have one output function. Draw the k-map for function F with marking 1 for X'Y, XY and XY positions. Now combine two 1's as shown in Figure 2.2.4(g) to form two single terms.

**F = X + Y**

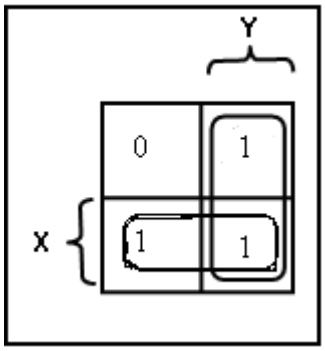



Figure 2.2.4(g)

**3-Variable K-Map**

There are 8 minterms for 3 variables (X, Y, Z). Therefore, there are 8 cells in a 3-variable K-map. One important thing to note is that K-maps follow the gray code sequence, not the binary one.

Using gray code arrangement ensures that minterms of adjacent cells differ by only one literal.

Each cell in a 3-variable K-map has 3 adjacent neighbours. In general, each cell in an *n*-variable K-map has *n* adjacent neighbours as shown in Figure 2.2.4(h)

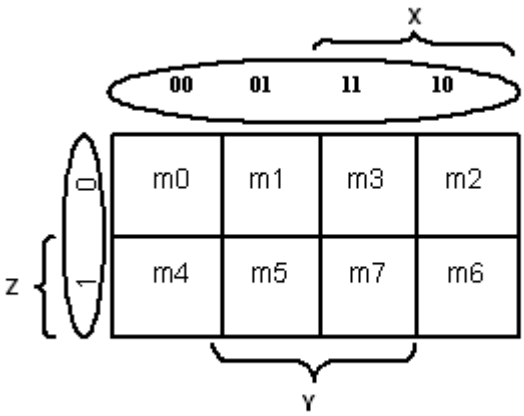



Figure 2.2.4(h)

There is wrap-around in the K-map

- X'Y'Z' (m0) is adjacent to X'YZ' (m2)

XY'Z' (m4) is adjacent to XYZ' (m6) as shown in Figure 2.2.4(i)

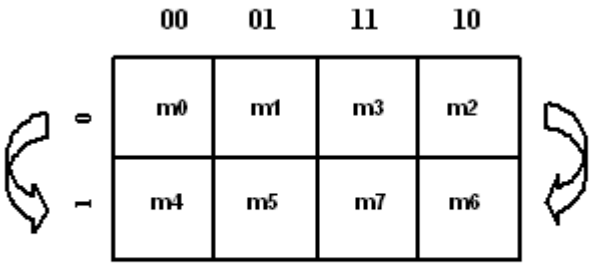
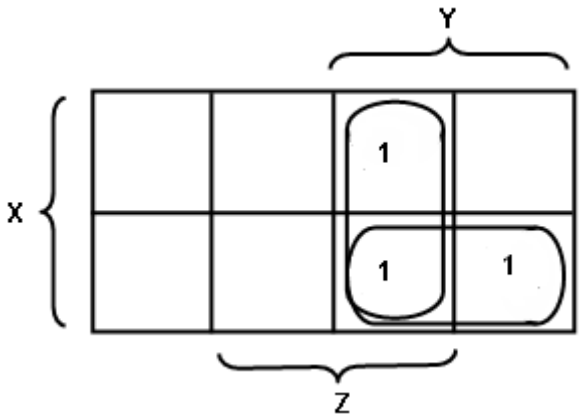



Figure 2.2.4(i)

**Example**

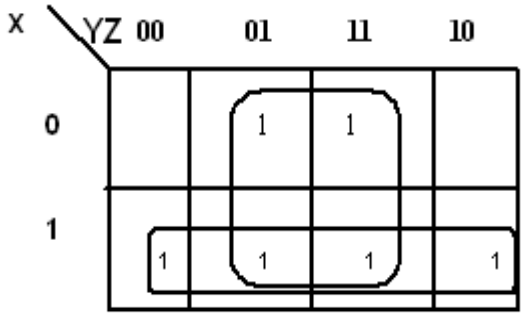
F = XYZ'+XYZ+X'YZ




F = XY + YZ

**Example**

F(X, Y, Z) = $\Sigma$(1, 3, 4, 5, 6, 7)




F = X + Z

**4-Variable K-Map**

There are 16 cells in a 4-variable (W, X, Y, Z) K-map as shown in the Figure 2.2.4 (j).

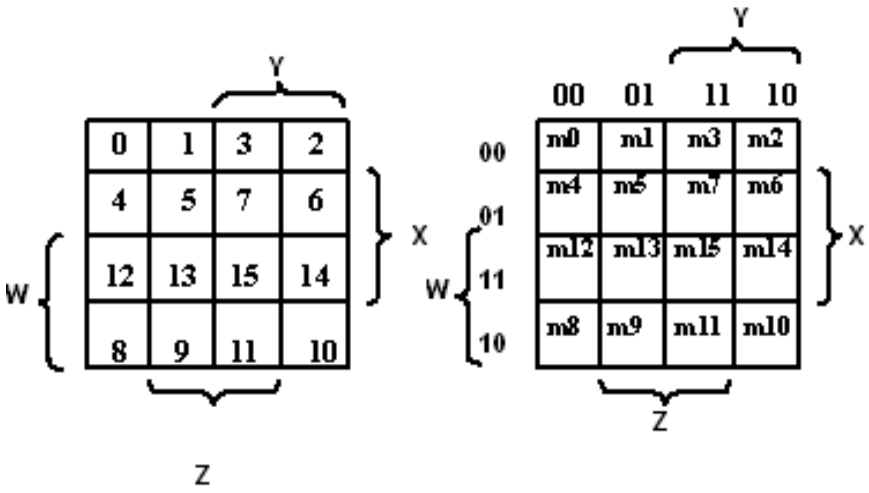



Figure 2.2.4(j)

There are 2 wrap-arounds: a horizontal wrap-around and a vertical wrap-around. Every cell thus has 4 neighbours. For example, the cell corresponding to minterm m0 has neighbours m1, m2, m4 and m8 as shown in Figure 2.2.4(k).

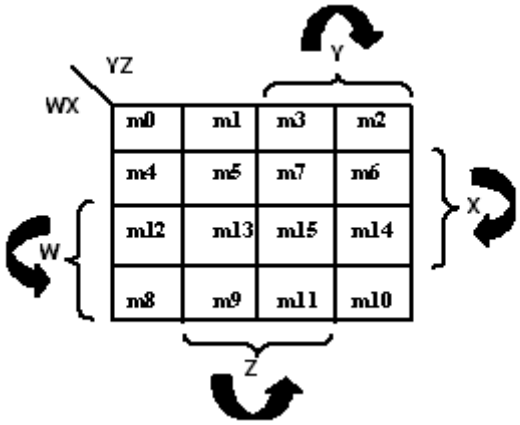
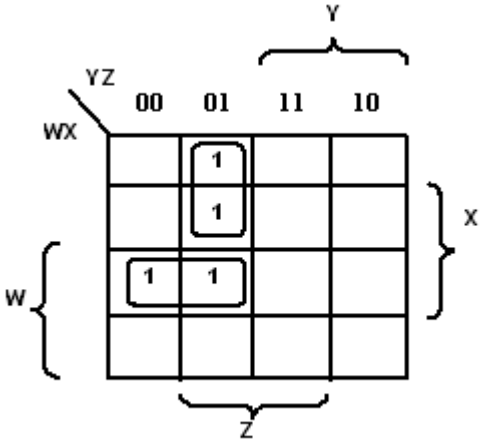



Figure 2.2.4(k)

**Example**

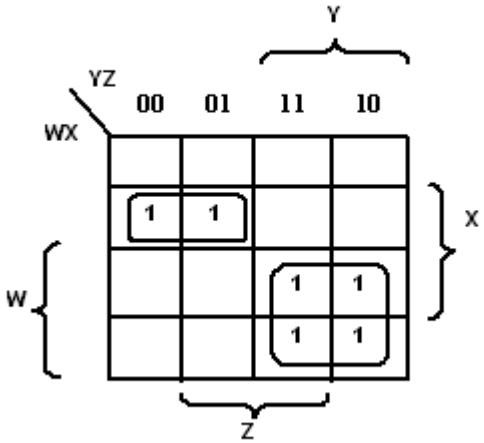
F (W, X, Y, Z) = (1, 5, 12, 13)




F=WXY'+W'Y'Z

**Example**

F (W, X, Y, Z) = (4, 5, 10, 11, 14, 15)




F = W'XY' + WY

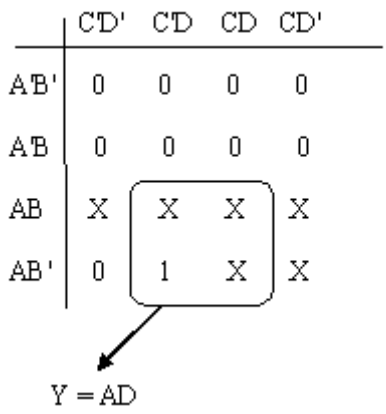
**Don't Care:**

In some digital systems, certain input conditions never occur during normal operations; therefore the corresponding output never appears. Since the output does not appear it is indicated by an X in the truth table.

X is called don't care condition. So don't cares can be treated as 0's and 1's which ever is more convenient in the process of k-map simplification.

Consider the following truth table in which the output is low for all input entries from 1001 and ‘X’ from 1010 through 1111. The don’t care conditions are denoted by ’X’.

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 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 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | X |
| 1 | 0 | 1 | 1 | X |
| 1 | 1 | 0 | 0 | X |
| 1 | 1 | 0 | 1 | X |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | X |



Here three don't cares are treated as 1's to get a quad which eliminates two variables. The remaining don't cares are treated as 0's.

Steps to be followed to apply don't care conditions:

1. For the given truth table, draw a K-map with 0's, 1's and don't cares.
2. Encircle the actual 1's on the K-map in the largest groups, by treating the don't cares as 1's.
3. After the actual 1's have been included in groups discard the remaining don't cares visualizing them as 0's.

### 2.2.5 Implementing Boolean Expressions Using NAND Gates:

The implementation of a Boolean function with NAND-NAND logic requires that the function be simplified in the sum of product form. The relationship between AND-OR logic and NAND-NAND logic is explained using the following example.

Consider the Boolean function: Y = A B C + D E + F

This Boolean function can be implemented using AND-OR logic as shown in Figure 2.2.5 (a).

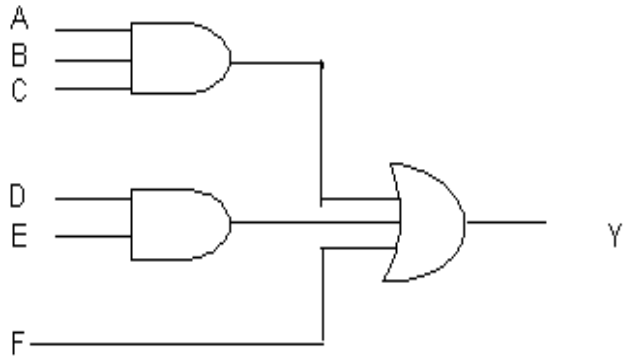



Figure 2.2.5 (a) AND-OR

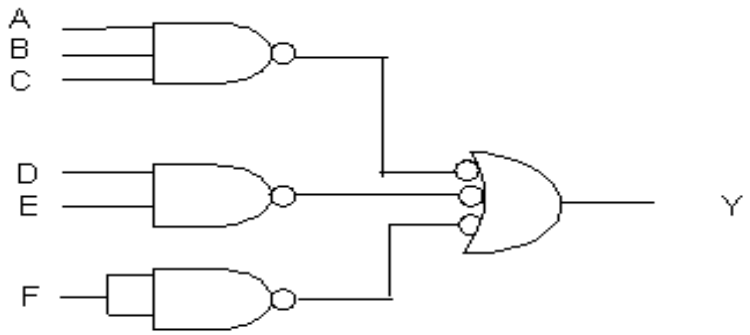



Figure 2.2.5 (b) NAND-Bubbled OR

Figure 2.2.5 (b) shows the AND gates are replaced by NAND gates and the OR gate is replaced by a bubbled OR gate. The implementation shown in Figure 2.2.5(b) is equivalent to implementation in Figure 2.2.5 (a), because two bubbles on the same line represent double inversion (complementation) which is equivalent to having no bubble on the line. In case of single variable, F, the complemented variable is again complemented by bubble to produce the normal value of F.

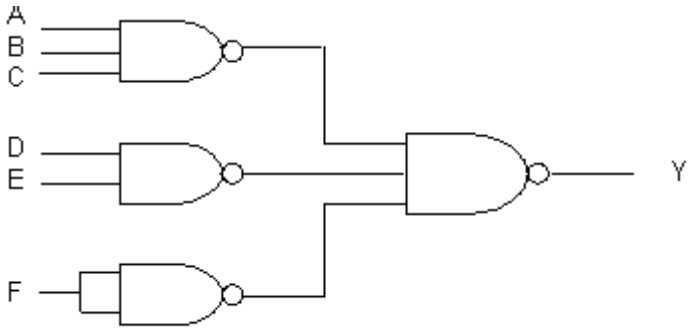



Figure 2.2.5(c) NAND-NAND

In Figure 2.2.5 (c), the output NAND gate is redrawn with the conventional symbol. The NAND gate with same inputs gives complemented result; therefore F′ is replaced by NAND gate with F input to its both inputs. Thus all the three implementations of the Boolean function are equivalent.

From the above example we can summarize the rules for obtaining the NAND-NAND logic diagram from a Boolean function as follows:

1. Simplify the given Boolean function and express it in sum of products form (SOP form).
2. Draw a NAND gate for each product term of the function that has two or more literals. The inputs to each NAND gate are the literals of the term. This constitutes a group of first-level gates.
3. If Boolean function includes any single literal or literals draw NAND gates for each single literal and connect corresponding literal as an input to the NAND gate.
4. Draw a single NAND gate in the second level, with inputs coming from outputs of first level gates.

### 2.2.6 Implementing Boolean Expressions Using NOR Gates:

The NOR function is a dual of the NAND function. For this reason, the implementation procedures and rules for NOR-NOR logic are the duals of the corresponding procedures and rules developed for NAND-NAND logic.

The implementation of a Boolean function with NOR-NOR logic requires that the function be simplified in the product of sums form. In product of sums form, we implement all sum terms using OR gates. This constitutes the first level. In the second level all sum terms are logically ANDed using AND gates. The relationship between OR-AND logic and NOR-NOR is explained using following example

Consider the Boolean function: $Y = (A + B + C)(D + E)F$

The Boolean function can be implemented using OR-AND logic, as shown in the Figure 2.2.6 (a)

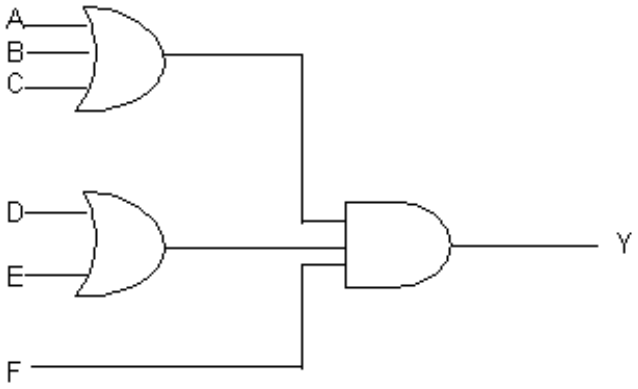



Figure 2.2.6 (a) OR-AND

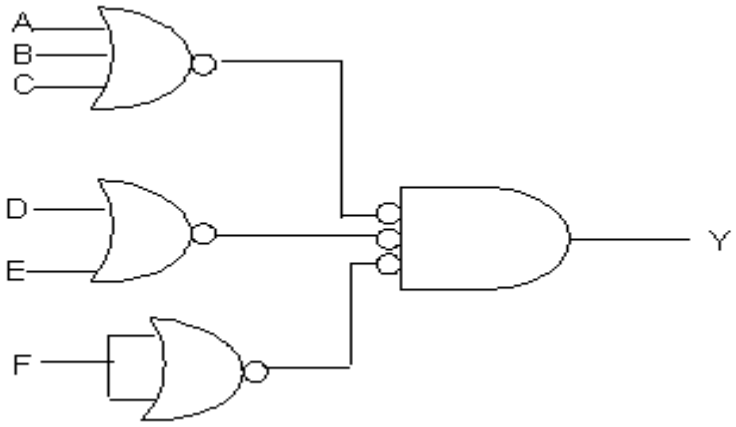



Figure 2.2.6 (b) NOR-Bubbled AND

In Figure 2.2.6 (b) the OR gates are replaced by NOR gates and the AND gate is replaced by a bubbled AND gate. The implementation shown in Figure 2.2.6 (b) is equivalent to implementation shown in Figure 2.2.6 (a) because two bubbles on the same line represent double inversion (complementation) which is equivalent to having no bubble on the line. In case of single variable, F, the complemented variable is again complemented by bubble to produce the normal value of F.

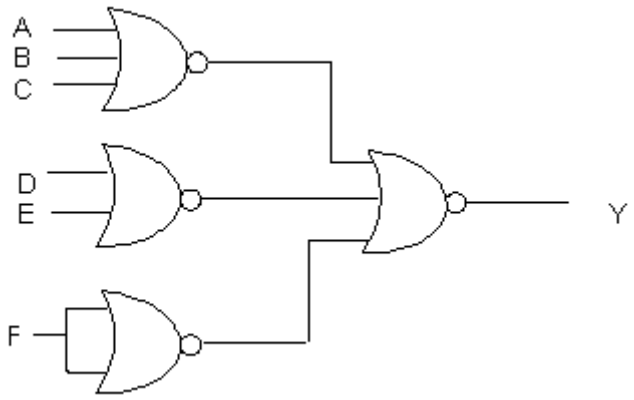



Figure 2.2.6(c) NOR-NOR

In Figure 2.2.6 (c), the output NOR gate is redrawn with the conventional symbol. The NOR gate with same inputs gives complemented result, therefore, F is replaced by NOR gate with F input to its both inputs. Thus all the three implementations of the Boolean function are equivalent.

From the above example, we can summarize the rules for obtaining the NOR-NOR logic diagram from a Boolean function as follows:

1. Simplify the given Boolean function and express it in product of sums form(POS form)
2. Draw a NOR gate for each sum term of the function that has two or more literals. The inputs to each NOR gate are the literals of term. This constitute a group of first level gates.
3. If Boolean function includes any single literal or literals, draw NOR gate for each single literal and connect corresponding literal as an input to the NOR gate.
4. Draw a single NOR gate in the second level, with inputs coming from outputs of first level gates

**Check Your Progress 1**

**1.** The simplified form of Boolean expression(X+Y+XY) (X+Z) is

(a) X+Y+Z (b) XY+YZ

(c)X+YZ (d) XZ+Y

2. The simplified form of Boolean expression(X +Y'+Z) (Z+ Y'+Z') is

(a) X' Y+Z' (b) X+Y' +Z

(c) X (d) XY+Z'

3. The canonical form of logical expression A+A' B is

(a)AB+AB'+A'B (b) A'B' +AB+AB'

(c) AB'+A'B+AB' (c) A'B+A B' +A'B'

4. The canonical form of logical expression (A+B') (B'+C) is

(a) (A+B'+C') (A+B'+C) (A'+B'+C)

(b) (A+B'+C') (A+B'+C) (A'+B+C')

(c) (A+B+C') (A+B'+C') (A'+B'+C)

(d) (A+B'+C) (A+B'+C) (A'+B'+C)

## 2.3 Combinational Circuits

A combinational circuit consists of input variables, logic gates and output variables. The logic gates accept signals from the input variables and generate output signals. This process transforms binary information from the given input data to the required output data. Figure 2.3 shows the block diagram of a combinational circuit. As shown in the figure the combinational circuit accepts *n* input binary variables and generates *m* output variables depending on the logical combination of gates.

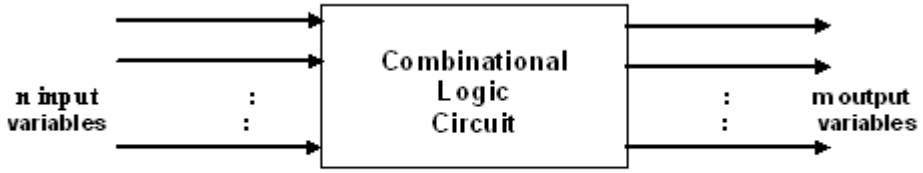



Figure 2.3

In this section we shall discuss about the functions of Half Adder, Full Adder, Half Subtractor, Full Subtractor, Parallel Binary Adder, BCD Adder, Encoders, Decoders, Multiplexers and Demultiplexers.

### 2.3.1 Half Adder

Half adder is a logic circuit that finds the arithmetic sum of two binary digits at a time. Its logic circuit is shown in Figure 2.3.1(a).

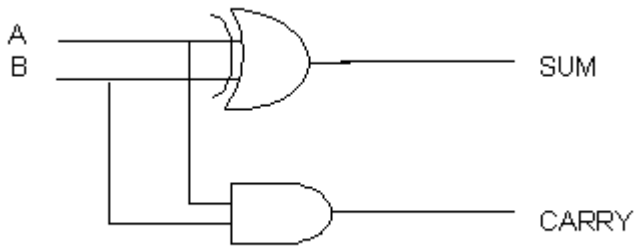


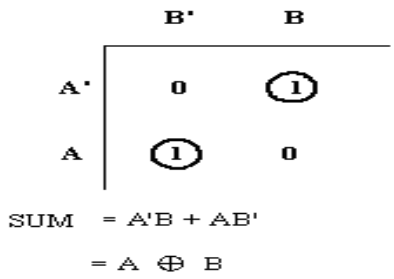

Figure 2.3.1(a) Half Adder

The outputs of the XOR and AND gates produces the sum and carry respectively.

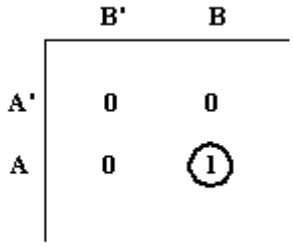
THE TRUTH TABLE:

| A | B | SUM $A \oplus B$ | CARRY A.B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Map for SUM




Map for CARRY




CARRY = A . B

The input variables of half adder are augend and addend. The output variables are sum and carry. It is necessary to specify two output variables, because the sum of 1+1=10. Let A & B be input variables SUM and CARRY be output variables.

The output 'CARRY' represents an AND function. The output SUM represents exclusive OR function. The Boolean functions of the two outputs are

SUM = A ⊕ B and

CARRY = A . B

### 2.3.2 Full Adder

When two binary numbers are added a carry may be generated onto the subsequent bit positions. Hence, it is required to add three bits for the subsequent additions. A combinational circuit that finds the arithmetic sum of three bits is called a *Full adder*. A Full adder can be constructed using two half adders and an OR gate as shown in the Figure 2.3.2(a).

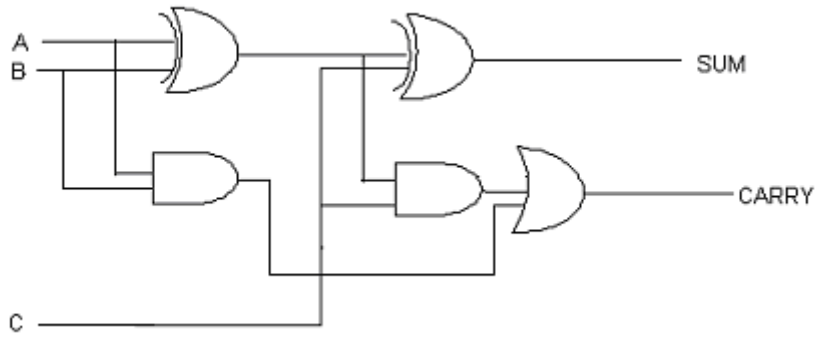



Figure 2.3.2(a) Full Adder

Truth table:

| A | B | C | CARRY | SUM |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Thus a full-adder is a combinational circuit that performs the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables denoted by A, B represents the two significant bits to be added. The third input C represents the carry from the lower significant position. The two outputs are denoted by SUM and CARRY. The Boolean expressions for SUM and CARRY outputs are given below.

SUM= A' B 'C + A' B C' + A B' C' + A B C

SUM= (A' B + A B') C' + (A' B' + A B) C

SUM = A ⊕ B ⊕ C

CARRY = A B' C + A' B C + A B C + A B C'

CARRY = C [A' B + A B'] + A B [C + C']

= (A⊕B).C + A B

### 2.3.3 Half Subtractor:

*A Half subtractor* is a combinational logic circuit which is used to find the difference between two binary digits. Its logic circuit is shown in Figure 2.3.3(a).

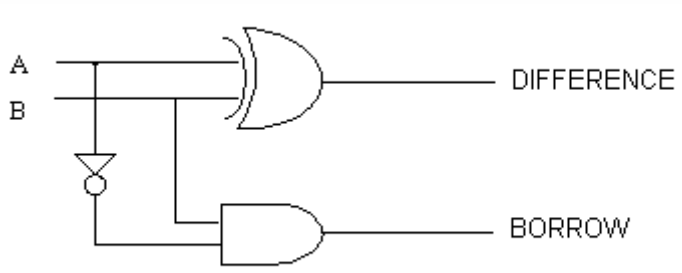
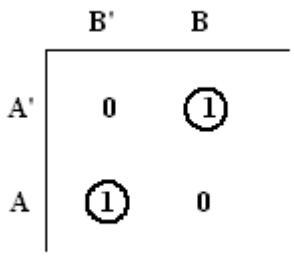



Figure 2.3.3(a) Half Subtractor

TRUTH TABLE:

| A | B | BORROW | DIFFERENCE |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

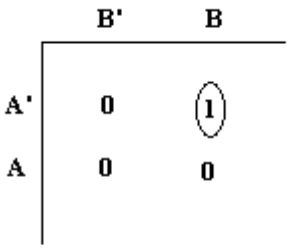
Map for DIFFERENCE:




DIFFERENCE = A'B + AB'

= A⊕ B

Map for BORROW:




BORROW = A'B

A half subtractor consists of two input variables A and B (minuend and subtrahend) and two output variables DIFFERENCE and BORROW. The DIFFERENCE output is obtained by a 2-input XOR gate. The BORROW output is obtained by the expression A'B

Hence DIFFERENCE = A⊕B

BORROW = A'B

### 2.3.4 Full Subtractor:

A full subtractor (Figure 2.3.4 (a)) is a combinational circuit that performs a subtraction between two bits taking into account that a 1 may have been borrowed by a lower significant stage.

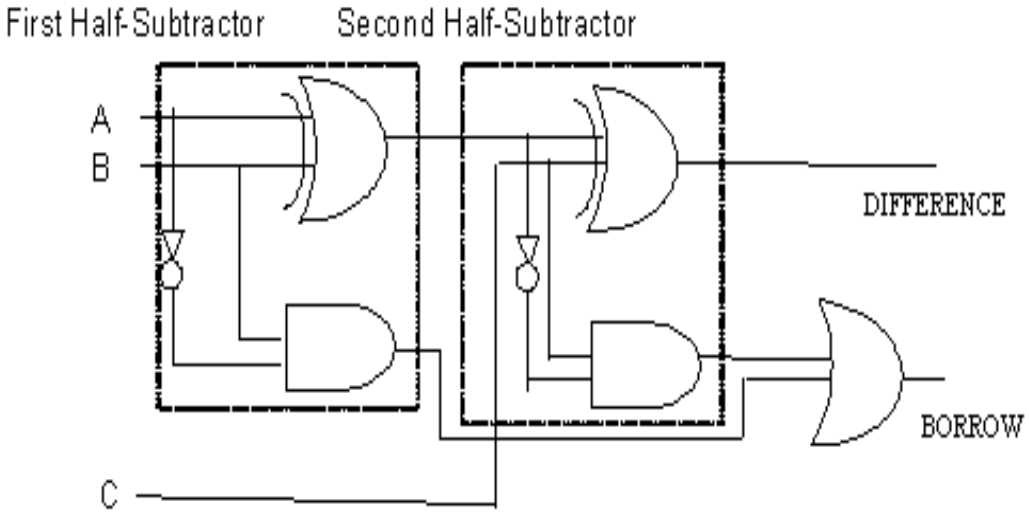


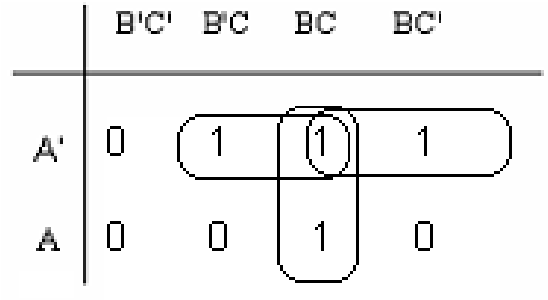

Figure 2.3.4 (a) Full Subtractor

This circuit has three inputs and two outputs. The three inputs A, B and C denote the minuend, subtrahend and previous borrow respectively. The two outputs DIFFERENCE and BORROW represent the difference and borrow, respectively. The truth table for the circuit is as follows.

| A | B | C | BORROW | DIFFERENCE |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

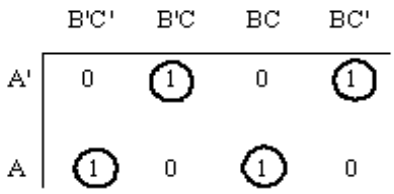
The Boolean functions for the two outputs of the full subtractor are derived in the K-map as shown below.

Map for BORROW




BORROW = A'C + A'B + B

Map for DIFFERENCE




DFFERENCE = A'B'C + A'BC' + AB'C' + ABC
= A⊕B⊕C

**2.3.5 Parallel Binary Adder:**

A parallel binary adder is a digital circuit that produces the arithmetic sum of two binary numbers in parallel. It consists of full adders connected in cascade, with the output carry from one full adder connected to the input carry of the next full adder. Figure 2.3.5 shows the circuit diagram of a 4-bit parallel binary adder.

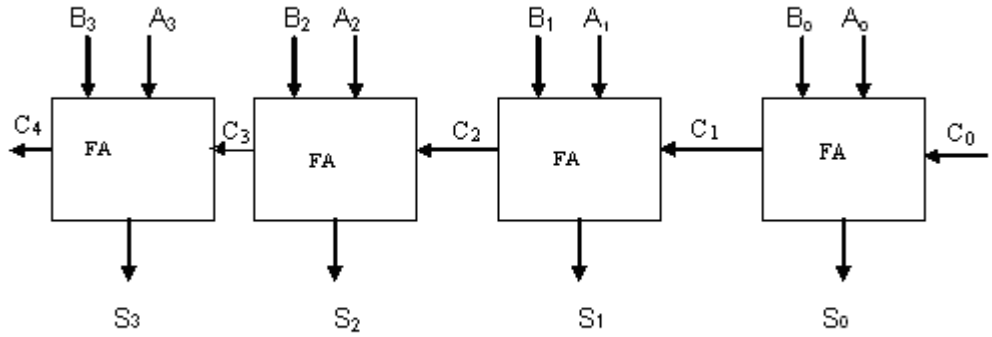
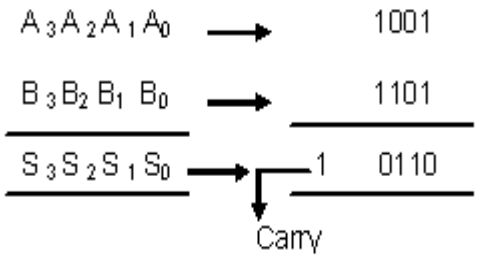
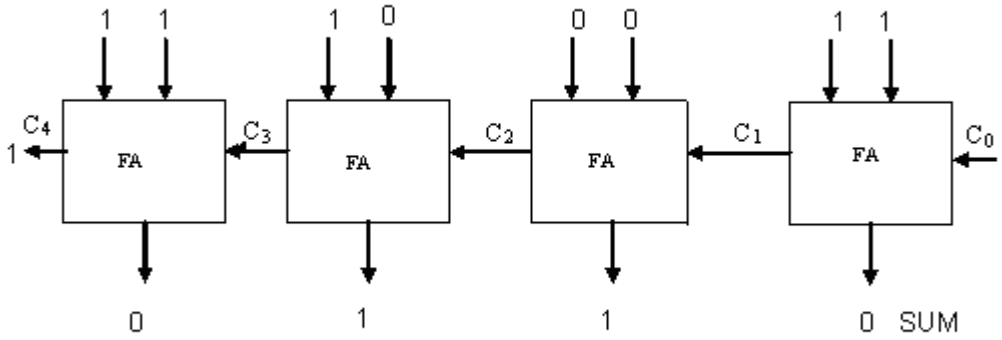



Figure 2.3.5 Parallel Binary Adder

The augend bits of A and the addend bits of B are designated by subscript number from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full adders. The input carry to the adder is $C_0$ and the output carry is $C_4$. The S outputs generate the required sum bits. An *n*-bit parallel binary adder requires *n* full adders.

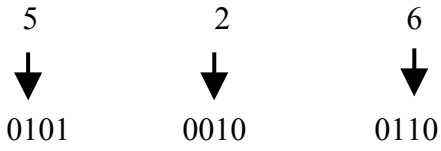
The following example illustrates the parallel binary addition



### 2.3.6. BCD adder

A BCD adder is a circuit that adds two BCD digits and produces a sum digit also in BCD. BCD numbers use 10 digits, 0 to 9 which are represented in the binary form 0000 to 1001, i.e. each BCD digit is represented as a 4-bit binary number. When we write BCD number say 526, it can be represented as




Here, we should note that BCD cannot be greater than 9.

The addition of two BCD numbers can be best understood by considering the two cases that occur when two BCD digits are added.

**Sum Equals 9 or less with carry 0 :**

Let us consider additions of 3 and 6 in BCD.

| | | |
|---|---|---|
| 6 | 0110 | ← BCD for 6 |
| + 3 | 0011 | ← BCD for 3 |
| 9 | 1001 | ← BCD for 9 |

The addition is carried out as in normal binary addition and the sum is 1001, which is BCD code for 9.

**Sum greater than 9 with carry 0 :**

Let us consider addition of 6 and 8 in BCD

| | | |
|---|---|---|
| 6 | 0110 | ← BCD for 6 |
| + 8 | 1000 | ← BCD for 8 |
| 14 | 1110 | ← Invalid BCD number |

The sum 1110 is an invalid BCD number. This has occurred because the sum of the two digits exceeds 9. Whenever this occurs the sum has to be corrected by the addition of six (0110) in the invalid BCD number, as shown below

| | | | |
|---|---|---|---|
| 6 | | 0110 | ← BCD for 6 |
| + 8 | | 1000 | ← BCD for 8 |
| 14 | | 1110 | ← Invalid BCD number |
| | + | 0110 | ← add 6 for correction |
| | 0001 | 0100 | ← BCD for 14 |

After addition of 6, carry is produced into the second decimal position.

Going through these two cases of BCD addition we can summarize the BCD addition procedure as follows:

1. Add two BCD numbers using ordinary binary addition.
2. If the 4-bit sum is equal to or less than 9, no correction is needed. The sum is in proper BCD form.
3. If the 4-bit sum is greater than 9 or if a carry is generated from the 4-bit sum, the sum is invalid.
4. To correct the invalid sum, add $0110_2$ to the 4-bit sum. If a carry results from this addition, add it to the next higher-order BCD digit.

Thus to implement BCD adder we require:

- A 4-bit binary adder for initial addition
- Logic circuit to detect sum greater than 9 and
- One more 4-bit adder to add $0110_2$ if the sum is greater than 0 or carry is 1.

Figure 2.3.6 shows the block diagram of a BCD adder.

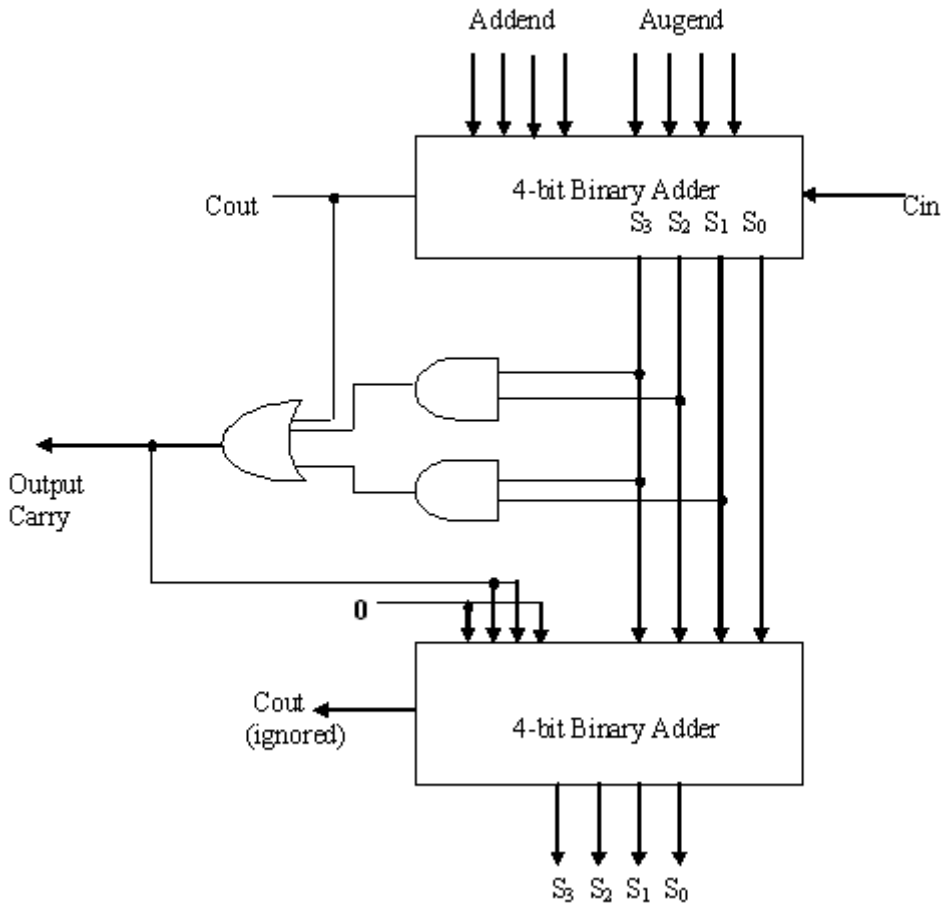



Figure 5.3.6 BCD adder

As shown in Figure 5.3.6 the two BCD numbers, together with input carry, are first added in the top 4-bit binary adder to produce a binary sum. When the output carry is equal to zero (i.e. when sum <=9 and $C_{out}$=0) nothing (zero) is added to the binary sum. When it is equal to one (i.e. when sum>9 or $C_{out}$=1), binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output carry generated from the bottom binary adder can be ignored.

### 2.3.7 Encoders

An *encoder* (Figure 2.3.7(a)) converts an active input signal into a coded output signal. There is ***n*** input lines of which only one is active. Internal logic within the encoder converts this active input to a coded binary output with ***m*** bits.

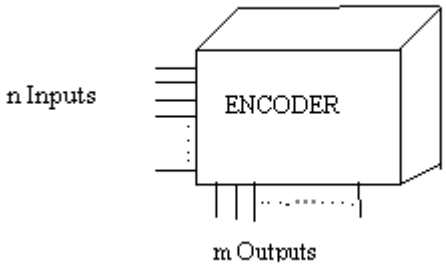



Figure 2.3.7(a) Encoders

**Decimal to BCD Encoder**

The Figure 2.3.7 (b) shows a common type of encoder such as a Decimal to BCD Encoder. The switches are push-button switches like those of a pocket calculator.
When button 3 is pressed, the C and D OR gates receive high inputs.
Therefore the output is
ABCD=0011
If button 5 is pressed, the output becomes
ABCD=0101
When switch 9 is pressed the output is
ABCD=1001

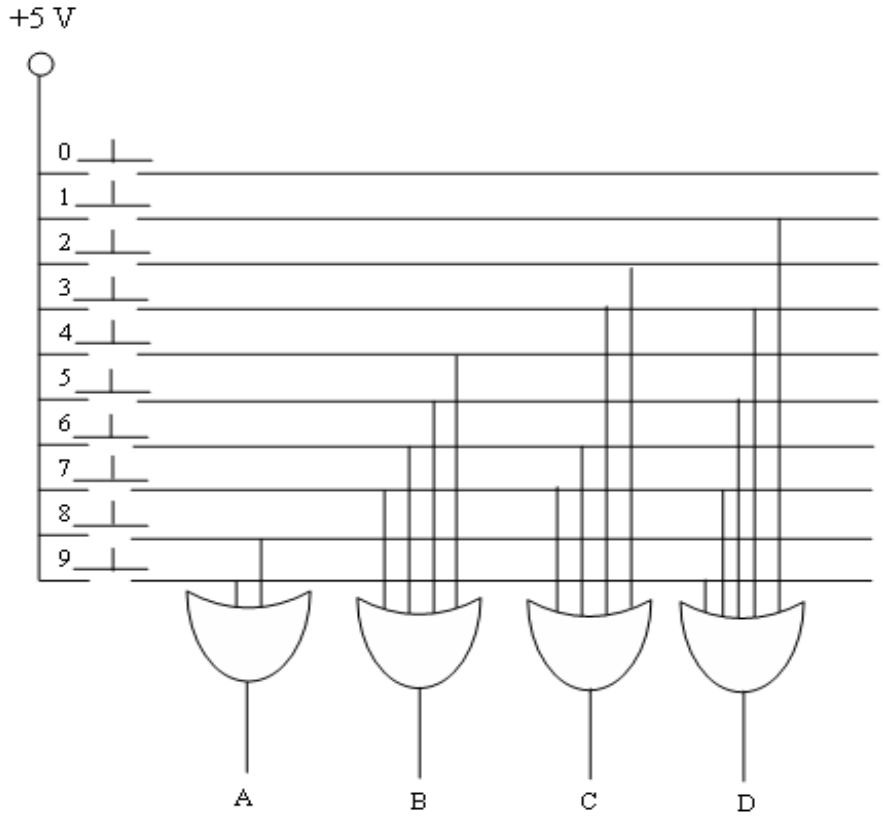



Figure 2.3.7 (b) Decimal to BCD Encoder

### 2.3.8 Decoders

A *decoder* is a combinational circuit that converts binary information from '*n*' input lines to a maximum of $2^n$ unique output lines. The circuit in Figure 2.3.8(a) represents a 2-to-4 line decoder.

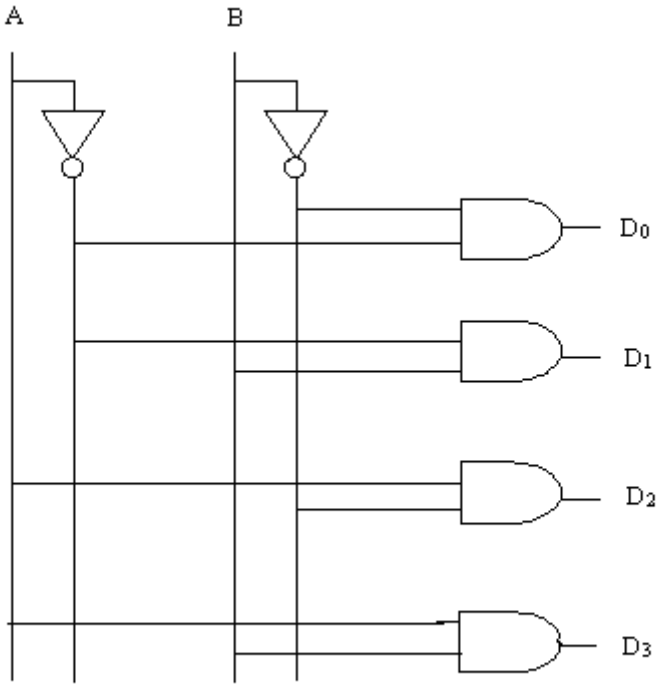



Figure 2.3.8 (a) 2-to-4 decoder.

The two inputs are decoded into 4 outputs each output representing one of the *minterms* of the 2-input variables. The two inverters provide the complement of inputs and each of the four AND gates generate one of the *minterms*.

The following is the truth table of the 2-to-4 line decoder with two inputs and 4 outputs.

| A | B | $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

### 2.3.9 Multiplexer

A *multiplexer* is circuit with many inputs but only one output. By applying control signals, we can steer any one of the inputs to the output. Figure 2.3.9 (a) illustrates the general idea.

The circuit has *n* input signals**,** *m* control signals and one output signal.

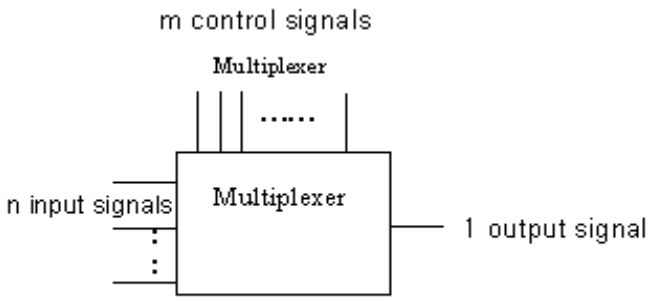



Figure 2.3.9(a) Multiplexer

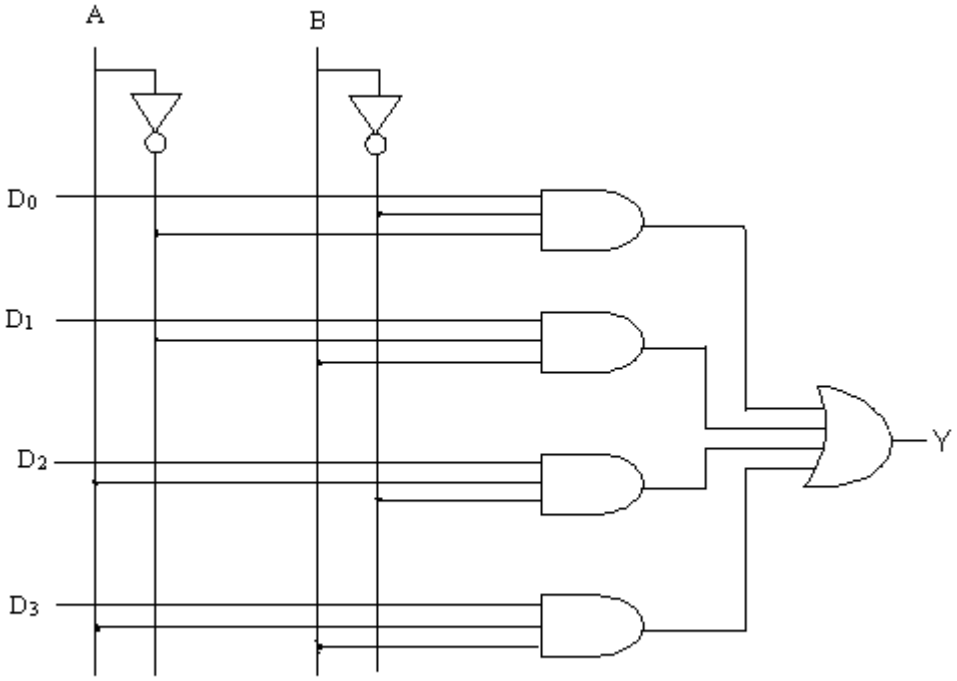



Figure 2.3.9 (b) 4-to-1 Multiplexer

| A | B | Y |
|---|---|---|
| 0 | 0 | $D_0$ |
| 0 | 1 | $D_1$ |
| 1 | 0 | $D_2$ |
| 1 | 1 | $D_3$ |

Figure 2.3.9 (b) shows a 4-to-1 Multiplexer. A multiplexer is also called *Data selector* because the output bit depends on the input data bit that is selected. The input bits are labeled $D_0$ through $D_4$. Only one of these inputs is transmitted to the output, depending on the control inputs AB.

For instance, when AB=00 the upper AND gate is enabled while all other AND gates are disabled. Therefore, data bit $D_0$ is transmitted to the output, giving $Y=D_0$. If $D_0$ is low, Y is low; If $D_0$ is high, Y is high. The point is that Y depends only on the value of $D_0$. If control bits are changed to AB=11, all gates are disabled except the bottom AND gate. In this case $D_3$ is the only bit transmitted to the output and $Y= D_3$. As you can see, the control bits determine which of the input data bits is transmitted to the output.

### 2.3.10 Demultiplexer

A *demultiplexer* is a logic circuit with one input and may outputs. By applying control signals, we can steer the input signal to one of the output lines. Figure 2.3.10(a) illustrates the general idea. The circuit has 1 input signal, *m* control signals and *n* output signals.

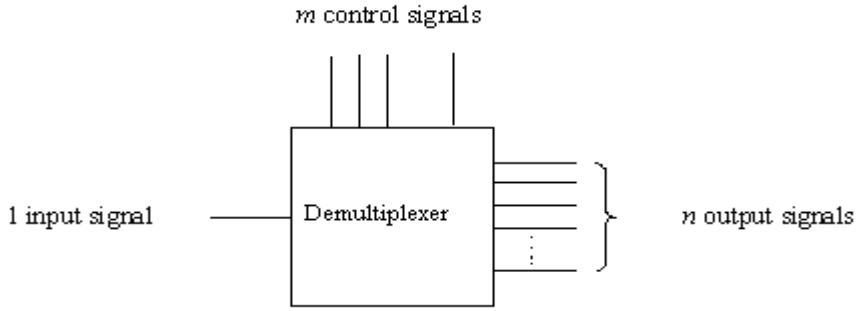



Figure 2.3.10 (a) Demultiplexer

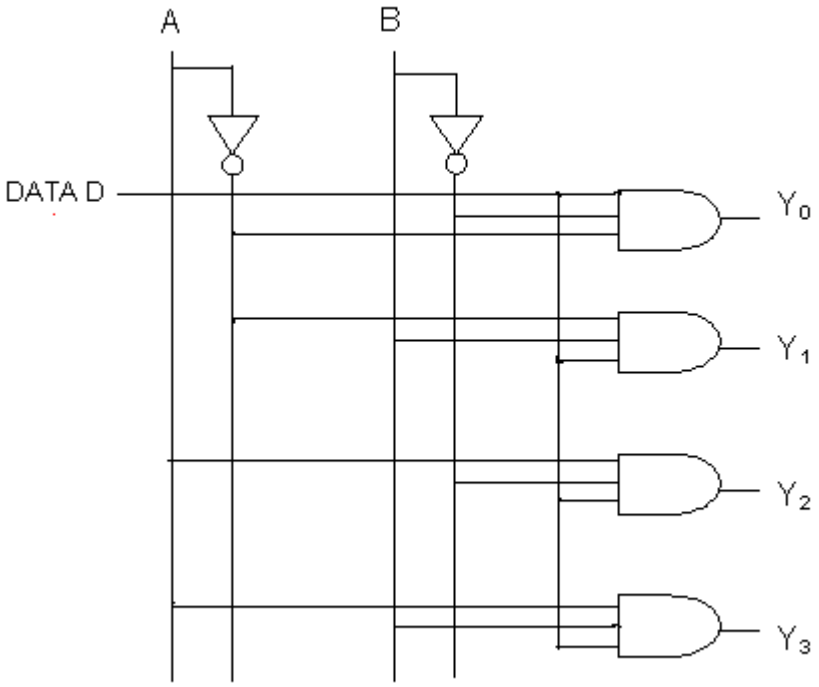



Figure 2.3.10 (b) 1x4 Demultiplexer

Figure 2.3.10 (b) shows a 1x4 Demultiplexer. The input bit is labeled as *D*. This data bit (*D*) is transmitted to the data bit of the output lines. This depends on the value of *AB*, the control inputs. When *AB=00* the upper AND gate is enabled while all other AND gates are disabled. Therefore the data bit (*D*) is transmitted only to the $Y_0$ output, giving $Y_0 = D$. If D is low, $Y_0$ is low. If D is high, $Y_0$ is high. As you can see, the value of $Y_0$ depends on the value of *D*. All other outputs are in the low state. If the control bits are changed to AB=11 all gates are disabled except the bottom AND gate. Then D is transmitted only to the $Y_3$ output and $Y_3$=D.

**Check Your Progress 2**

1. A half adder adds……..bits.

(a) 16 (b) 10 (c) 8 (d) 2

2. Parallel binary adders are

(a)Combinational logic circuits (b) Sequential logic circuits

(c) Both of the above (d) None of the above

3. A combinational circuit which is used to change a decimal number into an equivalent BCD number is

(a) Decoder (b) Encoder (c) Multiplexer (d) Demultiplexer

4. A combinational circuit which is used to change a BCD number into an equivalent decimal number is

(a) Decoder (b) Encoder (c) Multiplexer (d) Demultiplexer

5. Multiplexer is also known as

(a) Data selector (b) Data distributor (c) Multiplexer (d) Encoder

6. A combinational circuit which is used to send data coming from a single source to two or more separate destinations is called as:

(a) Decoder (b) Encoder (c) Multiplexer (d) Demultiplexer

**2.4 Summary**

With Boolean algebra you may be able to simplify a Boolean equation.

Given the truth table, you can identify the fundamental products that produce output 1s. By ORing these products, you get the sum of products for the truth table. Therefore sum-of-product equation always results in an AND-OR circuit or its equivalent NAND-NAND circuit.

The Karnaugh method of simplification starts by converting a truth table into a karnaugh map. Next, you encircle all the octets, quads and pairs. This allows you to write a simplified Boolean equation and to draw a simplified logic circuit. When a truth table contains don't cares, you can treat them as 0s or 1s, whichever produces the greatest simplification.

Half adder is a logic circuit with two inputs and two outputs. It adds two bits at a time, producing a sum and a carry output.

Full adder is a logic circuit with three inputs and two outputs. The circuit adds three bits at a time, giving a sum and a carry output.

Half subtractor is a logic circuit that subtracts two bits and produces their difference.

Full subtractor is a logic circuit that performs a subtraction between two bits, taking into account borrowing by lower significant stage. It has three inputs and two outputs.

BCD adder adds two BCD digits and produces a sum digit also in BCD form.

Encoder is circuit that converts an active input signal into coded output form.

A decoder is a combinational circuit that converts binary information from 'n' input lines to a maximum of $2^n$ unique output lines.

A multiplexer is circuit with many inputs but only one output. By applying control signals, we can steer any one of the inputs to the output.

Demultiplexer is a circuit with one input and many outputs. By applying control signals, we can steer the input signal to one of the outputs.

## 2.5 Glossary

**BCD adder** A logic circuit that adds two BCD digits and produces a sum digit also in BCD.

**Decoder** is a combinational circuit that converts binary information from 'n' input lines to a maximum of $2^n$ unique output lines.

**Demultiplexer** A circuit with one input and many outputs.

**Don't care conditions** An input output condition that never occurs during normal operations. Since the condition never occurs, you can use X in the truth table.

**Encoder** An circuit that converts an active input signal into coded output form.

**Full adder** A logic circuit with three inputs and two outputs. The circuit adds three bits at a time, giving a sum and a carry output.

**Half adder** A logic circuit with two inputs and two outputs. It adds two bits at a time, producing a sum and a carry output.

**Half subtractor** A logic circuit that subtracts two bits and produce their difference.

**Full subtractor** A logic circuit that performs a subtraction between two bits, taking into account borrowing by lower significant stage. It has three inputs and two outputs.

**Karnaugh map** A map that shows all the fundamental products and the corresponding output values of a truth table.

**Multiplexer** A circuit with many inputs but with only one output.

**Octet** Eight adjacent 1s in a karnaugh map.

**Overlapping groups** Using the same 1 more than once when looping the 1s of a karnaugh map.

**Pair** Two horizontally or vertically adjacent 1s in a Karnaugh map..

**Parallel binary adder** A logic circuit with number of full adders connected in cascade. The carry output of each adder is connected to the carry input of the next higher adder.

**Product of sum equation** The logical product of those fundamental sums that produce output 1s in the truth table.

**Quad** Four horizontal, vertical, or rectangular 1s in a Karnaugh map.

**Redundant group** A group of 1s in a karnaugh map that is a part of other groups.

**Sum of products equation** The logical sum of those fundamental products that produce output 1s in the truth table.

**Truth table** A table that shows all the input-output possibilities of a logic circuit.

## 2.6 References


1."Digital logic and computer design" M.Moris Mano, prentice-hall of India private limited.

2."Digital principles and applications" Albert Paul Malvino, Tata McGraw-Hill book company

3."Digital computer fundamentals" Thomous c. Bartee, Tata McGraw-Hill book company.

4."Computer fundamentals- architecture and organization", B.Ram, New age international (P) Ltd.


## 2.7 Answers to Check Your Progress Questions

Check your progress1

1. c
2. b
3. a
4. a

Check your progress 2

1. d
2. a
3. b
4. a
5. a

6. d