UNIT I

NUMBER SYSTEM AND BINARY CODES

1.0 Aims and Objectives
1.1 Introduction
1.2 Number System
   1.2.1 Decimal Number System
   1.2.2 Bi-stable Devices
   1.2.3 Binary Number System
   1.2.4 Octal number System
   1.2.5 Hexadecimal Number System
1.3 Conversions
   1.3.1 Decimal to Binary
   1.3.2 Decimal to Octal
   1.3.3 Decimal to Hexadecimal
   1.3.4 Binary to Decimal
   1.3.5 Binary to Octal
   1.3.6 Binary to Hexadecimal
   1.3.7 Octal to Decimal
   1.3.8 Octal to Binary
   1.3.9 Octal to Hexadecimal
   1.3.10 Hexadecimal to Binary
   1.3.11 Hexadecimal to Octal
   1.3.12 Hexadecimal to Decimal
1.4 Binary Addition and Subtraction
1.5 Binary Multiplication and Division
1.6 Floating point Representation
1.7 Complements
   1.7.1 The (r-1)’s Complement
   1.7.2 The r’s Complement
1.8 Binary Coded Decimal Number Representation
1.9 Excess 3 Code
1.10 Gray Code
1.11 Arithmetic Circuits
   1.11.1 Half Adder
1.11.2 Full Adder
1.11.3 Parallel Binary Adder
1.11.4 BCD Adder
1.11.5 Half Subtractor
1.11.6 Full Subtractor
1.11.7 Parallel Binary Subtractor

1.12 Digital Logic
1.12.1 The Basic Gates
1.12.2 NOR Gate
1.12.3 NAND Gate
1.12.4 XOR Gate

1.13 Let us Sum Up

1.14 Lesson – End Activities

1.16 Points for Discussion

1.16 Model Answers to “Check your Progress”

1.17 References

1.0 AIM AND OBJECTIVES

This unit being the first unit, it introduces you to the world of computers. At the end of the unit you will be able to know how:

1. An explanation of positional notation is given and the idea of the base, or radix, of a number system is presented.
2. The binary number system is explained as well as how to add, subtract, multiply and divide in this system. The techniques for converting from binary to decimal and decimal to binary are given.
3. Negative numbers are represented in computers by using a sign bit, and this concept is explained. Negative numbers are often represented by using a complemented form rather than a signed magnitude form. The two major complemented forms, true complement and radix minus one are described.
4. The representation of decimal numbers using bi-stable devices can be accomplished with a binary coded decimal (BCD) system and several of these are explained.
5. The octal and hexadecimal number systems are widely used in computer literature and manufacturer’s manuals. These number systems are explained along with conversion techniques to and from decimal and binary.

1.1 INTRODUCTION

As a mathematician, Laplace could well appreciate the decimal number system. He was fully aware of the centuries of mental effort and sheer good luck, which had gone into the development of the number system we use, and he was in a position to appreciate its advantages.
Our present number system provides modern mathematicians and scientists with a great advantage over those of previous civilizations and is an important factor in our rapid advancement.

Since hands are the most convenient tools nature has provided, human beings have always tended to use them in counting. So the decimal number system followed naturally from this usage.

As even simpler system, the binary number system has proved the most natural and efficient system for computer use, however, and this chapter develops this number system along with other systems used by computer technology.

1.2 NUMBER SYSTEM

A number system of base (also called radix) \( r \) is a system, which have \( r \) distinct symbols for \( r \) digits. A number is represented by a string of these symbolic digits. To determine the quantity that the number represents, we multiply the number by an integer power of \( r \) depending on the place it is located and then find the sum of weighted digits.

1.2.1 DECIMAL NUMBER SYSTEM

Decimal system is the most commonly used number system. Our present system of numbers has 10 separate symbols namely 0,1,2,3,4,5,6,7,8 and 9, which are called Arabic numerals. We would be forced to stop at 9 or to invent more symbols if it were not for the use of positional notation.

- The digit of a number system is a symbol, which represents an integral quantity.
- The base or radix of a number system is defined as the number of different digits, which can occur in each position in the number system. The decimal system has a base or radix of 10.

An example of earlier types of notation can be found in Roman Numerals, which are essential additive: III=I+I+I, XXV=X+X+V. The only importance of position in Roman numerals lies in whether a symbol precedes or follows another symbol (IV= 4, while VI=6).

In the beginning it was so difficult for the mathematicians to calculate the roman numerals and to perform arithmetic operations but now it is a great beauty that it is enough to learn only the 10 basic numerals and the positional notational system in order to count as to any desired figure. After memorizing the addition and multiplication tables and learning a few simple rules, we can perform all arithmetic operations.

The actual meaning of 168 can be seen more clearly if we notice that it is spoken as “one hundred and sixty eight”. Basically, the number is a contraction of 1*100+6*10+8. The important point is that the value of each digit is determined by its position. Written numbers are always contracted, however, and only the basic 10 numerals are used, regardless of the size of the integer written. The general rule for representing numbers in the decimal system by using positional notation is as follows:
\[ a_{n-1} \cdot 10^{n-1} + a_{n-2} \cdot 10^{n-2} + \ldots + a_0 \text{ is expressed as } a_{n-1}, a_{n-2}, \ldots, a_0 \]

where \( n \) is the number of digits to the left of the decimal point.

### 1.2.2 BISTABLE DEVICES

The basic elements in early computer are relays and switches. The operation of a switch or relay can be seen to be essentially bi-stable, or binary in nature; that is, the switch is either on (1) or off (0). The principal circuit elements in modern computers are transistors. Because of the large number of electronic parts used in computers, it is highly desirable to utilize them in such a manner that slight changes in their characteristics will not affect their performance. The best way of accomplishing this is to use circuits, which are basically bi-stable (have two possible states).

### 1.2.3 BINARY NUMBER SYSTEM

Digital computers use the binary number system, which has only two symbols: 0 and 1. The numbers in binary system are represented as combinations of these two symbols. The decimal system uses powers of 10 and binary system uses powers of 2.

The binary digit is also referred to as Bit (the acronym for Binary Digit). A string of 4 bits is called a nibble and a string of 8 bits is called a byte. A byte is the basic unit of data in computers. The number 125 actually means \(1 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0\). In binary system, the same number (125) is represented as 111101 meaning

\[1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0\]

The table 1.1 lists the first 20 binary numbers.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>15</td>
<td>1111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>16</td>
<td>10000</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>17</td>
<td>10001</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>18</td>
<td>10010</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>19</td>
<td>10011</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>20</td>
<td>10100</td>
</tr>
</tbody>
</table>

Table 1.1 First 20 Binary Numbers
To express the value of a binary number, therefore, \( a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \ldots \ldots + a_0 \) is expressed as \( a_{n-1}, a_{n-2} \ldots a_0 \) where \( a_i \) is either 1 or 0 and \( n \) is the number of digits to the left of the binary (radix) point.

### 1.2.4 OCTAL NUMBER SYSTEM

The octal number system has a base, or radix as 8: eight different symbols are used to represent numbers. These are commonly 0,1,2,3,4,5,6,7. We show the first 20 octal numbers and their decimal equivalents in the table 1.2.

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>OCTAL</th>
<th>DECIMAL</th>
<th>OCTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>21</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 1.2 First 20 Octal Numbers

To convert an octal number to a decimal number, we use the same sort of polynomial as was used in the binary case, except that we now have a radix of 8 instead of 2. Therefore 1213 in octal is

\[
=1 \times 8^3 + 2 \times 8^2 + 1 \times 8^1 + 3 \times 8^0 \\
=512 + 128 + 8 + 3 = 651
\]

in decimal. Also, 1.123 in octal is \(1 \times 8^0 + 1 \times 8^{-1} + 2 \times 8^{-2} + 3 \times 8^{-3} = 1.83/512 \) in decimal.

### 1.2.5 HEXADECIMAL NUMBER SYSTEM

When the machine is handling numbers in binary but in groups of four digits, it is convenient to have a code for representing each of these sets of four digits. Since 16 possible different numbers can be represented, the digits 0 through 9 will not suffice. So the letters A, B, C, D, E and F are also used. Hexadecimal numbers are strings of these digits. The numbers in decimal, binary and hexadecimal is shown in the table 1.3.

<table>
<thead>
<tr>
<th>BINARY</th>
<th>HEXADECIMAL</th>
<th>DECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
The base or radix of a number system is defined as the number of different digits, which can occur in each position in the number system. The decimal number system has a base, or radix of 10. Thus the system has 10 different digits (0,1,2,...,9), any one of which may be used in each position in a number. History records the use of several other number systems.

### Table 1.3 First 16 Hexadecimal Numbers

| 0011 | 3   | 3   |
| 0100 | 4   | 4   |
| 0101 | 5   | 5   |
| 0110 | 6   | 6   |
| 0111 | 7   | 7   |
| 1000 | 8   | 8   |
| 1001 | 9   | 9   |
| 1010 | A   | 10  |
| 1011 | B   | 11  |
| 1100 | C   | 12  |
| 1101 | D   | 13  |
| 1110 | E   | 14  |
| 1111 | F   | 15  |

The base or radix of a number system is defined as the number of different digits, which can occur in each position in the number system. The decimal number system has a base, or radix of 10. Thus the system has 10 different digits (0,1,2,...,9), any one of which may be used in each position in a number. History records the use of several other number systems.

### Self Check Exercise 1

**Say True or False**

1. A byte is equal to 8 bits and can represent a character internally.  \[ \text{True/False} \]
2. A program is a sequence of instructions designed for achieving a task/goal. \[ \text{True/False} \]
3. One MB is equal to 1024 KB \[ \text{True/False} \]

### 1.3 CONVERSIONS

#### 1.3.1. DECIMAL NUMBER TO BINARY NUMBER

To convert a decimal number into binary number it requires successive division by 2 writing down each quotient and its remainder. The remainders are taken in the reverse order, which is the binary equivalent of the decimal number. For example, let it is required to convert the decimal number 25 to its binary equivalent.

\[
\begin{align*}
2 & \mid 25 \\
2 & \mid 12 -1 \\
2 & \mid 6 -0 \\
2 & \mid 3 -0 \\
1 & \mid 1 -1 \\
\end{align*}
\]

The binary equivalent for \(25_{10} = 11001_2\)
To convert decimal fractions into equivalent binary fractions repeatedly double the
decimal fraction. The number (0 or 1) that appears on the left is written separately. The bits that
are written in this manner are read from top to bottom with a decimal point on the left. For
example if the given number is 0.0625, conversion is done in the following manner.

\[
\begin{array}{l}
0.625 \times 2 = 0.1250 \\
0.125 \times 2 = 0.25 \\
0.25 \times 2 = 0.5 \\
0.5 \times 2 = 1.0 \\
\end{array}
\]

Bits on the left: 0 0 0 1

The Multiplication cannot be continued further, as the fractional part in the previous step
has already become zero. Therefore, \(0.0625_{10} = .0001_2\)

1.3.2. DECIMAL NUMBER TO OCTAL NUMBER

Conversion from decimal to octal can be performed by repeatedly dividing the decimal
number by 8 and using each remainder as a digit in the octal number being formed. For instance,
to convert decimal number 200 to an octal representation, we divide as follows.

\[
\begin{array}{c|c}
8 & 200 \\
8 & 25 \\
3 & -0 \\
\end{array}
\]

Therefore \((200)_{10} = (310)_8\)

1.3.3. DECIMAL NUMBER TO HEXADECIMAL NUMBER

One way to convert decimal to Hexadecimal is the hex dabbles. The idea is as divide
successively by 16, writing down the remainders. Here is a sample of how it is done. To convert
decimal 2429 to hexadecimal,

\[
\begin{array}{c|c}
16 & 2429 \\
16 & 154 - 15 \\
9 & -10 \\
\end{array}
\]

Therefore \((2429)_{10} = (9AF)_{16}\)

1.3.4. BINARY NUMBER TO DECIMAL NUMBER

For converting the value of Binary numbers to decimal equivalent we have to find its
quantity, which is found by multiplying a digit by its place value. The following example
illustrates the conversion of binary numbers to decimal system.
101 = \(1 \times 2^3 - 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 1 = 9\)

1001 = \(1 \times 2^4 - 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 16 + 0 + 0 + 0 + 1 = 17\)

11.011 = \(1 \times 2^1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 2 + 0.5 + 0.125 = 2.625\)

1.3.5. **BINARY NUMBER TO OCTAL NUMBER**

There is a simple trick for converting a binary number to an octal number. Simply group the binary digits into groups of 3, starting at the octal point, and read each set of three binary digits according to the following table 1.4.

<table>
<thead>
<tr>
<th>BINARY</th>
<th>OCTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

*Table 1.4 First 8 Octal Numbers*

Let us convert the binary number 011101 into octal. First, we break binary number into 3 digits (011 101). Then converting each group of three binary digits, we get 35 in octal. Therefore 011101 binary = 35 in octal.

\[
011101 = 011 \ 101 \\
= 3 \ 5 \\
= (3 \ 5)_8
\]

1.3.6. **BINARY NUMBER TO HEXADECIMAL NUMBER**

To convert binary to hexadecimal, we simply break a binary number into groups of four digits and convert each group of four digits according to the preceding code. Here are some examples:

**Example:**

\[
(10111011)_2 = 1011 \ 1011 = B \ B
\]
Example: 2

\[ (10010101)_2 = 1001 \ 0101 \]
\[ = 9 \ 5 \]
\[ = (9\ 5)_{16} \]

1.3.7. OCTAL NUMBER TO DECIMAL NUMBER

To convert an octal number to a decimal number, we use the same sort of polynomial as was used in the binary case, except that we now have a radix of 8 instead of 2. Therefore 1213 in octal is

\[ = 1 \times 8^3 + 2 \times 8^2 + 1 \times 8^1 + 3 \times 8^0 \]
\[ = 512 + 128 + 8 + 3 = 651 \]
in decimal. Also, 1.123 in octal is \[ 1 \times 8^0 + 1 \times 8^{-1} + 2 \times 8^{-2} + 3 \times 8^{-3} = 1 \frac{83}{512} \] in decimal

1.3.8. OCTAL NUMBER TO BINARY NUMBER

The conversion from octal number to binary number is easily accomplished. Each octal bit is converted to its three digit binary equivalent.

Example: 1

\[ (2\ 6\ 1\ 5\ 3\ .\ 7\ 4\ 0\ 6)_{8} \]
\[ = (010\ 110\ 001\ 101\ 011.\ 111\ 111\ 100\ 000\ 110)_{2} \]

1.3.9. OCTAL NUMBER TO HEXADECIMAL NUMBER

The method of converting octal to hexadecimal number is to convert the given octal number to binary number and then arrange the binary digits into groups of 4 starting at the binary point.

Example: 1

Convert octal number 714.06 to hexadecimal.

\[ (714.06)_{8} = (111\ 001\ 100.000\ 110)_{2} \]
\[ = 0001\ 1100\ 1100.\ 0001\ 1000 \]
\[ = 1\ C\ C.\ 1\ 8 \]

The hexadecimal equivalent of \( (714.06)_{8} \) is \( (1\ CC.18)_{16} \)

1.3.10. HEXADECIMAL NUMBER TO BINARY NUMBER

To convert a hexadecimal number to a binary number, convert each hexadecimal digit to its 4-bit equivalent using the code. For instance, here’s how 9AF converts to binary.

\[ (3\ 0\ 6\ .\ D)_{16} \]
\[ = (0011\ 0000\ 0110.\ 1101)_{2} \]
\[ = 9\ A\ F \]
As another example, C5E2

\[
\begin{array}{c}
1001 \\
1010 \\
1111 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
C & 5 & E & 2 \\
1100 & 0101 & 1100 & 0010 \\
\end{array}
\]

### 1.3.11. HEXADECIMAL NUMBER TO OCTAL NUMBER

The conversion of Hexadecimal number to octal number involves two steps. First the method suggests to go from hexadecimal to binary numbers and second from binary to octal numbers. Convert the hexadecimal into binary by writing 4 bits binary value for each bit in hexadecimal number and then arrange the binary digits into groups of three starting at the binary point.

Example: convert Hexadecimal (1E.C) to octal conversion

\[
(1E.C)_{16} = (0001 1110.1100)_2 \\
= (011 110.110) \\
= 36.6
\]

The octal equivalent of \((1E.C)_{16}\) is \((36.6) _8\)

### 1.3.12. HEXADECIMAL NUMBER TO DECIMAL NUMBER

The conversion of Hexadecimal to decimal is straightforward but time consuming. In Hexadecimal number system each digit position corresponds to a power of 16. The weights of the digit positions in a hexadecimal number are as follows: For instance, BB represents

\[
BB = B*16^1 + B*16^0 \\
= 11*16 + 11*1 \\
= 176 + 11 \\
= 187
\]

**Self – Check Exercise 2**

1. Convert the following binary numbers to decimal.

   (a) 1100.1101 \\
   (b) 10101010

2. Convert the following decimal numbers to binary.

   (a) 23 \\
   (b) 49.25 \\
   (c) 892
1.4 BINARY ADDITION AND SUBTRACTION

Binary addition is performed in the same manner as decimal addition. The complete table for binary addition is as follows:

\[
\begin{align*}
0+0 &= 0 \\
0+1 &= 1 \\
1+0 &= 1 \\
1+1 &= 0 \text{ plus a carry over of 1}
\end{align*}
\]

‘Carry over’ are performed in the same manner as in decimal arithmetic. Since 1 is the largest digit in the binary system, any sum greater than 1 requires that a digit be carried.

Examples:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>101</td>
<td>3 ¼</td>
<td>11.01</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>5 ¾</td>
<td>101.11</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>9</td>
<td>1001.00</td>
</tr>
</tbody>
</table>

Subtraction is the inverse operation of addition. To subtract, it is necessary to establish a procedure for subtracting a larger from a smaller digit. The only case in which this occurs with binary numbers is when 1 is subtracted from 0. It is necessary to borrow 1 from the next column to the left. This is the binary subtraction table.

\[
\begin{align*}
0-0 &= 0 \\
1-0 &= 1 \\
0-1 &= 1 \text{ with a borrow of 1} \\
1-1 &= 0
\end{align*}
\]

Examples:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1001</td>
<td>16</td>
<td>10000</td>
</tr>
<tr>
<td>-5</td>
<td>-101</td>
<td>-3</td>
<td>-11</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>13</td>
<td>1111</td>
</tr>
</tbody>
</table>

1.5 BINARY MULTIPLICATION AND DIVISION

The table for binary multiplication is very short, with only four entries instead of the many for normal decimal multiplication

\[
\begin{align*}
0*0 &= 0 \\
0*1 &= 0 \\
1*0 &= 0 \\
1*1 &= 1
\end{align*}
\]

The following examples of binary multiplication illustrate the simplicity of each operation. It is only necessary to copy the multiplicand if the digit in the multiplier is 1 and to copy all 0’s if the digit in the multiplier is a 0.
The complete table for binary division is as follows:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0/1=0</td>
<td>1/1=1</td>
</tr>
</tbody>
</table>

Examples:

```
111

110

110 110

110 110

110 0
```

1.6 FLOATING POINT REPRESENTATION

Floating point numbers consists of two parts. The first part of the number is a signed fixed-point number, which is termed as mantissa, and the second part specifies the decimal or binary point position and is termed as an Exponent. The mantissa can be an integer or a fraction.

Example:

A decimal +12.34 in a typical floating-point notation is 12.34=0.1234 * 10^2

```
<table>
<thead>
<tr>
<th>Sign</th>
<th>Sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1234</td>
<td>02</td>
</tr>
</tbody>
</table>
```

Mantissa (fraction)  Exponent
12.34 = 1234 * 10^{-2}

<table>
<thead>
<tr>
<th>Sign</th>
<th>0 1234</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign</td>
<td>1 02</td>
</tr>
</tbody>
</table>

This number in any of the above form (if represented in BCD) requires 17 bits for mantissa (1 for sign and 4 each decimal digit as BCD) and 9 bits for exponent (1 for sign and 4 for each decimal digit as BCD). Exponent indicates the correct decimal location. In the first case where exponent is $+2$, indicates that actual position of the decimal point is 2 places to the right of the assumed position, while exponent $-2$ indicates that the assumed position of the point is 2 places towards the left of assumed position. The assumption of the position of the point is normally the same in a computer resulting in a consistent computational environment.

Floating-point numbers are often represented in normalized form. A floating-point number whose mantissa does not contain zero as the most significant digit of the number is considered to be in a normalized form. For example, a BCD mantissa $+370$ which is 0 0011 0111 000 is in normalized form because these leading zero’s are nor part of a 0 digit. On the other hand a binary number 0 01100 is not in a normalized form. The normalized form of this number will be 0 1100

Arithmetic operations involved with floating point numbers are more complex in nature, takes longer time for execution and require complex hardware. Yet the floating point representation is a must as it is useful in scientific calculations. Real numbers are normally represented as floating point numbers.

1.7 COMPLEMENTS

Complements are quite often used to represent negative numbers in digital computers for simplifying the subtraction operation and logical manipulation. For instance, the number $N_2$ has to be subtracted from $N_1$ i.e, $N_1 - N_2$ then without using subtraction the complement form of negative number is formed and then added. It can be pointed out that since the subtraction of number $N_2$ from $N_1$ is same as the addition of $N_1$ and complement of $N_2$ (i.e) $N_1 + (-N_2)$. There are two types of complements for each base $-1$ system.

2. The r’s Complement
3. The (r-1)’s Complement

When the value of the base is substituted the two types receive the names 2’s and 1’s complement for binary number or 10’s and 9’s complement for decimal numbers. The r’s complement is sometimes called as “True Complement” and the (r-1)’s complement as “Radix minus one’s complement”.
1.7.1 The (r-1)’s complement

Given a positive number N in base r with an integer part of N digits and a fraction part of m digits, then (r-1)’s complements can be defined as $r^n - r^m - N$. The (r-1)’s complement in decimal system is 9’s complement and 1’s complement in case of binary. Some numerical examples of 9’s complement is as follows:

Example: 1 The 9’s complement of 52510 is

$(10^5 -1) - 52510$

$=99999 - 52510$

$=47489$

The 9’s complement of 0.3266 is

$=(1-10^{-4}) - 0.3266$

$=0.9999 - 0.3266$

$=0.6733$

The 9’s complement of 25.638 is

$=(10^2 - 10^{-3}) - 25.638$

$=99.999 - 25.638$

$=74.361$

Example: 2

The 1’s complement of 101100 is

$=(2^6 - 1) - 101100$

$=(1000000 - 1) - 101100$

$=111111 - 101100$

$=010011$

The 1’s complement of (0.0110)$_2$ is

$=(1-2^{-4})_{10} - 0.0110$

$=1 - 0.0001 - 0.0110$

$=0.1111 - 0.0110$

$=0.1001$

The 1’s complement of 10.101 is

$=(2^2 - 2^{-3})_{10} - 10.101_B$

$=(100 - 0.001)_B - 10.101_B$

$=11.1111 - 10.101_B$

$=1.010$

1.7.2 The r’s complement

Given a positive number N in base r with an integer part of n digits the r’s complement of N is defined as $r^n - N$ for N $\neq$ 0 and 0 for N=0. The r’s complement in decimal system is 10’s complement and 2’s complement in case of binary system.

Example: 1

The 10’s complement of $(52510)_{10}$ is

$=10^5 - 52510$

$=47490$
The 10’s complement of 0.3266_{10} is
= 1 - 0.3266
= 0.6734

The 10’s complement of 25.638_{10} is
= 10^2 - 25.638
= 74.362

The 2’s complement of 101100 is
= (2^6) – 101100_B
= (1000000 - 101100)_B
= 010011

The 2’s complement of (0.0110)_B is
= (1 - 0.0110)_B
= 0.1010

The 2’s complement of 10.101 is
= (2^2)_D - 10.101
= (100 - 10.101)_D
= 01.011

The r’s complement can be obtained from the (r-1)’s complement after the addition of r to the least significant digit.

From the examples, one can notice that:
- 9’s complement of a decimal number is formed simply by subtracting every digit by 9.
- The 1’s complement of a binary number is even simpler to form, the 1’s are changed to 0’s and 0’s are changed to 1’s.

For example, one can notice that:
- 2’s complement of 101101 is obtained from the 1’s complement 010010 by adding 1 to give 010011.
- 10’s complement can be formed by forming 9’s complement and then adding a 1 to the least significant digit.

Self – Check Exercise 3

1. Find the 1’s and 2’s complement of the following fixed point numbers
   (a) 10100010
   (b) 00000000
   (c) 11001100

2. Add the following numbers in 8 – bit register using signed 2’s complement notation.
   (a) +50 and -5
   (b) +45 and -65
   (c) +75 and +85
   (d) -75 and -85

Also indicate the overflow condition if any.
1.8 BINARY CODED DECIMAL NUMBER REPRESENTATION

In BCD number system a group of binary bit is used to represent each of 10 decimal digits. For instances, an obvious and natural code is a simple weighted binary code as shown in table 1.5.

<table>
<thead>
<tr>
<th>BINARY CODE</th>
<th>DECIMAL DIGIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1.5 Binary Coded Decimal

This is known as a binary coded decimal 8421 code or simply BCD. Notice that 4 binary bits are required for each decimal digit and each bit is assigned a weight; for instance the rightmost bit has a weight of 1, and the leftmost bit in each code group has a weight of 8. By adding the weights of the positions in which 1’s appear, the decimal digit represented by a code group may be derived. This is somewhat uneconomical since $2^4 = 16$, and thus the 4 bits could actually represent 15 different values. For the decimal number 1246 to be represented, 16 bits are required: 0001 0010 0100 0110

Examples:
Convert decimal 4019 to BCD

\[
\begin{array}{cccc}
4 & 0 & 1 & 9 \\
0100 & 0000 & 0001 & 1001 \\
\end{array}
\]

The BCD equivalent of $(4019)_{10}$ is 0100 0000 0001 1001

Convert BCD number 0001 1001 0000 0111 to decimal

\[
\begin{array}{c|c|c|c}
0001 & 1001 & 0000 & 0111 \\
1 & 9 & 0 & 7 \\
\end{array}
\]

The decimal equivalent of BCD Number 0001 1001 0000 0111 is 1907. BCD numbers are useful wherever decimal information is transferred into a computer. The pocket calculator is one of the best examples for the application of BCD numbers. Other examples of BCD system are electronic counters, digital voltmeter and digital clocks.
1.9 EXCESS – 3 CODE:

The Excess-3 code is a decimal code that has been used in older computers. This is an unweighted code. Its binary code assignment is obtained from the corresponding BCD equivalent binary number after the addition of binary 3 (0011).

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>BCD</th>
<th>EXCESS 3 CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0100</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0101</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0110</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0111</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>1001</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1010</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1011</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>1100</td>
</tr>
<tr>
<td>10</td>
<td>0001 0000</td>
<td>0100 0011</td>
</tr>
<tr>
<td>11</td>
<td>0001 0001</td>
<td>0100 0100</td>
</tr>
</tbody>
</table>

Table 1.6 Excess – 3 code

1.10 GRAY CODE

Digital systems can process data in discrete form only. Many physical systems supply continuous output data. The data must be converted into digital form before they can be used by a digital computer. Continuous, or analog information is converted into digital form by means of an analog to digital converter. The reflected binary or gray code is shown in the table is sometimes used for the converted digital data. The advantage of the gray code over straight binary numbers is that the gray code changes by only one bit as it sequences from one number to the next. In other words, the change from any number to the next in sequence is recognized by a change of only one bit from 0 to 1 data is represented by the continuous change of a shaft position. The shaft is partitioned into segments with each segment assigned a number. If adjacent segments are made to correspond to adjacent Gray code numbers, ambiguity is reduced when the shaft position is in the line that separates any two segments.

Gray code counters are sometimes used to provide the timing sequences that control the operations in a digital system. A gray code counter is a counter whose flip-flop go through a sequence of states. Gray code counters remove the ambiguity during the change from one state of the counter to the next because only one bit can change during the state transition.

<table>
<thead>
<tr>
<th>Gray Code</th>
<th>Decimal Equivalent</th>
<th>Gray Code</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>1100</td>
<td>8</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1101</td>
<td>9</td>
</tr>
<tr>
<td>0011</td>
<td>2</td>
<td>1111</td>
<td>10</td>
</tr>
<tr>
<td>0010</td>
<td>3</td>
<td>1110</td>
<td>11</td>
</tr>
</tbody>
</table>
Table 1.7 : 4 Bit Gray Code

<table>
<thead>
<tr>
<th>0110</th>
<th>4</th>
<th>1010</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>5</td>
<td>1011</td>
<td>13</td>
</tr>
<tr>
<td>0101</td>
<td>6</td>
<td>1001</td>
<td>14</td>
</tr>
<tr>
<td>0100</td>
<td>7</td>
<td>1000</td>
<td>15</td>
</tr>
</tbody>
</table>

1.11 ARITHMETIC CIRCUITS

Arithmetic circuits such as binary adders, parallel binary adder and BCD adder are explained with circuit diagram.

1.11.1 HALF ADDER

A basic module used in binary arithmetic elements is the half-adder. The function of the half-adder is to add two binary digits, producing a sum according to the binary addition rules shown in the table 1.8.

<table>
<thead>
<tr>
<th>INPUT</th>
<th>SUM OF BITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0+0</td>
<td>0</td>
</tr>
<tr>
<td>0+1</td>
<td>1</td>
</tr>
<tr>
<td>1+0</td>
<td>1</td>
</tr>
<tr>
<td>1+1</td>
<td>0 With a carry of 1</td>
</tr>
</tbody>
</table>

Table 1.8 Addition Table

The following Figure 1.1 shows a design for a half-adder, two inputs are designated as X and Y and two outputs, designated as S and C. The half-adder perform binary addition operation for two binary inputs as shown in table 1.9. This is arithmetic addition, not logical or Boolean algebra addition.

Figure 1.1 Half Adder – Block Diagram

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1.9 Truth Table

Figure 1.2 Logic Diagram
In the half-adder diagram there are two inputs to the half-adder and two outputs. If either of the inputs is a 1 but not both, then the output on the S line will be a 1. If both inputs are 1s, the output on the C line will be a 1. For all other states, there be a 0 output on the carry line. These relationships may be written in Boolean form as follows.

\[ S = XY' + X'Y \]
\[ C = XY \]

### 1.11.2 FULL ADDER

The adder circuit is capable of adding the content of two registers. It must include provision for handling carries as well as an addend and augends bits. So there must be three inputs to each stage of a multi digit adder, except the stage for the least significant bits. One for each input from the numbers being added, one for any carry that might have been generated or propagated by the previous stage.

There are three inputs to the full-adder X and Y inputs from the respective digits of the registers to be added, the \( C_i \) input, which is for any carry generated by the previous stage. The two outputs are \( S \), which is the output value for that stage of the addition, and \( C_0 \), which produces the carry to be added into the next stage. The Boolean expressions for the input output relationships for each of the two outputs are as follows:

\[
\begin{array}{c|c|c|c|c|c}
X & Y & C_i & S & C_0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

**Table 1.10 Truth Table**
1.11.3 PARALLEL BINARY ADDER

The purpose of this adder is to add two 4-bit binary integers. The adder inputs are named \( X_0 \) through \( X_3 \) and the augend bits are represented by \( Y_0 \) through \( Y_3 \). A 4-bit parallel binary adder is illustrated in the figure 1.5.

Consider the addition of the following two 4-bit binary numbers:

\[
\begin{align*}
0111 & \text{ where } X_3 = 0, X_2 = 1, X_1 = 1, \text{ and } X_0 = 1 \\
0011 & \text{ where } Y_3 = 0, Y_2 = 0, Y_1 = 1, \text{ and } Y_0 = 1
\end{align*}
\]

**Sum** = 1010

The sum should therefore be \( S_3 = 1, S_2 = 0, S_1 = 1 \) and \( S_0 = 0 \). The operation of the adder may be checked as follows. Since \( X_0 \) and \( Y_0 \) are the least significant digits, they cannot receive a carry from a previous stage. In the problem above \( X_0 \) and \( Y_0 \) are both 1s, their sum therefore 0 and a carry is generated and added into the full-adder for bits \( X_1 \) and \( Y_1 \). Bits \( X_1 \) and \( Y_1 \) are also both 1s, as is the carry input to this stage.

Therefore the sum output line \( S_1 \) carries a 1 and the carry line to the next stage also carries a 1. Since \( X_2 \) is a 1, \( Y_2 \) is a 0 and the carry input is 1. The sum output line \( S_2 \) will carry a 0, and the carry to the next stage will be a 1. Both inputs \( X_3 \) and \( Y_3 \) are equal to 0, and the carry input line to this adder stage is equal to 1. Therefore, the sum output line \( S_3 \) will represent a 1 and the carry output line designated as “overflow” will have a 0 output.
1.11.4 BINARY CODED DECIMAL ADDER

Arithmetic units which perform operations on numbers stored in BCD form must have the ability to add 4-bit representations of decimal digits. To do this BCD adder is used. A block diagram symbol for an adder is shown in the figure 1.6.

Figure 1.6 BCD Adder

The adder has an augends digit input consisting of four lines, an addend digit input for four lines, a carry-in and a carryout, and a sum digit with four output lines. The augend digit, addend digit and sum digit with four output line. The augend digit, addend digit and sum digit are each represented 8, 4, 2, 1 BCD form. The purpose of the BCD adder in figure 1.6 is to add the augend and addend digits and the carry-in and produce a sum digit and carry out.

There are eight inputs to the BCD adder; four $X_i$, or augend, inputs and four $Y_i$ (or) addend digits. Each input will represent a 0 or a 1, during a given addition. If 3 (0011) is to be added to 2 (0010), then $X_8=0$, $X_4=0$, $X_2=1$ and $X_1=1$; $Y_8=0$, $Y_4=0$, $Y_2=1$ and $Y_1=0$.

A further difficulty arises when a carry is generated. If $7_{10}$ (0111) is added to $6_{10}$ (0110), a carry will be generated, but the output from the base – 16 adder will be 1101. This 1101 does not represent any decimal digit in the 8,4,2,1 system and must be connected. The method used to correct this is to add $6_{10}$ (0110) to the sum from the base – 16 address whenever a carry is generated. This addition is performed by adding 1’s to the weight 4 and weight 2 position output lines from the base – 16 adder when a carry is generated.

The adder performs base – 16 addition and corrects the sum, if it is greater than a1 by adding 6. Several Examples of this are shown below.
(i) $8+7=15$  
$1000+0111$

(ii) $9+5=14$  
$1001$
$0101$
$1110$
$+0110$
Carry Generated $10100 = 14$

1.11.5 HALF SUBTRACTOR

A half subtractor subtracts a bit from another. The subtraction table (or truth table) of a half subtractor is shown below. The half subtractor has two input bits $A$ and $B$ two output bits, a difference $DIFF = (A - B)$ and a Borrow.

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>BORROW</th>
<th>DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1.11 Truth Table

From the truth table shown in the table 1.11, it can be seen that $DIFF = A - B$ and borrow which has been implemented on the logic circuit for half subtractor.

1.11.6 FULL SUBTRACTOR

A full subtractor subtracts with three bits ($A - B - C$). The third bit $C$ is the borrow from previous stage. The truth table of a full subtractor is given in table 1.12.

From the truth table of the full subtractor it can be seen that $DIFF = (A \ B \ C)$ and borrow $= A' B + BC + CA'$. This logic has been implemented.
When a binary number is to be subtracted from another using 1’s complement method, the following circuit can be used. The number to be subtracted is first complemented using inverters. The complemented number is then added to the minuend using full adders. The carry resulting from the addition is added to the least significant bit as shown in Figure 1.8.

**PARALLEL BINARY ADDER - SUBTRACTOR**

In implementing a combined adder and subtractor circuit, the primary problem of complementing a number has to be taken care, that is, to form the complement of the number to be subtracted. For the 1’s complement system, if the storage register is composed of flip flops, the 1’s complement can be formed by simply connecting the complement of each input to the
adder. The 1 which must be added to the least significant position to form a 2’s complement may be added when the two numbers are added by connecting a 1 at the CARRY input of the adder for the least significant bits.

Figure. 1.8.(a) Parallel Arithmetic Element

A complete logical circuit capable of adding or subtracting two signed 2’s complement number is shown below.

To add: Add line is made 1.
To Subtract: Subtract line is made 1.

One number is represented by $X_1 X_0$ and the other number is represented by $Y_1 Y_0$. There are two control signals, ADD and SUBTRACT. If the add control line is made a 1 then the sum of the number $X$ and $Y$ will appear as $S_1 S_0$. If the subtract line is made a 1, then the difference between $X$ and $Y$ will appear $S_1 S_0$.

The AND to OR gate network connected to each input selects either $Y$ or $Y$. An ADD causes $Y_1$ to enter the appropriate full adder, while a subtract causes $Y_1$ to enter the full adder, to either add or subtract each $x$ input is connected to the appropriate full adder, and a 1 is added by connecting the SUBTRACT signal to the $C_i$ input of the full adder for the lowest order bits $X_0$ and $Y_0$. 
The configuration in the figure 1.8.1 is the most frequently used for addition and subtraction because it provides a simple, direct means for either adding or subtracting positive numbers or negative numbers.

**Self Check Exercise 4**

1. Can a full adders be constructed using two half adders?

   

   

   

   

1.12 Digital Logic

Since Boolean functions are expressed in terms of AND, OR & NOT operations, it is easier to implement a Boolean function with these types of gates. The possibility of constructing gates for other logic operations is of practical interest. Factors to be weighted when considering the constructing of other types of logic gates are

- (1) The feasibility and economy of producing the gate with physical components,
- (2) The possibility of extending the gate to more than two inputs,
- (3) The basic properties of the binary operator, such as commutativity and associativity, and
- (4) The ability of the gate to implement Boolean functions alone or in conjunction with other gates.

<table>
<thead>
<tr>
<th>Boolean Function</th>
<th>Operator Symbol</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>F₀=0</td>
<td></td>
<td>Null</td>
<td>Binary constant 0</td>
</tr>
<tr>
<td>F₁=xy</td>
<td>x.y</td>
<td>AND</td>
<td>x and y</td>
</tr>
<tr>
<td>F₂=xy’</td>
<td>x/y</td>
<td>Inhibition</td>
<td>x but not y</td>
</tr>
<tr>
<td>F₃=x</td>
<td></td>
<td>Transfer</td>
<td>x</td>
</tr>
<tr>
<td>F₄=x’y</td>
<td>y/x</td>
<td>Inhibition</td>
<td>y but not x</td>
</tr>
<tr>
<td>F₅=y</td>
<td></td>
<td>Transfer</td>
<td>Y</td>
</tr>
<tr>
<td>F₆=xy’+x’y</td>
<td>x  y</td>
<td>Exclusive – OR</td>
<td>x or y, but not both</td>
</tr>
<tr>
<td>F₇=x+y</td>
<td>x+y</td>
<td>OR</td>
<td>X or y</td>
</tr>
<tr>
<td>F₈=(x+y)’</td>
<td>x  y</td>
<td>NOR</td>
<td>Not – OR</td>
</tr>
<tr>
<td>F₉=xy+x’y’</td>
<td>(x  y)’</td>
<td>Equivalence</td>
<td>x equals y</td>
</tr>
<tr>
<td>F₁₀=y’</td>
<td>y’</td>
<td>Complement</td>
<td>Not y</td>
</tr>
<tr>
<td>F₁₁=x+y’</td>
<td>x  y’</td>
<td>Implication</td>
<td>If y, then x</td>
</tr>
<tr>
<td>F₁₂=x’</td>
<td>x  y’</td>
<td>Complement</td>
<td>Not x</td>
</tr>
<tr>
<td>F₁₃=x’+y</td>
<td>(x  y)’</td>
<td>Implication</td>
<td>If x, then y</td>
</tr>
<tr>
<td>F₁₄=(xy)’</td>
<td>x^ y</td>
<td>NAND</td>
<td>Not - AND</td>
</tr>
<tr>
<td>F₁₅=1</td>
<td>All ones</td>
<td>Identity</td>
<td>Binary Constant 1</td>
</tr>
</tbody>
</table>

Table 1.13 Boolean Expression for the 16 Functions of two variables
Of the 16 functions defined in the table 1.13, two are equal to a constant and four are repeated twice. There are only ten functions left to be considered as candidates for logic gates. Two-inhibition and implication-are not commutative or associative and thus are impractical to use as other logic gates. The other eight: complement, transfer, AND, OR, NAND, NOR, exclusive-OR, and equivalence, are used as standard gates in digital design.

Each gate has one or two binary input variables designated by \( x \) and \( y \) and one binary output variable designated by \( F \). The AND, OR, and inverter circuits were defined. The inverter circuit inverts the logic sense of a binary variable. It produces the NOT, or complement, function. The small circle in the output of the graphic symbol of the inverter (referred to as a *bubble*) designates the logic component. The triangle symbol by itself designates a buffer circuit. A buffer produces the transfer function, but does not produce a logic operation, since the binary value of the output is equal to the binary value of the input. This circuit is used for power amplification of the signal and is equivalent to two inverters connected in cascade.

The NAND function is the complement of the AND function, as indicated by a graphic symbol that consists of an AND graphic symbol followed by a small circle. The NOR function is the complement of OR function and uses the OR graphic symbol followed by a small circle. The NAND and NOR gates are extensively used as standard logic gates and are in fact far more popular than the AND and OR gates. This is because NAND and NOR gates are easily constructed with transistor circuits and because digital circuits can be easily implemented with them.

The exclusive-OR gate has a graphical symbol similar to that of the OR gate, except for the additional curved line on the input side. The equivalence, or exclusive-NOR, gate is the complement of the exclusive-OR, as indicated by the small circle on the output side of the graphic symbol.
<table>
<thead>
<tr>
<th>Name</th>
<th>Graphic Symbol</th>
<th>Algebraic Function</th>
<th>Truth Table</th>
</tr>
</thead>
</table>
| AND    | ![AND Symbol](image) | $F = xy$         | \[
|        |                |                   | $\begin{array}{c|c|c}
| x & y & F \\
| 0 & 0 & 0 \\
| 0 & 1 & 0 \\
| 1 & 0 & 0 \\
| 1 & 1 & 1 \\
| \end{array}$ | |
| OR     | ![OR Symbol](image) | $F = x + y$      | \[
|        |                |                   | $\begin{array}{c|c|c}
| x & y & F \\
| 0 & 0 & 0 \\
| 0 & 1 & 1 \\
| 1 & 0 & 1 \\
| 1 & 1 & 1 \\
| \end{array}$ | |
| Inverter | ![Inverter Symbol](image) | $F = x'$        | \[
|        |                |                   | $\begin{array}{c|c}
| x & F \\
| 0 & 1 \\
| 1 & 0 \\
| \end{array}$ | |
| Buffer | ![Buffer Symbol](image) | $F = x$         | \[
|        |                |                   | $\begin{array}{c|c}
| x & F \\
| 0 & 0 \\
| 1 & 1 \\
| \end{array}$ | |
| NAND   | ![NAND Symbol](image) | $F = (xy)'$      | \[
|        |                |                   | $\begin{array}{c|c|c}
| x & y & F \\
| 0 & 0 & 1 \\
| 0 & 1 & 1 \\
| 1 & 0 & 1 \\
| 1 & 1 & 0 \\
| \end{array}$ | |
EXTENSION TO MULTIPLE INPUTS

The gates shown in Figure 1.9 – except the inverter and buffer – can be extended to have more than two inputs. A gate can have extended to have multiple inputs if the binary operation it represents is commutative and associative. The AND and OR operations, defined in Boolean algebra, possess these two properties. For the OR function, we have

\[ x + y = y + x \] (commutative)

and

\[ (x + y) + z = x + (y + z) = x + y + z \] (associative)

which indicates that the gate inputs can be interchanged and that the OR function can be extended to three or more variables.

---

**Figure 1.9 Digital Logic Gates**

<table>
<thead>
<tr>
<th>NOR</th>
<th>x</th>
<th>y</th>
<th>F = (x + y)’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exclusive-OR (XOR)</th>
<th>x</th>
<th>y</th>
<th>F = xy’ + x’y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exclusive-NOR or equivalence</th>
<th>x</th>
<th>y</th>
<th>F = xy + x’y’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The NAND and NOR functions are commutative, and their gates can be extended to have more than two inputs, provided that the definition of the operation is slightly modified. The difficulty is that the NAND and NOR operators are not associative \([i.e., (x \sim y) \sim z \neq x \sim (y \sim z)\], as shown in Figure 1.10 and the following equations:

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

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

\[
x \sim y \sim z = ( x + y + z )' \\
x \^ y \^ z = ( xyz )'
\]

The graphic symbols for the 3-input gates are shown in Figure 1.11. In writing cascaded NOR and NAND operations, one must use the correct parenthesis to signify the proper sequence of the gates. To demonstrate this, consider the circuit of Figure 1.11 (c). The Boolean function for the circuit must be written as

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

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

\[
\text{(a) 3 - input NAND gate} \quad \text{b) 3 - input NOR gate}
\]
The exclusive-OR and equivalence gates are both commutative and associative and extended to more than two inputs. However, multiple-input exclusive-OR gates are uncommon from the hardware standpoint. In fact, even a 2-input function is usually constructed with other
types of gates. Moreover, the definition of the function must be modified when extended to more than two variables. The exclusive-OR is an odd function, i.e., it is equal to 1 if the input variables have an odd number of 1’s. The construction of a 3-input exclusive-OR function is shown in Figure 1.12. It is normally implemented by cascading 2-input gates, as shown in (a). Graphically, it can be represented with a single 3-input gate, as shown in (b). The truth table in table 1.14 clearly indicates that the output F is equal to 1 only if one input is equal to 1 or if all three inputs are equal to 1, i.e., when the total number of 1’s in the input variable is odd.

**POSITIVE AND NEGATIVE LOGIC**

The binary signal at the inputs and outputs of any gate has one of the two values, except during transition. One signal value represents logic-1 and the other logic-0. Since two signal values are assigned to two logic values, there exists two different assignments of signal level to logic value, as shown in Figure 1.13. The higher signal level is designated by H and the lower signal level is designated by L. Choosing the high-level H to represent logic-1 defines a positive logic system. Choosing the low-level L to represent logic-1 defines a negative logic system. The terms positive and negative are somewhat misleading since both signals may be positive or both may be negative. It is not the actual sign values that determine the type of logic, but rather the assignment of the logic values to the relative amplitudes of the two signal levels.

<table>
<thead>
<tr>
<th>Logic Value</th>
<th>Signal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>L</td>
</tr>
</tbody>
</table>

(a) Positive Logic

<table>
<thead>
<tr>
<th>Logic Value</th>
<th>Signal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>L</td>
</tr>
</tbody>
</table>

(b) Negative Logic

---

**Figure 1.13 Signal Assignment and Logic Polarity**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
</tbody>
</table>

(a) Truth table with H and L

---

**Figure 1.14 Digital Gate Block Diagram**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Hardware digital gates are defined in terms of signal values such as H and L. It is up to the user to decide on a positive or negative logic polarity. Consider, for example, the electronic gate shown in Figure. 1.14. The truth table for this gate is listed in Figure. 1.14 (a). It specifies the physical behavior of the gate when H is 3 volts and L is 0 volts. The truth table of Figure. 1.14 (c) assumes positive logic assignment, with H=1 and L=0. This truth table is the same as the one for the AND operation. The graphical symbol for a positive logic AND gate is shown in Figure. 1.14 (d).

Now consider the negative logic assignment for the same physical gate with L=1 and H=0. The result is truth table of Figure. 1.14 (e). This table represents the OR operation even though the entries are reversed. The graphical symbol for the negative logic OR gate is shown in Figure. 1.14 (f). The small triangles in the inputs and output designate a polarity indicator. The presence of this polarity indicator along a terminal signifies that negative logic is assumed for the signal. Thus, the same physical gate or as negative logic OR gate.

The conversion from positive logic to negative logic, and vice versa, is essentially an operation that changes 1’s to 0’s and 0’s to 1’s in both the inputs and the output of a gate. Since this operation produces the dual of a function, the change of all terminals from one polarity to the other results in taking the dual of the function. The result of this conversion is that all AND operations are converted to OR operations (or graphic symbols) and vice versa. In addition, one must not forget to include the polarity-indicator triangle in the graphic symbols when negative logic is assumed.

1.12.1 The Basic Gates
Logical Addition (OR Gate)

If A and B are the input logic variable and Y is the output Variable, the truth table for a two input OR gate is given in Figure. 1.15.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The OR gate produces the inclusive – OR function, that is, the output is 1. If input A or input B or both inputs are 1, otherwise the output is a 0. In other words, in an n input OR gate, if any input is a 1, the output is a 1. For an OR gate the logical relationship of inputs and output can be written as,

\[ Y = A + B + C + D + E + \ldots \]

**Logical Multiplication (AND Gate)**

The AND gate has one or more inputs and a single output. The output of an AND gate is equal to the multiplication of its inputs. The truth table for a two input AND gate is shown in Figure 1.16. The output is 1 if both A and B are 1 and 0 otherwise. In general, in an n input AND gate, only if all the inputs are at logic 1, the output will be 1.

The Input – Output relationships can be written as

\[ Y = A \cdot B \cdot C \cdot D \cdot E \cdot \ldots \]

Or \[ Y = ABCDEF \ldots \]

(a) Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Graphic Symbol

![AND Gate](image)

Figure 1.16 AND Gate

**Logical Inversion: Complement (NOT gate)**

The complement function is nothing but inversion, 0 is changed to 1 and 1 to 0. The inverter circuit is also referred to as a NOT gate and it has a single input and single output. The truth table for a NOT gate is shown in table 1.17.

(a) Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Graphic Symbol

![NOT Gate](image)

Figure 1.17 NOT Gate

The NOT logic can be written as, \[ Y = A' \] or \[ Y = A \]. The small circle in the output of the graphic symbol of an inverter designates a logic complement.
1.12.2 NOR Gate

A NOR gate is shown in Figure 1.18. Figure shows the NOR gate block diagram symbol with inputs A, B, C and the output ABC. This shows the NOR gate’s output will be a 1 only when all three inputs are 0’s. If any input represents a 1, then the output of a NOR gate will be a 0. The operation of the gate can be analyzed using the equivalent block diagram circuit as shown in the figure 1.18 which has an equivalent circuit showing an OR gate and an inverter. The inputs A, B, C are OR ed by the OR gate, giving A+B+C, which is complemented by the inverter, yielding (A+B+C)=ABC. The truth table for a NOR gate is shown in table.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Truth Table

![NOR Gate Symbol](image)

(b) Graphic Symbol

Figure. 1.18 NOR Gate

1.12.3 NAND Gate

A NAND gate is shown in figure 1.19. The inputs A, B and C and the output from the gate is written A+B+C. The output will be a 1 if A is a 0 or B is a 0 or C is a 0, and the output will be a 0 only if A, B and C are all 1’s. The operation of the gate can be analyzed using the equivalent block diagram circuit show in the figure1.19 which has an AND gate followed by an inverter. If the inputs are A, B, and C then the output of the AND gate will be A.B.C and the complement of this is (A.B.C) =A+B+C as shown in the figure 1.19. The truth table for a NAND gate is shown in table.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Truth Table

![NAND Gate Symbol](image)

(b) Graphical Symbol

Figure. 1.19 NAND Gate

1.12.4 XOR Gate

The exclusive – OR (XOR), denoted by ⊕, is a logical operation that performs the following Boolean operation:

\[ x \oplus y = x'y + x'y \]
It is equal to 1 if only x is equal to 1 or if only y is equal to 1, but not when both are equal to 1. The truth table for a XOR gate is shown in the Figure. 1.20.

(a) Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Graphical Symbol

Figure. 1.20 NOR Gate

1.13 LET US SUM UP

This completes our discussion on the introductory concepts of Computer Architecture. The information given on various topics such as data representation, Arithmetic Circuits etc. although is exhaustive yet can be supplemented with additional readings. In fact, a course in an area of computer science must be supplemented by further readings to keep your knowledge up to date, as the computer world is changing with leaps and bounds. In addition to further readings the student is advised to study several Indian Journals on computers to enhance his knowledge.

1.14 LESSON – END ACTIVITIES

1. Correct the following into octal and Hex Decimal
   (i) \( (25)_{10} \)
   (ii) \( (110101)_2 \)
   (iii) \( (125.73)_{10} \)

2. Discuss in detail about Binary adders.
3. Discuss about Half adder and Full adders.

11.15 POINTS FOR DISCUSSION

1. Try to differentiate the full adder and full subtractor.
3. Discuss about digital logic circuits.

1.16 CHECK YOUR PROGRESS: MODEL ANSWERS

Self-Check Exercise 1

1. 
   (a) True
   (b) True
   (c) True

Self – Check Exercise 2

1. 
   (a) 12.8125
(b) 170

2.
   (a) 10111
   (b) 110001.01
   (c) 1101111100

Self- Check Exercise 3
1.
   Numbers                      10100010    00000000    11001100
   1’s Complement               01011101    11111111    00110011
   2’s Complement               01011110    00000000    00110100

2.
   (a) 0 0101101 +45
   (b) 1 1101100 -25
   (c) Overflow
   (d) Underflow

Self – Check Exercise 4

1 Yes

1.17 REFERENCES

UNIT II
COMBINATIONAL LOGIC CIRCUITS & SEQUENTIAL CIRCUITS

2.0 Aims and Objectives

2.1 Introduction

2.2 Combinational Logic Circuits
   2.2.1 Boolean algebra
   2.2.2 Karnaugh Map
   2.2.3 Canonical Form
   2.2.4 Construction and Properties
   2.2.5 Implicants
   2.2.6 Don’t Care Combinations
   2.2.7 Sum of Products
   2.2.8 Product of Sums
   2.2.9 Simplification

2.3 Sequential Circuits
   2.3.1 Flip Flops
      2.3.1.1 RS Flip Flop
      2.3.1.2 D Flip Flop
      2.3.1.3 JK Flip Flop
      2.3.1.4 T Flip Flop

2.4 Multiplexers

2.5 Demultiplexers

2.6 Decoder

2.7 Encoder

2.8 Counters

2.9 Let us Sum Up

2.10 Lesson – End Activities

2.11 Points for Discussion

2.12 Model Answers to “Check your Progress”

2.13 References

2.0 AIMS AND OBJECTIVES

At the end of this unit you will be able to describe:

- What are flip-flops and gates?
- Combinational and sequential circuits and their applications thereof.
2.1 INTRODUCTION

In this unit you will be exposed to some of the basic components, which forms the most essential part of a computer. You will come across the terms like computer bus, binary adders, and logic gates, flip flop, combinational and sequential circuits. These circuits are the backbone for any computer system and knowing them will be quite essential.

2.2 BOOLEAN ALGEBRA

Basic concepts of Boolean algebra

Digital computers use the binary number system with bits 0 and 1. The human logic tends to be binary in a number of situations (i.e., true or false, yes or no).

Modern digital computers are designed and maintained, and their operation is analyzed, by using techniques and symbology from a field of mathematics called modern Algebra.

Algebraists have studied for over a hundred years that mathematical systems are called Boolean algebra. The name Boolean algebra honors a fascinating English Mathematician, George Boole, who in 1854 published a classic book. An investigation of the laws of thought, on which are founded the mathematical theories of Logic and Probabilities. Boole’s stated intention was to perform a mathematical analysis of logic.

Starting with his investigation of the laws of thought, Boole constructed a “Logical algebra”.

The three basic functions in Boolean algebra are

- Logical addition
- Logical Multiplication and
- Complementation

The addition of two logic variables are referred to as OR and addition symbol ‘+’ is used for this operation. Multiplication of two logic variables is referred to as AND, representing this operation with the symbol ‘.’ And complementation of a logic variable is NOT represented by the symbols ‘~’ or ‘’. In the equation \( Z = X \cdot Y + X' \), if X, Y and X’ are logic variables, the first term is the logical multiplication of x and y, second term is the complementation of X and the ‘+’ sign in between the two terms represent the logical addition.

1. Logical Addition (OR Gate)

If A and B are the input logic variable and Y is the output Variable, the truth table for a two input OR gate is given in Figure. 2.1.
The OR gate produces the inclusive – OR function, that is, the output is 1, if input A or input B or both inputs are 1, otherwise the output is a 0. In other words, in an n input OR gate, if any input is a 1, the output is a 1. For an OR gate the logical relationship of inputs and output can be written as,

$$Y = A + B + C + D + E + \ldots$$

### 2. Logical Multiplication (AND Gate)

The AND gate has one or more inputs and a single output. The output of an AND gate is equal to the multiplication of its inputs. The truth table for a two input AND gate is shown in Table. The output is 1 if both A and B are 1 and 0 otherwise. In general, in an n input AND gate, only if all the inputs are 1, the output will be 1.

The Input – Output relationships can be written as

$$Y = A \cdot B \cdot C \cdot D \cdot E \cdot \ldots$$

Or $$Y = ABCDEF \ldots$$

### 3. Logical Inversion: Compliment (NOT gate)

The complement function is nothing but inversion, 0 is changed to 1 and 1 to 0. The inverter circuit is also referred to as a NOT gate and it has a single input and single output. The truth table for a NOT gate is shown in Figure. 2.3.
Figure 2.3 NOT Gate
The NOT logic can be written as, $Y = A'$ or $Y = \overline{A}$. The small circle in the output of the graphic symbol of an inverter designates a logic complement.

**Demorgan’s Theorems**

(i) $(A+B)' = A'.B'$
(ii) $(A.B)' = A' + B'$

Demorgan’s first theorem states that the complement of a sum equals the product of the complements. Demorgan’s second theorem states that the complement of a product equals the sum of the complement.

Demorgan’s theorems can be extended to any number of variables. The two relationships can be written as,

(i) $(A+B+C+D+E+\ldots)' = A'.B'.C'.D'.E'\ldots$
(ii) $(ABCD\ldots)' = A'+B'+C'+D'\ldots$

These equalities can be easily proved as shown in truth tables.

**Table 2.1** Demorgan’s Theorem:1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A+B</th>
<th>(A+B)'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A'</th>
<th>B'</th>
<th>A'.B'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 2.2** Demorgan’s Theorem:2

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>AB</th>
<th>(AB)'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The last column in the left hand side tables is equivalent to the corresponding columns in the right hand side. The complement of any Boolean expression or part of any expression can be found as follows:

1. The + symbols in the given expression should be changed to dots (.) and dot symbols should be changed to +;
2. Each variable in the given expression should be complemented.
For example, to find the complements of the following expression, \( Y = A \cdot B \cdot C + A \cdot B' \cdot C \)

Changing the + symbols to dots and vice versa we get, \((A + B + C). (A + B' + C)\); Applying the second rule the expression changes to \((A' + B' + C') - (A' + B + C)\)

Hence \( Y' = (A' + B' + C'). (A' + B + C') \).

Demorgan’s theorem’s expresses a basic duality. For example.

\((A + B) + C = A + (B + C)\) is the dual of \((AB)C = A(BC)\)

**2.2.2 KARNAUGH MAP**

**Karnaugh map method for simplifying expressions:**

We have examined the derivation of a Boolean algebra expression for a given function by using a table of combinations to list desired function values. To derive a sum of products expression for the function, a set of product terms was listed, and those terms for which the function was to have a value 1 were selected and logically added to form the desired expressions.

The table of combinations provides a nice, natural way to list all values of Boolean function. There are several other ways to represent or list function values, and the use of certain kinds of maps, which we will examine, also permits minimization of the expression formed in a nice graphic way.

For example, if the function has only two variables, the number of squares will be \(2^n\), (i.e., \(2^2 = 4\)). Similarly, for a three input variable function the number of squares in the karnaugh map will be \(2^3 = 8\) and for four variable functions, \(2^4 = 16\) square will be present.

\[
\begin{array}{cccc}
B' & A' & A & \\
A'B' & AB' & AB & \\
B & \\
A' & \begin{array}{cccc}
B'C' & B'C & BC & BC' \\
A'B'C & A'B'C & A'BC & A'BC' \\
A'B'C' & AB'C' & AB'C & ABC & ABC' \\
A & \begin{array}{cccc}
C'D' & C'D & CD & CD' \\
A'B' & AB'C'D & A'B'CD & A'B'CD' \\
A'B & A'BC'D & A'BCD & A'BCD' \\
AB & ABC'D & ABC'D & ABCD & ABCD' \\
AB' & AB'C'D & AB'C'D & AB'CD & AB'CD' \\
\end{array}
\end{array}
\end{array}
\]
Table Two input variable function

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Product</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A’B’</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A’B</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>AB’</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>AB</td>
<td>0</td>
</tr>
</tbody>
</table>

The position of A and B can be interchanged

Table Three input variable function

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Product</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A’B’C’</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A’B’C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A’BC’</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A’BC</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AB’C’</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AB’C</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABC’</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ABC</td>
<td>1</td>
</tr>
</tbody>
</table>

Karnaugh map for 2 input variable function

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

As there are $2^n$ different squares available, each combination of input variable is represented by the square.

Table Four input variable function

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Product</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A’B’C’D’</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A’B’C’D</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A’B’CD’</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A’B’CD</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A’BC’D’</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A’BC’D</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A’BCD’</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A’BCD</td>
<td>1</td>
</tr>
</tbody>
</table>
Karnaugh map for 4 input variable functions

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>AB'C'D'</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>AB'C'D</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>AB'CD'</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>AB'CD</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ABC'D'</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>ABC'D</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABCD'</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ABCD</td>
</tr>
</tbody>
</table>

The input variables are split to represent all the combinations of the inputs. For example, in the two variable function, the two variables are treated as x and y. Coordinates and the map is drawn. In the variable function variable A is treated as x and B and C are treated as Y and so on.

In a karnaugh map the adjacent squares differ in bit combination in only one position in the fourth row entries for the same column one, in a four variable function are ABC'D', AB'C'D'.

For a two variable function F (A, B) there can be four minterms in all A B, AB, AB. For a three variables F (A, B, C) the possible minterms are ABC, ABC, ABC, ABC, ĀBC, ABC (Eight minterms all). In all general when there are n variables. $2^n$ minterms are possible. In a Boolean function, if all the terms are minterm, then it is said to be in canonical sp form.

Example:

$Y = ABC + ABC + ABC$ is a function in canonical form.

Product of sum canonical form:

Maxterm: If a sum term (or term) of a Boolean function of n variables, contain all the n variables, either complemented or uncomplemented, then it is called a maxterm or a standard sum.

For a two variable function F(A, B) there can be four maxterms in all the (A+B), (A+B), (A+B), (A+B). For a three variable function F (A, B, C) there can be eight maxterm and so on....
2.2.3 Canonical Form:

Canonical forms for Boolean functions are sum of products (SOP) form and basic forms. Canonical forms of Boolean expressions go into two level logic networks. That is the longest path through which a signal must pass through from input to output is through two gates. This characteristic makes the Canonical forms most desirable.

2.2.4 CONSTRUCTION AND PROPERTIES

Since there are a finite number of Boolean functions of n input variables, yet an infinite number of possible logic expressions you can construct with those n input values, clearly there are an infinite number of logic expressions that are equivalent (i.e., they produce the same result given Boolean function using a canonical, or standardized, form. For any given boolean function there exists a unique canonical form. This eliminates some confusion when dealing with Boolean functions.

Actually, there are several different canonical forms. We will discuss only two here and employ only the first of the two. The first is the so-called sum of minterms and the second is the product of maxterms. Using the duality principle, it is very easy to convert between these two.

2.2.5 PRIME IMPLICANTS

When choosing adjacent squares in a map, we must ensure that all the minterms of the function are covered when combining the squares. At the same time, it is necessary to minimize the number of terms in the expression and avoid any redundant terms whose minterms are already covered by other terms. Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining the squares in the map may be made more systematic if we understand the meaning of the referred to as prime implicants and essential prime implicants. A prime implicant is a product term obtained by combining the maximum possible number of the adjacent squares in the map. If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be essential.

The prime implicant of a function can be obtained from the map by combining all possible maximum number of squares. This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1’s. Two adjacent 1’s form a prime implicant, provided that they are not within a group of four adjacent squares. Four adjacent 1’s form a prime implicant are found by looking at each squares, and so on. The essential prime implicants are found by looking at each square marked with a 1 and checking the number of prime implicant that covers the minterm.

Consider the following four-variable Boolean functions:

\[ F (A, B, C, D) = \Sigma (0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15) \]
The minterm of the function are marked with 1’s in the map of Fig. 3-11. Part (a) of the figure shows two essential prime implicants. One term is essential because there is a only one way to include minterm \( m_{0} \) with in four adjacent squares. These four squares define the term B’D’ similarly, there is only one way that minterm \( m_{5} \) can be combined with four adjacent squares.

![Diagram showing minterms and essential prime implicants](image)

Implicants CD, B’C AD, and AB’

**2.2.6. SIMPLIFICATION USING-PRIME IMPLICANTS**

And this gives the second term BD. The two essential prime implicant cover eight minterms. The remaining three minterms, \( m_{3}, m_{9}, \) and \( m_{11} \) must be considered next.

Figure 3-11(b) shows all possible ways the three minterms can be covered with prime implicants. Minterms \( m_{3} \) can be covered with either prime implicant CD or B’C. Minterm \( m_{9} \) can be covered with either AD and AB’. Minterm \( m_{11} \) any one of the four prime implicants. The simplified expression is obtained from the logical sum of the two essential are four possible ways that the function can be expressed with four product terms of the literals each:

\[
F = BD + B'D' + CD + AD \\
F = BD + B'D' + CD + AB' \\
F = BD + B'D' + B'C + AD \\
F = BD + B'D' + B'C + AB'
\]

The previous example has demonstrated that the identification of the prime implicants in the map helps in determining the alternatives that are available for obtaining a simplified expression.

The procedure for finding the simplified expression from the map requires that we first determine all the essential prime implicants. The simplified expression is obtained from the logical sum of all the essential prime implicants plus other prime implicants that may be needed.
to cover any remaining minterms not covered by the essential prime implicants. Occasionally, there may be more than one way of combining squares and each combining squares and each combination may produce an equally simplified expression.

2.2.6 DON’T CARE CONDITIONS

The logical sum of the minterms associated with a Boolean function specifies the condition under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. This assumes that all the combinations of the values for the variables of the function are valid. In practice, there are some applications where the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered as unspecified. Functions that have unspecified outputs for some input combinations are incompletely specified functions. In most applications, we simply don’t care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function don’t-care condition. These don’t-care condition can be used on a map to provide further simplification of the Boolean expression.

It should be realized that a don’t-care minterm is a combination of variables whose logical value is not specified. It cannot be marked with a 1 in the map because it would require that the function always be a 1 for such combination. Likewise, putting A 0 on the square requires the function to be a 0. To distinguish the don’t-care condition from 1’s and 0’s, an X is used. Thus, an X inside a square in the map indicates that we don’t care whether the value of 0 or 1 is assigned to F for the particular minterm.

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

EXAMPLE 3-9

Simply the Boolean function

\[ F(w,x,y,z) = (1,3,7,11,15) \]

Which has the don’t-care conditions

\[ d (w,x,y,z) = (0,2,5) \]

The minterms of the variable combination that make the function equal to 1. The minterms of d are the don’t-care minterms that may be assigned either 0 or 1. The map simplification is shown in Figure.2.4. The minterms of F are marked b 1’s, those of d are marked by X’s, and the remaining squares are filed with 0’s. To get simplified expression in sum of
products, we must include all five 1’s in the map, but we may or may not include any of the X’s depending on the way the function is simplified. The term $yz$ covers the four minterms in the third column. The remaining minterm $m_1$ can be combined with minterm $m_3$ to give the three-literal term $w’x’z$. However, by including one or two adjacent X’s we can combine four adjacent squares to give a two-literal term. In part(a) of the diagram, don’t-care minterms 0 and 2 are included with the 1’s, which results in the simplified function

$$F = yz + w’x$$

Figure 2.4 Karnaugh Map
In part (b), don’t-care minterm 5 is included with the 1’s and the simplified function now is

\[ F = yz + w'z \]

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

The previous example has shown that the don’t-care minterms in the map are initially marked with X’s and are considered as being either 0 or 1. The choice between 0 and 1 is made depending on the way the incompletely specified function is simplified. Once the choice is made, the simplified function obtained will consist of a sum of minterms that includes those minterms that were initially unspecified and have been chosen to be included with the 1’s. Consider the two simplified expressions obtained in example

\[
\begin{align*}
F(w,x,y,z) &= yz + w'x' = (0,1,2,3,7,11,15) \\
F(w,x,y,z) &= yz + w'z = (1,3,5,7,11,15)
\end{align*}
\]

Both expressions include minterms 1, 3, 7, 11 and 15 that make the function \( F \) equal to 1. The don’t-care minterms 0, 2 and 5 are treated differently in each expression. The first expression includes minterms 0 and 2 with 1’s and leaves minterms 5 with the 0’s. The second expression represents two functions that are algebraically unequal. Both cover the specified minterms of the function, but each covers different don’t-care minterms. As far as the incompletely specified function is concerned, either expression is acceptable because the only difference is in the value of \( F \) for the don’t-care minterms.
It is also possible to obtain a simplified product of sums expression for the function of Figure.3-17. In this case, the only way to combine the 0’s is to include don’t-care minterms 0 and 2 with the 0’s to give a simplified complemented function:

$$F' = z' + wy'$$

Taking the complement of $F_1$ gives the simplified expression in product of sums:

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

For this case, we include minterms 0 and 2 with 0’s and minterms 5 with the 1’s.

**2.2.7 SUM OF PRODUCTS**

Minterm: If a product term (AND) of a function of ‘n’ variables, contain all the variables, in complemented form or uncomplemented form, then it is called a “minterm” or “standard product”.

**Sum of product method:**

The four possible ways to AND two input signals that are in complemented and uncomplemented from. These outputs are called fundamental products. The following table lists each fundamental product next to the input conditions producing a high input for instance, $AB$ is high when $A$ and $B$ are low. $AB$ is high when $A$ is high and so on.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>FUNDAMENTAL PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$A'B'$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$A'B$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$AB'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$AB$</td>
</tr>
</tbody>
</table>

The idea of fundamental products applies to three or more input variables. For example, assume three input variables: $A$, $B$, $C$ and their complements. There are eight ways to AND three input variables and their complements resulting in fundamental products of $ABC$, $ABC$, $ABC$, $ABC$, $ABC$, $ABC$, $ABC$, and $ABC$.

The following table summaries the fundamental products by listing each one next to the input condition that results in a high output. For instance, when, $A=1$, $B=0$ and $C=0$, the fundamental products results in an output of $Y= ABC = 1.0.0=1$

**Fundamental products of three inputs**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>FUNDAMENTAL PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$A'B'C'$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$A'B'C$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A'BC'$</td>
</tr>
</tbody>
</table>
Logic circuit:

After you have a sum of products equation, you can derive the corresponding logic circuit by drawing an AND-OR network, or what amounts to the same thing, a NAND-NAND network.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A’BC</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AB’C’</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AB’C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABC’</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ABC</td>
</tr>
</tbody>
</table>

2.2.8 PRODUCTS OF SUMS

In a Boolean function, if all the terms are maxterm, then it is said to be in canonical product of the sum form.

\[ Y = (A+B+C)(A+B+C)(A+B+C) \]

is a canonical product of sums function.

Products of sums method:

With the sum of products method and design starts with a truth table that summarizes the desired input-output conditions. The next step is to convert the truth table into an equivalent sum of products equation. The final step is to draw the AND-OR network or its NAND-NAND equivalent.

The product of sums method is similar. Given a truth table you identify the fundamental sums needed for a logic design. Then by ANDing these sums equation corresponding to the truth table.

With the sum of products method, the fundamental product produce an output 1 for the corresponding input condition. But with the product of sum method, the fundamental sum products an output 0 per the corresponding input condition.
Converting the truth table to an equation

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Here consider the following truth table into equation you want to get the product of sum equation. Here A=0, B=0, C=0 then sum for three inputs is A+B+C

There are four squares marked with 1’s. One for each minterm that produces 1 for the function. These squares belong to minterms 3, 4, 6 and 7. Two adjacent squares are combined in the third column. This column belongs to both B and C and produces the term BC. The remaining two squares with 1’s in the two corners of the second row are adjacent and belong to row. A and the two columns of C’, so they produce the term AC’. The simplified algebraic expression for the function is the OR of the two terms.

\[ F = BC + AC' \]

The second example simplifies the following Boolean function.

\[ F (A, B, C) = (0, 2, 4, 5, 6) \]

The five minterms are marked with 1’s in corresponding squares of the three variable maps shown in the above figure. The four squares in the first and fourth column are adjacent and represent the term C’. The remaining square marked with 1 belongs to minterm 5 and can be combined with the square of minterm 4 to produce the term A. The simplified function is \[ F = C' + AB' \]
The third example needs a four variable map.

Map for \( F(A, B, C, D) = (0, 1, 6, 8, 9, 10) \)

The area in the map covered by these four variable function consists of the squares marked with 1’s in the four corners that, when taken as a group, gives the term \( B'D' \). This is possible because these four squares are adjacent when the map is considered with top and bottom or left and right edges touching. The two 1’s on the left of the top row are combined with the two 1’s on the left of the bottom row to give the term \( B'C' \). The remaining 1 in the square of minterm 6 is combined with minterm 2 to give the term \( A'CD' \). The simplified function is

\[ F = B'D' + B'C' + A'CD' \]

2.2.9 SIMPLIFICATION

The product denotes the AND terms and the sum denotes the OR of these terms. It is convenient to obtain the algebraic expression for the function in a product–of–sums form. With a minor modification, a product of sums form can be obtained from a map.

The procedure for obtaining a product of sums expression follows from the basic properties of Boolean algebra. The 1’s in the map represent the minterms that produce 1 for the function. The squares not marked by 1 represent the minterms that produce 0 for the function. If we mark the empty squares, we obtain the complement of the function’. Taking the complement of \( F' \) produces an expression for \( F \) in product of sums form. The best way to show this is by example.

We wish to simplify the following Boolean function in both sum of products form and product of sums form.
The 1’s marked in the above map represent the minterms that produce a 1 for the function. The square marked with 0’s represent minterms not included in $F$ and therefore denote the complement of $F$. Combining the square with 1’s gives the simplified function in sum of products form:

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

If the square marked with 0’s are combined as shown in the diagram, we obtain the simplified complemented function.

$$F' = AB + CD + BD'$$

Taking the complement of $F'$, we obtain the simplified function in product of sums form:

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

The logic diagrams of the two simplified expressions are shown in the figure 2.6.

Logic diagrams with AND, OR gates.

Figure 2.6 Logic diagrams with AND, OR gates.

The sum of products expression is implemented in figure 2.6 (a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The product of sums expression is implemented in figure 2.6 (b) with a group of OR gates, one for each OR term. The outputs of the OR gates are connected to the inputs of a single AND gate. In each case it is assumed that the input variables are directly available in their
complement, so inverters are not included. The general form by which any Boolean function is implemented when expressed in one of the standard forms. AND gates are connected to a single OR gate when in sum of products form. OR gates are connected to a single AND gate when in product of sums form.

Self Check Exercise: 1

1. Draw a Karnaugh Map for five variables.

2. Map the function having four variables in Karnaugh’s map. The function is \( F(A,B,C,D) = \Sigma (2,6,10,14) \).

3. Find the optimal logic expression for the above function. Draw the resultant logic diagram.

4. Can a full adder be constructed by using two half adders?

2.3 SEQUENTIAL CIRCUITS

2.3.1 Flip Flop

A flip flop is a bi-stable device, that is, it can remain in one of the two stable states which are designated as “0” and “1” states. It is the fundamental logic circuit used for storing information in digital systems. Different types of shift registers and counters are designed only using flip flops, which can be built using NOR gates or NAND gates. A flip flop has two outputs, one of which is the complement of the other. They are called Normal and complement outputs.

2.3.2 RS Flip Flop

The RS Flip flop can be implemented in many ways. One such implementation is shown in Figure 2.7. There are two inputs to the RS Flip Flop. These lines are used to control the output of the flip-flop. The working of the flip-flop is as follows:
Case: 1

When \( S=0 \) and \( R=0 \). Now both the NAND gates A and B output logic 1. Hence the outputs of C and D depend only on the feedback inputs (secondary inputs). The primary inputs, that is outputs of gates A and B are don’t cares now. In other words the entire circuit behaves as a latch. Thus the circuit will “hold on” to its previous output. This state is called “HOLD” state.

![Figure 2.7 RS Flip Flop](image)

Figure 2.7 shows an RS Flip Flop constructed with four NAND gates A, B, C and D. It has two inputs S and R and two outputs Q and Q’. (The state of any flip-flop is known by the state of a output only).

Case: 2

When \( S=0 \) and \( R=1 \). Now gate A will output 1 and B will output as 0. The output of D that is \( Q’ \) will be 1, making both inputs to gate c as 0. Hence Q will be in a 1 state. This state of the flip flop is known as “RESET” state.

Case: 3

When \( s=1 \) and \( R=1 \). This input condition is prohibited. This is because when both \( S \) and \( R \) are equal to 1, gates A and B will output 0’, which will force both Q and Q’ to 1. This is against the principle of operation of a flip flop. Moreover, if the inputs are now changed, the next state of the flip flop is unpredictable. The next state actually depends on which gate is faster to change its present state. This prohibited state is also called “RACE” condition.

The above stated input output relations are represented below on a table. This table is called truth table.

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>OUTPUT</th>
<th>MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>R</td>
<td>Q(N+1)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Q(N)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>*</td>
</tr>
</tbody>
</table>
Case 4:
When S=1 and R=0. This is just the reverse of case 2. On similar arguments we can see that now Q=1 and Q’=0. This state of the flip-flop is known as “SET” state.

2.3.3 D flip flop

In the RS Flip flop the condition R=1 and S=1 is forbidden. This state can be avoided by connecting an inverter between S and R inputs. The flip flop with this modified connection is called a D flip flop

![Logic Diagram]

(a) Logic Diagram

<table>
<thead>
<tr>
<th>CLK</th>
<th>D</th>
<th>Q</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Characteristics Table

Figure 2.8: D Flip Flop

When the clock is 0 and the D input does not affect the output. So when the clock is 0, D is treated as don’t care. The value of D is prevented from reading the output until a clock pulse occurs. When the clock is high, both the AND gates are enabled and the value of D appears at Q. When the clock goes low and last value is retained by Q. This Flip-flop is also called as a delay Flip-flop or data flip-flop.

2.3.4 JK Flip-flop

A flip-flop is a refinement of the SR flip-flop in that the in determinate condition of the SR type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the flip-flop, respectively. When inputs J and K are both equal to 1, a clock transition switches the outputs of the flip-flop to their complement state.

The graphic symbol and characteristic table of the JK flip-flop are shown in Figure. 2.9. The J input is equivalent to the S (set) input of the SR flip-flop, and the K input is equivalent to the R (clear) input. Instead of the indeterminate condition, the Jk flip-flop has a complement condition Q (t+1)= Q’ (t) when both J and K are equal to 1.
2.3.5 T Flip Flop

Another type of flip flop is the toggle flip flop. This flip flop is obtained from a JK flip flop when inputs J and K are connected to provide a single input designated by t. The T flip flop therefore has only two conditions. When T=0 (J=K=0) a clock transition does not change the state of the flip flop. When T=1(J=K=1) a clock transition complements the state of the flip flop. These conditions can be expressed by a characteristics equation.

\[ Q(t+1)=Q(t) \oplus T \]

<table>
<thead>
<tr>
<th>T</th>
<th>Q(t+1)</th>
<th>Q'(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q(t)</td>
<td>No Change</td>
</tr>
<tr>
<td>1</td>
<td>Q'(t)</td>
<td>Complement</td>
</tr>
</tbody>
</table>

Figure 2.9  JK Flip flop

Figure 2.10  T flip flop
A multiplexer is a combinational logic circuit, which has many inputs and only one output. (Multiplexer means “Many to one”). By applying a suitable control input, any data input can be sent to the output. Figure 2.11 shows a four data input multiplexer. D₀, D₁, D₂ and D₃ are data inputs. A and B are control inputs. Y is the output of the multiplexer. (Note that the multiplexer can be obtained by modifying a decoder circuit).

When control input AB=00, Gate=0 is enabled and hence Y=D₀. Similarly if AB=10, Gate 2 is enabled and hence Y=D₂ and so on.

A demultiplexer performs a function opposite to that of a multiplexer. It has one data input and several output lines. Based on the value of the control input, one of the output lines will become active and will output the data input across it. Figure 2.12 shows a 1 to 4 demultiplexer. D is the data input Y₃, Y₂, Y₁ and Y₀. ie. Y₀=D. Similarly when the control input AB=10, Y₂=D.
2.6 DECODER

A decoder has several output lines and control input lines. Based on the value of the control input (or select input), one of the output lines will become active. If there are four control input lines, then the decoder can have up to a maximum of sixteen (2 power 4) output lines. The decoder is generally used to select one among the many devices. It is widely used as address decoder in a computer system. (The outputs of a decoder can become the select inputs to the memory locations. In such case, the control inputs become the address of the memory location). Figure 2.13 shows a simple decoder with two control inputs and four outputs.
When the control input $AB=00$, the AND gate $G_0$ is enabled and its output $Y_0$ is high. Suppose $AB=10$, then $G_2$ is enabled and hence $Y_2$ will be high.

2.7 ENCODER

An encoder is a digital circuit that generates the binary code corresponding to the input number. An octal-to-Binary encoder takes an octal input (in some symbolic form) and generates its binary equivalent as output. Similarly, a Decimal-to-Binary encoder takes a decimal input and generates an equivalent binary output. Figure 2.14 shows an Octal-to-Binary encoder using OR gates. (A Decimal-to-Binary encoder has ten input lines and uses four OR gates at the output).

2.8 COUNTERS

A counter is a sequential circuits that counts the number of incoming clock pulses. It consists of an array of flip-flops. In the following sections, negative edge triggered JK flip-flops are used for discussions on the working of the various types of counters or basically there are two types of counters. “Parallel counters (or) Synchronous counters” and “Ripple counters (or) Asynchronous counters”.

![Figure 2.13 Decoder](image)

![Figure 2.14 Encoder](image)
Asynchronous Binary Up Counter

Let us consider a three bit counter for simplicity. It must have a count sequence 000 001…111, as shown in the table below. Q₀, Q₁, Q₂ are outputs of the flip-flops are cleared with a PC RESET input such that Q₂Q₁Q₀=000. Thereafter with every clock, the output increments as shown.

<table>
<thead>
<tr>
<th>CK</th>
<th>Q₂</th>
<th>Q₁</th>
<th>Q₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

After every eight clocks the count sequence repeats itself. From the count sequences table given above it can be seen that Q₀ toggles with every clock. Hence the CK pulse is directly
applied to the first flip-flop. But Q (i) toggles whenever Q (i-1) makes a negative transition that is when Q(i-1) changes from 1 to 0. Hence the clocks for subsequent flip-flops are obtained from the Q (i) output of the previous flip-flop. Note that in an asynchronous counter the J and K inputs are always kept high so that the flip-flop toggles when a clock arrives.

**Asynchronous Binary down Counter**

A binary down counter counts with every clock and hence it is initially SET with DCSET input.

<table>
<thead>
<tr>
<th>CK</th>
<th>Q2</th>
<th>Q1</th>
<th>Q0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

And so on.

The counter state table is given below. It can be seen from the count sequence table that Q0 toggles with every clock. Hence the first flip-flop is clocked directly. For the other flip-flops Q (i) toggles only when Q (i-1) makes a positive transition. That is when Q (i-1) makes a negative transition. Hence the clocks are derived from the complementary outputs for these flip-flops.
For convenience, a three bit natural binary counter is treated in the above discussions. The same logic can be extended to 4-bit (or) 5-bit counter or an \( n \)-bit counter.

**Synchronous Binary Counter**

In a synchronous counter, the flip-flops are all clocked simultaneously. Hence the \( J \) and \( K \) inputs are made high only when necessary. (Recall that in asynchronous counters, \( J \) and \( K \) inputs are always high, that is the flip-flops are in toggle mode and when a clock arrives the flip-flop toggles).

<table>
<thead>
<tr>
<th>CK</th>
<th>Q2</th>
<th>Q1</th>
<th>Q0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

And so on.

A careful study of the count sequence table will reveal that a flip-flop toggles only when the outputs of the previous flip-flops are high. ie. \( Q(i) \) toggles only when \( Q(i-1) \) AND \( Q(i-2) \) AND \( Q(i-3) \) … AND \( Q(0)=1 \).

This fact is made use of in realizing a synchronous counter.

**Self Check Exercise : 2**

1. What are sequential Circuits? How are they different from combinational circuit?
   - ……………………………………………………………………………………………………………………………………………………

2. Can Ripple counter be constructed from a shift register?
   - ……………………………………………………………………………………………………………………………………………………
2.9 LET US SUM UP

This unit provides you the information regarding the basis of a computer system. The key elements for the design is the combinational and sequential circuits. With this developing scenario in the forefront and the expectations of Ultra Large Scale Integration(ULSI) in view, it is not far off when design of logical circuit will be confined to single microchip components.

2.10 LESSON – END ACTIVITIES

(1) Construct a sequential circuits for J.K. and D flip flops.
(2) Discuss about product of SUM and SUM of the product.

2.11 POINTS FOR DISCUSSIONS

(1) Study about the combinational circuit and sequential circuits.
(2) Analysis the usage of sequential circuits on the basis of digital computer.

2.12 CHECK YOUR PROGRESS: MODEL ANSWERS

Self-Check Exercise: 1

1.

\[
\begin{array}{c|ccccc}
\text{AB} & 00 & 01 & 11 & 10 \\
\hline
\text{CDE} & & & & \\
\end{array}
\]

2. The logic Expression is: \( F=\overline{CD} \)

3.

4. Yes

Self Check Exercise: 2
1. The logic circuits present output depends on the past inputs. These circuits store and remember information. The sequential circuits unlike combinational circuits are time dependent. Normally the current output of a sequential circuit depends on the state of the circuit and on the current input to the circuit. It is a connection of flip flops and gates.

2. Yes.

2.13 REFERENCES