Digital Techniques

SEMESTER I

MASTER OF COMPUTER APPLICATION /

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

BLOCK - 1

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Expert

Prof. Anjana Kakati Mahanta, Gauhati University
Prof. (Retd.) Prannari Talukdar, Gauhati University
Prof. Sikhar Kumar Sarma, Cotton University
Dr. Jyotiprokash Goswami, Assam Engineering College

Course Coordinator

Dr. Sanjib Kr. Kalita, Associate Professor, Dept. of Computer Science, KKHSOU
Dr. Tapashi Kashyap Das, Assistant Professor, Dept. of Computer Science, KKHSOU
Sruti Sruba Bharali, Assistant Professor, Dept. of Computer Science, KKHSOU

SLM Preparation Team

UNITS	CONTRIBUTORS
1, 2, 3, 4	Dr. Sangeeta Kakoty, KKHSOU
5, 6	Dr. Jyotiprokash Goswami, Assam Engineering College
7, 8	Prof. Kandarpa Kr. Sarma, Gauhati University

Editorial Team

Content : Dr. Sanjib Kr. Kalita, KKHSOU
Language : Prof.(Retd) Robin Goswami, Cotton College

Structure, Format & Graphics : Dr. Sanjib Kr. Kalita, KKHSOU

May 2018
ISBN NO: 978-93-87940-21-5

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State Open University is made available under a Creative Commons Attribution-Non Commercial-Share Alike 4.0 License (international): http://creativecommons.org/licenses/by-nc-sa/4.0/

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University

Headquarters : Patgaon, Rani Gate, Guwahati - 781017
Housefed Complex, Dispur, Guwahati-781006; Web: www.kkhsou.in

The University acknowledges with thanks the financial support provided by the **Distance Education Bureau**, **UGC** for the preparation of this study material.

COURSE INTRODUCTION

This course is designed for the learners who are beginners in the field of computer architecture. This is the age of digital technology. In this age it is very important to have some idea about digital techniques used in computer system. After going through this course learners will be able to understand the different number systems like binary, octal, decimal and hexadecimal. Different types of gates, basic memory circuits, combinational and sequential circuits are discussed in this course.

There are two blocks in this course.

Block 1 deals with the basic concept of number system, data representation, logic gates and logic family. After going through this block learners will be comfortable to learn block 2.

Block 2 concentrates on basic principles of combinational and sequential circuits. At the end of this block learners will have a clear concept of register, counter and memory organization.

Each unit of these blocks includes some along-side boxes to help you know some of the difficult, unseen terms. Some “EXERCISES” have been included to help you apply your own thoughts. You may find some boxes marked with: “LET US KNOW”. These boxes will provide you with some interesting and relevant additional information. Again, you will get “CHECK YOUR PROGRESS” questions. These have been designed to self-check your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with “ANSWERS TO CHECK YOUR PROGRESS” given at the end of each unit.

BLOCK INTRODUCTION

This course contains fifteen units in two blocks. **Block I** contains 8 units. These units discuss the basic concept of digital logic. Unit 1 deals with the basic concept of different number systems and their conversion. Unit 2 introduces binary arithmetic like r's and (r-1)'s complement etc.The basic of data representation like fixed point and floating point are discussed in Unit 3. Introduction to code conversion techniques like Gray code, BCD, Excess - 3 are discussed in unit 4. Unit 5 discusses the introduction of boolean algebra and their properties, De - Morgan's theorem etc. The concept of logic gates like AND, OR, NAND,NOR, XOR etc. are discussed in unit 6. Unit 7 discusses the details of floating point number representation. Logic family and their properties are discussed in unit 8.

MASTER OF COMPUTER APPLICATION / MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

Digital Techniques

DETAILED SYLLABUS

BLOCK-1

Semester I

Page No.

Unit 1: Introduction to Number Systems [5 marks] **7-18**

Decimal, Binary, Hexadecimal and Octal number system, Number system conversions

Unit 2: Binary Arithmetic [6 marks] **19-30**

Complement: r's and (r-1)'s complement, Binary addition, Binary subtraction, Binary multiplication, Binary division.

Unit 3: Introduction to Data Representation [6 marks] **31-38**

Fixed Point representation and Floating point representation

Unit 4: Code Conversion [5 marks] **39-46**

Gray code, BCD, ASCII, EBCDIC, Conversion from Binary to Gray and Vice-versa

Unit 5: Boolean Algebra [5 marks] **47-55**

Introduction, Properties, De-Morgan's Theorem, Duality Principle

Unit 6: Logic Gates [5 marks] **56-75**

Logic Gates: AND, OR, NOT, NAND, NOR, XOR; Conversion of the logic gates

Unit 7: Floating Point Number Representation [5 marks] **76-94**

Floating point number, Normalization of floating point, overflow and underflow, detection of overflow, IEEE floating point standard

Unit 8: Logic Families [6 marks] **95-112**

Introduction, Registror Transistor Logic(RTL), Integrated Injection logic(IIL), Diode- Transistor Logic(DTL), Emitter-Coupled Logic(ECL), Transistor- Transistor Logic(TTL), TTL-NAND, Tri State Logic, MOS devices, Logic gates with MOSFET's

*****

UNIT 1 : INTRODUCTION TO NUMBER SYSTEMS

UNIT STRUCTURE

1.1 Learning Objectives

1.2 Introduction

1.3 Number System

- 1.3.1 Decimal Number System
- 1.3.2 Binary Number System
- 1.3.3 Octal Number System
- 1.3.4 Hexadecimal Number System

1.4 Number System Conversion

- 1.4.1 Binary to Decimal Conversion
- 1.4.2 Decimal to Binary Conversion
- 1.4.3 Octal to Decimal Conversion
- 1.4.4 Decimal to Octal Conversion
- 1.4.5 Hexadecimal to decimal Conversion
- 1.4.6 Decimal to Hexadecimal Conversion

1.5 Let Us Sum Up

1.6 Further Reading

1.7 Answers to Check Your Progress

1.8 Model Questions

1.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define and describe number systems like decimal, binary, octal, and hexadecimal.
- convert from one number system to another number system.

1.2 INTRODUCTION

Computer system uses different number systems to represent data. In this unit, we will be able to understand how numbers are represented in different number systems.

In this unit, we will learn about decimal, binary, octal and hexadecimal number systems. In addition to this, we will also discuss the ways for converting from one number system to one another number system. Once, we have the knowledge of number systems, then in the next units, we will discuss about binary arithmetic and data representation methods.

1.3 NUMBER SYSTEM

Number system is a fundamental concept used in micro computer system. They are of different types and can be represented by digit symbols. The knowledge of binary, octal and hexadecimal number system is essential to understand the operation of a computer. This unit deals with all these number systems. In this unit we will discuss about all the representation of number system and their conversion from one number system to another number system.

We are familier with the decimal number system which is used in our day-to-day work. Ten digits are used in decimal number system. To represent these decimal digits, ten separate symbols 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 are used. But a digital computer stores, understands and manipulates information composed of only zeros and ones. So, each decimal digits, letters, symbols etc. written by the programmer (an user) are converted to binary codes in the form of 0's and 1's within the computer. The number system is divided into different categories according to the base (or radix) of the system as binary, octal and hexadecimal. If a number system of base r is a system, then the system have r distinct symbols for r digits. The knowledge of the number system is essential to understand the operation of a computer.

1.3.1 Decimal Number System

Decimal number system have ten digits represented by 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. So, the base or radix of such a system is 10.

In this system, the successive position to the left of the decimal point represent units, tens, hundreds, thousands etc. For example, if we consider a decimal number 1257, then the digit representations are :

1	2	5	7
↓	↓	↓	↓
thousands positions	hundreds position	tens position	units position

The weight of each digit of a number depends on its relative position within the number.

Example 1.1 : Find the weight of each digit in 6472.

The weight of each digit of the decimal no. 6472

$6472 = 6000 + 400 + 70 + 2 = 6 \times 10^3 + 4 \times 10^2 + 7 \times 10^1 + 2 \times 10^0$

The weight of digits from right hand side are :

Weight of 1st digit = 2×10^0

Weight of 2nd digit = 7×10^1

Weight of 3rd digit = 4×10^2

Weight of 4th digit = 6×10^3

The above expressions can be written in general forms as the weight of n^{th} digit of the number from the right hand side :

$= n^{th}$ digit $\times 10^{n-1}$

$= n^{th}$ digit $\times$ (base)$^{n-1}$

The number system in which the weight of each digit depends on its relative position within the number is called positional number system. The above form of general expression is true only for positional number system.



Decimal Number System uses 10 digits from 0 to 9.

1.3.2 Binary Number System

Only two digits 0 and 1 are used to represent the binary number system. So the base or radix is two (2). The digits 0 and 1 are called bits (Binary Digits). In this number system the value of the digit will be two times greater than its predecessor. Thus, the value of the places are :

$\leftarrow 32 \leftarrow 16 \leftarrow 8 \leftarrow 4 \leftarrow 2 \leftarrow 1$

The weight of each binary bit depends on its relative position within the number. It is explained by the following example--

Example 1.2 : Find the weight of binary number 10110.

The weight of bits of the binary number 10110 is :

$= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$= 16 + 0 + 4 + 2 + 0 = 22$ (decimal number)

The weight of each bit of a binary no. depends on its relative pointer within the no. and explained from right hand side as :

Weight of 1st bit = 1st bit X 2^0

Weight of 2nd bit = 2nd bit X 2^1

.....

.....

and so on.

The weight of the n^{th} bit of the number from right hand side

$= n^{th}$ bit $\times 2^{n-1}$

$= n^{th}$ bit $\times$ (Base)$^{n-1}$



A ***Binary Number System*** uses only digit 0 and 1

It is seen that this rule for a binary number is same as that for a decimal number system. The above rule holds good for any other positioned number system. The weight of a digit in any positioned number system depends on its relative positon within the number and the base of the number system.

Table 1.1 shows the binary equivalent numbers for decimal digits.

Table 1.1 : Binary equivalent of decimal numbers

Decimal Number	Equivalent Binary Number
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Binary Fractions : A binary fractions can be represented by a series of 1s and 0s to the right of a binary point. The weight of digit positions to the right of the binary point are given by 2^{-1}, 2^{-2}, 2^{-3} and so on.

Example 1.3 : Show the representation of binary fraction 0.1101.

Solution : The binary representation of 0.1101 is :

$$0.1101 = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$$
$$= 1 \times 0.5 + 1 \times 0.25 + 0 \times 0.125 + 1 \times 0.0625$$
$$= 0.8125$$

So, $(0.1101)_2 = (0.8125)_{10}$

1.3.3 Octal Number System

A commonly used positional number system is the Octal Number System. This system has eight (8) digit representation as 0,1,2,3,4,5,6 and 7. The base or radix of this system is 8. The values increase from left to right as 1,8,64,512, 4096 etc. The decimal value 8 is represented in octal as 10,9 as 11,10 as 12 and so on. As $8=2^3$, an octal number is represented by a group of three binary bits. For example, 3 is represented as 011, 4 as 100 etc.

Table 1.2 The octal number and their binary representations.

Decimal Number	Octal Number	Binary Coded Octal No.
0	0	000
1	1	001
.	.	.
.	.	.
7	7	111
8	10	100 000
15	17	001 111

1.3.4 Hexadecimal Number System

The hexadecimal number system is now extensively used in computer industry. Its base (or radix) is 16, and the digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. The hexadecimal numbers are used to represent binary numbers incase of conversion and compactness.



Octal Number System
uses 8 digits from 0 to 7.

As $16 = 2^4$, hexadecimal number is represented by a group of four binary bits. For example, 5 is represented by 0101. Table 1.3 shows the binary equivalent of a decimal number and its hexadecimal representation.

Table 1.3 : Hexadecimal number and their Binary representation

Decimal No.	Hexadecimal No.	Binary coded Hex. No
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

1.4 NUMBER SYSTEM CONVERSION

As the computer uses different number systems, there is a process of converting generally used decimal number systems to other number systems and vice-versa.

1.4.1 Binary to Decimal Conversion

To convert a binary number to its decimal equivalent we use the following expression.The weight of the n^{th} bit of the number from right hand side can be represented as-

$= n^{th}$ bit $\times 2^{n-1}$

First we mark the bit position and then we give the weight of each bit of the number depending on its position. The sum of the weight of all bits gives us the equivalent number.

Example 1.4 : Convert binary $(100101)_2$ to its decimal equivalent.

Solution : $(100101)_2 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 32 + 0 + 0 + 4 + 0 + 1$

$= 37$

So, $(100101)_2 = (37)_{10}$

Mixed number contain both integer and fractional parts and can convert to its decimal equivalent is as follows :

Example 1.5 : Convert $(11011.101)_2$ to its equivalent decimal number.

Solutaion :

$(11011.101)_2 = (1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) + (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3})$

$= (16 + 8 + 0 + 1) + (0.5 + 0 + 0.125) = 27.625$

So, $(11011.101)_2 = (27.625)_{10}$

1.4.2 Decimal to Binary Conversion

There are different methods used to convert decimal number to binary numbers. The most common method is to repeatedly divide the decimal number by 2, then the remainder 0's and 1's obtained after division, is read in reverse order to obtain the binary equivalent of the decimal number.

Example 1.6 : Convert $(75)_{10}$ to its binary equivalent.

Solution :

		Remainder	
2 |75			
2 |37	LSB	1	
2 |18		1	á
2 |9		0	Read in
2 |4		1	reverse order
2 |2		0	
2 |1	MSB	0	
0		1	

So, $(75)_{10} = (1001011)_2$

Example 1.7 : Convert decimal fraction $(25.625)_{10}$ to its equivalent binary number.

Solution : 2|25 Remainder MSB 0.625

2 |12	1	↑	× 2
2 |6	0	|	1.250
2 |3	0	|	× 2
2 |1	1	|	0.500 ↓
0	1	↓	× 2
			1.000

$(25)_{10} = (11001)_2$

$(0.625)_{10} = (0.101)_2$

So, $(25.625)_{10} = (11001.101)_2$

1.4.3 Octal to Decimal Conversion

The method of converting octal numbers to decimal numbers is simple. The decimal equivalent of an octal number is the sum of the numbers multiplied by their corresponding weights.

Example 1.8 : Find decimal equivalent of octal number $(153)_8$

Solution : $1 \times 8^2 + 5 \times 8^1 + 3 \times 8^0 = 64 + 40 + 3 = 107$

So, $(153)_8 = (107)_{10}$

The fractional part can be converted by multiplying it by the negative powers of 8 as shown in the following example.

Example 1.9 : Find decimal equivalent of octal number $(123.21)_8$

Solution : $(1X8^2 + 2X8^1 + 3X8^0) + (2 \times 8^{-1} + 1 \times 8^{-2})$

$= (64 + 16 + 3) + (0.25 + 0.0156) = 83.2656$

So, $(123.21)_8 = (83.2656)_{10}$

1.4.4 Decimal to Octal Conversion

The procedure for conversion of decimal numbers to octal numbers is exactly similar to the conversion of decimal number to binary numbers except replacing 2 by 8.

Example 1.10 : Find the octal equivalent of decimal $(4121)_{10}$

Solution :

8 |4121		
8 |515	1	↑ read from MSB
8 |64	3	to LSB
8 |8	0	
8 |1	0	
0	1	

So, $(4121)_{10}=(10031)_8$

The fractional part is multiplied by 8 to get a carry and a fraction as shown in the following example.

Example 1.11 : Find the octal equivalent of $(.123)_{10}$

Solution : Octal equivalent of fractional part of the decimal number is :

$8 \times 0.123 = 0.984$	0	
$8 \times 0.984 = 7.872$	7	read from LSB
$8 \times 0.872 = 6.976$	6	to MSB
$8 \times 0.976 = 7.808$	7	↓

Read the integer to the left of the decimal point.

The calculation can be terminated after a few steps if the fractional part does not become zero.

The octal equivalent of $(0.123)_{10}= (0.0767)_8$

NOTE : The octal to binary and binary to octal conversion is very easy. Since, 8 is the third power of 2, we can convert each octal digit into its three-bit binary form and vice versa.

Example 1.12 : Convert $(567)_8$ to its binary form.

Solution :

5	6	7
↓	↓	↓
101	110	111

So, $(567)_8 = (101\ 110\ 111)_2$

Conversion from binary to octal is just opposite of the above example.

1.4.5 Hexadecimal to Decimal Conversion

The method of converting hexadecimal numbers to decimal numbers is simple. The decimal equivalent of an hexadecimal number is the sum of

the numbers multiplied by their corresponding weights.

Example 1.13 : Find the decimal equivalent of $(4A8C)_{16}$

Solution :

$$(4A8C)_{16} = (4 \times 16^3) + (10 \times 16^2) + (8 \times 19^{1)} + (12 \times 16^0)$$
$$= 16384 + 2560 + 128 + 12$$
$$=(19084)_{10}$$
$$(4A83)_{16} = (19084)_{10}$$

Example 1.14 : Find the decimal equivalent of $(53A.0B4)_{16}$

Solution :

$$(53A.0B4)16 = (5 \times 16^2) + (3 \times 16^1) + (10 \times 16^0) + (0 \times 16^{-1}) + (11 \times 16^{-2}) + (4 \times 16^{-3})$$
$$= 1280 + 48 + 10 + 0 + 0.04927 + 0.0009765$$
$$= (1338.0439)_{10}$$
$$(53A.0B4)_{16} =(1338.0439)_{10}$$

1.4.6 Decimal to Hexadecimal Conversion

The procedure for conversion from decimal no. and its fraction part to hexadecimal equivalent is exactly similar to the conversion of decimal to binary number except replacing 2 by 16.

Example 1.15 : Convert decimal $(1234.675)_{10}$ to hexadecimal.

Solution : First, we consider $(1234)_{10}$:

	Remainder Decimal	Hexadecimal
16 |1234	2	2
16 |77	13	D
16 |4	4	4

$(1234)_{10} = (4D2)_{16}$

Then, conversion of $(0.675)_{10}$:

	Decimal	Hexadecimal
0.675 × 16 = 10.8	10	A
0.800 × 16 = 12.8	12	C
0.800 × 16 = 12.8	12	C
0.800 × 16 = 12.8	12	C

$(0.675)_{10}= (0.ACC)_{16}$

Hence $(1234.675)_{10}= (4D2.ACC)_{16}$

If the decimal number is very large, it is tedious to convert the number to binary directly. So it is always advisable to convert the number into hexadecimal first, and then convert the hexadecimal to binary.



CHECK YOUR PROGRESS

Q.1. What is the largest numbar that can be represented using 8 bits?

.....

Q.2. What is the weight of 1 in $(10000)_2$.

.....

Q.3. Convert the following:

a) $(565.25)_{10}$ to its equivalent binary number.

b) $(256.24)_8$ to decimal equivalent.

c) $(A3B.BB)_{16}$ to decimal equivalent.

d) $(10010.110)_2$ to decimal equivalent.

e) $(3964.63)_{10}$ to octal equivalent.



1.5 LET US SUM UP

- We have learnt four different number systems used in digital systems in this unit.
- Conversion of one number system to another number system can be done.
- In our day-to-day life, we use the decimal number system. In this system, base is 10 and we use 10 digits from 0 to 9.
- The binary number system has two digits and so its base is 2.
- The octal number system has 8 digits from 0 to 7 and the base of the system is 8.
- The hexadecinal number system has 16 digits from 0 to 9 and A to F and the base of the system is 16.



1.6 FURTHER READING

- *Ram, B. (2000), Computer Fundamentals Architecture and Organization*, New Age International
- *Mano, M.M. (2017), Digital Logic and Computer Design*, Pearson Education India
- *Sinha, P. K. and Sinha P. (2010), Computer Fundamentals*, BPB Publication.
- *Talukdar, P. (2010), Digital Techniques*, N.L. Publications.



1.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : $(1111\ 1111)_2$

Ans. to Q. No. 2 : Weight is $2^4 = 16$

Ans. to Q. No. 3 : a) $(1000110101.01)_2$ b) $(174.44)_{10}$ c) $(2620.8)_{10}$
d) $(18.75)_{10}$ e) $(7574.50)_8$



1.8 MODEL QUESTIONS

Q.1.What is binary number system?

Q.2.What is meant by 'base' or 'radix' of a number system?

Q.3.Convert the following to its binary equivalent:

(i) $(67)_8$ (ii) $(A8D)_{16}$ (iii) $(81B6.F)_{16}$
(iv) $(64.3)_{10}$ (v) $(765.45)_8$ (vi) $(1725.23)_8$

Q.4. Convert the following to its decimal equivalent:

(i) $(11011)_2$ (ii) $(1111.11)_2$ (iii) $(45)_8$
(iv) $(671.23)_8$ (v) $(ABCD)_{16}$ (vi) $(12AF.E4)_{16}$

Q.5.Explain the process of converting a decimal number to binary.

Q.6.How can a binary number be converted into octal and hexadecimal numbers?

****

UNIT 2 : BINARY ARITHMETIC

UNIT STRUCTURE

2.1 Learning Objectives

2.2 Introduction

2.3 Complement of Numbers

- 2.3.1 (r–1)'s Complement
- 2.3.2 r's Complement

2.4 Binary Arithmetic

- 2.4.1 Addition
- 2.4.2 Subtraction
- 2.4.3 Multiplication
- 2.4.4 Division

2.5 Let Us Sum Up

2.6 Further Reading

2.7 Answers to Check Your Progress

2.8 Model Questions

2.1 LEARNING OBJECTIVES

After going through this unit you will be able to:

- describe r's and (r-1)'s complement
- describe binary addition and subtraction
- describe binary multiplication and division.

2.2 INTRODUCTION

In the previous unit, we have been introduced to the different number systems Decimal, Octal, Binary and Hexadecimal. We have also learnt how to convert from one number system to another number system in the previous unit. In this unit, we will learn about the complement systems. We will learn about R and (R-1)'s complement systems. We will also learn to perform binary arithmetic functions like addition, subtraction, multiplication and division in this unit. In the next few units are will learn more about data representation and code conversion methods.

2.3 COMPLEMENT OF NUMBERS



The 7's and 15's complement of a number is found by subtracting each digit of the number from 7 and 15 respectively.

Complements are used in digital computers for simplifying the substraction operation and for logical manipulation.The complement of a binary number is obtained by inverting all of the bits.

For example, the complement of 10011 is 01100 and 00101 is 11010 etc. The complement again depends on the base of the number.

There are two types of complements for a number of base r. These are :

- r's complement and
- (r-1)'s complement,

For example, for decimal numbers the base is 10. Therefore, complements will be 10's complement and (10–1)=9's complement. For binary numbers, the complement are 2's complement and 1's complement since base is 2.

2.3.1 (r–1)'s Complement

Given a number N in base r having n digits, the (r–1)'s complement of N is defined as $(r^n–1) – N$.

9's Complement : For decimal numbers, r=10 and r-1=9, so the 9's complement of N is $(10^n–1) – N$.

For example, with n = 4 we have $10^4 = 10000$ and $10^4–1 = 9999$. It follows the rule that the 9's complement of a decimal number is obtained by subtracting each digit from 9.



Like 8's complement and 16's complement of a number is found by adding 1 to the LSB of the 7's and 15's complement of an octal and hexadecimal number respectively.

For example, 9's complement of 49 is (99–49) = 50

9's complement of 127 is (999–127) = 872

1's Complement : For binary numbers, r = 2 and (r–1) = 1, so the 1's complement of N is $(2^n–1) – N$. Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. 2^n-1 is a binary number represented by n 1's. For example,with n = 4, we have $2^4 = (10000)_2$ and $2^4–1 = (1111)_2$. Thus the 1's complement of a binary number is obtained by subtracting each digit from 1.

However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0 and 0's into 1's. For example, 1's complement of 1010111 is 0101000.

2.3.2 R's Complement

The r's complement of a n-digit number N in base r is defined as $r^n - N$ for N =0 and 0 for N = 0. Comparing with the (r–1)'s complement, we note that the r's complement is obtained by adding 1 to the (r–1)'s complement.

10's Complement : The 10's complement of a decimal number is equal to the 9's complement of the number plus 1

i.e. 10's complement of decimal number = Its 9's complement +1

So, 10's complement of 49 is (99–49) + 1 = 50 + 1 = 51

10's complement of 127 is (999–127) + 1 = 872 + 1 = 873

2's Complement : It is obtained by adding 1 in the 1's complement form of the binary numbers.

i.e. 2's complement of binary number = 1's complement of that number + 1

For example, 2's complement of 1010111 is 0101000 + 1 = 0101001



CHECK YOUR PROGRESS

Q.1. Find 9's complement 10's complement of decimal numbers 44 and 182.

.....

.....

Q.2. Find 1's complement and 2's complement of binary numbers 1101001 and 0000.

.....

.....

2.4 BINARY ARITHMETIC

Like decimal arithmetic operations such as addition, subtraction we can perform arithmetic operation in binary number system too.

2.4.1 Addition

Binary addition is performed in the same manner as decimal addition. Since,

in binary system only two digit 0 and 1 are used, the addition follows the following rules.

0 + 0 = 0

0 + 1 = 1 = 1+0

1 + 1 = 0, Carry 1 to the next left column

1 + 1 +1 = 1, Carry 1 to the next column.

Carry overs are performed in the same manner as in decimal arithmetic.

Example 2.1 : Add the binary numbers

(i) 1011 and 1001

(ii) 10.011 and 1.001

Solution :

	Binary no.	**Equivalent decimal no.**
(i)	11 Carry	
	1011	11
	+1001	9
	10100	20
	1 Carry	
(ii)	10.011	2.375
	1.001	1.125
	11.100	3.500

Since the circuit in all digital systems actually can handle two numbers to performs addition, it is not necessary to consider the addition of more than two binary numbers. When more than two numbers are to be added, the first two are added first and then their sum is added to the third and so on.

The complexity may rise when we add combination of positive and negative binary numbers. In this case, the arithmatic addition is dependent on the representation of

a) Signed magnitude

b) Signed 1's complement

c) Signed 2's complement

This will be more clear if we discuss using the following example:

Example 2.2 : Add 25 and -30 in binary using 7 bit register in signed magnitude representation

a) Signed 1's complement representaion

b) Signed 2's complement representaion

Solution : Here, 25 is + 25 = 0011001 in binary system

–30 = 1011110 in binary system

To do arithmatic addition with one negative number, we have to check the magnitude of the numbers. The number having smaller magnitude is subtracted from the bigger number and the sign of bigger number is selected. To implement such a scheme in digits, hardware will require a long sequence of control decisions as well as circuits that will add, compare and subtract numbers. The better alternative of arithmatic addition with one negative number is signed 2's complement.

In signed 2's complement representation :

We get that +30 is 0 011110

–30 is 1 011110

Now, 2's complement of –30 (including sign bit) 1 100010

+25 is 0 011001

Addition

+25	0	011	001
–30	1	100	010
–05	1	111	011 (Just add the numbers)

The result for negative number will store in signed 2's complement form. So the above result in signed 2's complemnt form including sign bit is:

1 000 100 +1 = 1 000 101

Which is -05 in decimal system.

From the above example, it can be noticed that, signed 2's complement representation is simpler than signed magnitude representaion. This procedure requires only one central decision and only one circuit for adding the two numbers. But it puts additional condition that the negative numbers should be stored in signed 2's complment form in the register. This can be achieved by complementing the positive number bit by bit then incrementing the resultant by 1 to get signed 2's complement.

In signed 1's complement representation : This method is also simple. Here, we add the two numbers including the sign bit. If carry of the most significant bit or sign bit is one, then we increment the result by 1 and discard the carry over.

Addition :

+25 = 0 011 001

–30 = 1 100 001 (1's complement of -30)

–5 = 1 111 010

The result will store in 1's complement format. So, 1111 010 in 1's complement format including the sign bit is 1 000 101 which is the required result.

Example 2.3 : Add -25 and +30 using 7-bit register.

Solution :

–25	1	100	110	(1's complement of 25)
+30	0	011	110	
+5	1 0	000	100	

Carry bit, so add 1 to the sum and discard the carry.

This sum is now = 0 000 101 which is +5

Example 2.4 : Add –25 and –30 using 7-bit register.

Solution :

–25	1	100	110	(1's complement of 25)
–30	1	100	001	(1's complement of 30)
–55	1 1	000	111	

Carry bit, so add 1 to sum and discard the carry.

Now the sum is = 1 001 000, which is –55

Since, +55 is 0 110 111

So, –55 is in 1's complement 1 001 000

The interesting feature about these representation is the representation of 0 in signed magnitude and 1's complement. There are two representations for zero :

Signed magnitude +0 –0

	0	000000	1	000000
Signed 1's complement	0	000000	1	111111

But in signed 2's complement, there is just one zero and there are no positive or negative zero.

+0 000000

–0 in 2's complement is +0 = 1 111111

1

1 0 000000

↑

discard this carry

Thus, both +0 and –0 are same in 2's complement notation. This is an *added advantage* in favour of 2's complement notation. The maximum number which can be accomodated in registers also depends on the type of representation. In general, in a 8 bit register, 1 bit is used as sign. Therefore, the rest of 7 bits are used for representing the value. The value of maximum and minumum number which can be represented are :

For signed magnitude representation $2^7 - 1$ to $-(2^7 - 1)$

$= 128 - 1$ to $-(128 - 1)$

$= 127$ to -127,

which is for signed 1's complement representation.

For signed 2's complement representation is from + 127 to –128. The –128 is represented in signed 2's complement notation as 10000000.

2.4.2 Subtraction

Though there are many other methods for performing subtraction, we will consider the method of subtraction known as complementary subtraction. This is a more efficient method of subtraction while using electronic circuits. We have to follow the following three steps to subtract binary numbers.

In 1's complement method :

1. Find the 1's complement of the number which is subtracting.
2. Add the number which is subtracting from with the complement value obtained from step 1.

3. If there is a carry of 1, add the carry with the result of addition. Else, take complement of the result again and attach a negative sign with the result.

Example 2.5 : Subtract 5 – 6 by 1's complement method.

Solution : Binary equivalent of 5 is 101

Binary equivalent of 6 is 110

Step 1 : 1's complement of 6 is 001

Step 2 : Adding 001 with 101 gives the result :

$$\begin{array}{r} 001 \\ +\ 101 \\ \hline 110 \end{array}$$

Step 3 : Since there is no carry in step 2, we take the complement again which will be 001 and after attaching negative sign, the required result will be –001 which is -1.

In 2's complement method : It is same as 1's complement method except step 3. The steps are :

Step 1 : Find the 2's complement of the number which is subtracting.

Step 2 : Add the number which is subtracting from, with the complement value obtained from step 1.

Step 3 : If there is a carry of 1, ignore it. Else, take 2's complement of the result again and attach a negative sign with the result.

Example 2.6 : Subtract 5 – 7 by 2's complement method.

Solution : Binary equivalent of 5 is 101

Binary equivalent of 7 is 111

Step 1 : The 2's complement of 7 is 000 + 1 = 001

Step 2 : Adding 001 with 101 will give result as :

$$\begin{array}{r} 001 \\ 101 \\ \hline 110 \quad \text{(No carry)} \end{array}$$

Step 3 : Since there is no carry, the 2's complement of 110 is 001 + 1 = 010 and attaching a negative sign the required result is –10 i.e.,-2.

Overflow : An overflow is said to have occured when the sum of two n digit number occupies (n+1) digits. This definition is valid for both binary as well

as decimal digits. But what is the significance of overflow for binary numbers? Well, the answer lies in the representation of numbers. Every computer employs a limit for representation of numbers eg. in our examples we are using 8 bit registers of calculating the sum. But what will happen if the sum of the two numbers can be accommodated only in 9 bits? Where are we going to store the 9th bit? The problem will be more clear by the following example. In case of a +ve no. added to a –ve no., the sum of result will always be smaller than the two numbers. An overflow always occurs when the added numbers are both +ve or both -ve.

Example 2.7 : Add the numbers 65 and 75 in 8 bit register in signed 2's complement notation.

Solution :

65	0	1000001
75	0	1001011
140	1	0001100

This is a -ve number and the 2's complement of the result is equal to -115 which is obviously a wrong result. This has occured because of overflow.

Detection of Overflow : Overflow can be detected as :

If the carry out of the MSBs of number (or, carry into the sign bit) is equal to the carry out of the sign bit, then overflow must have occured. For example

–65	1	0111111	–65	1	0111111
–15	1	1110001	–75	1	0110101
–80	1 1	0110000	–140	1 0	1110100

Carry	= 1	Carry	= 10
Carry from MSB	= 1	Carry from MSB	= 0
Carry from sign bit	= 1	Carry from sign bit	= 1
Sign bit is	= 1	Sign bit is	= 0
No overflow		Overflow	

Thus, overflow has occured, i.e. the arithmatic results so calculated have exceeded the capacity of the representation. This overflow also implies that the calculated results might be errorneous.

2.4.3 Multiplication

Multiplication in binary follows the same rules that are followed in the decimal system. The rules to be remembered are:

$0 \times 0 = 0$ $1 \times 0 = 0$

$0 \times 1 = 0$ $1 \times 1 = 1$

For example, multiplying 10101 × 11001

```
   10101
 × 11001
 -------
   10101
  10101
 10101
 -------
1000001101
```

2.4.4 Division

The process of binary division is same as the decimal division. In binary division we have two rules:

$0/1 = 0$ $1/1 = 1$

The steps for binary division are :

1. Start from the left of the divided.
2. Perform subtraction in which the divisor is subtracted from the divedend.
   a) If subtraction is possible put a 1 in the quotient and subtract the divisor from the corresponding digits of the dividend. Else put a 0 in the quotient
   b) Bring down the next digit to the right of the remainder.
3. Execute step 2 till there are no more digits left to bring down from the dividend.

Example 2.8 : Divide 1011001 by 110

Solution :

```
                  0111      (Quotient)
(Divisor)  110 |1011001     (Dividend)
              110
               1010
```

110

1000

110

0101

Here 111 is the quotient and 101 is the remainder.



CHECK YOUR PROGRESS

Q.3.Add 35 and -40 in binary using 7 bit register in 2's complement representation.

.....

Q.4.Add binary no. 110011.010 and 1000.10

.....

Q.5.Subtract $(1010101)_2$ -$(1001001)_2$ by 2's complement method.

.....

Q.6.Multiply $(11001)_2$ by $(101)_2$.

.....

Q.7.Divide $(101101.101)_2$ by $(110)_2$.

.....



2.5 LET US SUM UP

- The 1's complement of a binary number is formed by changing 1's into 0 and 0's into 1's.
- The 2's complement of a binary number is obtained by adding 1 to the 1's complement of the binary number.
- Binary addition is performed in the same manner as decimal addition.
- Complementary subtraction method is considerd for binary subtraction.
- An overflow is said to have occured when the sum of two n digits number occupies (n+1) digits.



2.6 FURTHER READING

- *Ram, B. (2000), Computer Fundamentals Architecture and Organization*, New Age International
- *Mano, M.M. (2017), Digital Logic and Computer Design*, Pearson Education India
- *Sinha, P. K. and Sinha P. (2010), Computer Fundamentals*, BPB Publication.
- *Talukdar, P. (2010), Digital Techniques*, N.L. Publications.



2.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : 9's complement of 44 is 55 and 182 is 817
10's complement of 44 is 56 and 182 is 818

Ans. to Q. No. 2 : 1's complement of 1101001 is 0010110 and 0000 is 1111
2's complement of 1101001 is 0010111 and 0000 is 10000

Ans. to Q. No. 3 : $(000101)_2$

Ans. to Q. No. 4 : $(111011.110)_2$

Ans. to Q. No. 5: $(1100)_2$

Ans. to Q. No. 6 : $(1111101)_2$

Ans. to Q. No. 7 : $(111.10011)_2$



2.8 MODEL QUESTIONS

Q.1. Find 8's and 16's complement of the following:
(i) 67 (ii) 5672

Q.2. Find the 7's and 15's complement of the following:
(i) 643 (ii) 15AB

Q.3. Perform the following by 2's complement method:
(i) 10101 – 11011 (ii) 100011 – 1111

Q.4. Multiply 1101 to 11110.

Q.5. Divide 110011 by 110.

****

UNIT 3 : INTRODUCTION TO DATA REPRESENTATION

UNIT STRUCTURE

3.1 Learning Objectives

3.2 Introduction

3.3 Data Representation

3.4 Fixed Point Representation

3.5 Floating Point Representation

3.6 Let Us Sum Up

3.7 Further Reading

3.8 Answers to Check Your Progress

3.9 Model Questions

3.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- identify how data is represented in computers
- describe fixed point representation
- describe floating point representation

3.2 INTRODUCTION

In the previous units, we have learnt about different number systems like binary, octal, decimal and hexadecimal. We have also learnt how to do binary arithmetic operations like addition, subtraction, multiplication and division.

In this unit, we will discuss about the concept of data representation. Fixed and floating point representation are introduced in this unit. Floating point representation will be covered in detail in unit 7 later. In the next unit we will discuss about the different types of computer codes.

3.3 DATA REPRESENTATION

Data are usually represented by using the alphabets A to Z, numbers 0 to 9 and various other symbols. This form of representation is used to formulate problem and fed to the computer. The processed output is required

in the same form. This form of representation is called *external data representation*. However, the computer can understand data only in the form of 0's and 1's. The method of data representaion in a form suitable for storing in the memory and for processing by the CPU is called the internal data representation on digital computer.

Data, in general, are of two types : Numeric and non-numeric (Character data). The numeric data deals only with numbers and arithmetic operations and non numeric data deals with characters, names, addresses etc. and non-arithmetic operations. Numeric data can be represented using fixed point or floating point representations. Let us look at these in detail.

3.4 FIXED POINT REPRESENTATION

A fixed point number in binary system uses a sign bit. A positive number has a sign bit 0 while the negative number has a sign bit 1. A negative number can be represented in one of the following ways.

- Signed magnitude representaion
- Signed 1's complement representaion
- Signed 2's complement representaion

Assume that the size of the register is 7 bit and the 8th position bit in used for error checking and correction or other purposes.

a) Signed magnitude representation

	+6	–6	
0	000110	1	000110
↑		↑	No change in the
Sign bit		Sign bit	magnitude, only the sign bit changes

b) Signed 1's complement representation

	+6	–6	
0	000110	1	111001

Here 0 and 1 are sign bits. We get 1's complement for the –ve integer is by taking complement of all the bits of +ve no. including sign bit.

c) Signed 2's complement representation

	+6	–6	
0	000101	1	111011
↑		↑	2's complement of the
Sign bit		Sign bit	positive number
			including sing bit

The signed magnitude system is easier to interpret but computer arithmetic with this system is not efficient. The circuits for handling numbers are simplified if 1's or 2's complement systems are used and as a result one of these two is almost always adopted.

Note 1 : In 1's and 2's complements, all positive integers are represented as sign magnitude system.

Note 2 : When all the bits of the computer word are used to represent the number and no bit is used for signed representation, it is called unsigned representation of the number.

3.5 FLOATING POINT REPRESENTATION

A number which has both an integer part as well as a fractional part is called real number or floating point number. A floating point number is either positive or negative. Examples of real decimal numbers are 156.65, 0.893, –235.75, –0.253 etc. Examples of binary real numbers are 101.101, 0.11101, –1011.101, –0.1010 etc.

The first part of the number is a fixed point number which is called mantissa. It can be an integer or a fraction.

The second part specifies the decimal or binary point position and is termed exponent. It is not a physical point. Therefore, whenever we are representing a point it is termed as an exponent. It is only the assumed position. For example, for decimal 0. +15.37, the typical floating point notaion is :

$51.47 = 0.5147 \times 10^{2}$ or 5147×10^{-2}

Now, the floating point representation of 0.5147×10^{3} is :

Sign Sign

↓ ↓

|0 .5147| |0 02|

Mantissa (fraction) Exponent

The floating point representation of 5147 X 10^{-2} is

Sign Sign

↓ ↓

|0 5147| |0 02|

Mantissa (Integer) Exponent

Similarly, for example a floating point binary number 1011.1010 can be represented as : $1011.1010 = 0.10111010 \times 2^4$

This can be represented in a 16 bit register as follows

Sign Sign

↓ ↓

|0 .10111010| |0 000100|

Mantissa (fraction) Exponent

The mantissa occupies 9 bits (1 bit for sign and 8 bits for value) and the exponent 7 bits (1 bit for sign and 6 bits for value). The binary point (.) is not physically indicated in the register, but it is only assumed (position) to be there.

In general form, the floating point numbers is expressed as :

$N = M \times R^e$

Where, M – Mantissa

R – Radix (or base)

e – Exponent

The mantissa M and exponent e are physically present in register. But the radix R and the point (decimal or binary point) are not indicated in the register. There are only assumed for computation and manipulation.

Normalized Floating point Number : Floating point numbers are often represented in normalized forms. A floating point number where mantissa does not contain zero as the most significant digit of the

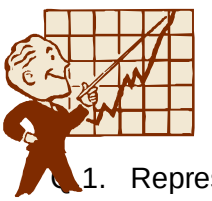
number is considered to be in normaliged form. For example, 0.00038695×10^5 and 0.0589×10^{-4} are not normaliged numbers. But 0.38695×10^2 and 0.589×10^{-5} are normaliged numbers. Similarly, for binary number also, 0.0011001×2^8 and 0.0001011×2^{-5} are not non-normaliged binary numbers. But 0.11001×2^6 and 0.1011×2^{-8} are normaliged binary numbers.

A zero cannot be normaliged as all the digits in the mantissa is zero.

Arithmetic operations involved with floating point numbers are more complex. It takes larger time for execution and requires complex hardware. But floating point representaion is frequently used in scientific calculations.

Overflow and Underflow : When the result is too small to be presented by the computer, an overflow or underflow condition exists. When two floating-point numbers of the same sign are added, a carry may be generated out of high-order bit position. This is known as mantissa overflow. In case of addition or subtraction floating point numbers are aligned. The mantissa is shifted right for the alignment of a floating point number. Sometimes, the low order bits are lost in the process of alignment. This is referred as mantissa underflow. To perform the multiplication of two floating point numbers, the exponents are added. In certain cases the sum of the exponents maybe too large and it may exceed the storing capacity of the exponent field. This is called exponent overflow. In case of division the exponent of the divisior is subtracted from the exponent of the dividend. The result of subtraction may be too small to be represented. This is called exponent underflow.

Overflow or underflow resulting from a mantissa operation can be corrected by shifting the mantissa of the result and adjusting the exponent. But the exponent overflow or underflow can not be corrected and hence, an error indication has to be displayed on the computer screen.



CHECK YOUR PROGRESS

Q.1. Represent 10 by the following representation method.

a) Signed magnitude representation.

b) Signed 2's complement representation.

Q.2. Represent $(1010.1010)_2$ with floating point representation method.

Q.3. Fill in the blanks

a) When all the bits of the computer word are used to represent the number, it is called _____ of the number.

b) A _____ cannot be normalized as all the digits in the mantissa is zero.

c) In general, _____ of a number is used to represent sign bit in signed magnitude representation.

d) _____ takes longer time for execution and requires complex hardware.

e) An _____ occurs when the result is too small to be represented by the computer.



3.6 LET US SUM UP

- A fixed point numbers in binary system uses a sign bit. A positive number has a sign bit 0 while the negative number has a sign bit 1.
- For fixed point we have discussed three different methods, they are:
  - Ø Signed magnitude representation method
  - Ø Signed 1's complement representation method
  - Ø Signed 2's complement representation method
- **Signed magnitude representation:** It uses one bit, usually the leftmost bit, to indicate the sign where '0' indicates the positive integer and the '1' indicates the negative integer. The rest of the bits are used as magnitude.

- A number which has both an integer part and a fractional part is called real number or floating point number. A floating point number can be either positive or negative.
- The first part of the number is a fixed point number which is called ***Mantissa***. It can be an integer or a fraction.
- The second part specified the decimal or binary point position and is termed ***exponent***.
- A floating point number where mantissa does not contain zero as the most significant digit of the number is considered to be in normalized form.
- When two floating-point numbers of the same sign are added, a carry may be generated out of high-order bit position. This is known as mantissa overflow. In case of division the exponent of the divisor is subtracted from the exponent of the dividend. The result may be too small to be represented and is calaled exponent underflow.



3.7 FURTHER READING

- Morris, M.M. (1987). DIgital Logic and Computer Design.
- Sinha, P.K., & Sinha, P. (2010). Computer Fundamentals (Vol. 4). BPB publications.
- Ram, B. (2000). Computer Fundamentals: Architecture and Organisation. New Age International.



3.8 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: a) 001010

b) 110110

Answer to Q2: Mantissa is 10101010, fraction is 000100 (in 16 bit representation)

Answer to Q3: a) unsigned representation

b) zero

c) leftmost bit

d) Arithmetic operation

e) overflow or underflow



3.9 MODEL QUESTIONS

Q.1. Explain the fixed point representation method with example.

Q.2. How can decimal number be represented?

Q.3. What is mantissa and exponent of decimal representation?

Q.4. How can a floating point number be normalized for computerized representation?

Q.5. What is overflow and underflow?

Q.6. What is data representation?

****

UNIT 4 : CODE CONVERSION

UNIT STRUCTURE

4.1 Learning Objectives

4.2 Introduction

4.3 Computer Code

4.4 BCD number

4.5 ASCII Code

4.6 EBCDIC

4.7 Gray Code

4.8 Let Us Sum Up

4.9 Further Reading

4.10 Answers to Check Your Progress

4.11 Model Questions

4.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- describe different computer code systems
- describe BCD and ASCII code
- describe EBCDIC and Gray code

4.2 INTRODUCTION

In the previous units we have learned about different number systems like decimal, binary, octal and hexadecimal. We have also covered binary arithmetic operations like addition, subtraction multiplication and division. We have also discussed about the methods for representing data. Fixed point representation and floating point representation were also introduced in the previous unit.

In this unit, we will introduce the concept of computer code and code conversion. Different computer codes like BCD, ASCII, EBCDIC and gray codes are discussed in this unit. We will also learn how to convert binary to gray code and vice versa. In the next units, we will discuss about boolean algebra and logic gates.

4.3 COMPUTER CODE

A **code** is a symbol or group of symbols that represents discrete elements. Coding of characters has been standardised to enable transfer of data between computers. *Numeric data* is not the only form of data handled by a computer. We often require to process *alphanumeric data*. An alphanumeric data is a string of symbols, where a symbol may be one of the letters A, B, C, ..., Z, or one of the digits 0, 1, 2, ..., 9, or a special character, such as + – */, . () = (space for blank) etc. However, the bits 0 and 1 must represent any data internally. Hence, computers use binary coding schemes to represent data internally. In binary coding, a group of bits represent every symbol that appears in the data. The group of bits used to represent a symbol is called a *byte*. To indicate the numbers of bits in a group, sometimes a byte is referred to as “n-bit byte”, where the group contains n bits. However, the term “byte” commonly means an 8-bit byte because most modern computers use 8 bits to represent a symbol.

4.4 BCD NUMBER

In computing and electronic systems, **Binary-Coded Decimal** (**BCD**) is a way to express each of the decimal digits with a binary code. Its main advantage is that it allows easy conversion to decimal digits for printing or display and faster decimal calculations. Decimal numbers with their BCD equivalent are given in the Table 4.1 :

Table 4.1 : BCD Code

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Unlike binary encoded numbers, BCD encoded numbers can easily be displayed by mapping each of the nibbles(4-bits) to a different character. Examples of conversion of decimal to BCD and BCD to decimal are included below:

Example 4.1 : Convert the decimal numbers 47 and 180 to BCD.

Solution : 4 = 0100 and 7 = 111

47 = 0100111

Similarly, 180=0001 1000 0000

Example 4.2 : Convert each of the BCD code 1000111 to decimal.

Solution : First we have to divide the whole BCD code by a set of 4-bits from right to left. Then from left to right we can put the corresponding decimal numbers.

0100	0111
4	7

Thus, 1000111(in BCD) = 47 (in Decimal)

BCD addition : BCD is a numeric code and can be used in arithmetic operations. Addition is the most important operation because the other three operations (subtraction, multiplication, and division) can be accomplished by the use of addition. Here is how to add two BCD numbers:

Step1 : Add the two BCD numbers, using the rules for binary addition.

Step 2 : If a 4-bit sum is equal to or less than 9, it is a valid BCD number.

Step 3 : If a 4-bit sum is greater than 9, or if a carry out of the 4-bit group is generated, it is an invalid result. Add 6(0110) to the 4-bit sum in order to skip the six invalid states and return the code to 8421. If a carry results when 6 is added, simply add the carry to the next 4-bit group.

Example 4.3 : Add the following BCD numbers :

a) 0001 + 0100

b) 10000111 + 01010011

Solution : The decimal number addition are shown for comparison.

(a)

0001	1
+0100	+4
0101	5

∴ 0001 + 0100 = 0101 Which is valid BCD no. (Value < 9)

(b)

1000	0111	87
+0101	0011	+ 53

	1101	1010	Both groups are invalid (>9) +140
	+0110	+0110	Add 6 (i.e.,0110) to both groups
0001	0100	0000	Valid BCD number which is 140 in decimal.

4.5 ASCII CODE

ASCII stands for American Standard Code for Information Interchange. It is a very well-known fact that computers can manage internally only 0s (zeros) and 1s (ones). By means of sequences of 0s and 1s the computer can express any numerical value as its binary translation, which is a very simple mathematical operation.

However, there is no such evident way to represent letters and other non-numeric characters with 0s and 1s. Therefore, in order to do that, computers use *ASCII tables*, which are tables or lists that contain all the letters in the roman alphabet plus some additional characters. In these tables each character is always represented by the same order number. For example, the ASCII code for the capital letter “A” is always represented by the order number 65, which is easily representable using 0s and 1s in binary: 65 expressed as a binary number is 1000001. ASCII has 128 character codes(from 0 to 127) and symbols represented by a 7-bit binary code.

ASCII is the common code for microcomputer equipment. The first 32 characters in the ASCII-table are unprintable control codes and are used to control peripherals such as printers. Examples of control characters are “NULL”, “line feed”, “start of text” and “escape”. The other characters are graphic symbols that can be printed or displayed and that include the letters of the alphabet (lowercase and upper case), the ten decimal digits, punctuation signs, and other commonly used symbols.

4.6 EBCDIC

EBCDIC is the abbreviation for **Extended Binary-Coded Decimal Interchange Code**. EBCDIC is an IBM code for representing characters as numbers. It uses 8 bits per character. Thus 256 charaters can be represented with 8 bits. The 9th position bit can be used for parity. The EBCDIC code is used in IBM mainframe models and other similar machines.

In EBCDIC, the first 4 bits are known as zone bits and remaining 4 bits represent digit values. Electronic circuits are available to transform characters from ASCII to EBCDIC and vice-varsa.

4.7 GRAY CODE

The gray code is an unweighted code not suited for arithmetic operations, but useful for input output devices, analog to digital converters etc. The **Gray code**, named after *Frank Gray*, is a binary numeral system where two successive values differ in only one digit. It is sometimes referred to as reflected binary, because the first eight values compare with those of the last 8 values, but in reverse order.

The Gray code was originally designed to prevent spurious output from electromechanical switches. Today, Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems.

Table 4.2 : Gray Code

Decimal	Gray Code	Binary
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

Conversion from Binary to Gray code : The following rules explain how to convert from a binary number to a Gray code **:**

a) The most significant bit (left most) in the gray code is the same as the corresponding most significant bit in the binary code.

b) Going from left to right, add each pair of adjacent pair of binary code to get the next Gray code bit. Discard carries.

For example, the conversion of the binary number 1100 to gray code is as follows:

Binary code 1 1 0 0

MSB (Most significant bit) LSB (Least significant bit)

MSB of Gray code will be same as MSB of Binary code. Here, it will be 1. Now , addition of each pair of adjacent bits of Binary code:

1 + 1 = 10 = 0 , discarding the carry 1

1 + 0 = 1, no carry

0 + 0 = 0, no carry

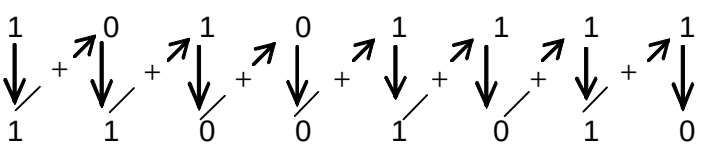
Therefore, 1100 in gray code is 1010

Conversion from Gray to Binary code : The following rules apply:

a) The most significant bit (left most) in the binary code is the same as the corresponding most significant bit in the gray code.

b) Add each binary code bit generated by Gray code bit in the next adjacent position. Discard carries.

For example, binary 10101111 in gray code will be like this :





CHECK YOUR PROGRESS

Q.1. Convert the following binary numbers to gray code

(i) 11011 (ii) 10110

Q.2. Convert the following Gray codes to Binary codes.

(i) 11011 (ii) 100111

Q.3. (i) The American Standard Code for Information Interchange is a standard _____ bits code.

(ii) EBCDIC uses _____ bits per character.

(iii) Code is a representation of _____.

(iv) In both binary and gray code, the _____ is same.

Q.4. Add the following BCD numbers :

(i) 00100011 + 00010001 (ii) 1001 + 0100



4.8 LET US SUM UP

- BCD code is a way to express decimal digits with a binary code.
- BCD, ASCII, EBCDIC,and Gray Code are the commonly used commputer codes.
- BCD is a numeric code and can be used in arithmetic operations.
- ASCII is a 7 bit binary code.
- EBCDIC uses 8-bits to represent character.
- Gray codes are used to facilitate error correction.
- ASCII stands for American Standard Code for Information Interchange.
- Gray codes area not not suited for arithmetic operations.



4.9 FURTHER READING

- *Ram, B. (2000), Computer Fundamentals Architecture and Organization*, New Age International
- *Mano, M.M. (2017), Digital Logic and Computer Design*, Pearson Education India
- *Sinha, P. K. and Sinha P. (2010), Computer Fundamentals*, BPB Publication.
- *Talukdar, P. (2010), Digital Techniques*, N.L. Publications.

4.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : (i) 10110 (ii) 11101

Ans. to Q. No. 2 : (i) 10010 (ii) 111010

Ans. to Q. No. 3 : (i) 7 (ii) 8 (iii) Discrete elements

(iv) MSB-bit

Ans. to Q. No. 4 : (i) Solution :

0010	0011	35	
+0001	0001	+17	
0011	0100	+52	Valid BCD no. (since both <9)

(ii) Solution :

1001		9
+0100		+4
1101	Invalid BCD number (>9)	+13
+0110	Add 6 (i.e., 0110)	
0001 0011	A valid BCD number which is 13 in decimal	

4.11 MODEL QUESTIONS

Q1. What is BCD code?

Q2. Convert the decimal numbers 15, 275 to BCD.

Q3. Add the two BCD numbers 10000111 and 01010011.

Q4. Write BCD for the following decimal number:

(i) 679 (ii) 45.96

Q.5. Give the importance of Gray codes.

Q.6. Write a short note on ASCII codes.

****

UNIT 5 : BOOLEAN ALGEBRA

UNIT STRUCTURE

5.1 Learning Objectives

5.2 Introduction

5.3 Boolean Operators

5.4 Basic Theorems and Postulates of Boolean Algebra

5.5 Let Us Sum Up

5.6 Further Reading

5.7 Answers to Check Your Progress

5.8 Model Questions

5.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define Boolean algebra
- explain basic concepts of Boolean Algebra
- define the basic theorems and postulates of Boolean Algebra
- define Boolean function

5.2 INTRODUCTION

In the previous units we have been introduced to concepts like binary arithmetic, data representation and code conversion. We have learnt about the methods for representing data. We have also learnt about the different types of computer codes like gray, BCD, ASCII and EBCDIC codes. The conversion process from one code to another is also discussed in previous units.

In this unit, you will be able to learn about the fundamentals of Boolean Algebra. You will also learn the basic theorems and postulates of Boolean algebra. Different concepts like the principles of Duality, De Morgan's theorem are also covered in this unit.

In the next unit, we will learn about the different types of logic gates. Conversion of logic gates is also covered in the next unit.

5.3 BOOLEAN OPERATORS

Boolean algebra may be defined with the help of three sets of components viz,

i) a set of elements,

ii) a set of operators and

iii) a number of unproved axioms or postulates.

A set of elements is any collection of objects having a common property. A set of operators are the binary operators, which are rules that are assigned to each pair of elements of the set, a unique element from the set. The axioms or postulates form the basic assumptions from which it is possible to deduce the rules, theorems and properties of Boolean algebra. A variable in Boolean algebra can take only two vales, 1 (TRUE) or 0 (FALSE). i.e. TRUE is represented by 1 and FALSE is represented by 0. Boolean algebra is used for designing and analysing digital circuits. There are three basic operations in Boolean algebra and these operations are done with the help of three operators, viz. AND, OR and NOT. (These are also called basic logic operation).

AND operation : Boolean AND operator for two variables A and B can be represented as:

A and B or A.B or AB

It results 1 or TRUE if both the operands A and B are 1 (TRUE), otherwise the result is 0 (FALSE).

OR Operation : The OR operator for the same variables can be represented as :

A OR B or A+B

The result of this operation in 0 (FALSE) if both the variables are 0 (FALSE); otherwise the result is 1 (TRUE).

NOT Operation : The NOT operation for a variable A can be represented as:

NOT $\bar{A}$ or A or $A^{/}$.

It returns the opposite value of the variable i.e. returns 0 (FALSE) if A is 1 (TRUE) and vice versa.

The results of these Boolean operations can be represented in a tabular form, which is referred to as the "***truth table***".

Table 5.1: Truth Table for AND, OR and NOT operation

A	B	(A AND B) A.B	(A OR B) A+B	NOT(A)
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Note : Boolean addition is same as the OR operation and Boolean multiplication is same as AND operation.

Truth Table : A truth table is a table which gives the output for all combination of input values. A truth table can be drawn for a particular function/operation. In the truth Table 5.1, three tables are merged into a single one.

In addition to these three basic Boolean operations, three more operators have been defined for Boolean algebra. They are: XOR (Exclusive OR), NOR (Not+OR) and NAND (Not+AND).

5.4 BASIC THEOREMS AND POSTULATES OF BOOLEAN ALGEBRA

The theorems and postulates are the most basic relationships in Boolean algebra. Six theorems and four postulates of Boolean algebra are listed in Table 5.2. The postulates are basic axioms of the algebric structure and need no proof, but the theorems must be proven with the help of the postulates. Both the postulates and theorems are listed in pairs: one is the dual of the other.

Table 5.2 Postulates and Theorems of Boolean Algebra

Postulate 2	(a) $x + 0 = z$	(b) $x.1 = x$
Postulate 5	(a) $x + x' = 1$	(b) $x. x' = 0$
Theorem 1	(a) $x + x = x$	(b) $x. x = x$
Theorem 2	(a) $x + 1 = 1$	(b) $x. 0 = 0$
Theorem 3, Involution $(x')' = x$		
Postulate 3, Commutative	(a) $x + y = y + x$	(b) $xy = yx$
Theorem 4, Associative	(a) $x + (y + z) = (x + y) + z$	

(b) x (yz) = (xy)z

Postulates 4, Distributive (a) x (y + z) = xy + xz

(b) x+yz = (x+y)(x+z)

Theorem 5, De Morgan's (a) $(x + y)' = x' y'$ (b) $(xy)' = x' + y'$

Theorem 6, Absorption (a) x + xy = x (b) x(x + y) = x

Duality Principle : This is an important property of Boolean algebra. It states that every algebric expression deducible from the postulates of Boolean algebra remains valid if the operator and identity elements are interchanged. In a two-valued Boolean algebra (which is defined on a set of two elements, B = {0, 1}, with rules for the two binary operators + and•), the identity elements and the elements of the set B are same: 1 and 0. If we need the dual of an algebric expression we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

The proofs of the theorems with one variable are given below:

Theorem 1 (a) x + x = x

x + x = (x + x).1 by postulate : 2 (b)

= (x + x)• $(x + x')$: 5 (a)

= x + x x' : 4 (b)

= x + 0 : 2 (a)

= x

Theorem 1 (b) : x. x = x

x. x = x. x + 0 by postulate : 2 (a)

= xx + x x' : 5 (b)

= x $(x + x')$: 4 (a)

= x.1 : 5 (a)

= x : 2 (a)

If we observe carefully we see that theorem 1 (b) is the dual of theorem 1 (a) and each step of the proof in part (b) is the dual of part (a). Thus any dual theorem can be similarly derived from the proof of its corresponding pair.

Theorem 2 (a): x + z = 1

x + 1 = 1. (x + 1) by postulate : 2 (b)

$= (x + x') (x + 1)$: 5 (a)

$= x + x'.1$: 4 (b)

$= x + x'$: 2 (b)

$= 1$: 5 (a)

Theorem 2 (b): $x. 0 = 0$ by duality.

Theorem 3: $(x')' = x$

We have $x. x' = 0$ from postulate 5 (b), which defines complement of x.

The complement of x' is x and is also $(x')'$. Since the complement is unique, therefore we have $(x')' = x$.

We can prove the theorems which involve two or three variables, algebraically, from the postulates and theorems which have already been proven. Let us consider the absorption theorem.

Theorem 6 (a) : $x + xy = x$

$x + xy$	$= x.1 + xy$	by postulate 2 (b)
	$= x (1 + y)$	by postulate 4 (a)
	$= x (y + 1)$	by postulate 3 (a)
	$= x.1$	by theorem 2 (a)
	$= x$	by postulate 2 (b)

Theorem 6 (b): $x (x + y) = x$ by duality.

The theorems of Boolean algebra can also be proved easily with the help of truth table. The truth table shown in Table 5.3 verifies the theorem 6 (b).

Table 5.3 Truth Table for verification of Theorem 6 (b).

1	**2**	**3**	**4**
x	**y**	**x + y**	**x (x + y)**
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Here we see that, column 4 and column 1 are same, i.e. $x (x + y) = x$. Since the algebric proof of the De Morgan's theorem and the associative law are very long, we can show their validity with truth tables easily. Let us consider the De Morgan's theorem: $(x + y)' = x' y'$.

Now, the truth table for this is shown in Table 5.4.

Table 5.4 : Truth table for De Morgan's theorem

1	2	3	4	5	6	7
x	**y**	**x + y**	**$(x + y)'$**	**x'**	**y'**	**x' y'**
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Column 4 is equal to column 7, so, $(x + y)' = x' y'$

Complement of a Boolean Function : The complement of a function F is F'. It can be obtained from an interchange of $0'$s for $1'$s and $1'$ s for $0'$s in the value of F. Algebraically, the complement of a function may be derived through De Morgan's theorem. The De Morgan's theorem can be extended to any number of variables.

The theorem can be generalized as :

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

$(ABC...G)' = A' + B' + C' + ... + G'$

This generalized form of De Morgan Theorem states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal. (A literal is a primed or unprimed variable). Let $F_1 = A'BC + A'B'C'$. Now F_1' i.e. complement of the function can be obtained by applying the De Morgan's theorem as follows:

$$F_1' = (A'BC + A'B'C')'$$
$$= (A'BC)' (A'B'C')'$$
$$= (A+B'+C') (A+B+C)$$

An easier procedure for deriving the complement of a function is to take the dual of the function and complement each literal. Thus, the dual of $F_1 = (A'+B+C) (A'+B'+C')$ and then complementing each literal, $F_1' = (A+B' +C') (A+B+C)$.



CHECK YOUR PROGRESS

Q.1. What is a truth table? Create a truth table for AND and OR operation?

Q.2. Define the principle of Duality.

Q.3. State and prove the De Morgan's theorem for two variables. (Using truth table)



5.5 LET US SUM UP

- Boolean addition is same as logical OR and Boolean multiplication is same as logical AND operation.
- The logical NOT operation changes logical 1 to 0 and vice versa.
- To get dual of any Boolean expression, you have to replace every 0 with 1 and every 1 with 0, and replacing every operator (+) with (.) and every (.) with (+).
- Any Boolean expression obtained by interchanging 0s and 1s and the operator (of an expression) is called the dual expression. This is the duality principle of Boolean algebra.
- De Morgan's theorem can be extended to any number of variables. It is useful in obtaining the complement of a Boolean function.



5.6 FURTHER READING

- Mano, M.M. (1982), Computer System Architecture, Englewood Cliffs, N. J. 53-54
- Mano, M.M. (2017), Digital logic and Computer Design, Pearson Education India.
- Ram, B. (2000), Computer Fundamentals Architecture and Organization, New Age International.

5.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : A truth table is a table, which has two sides, viz, input and output. In input side, we have all the possible combinations of input variables, i.e. for 2 variables, we have $2^2 = 4$ input combinations(4 rows), for 3, we have 8 combinations and so on. The output side gives us the result of the function or operation either 1 (TRUE) or 0 (FALSE). Output 1 indicates that the corresponding input combination (minterm) is present in the function.

Truth table for AND and OR operations

Input		Output	
A	B	A.B	A+B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Ans. to Q. No. 2 : The principle of duality states that for any Boolean expression, another valid expression can be obtained by replacing each 0 with 1 and 1 with 0 and interchanging the binary operators (.) and (+). For any pair of expression so obtained, the original one is called the primal and the new one is called the dual expression. For example – the expression $a\,b' + cd = 1$, its dual is $(a + b')(c + d) = 0$.

Ans. to Q. No. 3 : The De Morgan's theorem for two variables A and B can be state as:

I. $(A + B)' = A'\,B'$

II. $(AB)' = A' + B'$

i.e. complement of an expression can be obtained by complementing each literal and interchanging the binary operators (.) and (+).

Truth Table

Proof :

A	B	A'	B'	$A'B'$	(A+B)	$(A+B)'$	AB	$(AB)'$	$A'+B'$
0	0	1	1	1	0	1	0	1	1
0	1	1	0	0	1	0	0	1	1
1	0	0	1	0	1	0	0	1	1
1	1	0	0	0	1	0	1	0	0

= =



5.8 MODEL QUESTIONS

Q.1. State and prove De Morgan's Theorem.

Q.2. What are the different ways of representing a Boolean function?

Q.3. What is Duality Principle?

Q.4. Define truth table.

Q.5. Prove that x (y + z) = xy + yz using truth tables.

Q.6. Prove that x + xy = x.

****

UNIT 6 : LOGIC GATES

UNIT STRUCTURE

6.1 Learning Objectives

6.2 Introduction

6.3 Logic Gates

- 6.3.1 OR Gate
- 6.3.2 AND Gate
- 6.3.3 NOT Gate
- 6.3.4 NAND Gate
- 6.3.5 NOR Gate
- 6.3.6 XOR Gate
- 6.3.7 XNOR Gate

6.4 De Morgan's Theorem

6.5 Truth Table

6.6 Conversion of the Logic Gates

6.7 Let Us Sum Up

6.8 Further Reading

6.9 Answer to Check Your Progress

6.10 Model Questions

6.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- describe different logic gates
- prepare truth table for AND, OR, NOT, NAND, NOR, XOR and XNOR gates
- obtain boolean function from a truth table
- perform NAND implementation of a Boolean function
- perform NOR implementation of a Boolean function

6.2 INTRODUCTION

In the previous units, we have learnt about number system, binary arithmetic, boolean algebra and conversion codes. Different number systems have been covered along with different types of computer codes in previous units. In addition to these, boolean operators and basic postulates

and theorems are also covered in the previous units.

In this unit, we will discuss about the different types of logic gates. The graphical symbols and truth table of the logic gates are also discussed in detail. The conversion of logic gates is also covered in this unit. In the next unit, we will discuss in detail about the floating point representation.

6.3 LOGIC GATES

A logic gate is an electronic circuit, which is used or collectively can be used to transform a boolean function or expression into a logic diagram. Logic gates have only one output and atleast two inputs except for the NOT gate, which has only one input. The output signals appears only for certain combinations of input signals. Binary information available in the input lines are manipulated by the gates. Three basic logic circuits, commonly called gates are used to make logic decision : they are OR, AND and NOT gate. Logic gates are available in the form of various IC families and are the basic building block of various circuits. Each gate has a distinct graphic symbol and its operation can be described by means of an algebric function. The input-output relationship of the binary variables for each gate can be represented in tabular form in a truth table.

Before going to discuss about the functions of the logic gates, we have to know few basic terms that are associated with the functions of gates. Logic 1 and 0, that are applied as input or may be obtained as ouput of a gate, are represented by voltage levels. Positive logic (or active high levels) means that the most positive logic voltage level (also referred to as the high level) is defined to be the logic state1. On the other hand, the most negative logic voltage level (also referred to as the low level) is defined to be the logic state 0. Negative logic (or active law levels) is just the opposite, the most positive (high) level is 0, and the most negative (low) level is a 1. For instance, if the voltage levels are (–0.1)v and (-5)v, then in a positive logic system, the (-5)v level represents a zero and the –0.1v represents a 1. Conversely, if the voltage levels are 0.1v and 5v, then in a negative logic system, the 5v levels represents a zero and the 0.1v represents a one.

The choice of positive or negative logic is made by the individual logic designer. We cannot say one is advantageous over the other. It is common to see that most logic designers and text books on logic design use positive logic.

6.3.1 OR Gate

An OR gate has two or more inputs and a single output. It is an electronic circuit and the output of an OR gate is HIGH (logic1) if atleast one of the inputs is HIGH, otherwise (i.e. if all inputs are LOW) the output is LOW (0). Figure 6.1 shows the graphic symbol used for an OR gate.

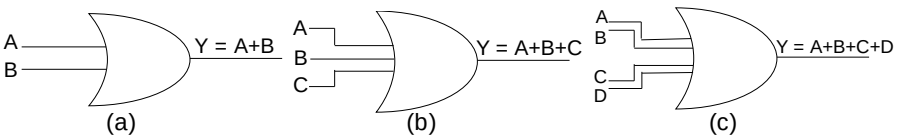


Figure 6.1 : Graphic symbols for (a) two inputs, (b) 3 inputs and (c) 4 inputs or gate

The algebric function for OR gate (2 inputs) is : F = x+y, and the truth table is :

Table 6.1 Truth table of OR gate (2inputs)

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

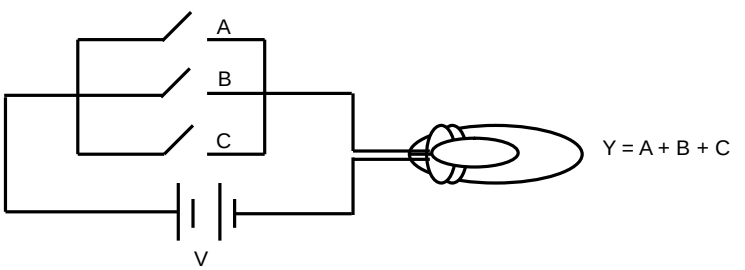


Figure 6.2 : Shows the switching circuit analogy of OR function

The circuit of an OR gate is arranged in such a way that the output is in state 1, when anyone of the inputs is in state 1 ; i.e. when input A or input B or input C is 1 (in case of 3 inputs OR gate). The circuit can be illustrated by the analogy shown in Figure 6.2. The circuit consists of a battery, a lamp and three parallel switches connected in series. Battery switches are the inputs to the lamp and the light from the lamp represents the circuit output.

Let us define an open switch as a 0 state, i.e., no light represents 0 state, and a closed switch represents state 1, i.e., a glowing lamp as a 1 state. We can list the various combinations (8 combinations) of switch states as inputs to the circuit and the resulting output states in a truth table. It is clear from the truth table that all switches must be opened (0 state) for the light to be off (output in 0 state). This type of circuit is called an OR gate.

Table 6.2 : Truth table of 3 inputs OR gate

Inputs			Outputs
A	B	C	Y = A + B + C
0	0	0	0
0	0	1	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

6.3.2 AND Gate

An AND gate also may have two or more inputs and a single output. In order to have a HIGH (1) output, all the inputs of the AND gate must be HIGH (1), otherwise the output is LOW. Figure 6.3 shows graphic symbols used for an AND gate.

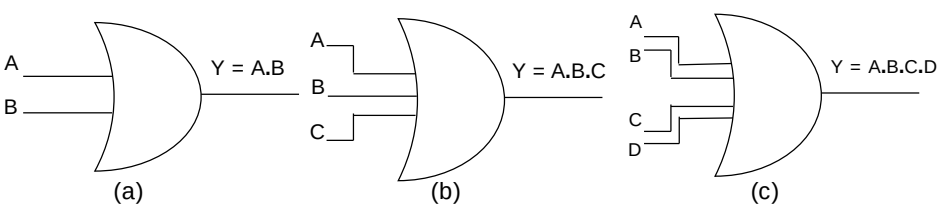


Figure 6.3 : Graphic symbol for (a) two inputs (b) three inputs and (c) four inputs AND gate

The algebric function for a two inputs AND gate is :

Y = A.B

Table 6.3 shows the truth table for this function.

Table 6.3 : Truth table of AND gate

X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

The AND function can be explained by a series switching circuit as shown in Figure 6.4. It has two switches X and Y in series with a bulb and a power supply. The bulb will glow if and only if both the switches X and Y are simultaneously on.

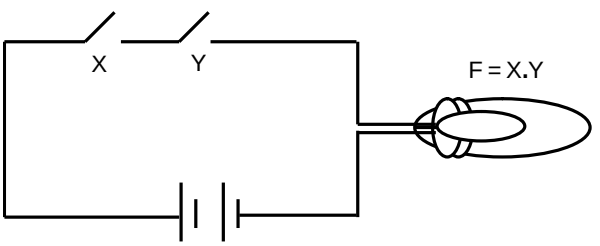


Figure 6.4 : Switching circuit analogy of AND function

In the truth table 6.3,

x = y = 0 represents that the switches are OFF

x = y = 1 represents that the switches are ON

F = 0 represents that the bulb will not glow

F = 1 represents that the bulb will glow.

6.3.3 NOT Gate

NOT gate circuit has a single input and single output. The NOT gate circuit is also called a complementary circuit or an inverter as it complements its input i.e. it accomplishes a logic negation. Figure 6.5 shows the different graphic symbols used for the NOT gate.



Figure 6.5 : Graphic symbols for NOT gate

The algebric function for a NOT gate is : F = x′. Table 6.4 is the

truth table for a NOT gate.

Table 6.4 : Truth table of NOT gate

X	F
0	1
1	0

The NOT gate is understood by the short circuit switch A which when it is closed, (ON) the bulb is bypassed and it does not glow, but when the switch is opened (OFF), the current will flow through the bulb and it would glow. i.e., when A is ON the bulb will be OFF and when A is OFF the lamp will be ON.

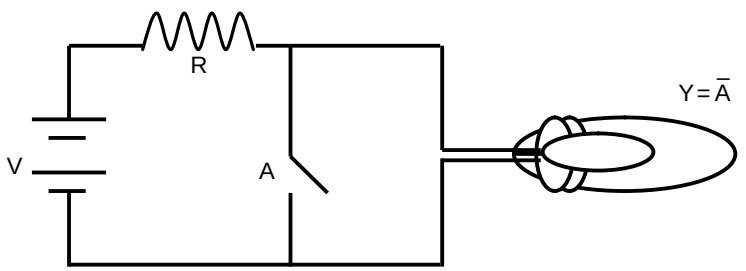


Figure 6.6 : Switching analogy of the NOT function

6.3.4 NAND Gate

A NAND gate is the cascade combination of all AND and a NOT gate. It is an AND gate followed by an inverter. The NAND operation is the complement of the AND operation. Fig. 6.7 shows the graphic symbols for a NAND gate. The algebric function is defined as : F = $(xy)^{/}$. Table 6.5 shows the truth table for NAND operation.

Table 6.5

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

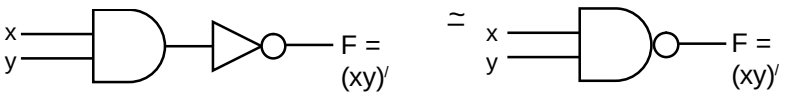


Figure 6.7 : Graphic symbols of NAND gate

6.3.5 NOR Gate

The negation of the OR function is called NOT-OR or NOR. A NOR gate is the cascade combination of NOT and OR gates. The NOR operation is the complement of OR operation. The graphic symbols used normally for a NOR gate are shown in figure 6.8. The NOR function is defined as : $F = (x+y)'$.

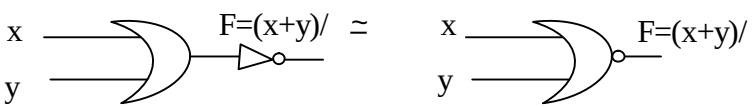


Figure 6.8 : Two Graphic or logic symbols of NOR gate

Table 6.6 : Truth table of NOR gate

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

Note : Any boolean function can be implemented using NAND or NOR gates. So, NAND and NOR gates are called universal gates.

6.3.6 Exclusive - OR (XOR) gate

The exclusive - OR (XOR) gate has a graphic symbol similar to that of the OR gate, except for the additional curved line on the input side. The output of a two input XOR gate is a logic 1, if the input x or input y is a logic 1 exclusively, i.e., they are not 1 simultaneously. The graphic symbol is shown in Figure 6.9 and the XOR function can be written as $F = X \oplus Y = XY' + X'Y$.

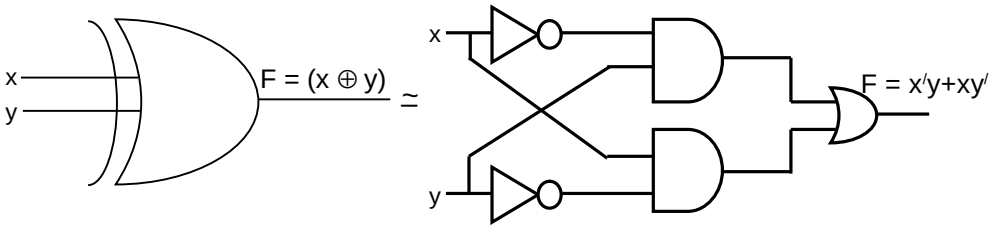


Figure 6.9 : (a) Graphic symbol fo XOR gate(b) XOR gate using basic gates

The truth table of XOR operation is shown in Table 6.7

Table 6.7 : Truth table for XOR operation

X	Y	F = X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

From the truth table, it is clear that the output is 1 (HIGH), when any one of the inputs is at 1 (HIGH). The output is 0 (LOW), when both the inputs are at 1 (HIGH) or at 0 (LOW), i.e. same. In case of more than two inputs, the output of a XOR gate is high when an odd number of inputs is HIGH, such as one or three or five etc. On the otherhand, when there is an even number of HIGH inputs, the output will be always LOW.

6.3.7 XNOR Gate

The XNOR i.e., exclusive NOR gate is the complement of the XOR gate. XNOR function is also called equivalence function. The graphic symbol of XNOR gate is similar to that of the XOR gate, except for the additional inverter gate (or a small bubble) on the output side. The output of a two inputs XNOR gate is a logic 1 (HIGH), if both the inputs are either 1 (HIGH) or 0 (LOW). If the inputs are different (not same), the output is 0 (LOW), In general, we can say the output is 0 (LOW), when the inputs to an XNOR gate have an odd numbers of 1s. The graphic symbol of XNOR gate is shown in Figure 6.10.

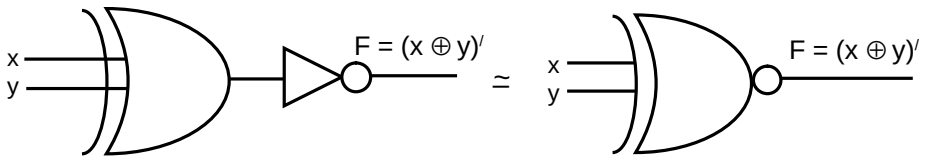


Figure 6.10 : Graphic symbols for XNOR or euivalence gate

The truth table is given in Table 6.8. Here, we see that the output F is the complement of output of the XOR gate. The boolean function for XNOR gate is :

$F = (x \oplus y)' = (x'y + xy')'$

$= (x'y)' (xy')'$ Applying DcMorgan's Theorem

$= (x + y') (x' + y)$

$= xx' + xy + x'y' + yy'$

$= xy + x'y'$

Table 6.8 : Truth table for XNOR gate

Input		Output
x	y	$F = (x \oplus y)'$
0	0	1
0	1	0
1	0	0
1	1	1

6.4 DE MORGAN'S THEOREM

In the previous unit, we have mentioned the De Morgan's Theorem and its proof for two variables also has been explained using truth table. De Morgan, a great mathematician contributed two most important theorems of Boolean algebra. These two theorem are extremely useful in simplifying an expression in which the product of the sum of variables is complemented. The two theorems can be extended to any number of variables and generalized as:

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

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

In words, we can write Theorem 1 as: The complement of an OR sum equals the AND product of the complements, and the theorem 2 as :

The complement of an AND product is equal to the OR sums of the complements.

The complement of any boolean function may be found by means of these theorems. It consists of two simple steps to form a complement of a function.

Step 1 : Interchange the symbols / operands "+" and "*".

Step 2 : Each and every term in the expression is complemented.

Example 6.1 : Find the complements of the following functions using De Morgan's Theorem.

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

(ii) $F = A'B + ABC + AB'C$

Solution : (i) $F = (A + B)' (A' + C') (B' + C)$

$F' = [(A + B)' (A' + C') (B' + C)]'$

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

$= (A + B) + AC + BC'$

$= A + B + AC + BC'$

(ii) $F = A'B + ABC + AB'C$

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

$= (A'B)' (ABC)' (AB'C)'$

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

6.5 TRUTH TABLE

Basic idea of truth table has already been given in the earlier unit. Here, we will learn the process of deriving an expression from a truth table.

The general procedure for obtaining the expression from a truth table in sum of products (SOP) can be summarized as follows:

1 Write an AND term (minterm) for each combination of input variables in the table for which output is 1.

2 Each AND term contains each input variable either in normal form or in complemented form. If the corresponding variable is 0 then it is complemented in the AND term.

3 All the AND terms are then ORed together to produce the final output expression.

Example : 6.2. Obtain the logic function specified by the following truth table. Simplify it using algebric manipulation and implement it with logic diagram.

x	y	z	F	
0	0	0	1 →	$x'y'z'$
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	1 →	$xy'z'$
1	0	1	0	
1	1	0	1 →	xyz'
1	1	1	1 →	xyz

Solution : The resultant boolean bunction is :

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

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

$= y'z' + xy$

This is the simplified function.

The logic diagram is :

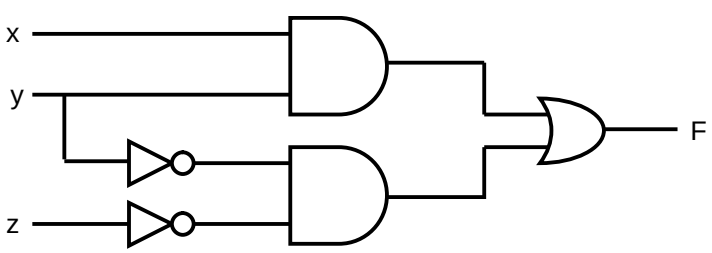


Figure 6.11 : Logic diagram for $F = xy + y'z'$

Note : Since the function has 2 AND terms, we need 2 AND gates and 1 OR gate to implement it.

6.6 CONVERSION OF LOGIC GATES

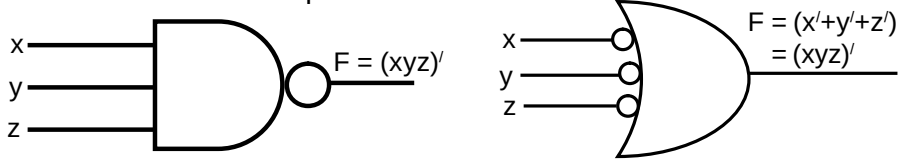
We have already discussed the functions of three basic gates, viz, OR, AND and NOT gates, which perform logical addition, logical multiplication and inversion operations respectively. Besides these, we also have come to know about NAND and NOR gates and their functions. In this section, we will discuss how the digital circuits can be implemented with NAND or NOR

gates.

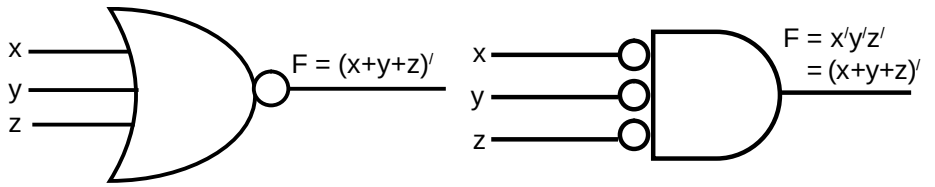
NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. So, digital circuits are more frequently constructed with NAND or NOR gates than with AND and OR gates. Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from boolean functions given in terms of AND, OR and NOT into equivalent NAND or NOR logic diagram. Here, we will consider the procedure for only two level implementation.

First, we will define two other graphic symbols for NAND and NOR gates, which will make the conversion procedure easily understandable. Two equivalent symbols for the NAND gate are shown in Figure 6.12 (a).

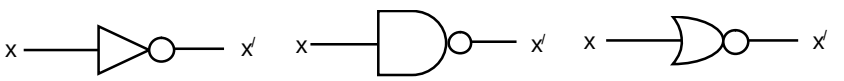
The AND invert symbol has been defined previously. It is possible to represent a NAND gate by an OR graphic symbol preceded by small circles in all the inputs. This symbol i.e. invert - OR symbol for the NAND gate follows from the De Morgan's theorem. and from the convention that the small circle denotes complementation.



(a) Two graphic symbols for NAND gate



(b) Two graphic symbols for NOR gate



(c) Three graphic symbols for inverter

Figure 6.12 : Graphic symbols for NAND, NOR and NOT gates

The invert AND is an alternative that uses De Morgan's theorem and the convention that small circles in the inputs denote complementation.

A one input NOR gate or NAND gate is equivalent to an inverter. So, an inverter gate can be drawn in three ways as shown in Figure 6.12 (c).

NAND Implementation : To implement a boolean function with NAND gates, it is required that the function is to be simplified in the sum of products (SOP) form. We can see the relationship between a sum of products expression and its equivalent NAND implementation, by considering the logic diagrams of Figure 6.13. All three diagrams are equivalent and implement the function : Y = ABC + DE + F

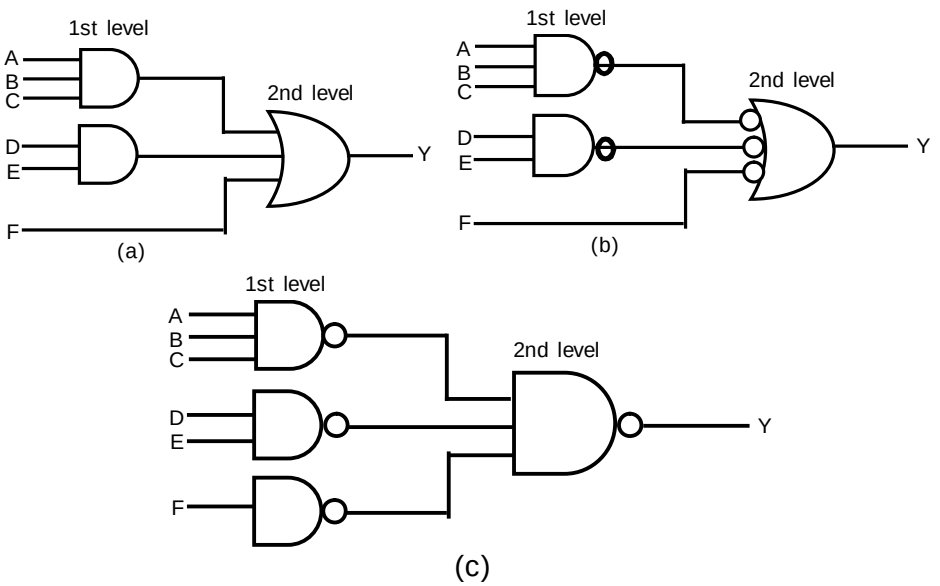


Figure 6.13 : Three ways to implement Y = ABC + DE + F

The function is implemented in SOP form with AND and OR gates in Figure 6.13 (a). The AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an invert OR symbol. The simple variable F is complemented and applied to the second level invert OR gate. A small circle represents complementation. Therefore two circles on the same line represent double complementation and both can be removed. The complement of F goes through a small circle which complements the variable again to produce the normal value of F. Thus, if we remove the small circles in the gates of Figure 6.13 (b), we get Figure 6.13 (a). Therefore, the two diagrams implement the same function and are equivalent.

In Figure 6.13(c),the output NAND gate i.e. the second level NAND

gate is replaced with the conventional symbol. The one input NAND gate complements variable F. The diagram in (c) is equivalent to the one in (b), which is turn is equivalent to the diagram in (a). Thus, we implement the circuit, with NAND gates in Figure 6.13(b) or 6.13 (c), which is first implemented with AND and OR gates in Figure 6.13 (a).

The NAND implementation can also be verified algebrically as:

$Y = [(ABC)' . (DE)' . F']'$

$= [(A' + B' + C') (D' + E') . F']'$

$= ABC + DE + F$

From, the transformation shown in Figure 6.13 we see that a boolean function can be implemented with two levels of NAND gates. The rule for obtaining the NAND logic diagram from a boolean function is as follows:

1. Simplify the function in sum of products (SOP).
2. For each product term of the function that has atleast two literals, draw a NAND gate.
3. Draw a single NAND gate (using the AND invert or INVERT -OR graphic symbol) in the second level, with inputs coming from outputs of first -level gates.
4. A term with a single variable requires an inverter in the first level or may be complemented and applied as an input to the second-level NAND gate.

There is a second way to implement a boolean function with NAND gates. We can combine the 0's in a map to obtain the simplified expression of the complement of the function in sum of products. The complement of the function so obtained can then be implemented with two levels of NAND gates using the above stated rules. To obtain the normal output of the circuit, it is required to insert a one input NAND or inverter gate to generate the true value of the output variable. When the designer wants to generate the complement of the function, the second method in preferred.

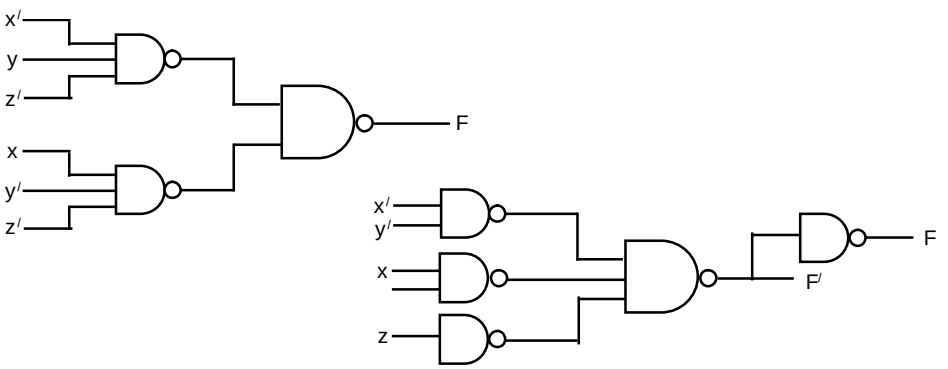
Example 6.3 : Implement the following function with NAND gates F (x, y, z) = $\sum(2,4)$.

The first step is to simplify the function in sum of products form. We draw a map and plot the terms (Figure 6.14).

There are only two 1's in the map, and they cannot be combined, since they are not adjacent to each other. The simplified form of the function in SOP is $F = x'yz' + xy'z'$

x\yz	00	01	11	10
0	0	0	0	1
1	1	0	0	0

(a) Map simplification in SOP $F = x'yz' + xy'z'$



(b) $F = x'yz' + xy'z'$ (c) $F' = x'y' + xy + z$

Figure 6.14 : Implementation of Function in eg. 6.3 with NAND gates

The two level NAND implementation is shown in Figure 6.14 (a) Now, we try to simplify the complement of the function in SOP. This is done by combining to 0's in the map: Thus , $F' = x'y' + xy + z$

The two-level NAND gate for generating F' is shown in Figure 6.14 (c). If output F is required, we have to add a one input NAND gate to invert the function. (We assume that the input variables are available in both the normal and complement forms).

NOR Implementation : The NOR function is the dual of the NAND function. So, all procedures and rules for NOR logic are the dual of the corresponding procedures and rules developed for NAND logic.

The implementation of a boolean function with NOR gates requires that the function be simplified in product of sums (POS) form. A product of sums expression specifies a group of OR gates for the sum terms, followed

by an AND gate to produce the product. The transformation from the OR-AND to the NOR-NOR diagram is shown in Figure 6.15, which is similar to the NAND transformation discussed already, except that here we use the product of sums expression :

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

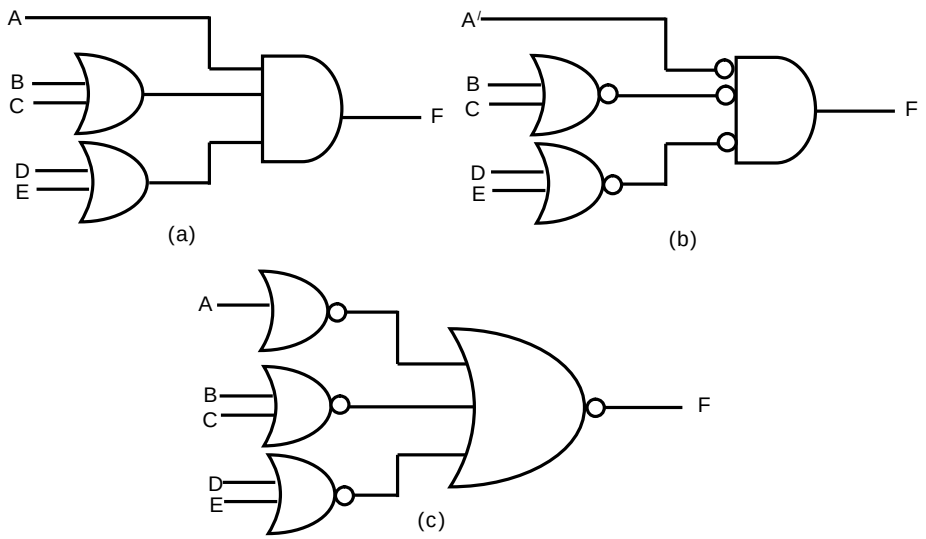


Figure 6.15 : Implementation of the function F = A(B + C) (D + E)

The procedure for obtaining the NOR logic diagram from a boolean function can be derived from this transformation. It is similar to the three step NAND rule, except that the simplified expression must be in the product of sums and the terms for the first level NOR gates are the sum terms. A term with a single variable requires a one input NOR or inverter gate or may be complemented and directly applied to the second-level NOR gate.

Another way to implement a function with NOR gate is to use the expression for the complement of the function in product of sums. It gives a two level implementation for $F^{/}$ and a three level implementation gives the normal output F.

Simplified product of sums can be obtained from a table by combining the 0's and then complementing the function. To obtain the simplified product of sums expression for the complement of the function, we have to combine the 1's in the map and then complement the function. The NOR gate implementation procedure is demonstrated in the following example.

Example 6.4 : Implement the function of e.g. 6.3 with NOR gates.

The map for this function is drawn in Figure 6.14 (a) Now, combining the 0's we obtain $F' = x'y' + xy + z$

This is the complement of the function in SOP. To obtain it in POS form, as required for NOR implementation, complement F'.

$$(F')' = F = (x + y)(x' + y')z'$$

The two level implementation with NOR gate is shown in Fig. 6.16.

Another implementation that is possible from the complement of the function in POS is left as an exercise.

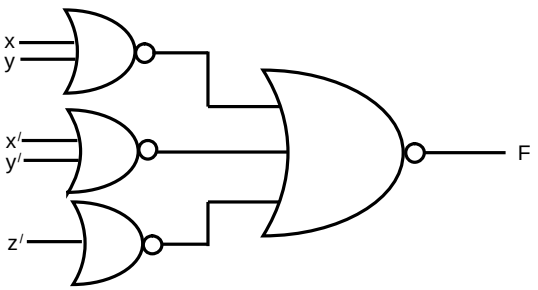
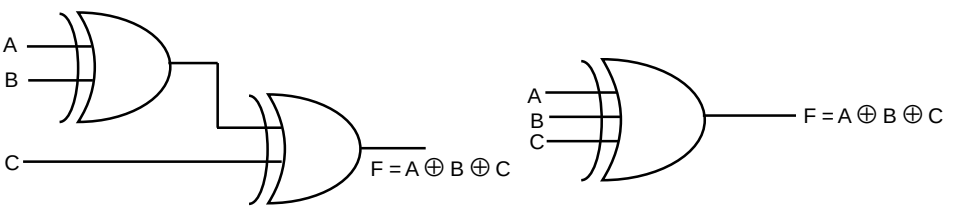


Fig. 6.16 : Implementation of function with NOR gates

Example 6.5 : Draw the logic diagram for the function $F = A \oplus B \oplus C$



(a) using 2-input gates (b) using 3-input gate



CHECK YOUR PROGRESS

Q.1. When the output of an OR gate is High?

Q.2. When the output of an AND gate is HIGH?

Q.3. What is the function of a NOT gate?

Q.4. When the output of a XOR gate is HIGH?

Q.5. Why the digital circuits are more frequently constructed with NAND or NOR gates?



6.7 LET US SUM UP

- A logic gate is an electronic circuit that is used to implement a booleam function by a logic diagram.
- An OR gate produces a HIGH output when atleast one input is HIGH; whereas an AND gate produces a HIGH output when all inputs are HIGH.
- A NAND gate is an AND gate followed by an inverter. It produces a LOW output if all its inputs are HIGH.
- A NOR gate is an OR gate followed by an inverter. It produces a HIGH output when all its inputs are LOW.
- The realization of basic qater viz., AND, OR, and NOT can be made by using either NAND or NOR gates. For this reason, NAND and NOR gates are called universal gates.



6.8 FURTHER READING

- Mano, Morris (2007), *Digital Logic and Computer Design*. Pearson Education.
- Kumar, A. Anand. (2003), *Fundamental of Digital Circuit*, New Delhi, PHI.



6.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : If atleast one of the inputs is HIGH.

Ans. to Q. No. 2 : If all the inputs to it are HIGH.

Ans. to Q. No. 3 : The function of the NOT gate is to invert / complement its single line input variable or function.

Ans. to Q. No. 4 : If both the input variables to a XOR gate is not equal then the output is HIGH.

Ans. to Q. No. 5 : Because, NAND and NOR gates are easier to fabricate.



6.10 MODEL QUESTIONS

Q.1. What is a logic gate?

Q.2. List the three basic logic operations.

Q.3. What are the positive and negative logic?

Q.4. What is the only set of input conditions that will produce a LOW output for an OR gate?

Q.5. Write a truth table for a 3 input OR gate?

Q.6. Write a Boolean expression for a 4 input AND gate

Q.7. What is the only input combination that will produce a HIGH at the output of a 4 input AND gate?

Q.8. What input/ logic level should be applied to the second input of a 2-input AND gate to inhibit the logic signal at the first input from reaching output?

Q.9. Develop a truth table for a 3- input AND gate.

Q.10. Is there any difference between 1 OR 1 and 1+1 (binary addition) ?

Q.11. Name the logic gate which has only one input, and show the logic / graphic sym bol.

Q.12. What are the NAND and NOR gates?

Q.13. Write the logic symbols of NAND /NOR gates and develop its truth table.

Q.14. What are the universal gates?

Q.15. Draw the logic diagram of OR gate using NOR /NAND gate.

Q.16. Draw the logic diagram of AND gate using NOR / NAND gate.

Q.17. Draw the logic diagram of NOT gate using NOR / NAND gate

Q.18. What is an XOR gate ? Write its truth table for 2 variables.

Q.19. Show the logic diagram of an XOR gate using basic gates.

Q.20. Draw the logic diagram of an XOR gate using NAND gates.

Q.21. What is an XNOR gate ? Write its truth table.

Q.22. Draw the logic diagram of an XNOR gate using basic gates.

Q.23. Draw the symbol of an XNOR gate and its Boolean expression

Q.24. Implement the following functions with (i) NAND gates (ii) NOR gates.

Q.25. Write the procedure for obtaing NAND logic implementation of a Boolean function.

Q.26. Write the procedure for obtaing NOR logic implementation of a Boolean function.

Q.27. Explain the operation of 3-input AND / OR gate and realize it using NAND /NOR gates

Q.28. Explain the operation of 2-input XOR gate and realize it using NAND /NOR gates

Q.29. Explain the operation of 2-input XNOR gate and realize it using NAND /NOR gates.

****

UNIT 7 : FLOATING POINT NUMBER REPRESENTATION

UNIT STRUCTURE

7.1 Learning Objectives

7.2 Introduction

7.3 Floating Point Number

7.4 Normalization of Floating Point Number

7.5 Overflow and Underflow

7.6 IEEE standard for Floating Point Representation

7.7 Floating-Point Arithmetic

- 7.7.1 Addition and Subtraction
- 7.7.2 Multiplication
- 7.7.3 Division

7.8 Let Us Sum Up

7.9 Further Reading

7.10 Answers to Check Your Progress

7.11 Model Questions

7.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define floating point number
- describe the method to normalize floating point number
- understand overflow and underflow
- know the accepted standard of floating point presentation
- become familiar with floating point arithmetic

7.2 INTRODUCTION

In the previous units, we have learnt about the different types of code conversion methods and logic gates. Different types of logic gates have been discussed in detail in the previous units. In this unit, we describe floating point number representation. We will also discuss how floating point number is normalized. Further, we shall describe what are overflow and underflow. The IEEE standard for floating point number representation shall also be described. We will also discuss floating point arithmetic in this unit.

7.3 FLOATING POINT NUMBER

The floating point number system is used to represent large fraction of numbers for scientific and computational purposes using two segments namely mania and exponent. Numbers that are too large for standard integer representations or that has fractional components are usually represented in scientific notation using floating point numbers. Thus, 875,000,000,000,000 can be represented as 8.75×10^{14} and 0.0000000000000875 can be represented as 8.75×10^{-14}. Here the decimal point is shifted to a convenient location and the exponent of 10 is used to keep track of that decimal point. This allows a range of very large and very small numbers to be represented with only a few digits. This same approach can be used for binary numbers. The floating point number system, based on scientific notation, is capable of representing very large (1.33×10^88) and very small (1.33×10^-88) numbers. A floating-point number can be represented in the form $\pm F\times B^{\pm E}$

This number can be stored in a binary word with three fields as shown below:

- Sign (plus or minus)
- Fraction F (also called mantissa or significand)
- Exponent E

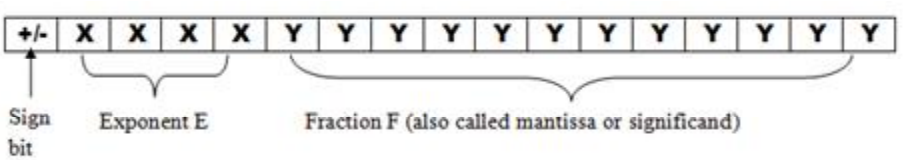


Figure 7.1: Floating point number representation

Figure 7.1 shows floating point number representation using mantissa and exponent. The base B is implicit and need not be stored because it is the same for all numbers. Decimal numbers use base 10 ($F\times10^E$) while binary numbers use base 2 ($F\times2^E$). The leftmost bit stores the sign of the number ('0' for positive numbers and '1' for negative numbers). The exponent value is stored in the next 'k' bits. The representation used is known as a biased

representation. A fixed value, called bias, is subtracted from the exponent field to get the true exponent value. Typically, the bias is equals to, $(2^{k-1}-1)$ where 'k' is the number of bits in the binary exponent. The 8-bit exponent field yields the numbers from 0 to 255. With a bias of 127 i.e. (2^7-1), the true exponent values are in the range -127 to+128. The significand is composed of an implicit leading bit (before the radix point) and fractional bits (after the radix point). The leading bit for normalized numbers is 1 and for the denormalized numbers is 0. Modern computer architecture adopt IEEE 754 standard for representing floating-point numbers. The IEEE standard defines both 32-bit single -precision and 64-bit double-precision format with 8-bit and 11-bit exponents, respectively. The implied base is 2.

7.4 NORMALIZATION OF FLOATING POINT NUMBER

A normalized number is defined to have the most significant digit of the significand, a nonzero value. Any floating-point number can be expressed in many ways. For example, the following are equivalent, where the significand is expressed in binary form: 0.110×2^6, 110×2^3, 0.0110×2^7. To simplify operations on floating-point numbers, it is required that they be normalized.

For example,

- $+(1.23356789)_{10}\times10^1$: Normalized decimal floating point number
- $-(9.97865231)_{10}\times10^{12}$: Normalized decimal floating point number
- $(23.2)_{10}\times10^4$: Unnormalized decimal floating point number
- $(101.011)_2\times2^3$: Unnormalized binary floating point number
- $(1.01011)_2\times2^5$: Normalized binary floating point number

Therefore, for base 2 representation, a normalized number is one in which the most significant bit (MSB) of the significand is one. Thus, a normalized nonzero number is one in the form $\pm1.F\times2^{\pm E}$. Since the most significant bit is always one, it is not necessary to store this bit. This is an implicit bit hidden in the significand. Thus, the 23-bit significand field (in 32-bit single-precision representation) is used to store a 24-bit value. Given a number

that is not normalized, it may be normalized by shifting the radix point to the right of the most significant binary digit (bit 1) and adjusting the exponent accordingly.

Example 7.1: Represent each of the following decimal number using the 8-bit floating-point format with 3 bits for the mantissa and 4 bits for the exponent.

a) $(5.5)_{10}$

b) $-(96)_{10}$

Solution: a) $(5.5)_{10}$

Steps : 1.Convert the decimal number to binary: $(101.1)_2=(1.011)_2\times2^2$

2. Store the binary number in the following binary word with three fields:

Sign (1 bit)	**Exponent** (4 bits)	**Significand** (3 bits)

Sign bit is equals to 0 since the number is positive. The MSB of the number $(1.011)_2\times2^2$ will not occupy a bit position because it is always a 1. Therefore, the significand is the fractional number 011. Here, exponent field is of 4 bits. So, bias is equal to 7 (2^3-1). Add 7 to the exponent to get the true exponent value. True exponent value (biased exponent) is $2+7=(9)_{10}=(1001)_2$. Therefore, the complete floating point number is:

0	**1001**	**011**

b) - $(96)_{10}$

Steps: 1.Convert the decimal number to binary: $(-1100000)_2= (1.100000)_2\times2^6$

2. Store the binary number in the following binary word with three fields:

Sign (1 bit)	**Exponent** (4 bits)	**Significand** (3 bits)

Sign bit is equals to 1 since the number is negative. The MSB of the number $(1.100000)_2\times2^6$ will not occupy a bit position because it is always a 1. Therefore, the significand is the fractional number 100000. The required three significand bits are the first three bits i.e. 100. Here, significand needs to be rounded off to the nearest eight bits to fit in the allowed range of the given representation. Here, exponent field is of 4 bits. So, bias is equal to 7 (2^3-1). Add 7 to the exponent to get the true exponent value. True exponent value (biased exponent) is $6+7=(13)_{10}=(1101)_2$. Therefore, the complete

floating point number is :

1	1101	100

Example 7. 2: Consider the 8-bit floating-point representation with 3 bits for the mantissa and 4 bits for the excess-7 (bias value is 7) exponent. What decimal number does the bit pattern 11101100 represent?

Solution:

1. The number is negative, since the sign bit is 1.
2. The exponent bits represent $(1101)_2 = (13)_{10}$. This is 7 more than the actual exponent, and so the actual exponent must be 13-7=6. Significand is 100. Thus, in binary scientific notation, we have - $(1.100)_2 \times 2^6$.
3. Convert this to binary number: - $(1.100)_2 \times 2^6$= - $(1100000)_2$.
4. Convert the binary number into decimal number: $(1100000)_2$= - $(96)_{10}$.
5. Therefore, the resultant decimal number is - $(96)_{10}$.

The 8-bit floating-point format can represent a wide range of both small numbers and large numbers. The smallest possible positive number that can be represented in 8-bit floating point format with 4 bit exponent and 3 bit significand is :

0 (sign bit)	0000 (Exponent bits)	000 (significand bits)

This gives 00000000 which represents,

$(1.000)_2 \times 2^0$-7= $(1.000)_2 \times 2^{-7}$. The decimal equivalent of this number is (0.0078)10.

The largest possible positive number that can be represented in this 8-bit floating point format is :

0 (sign bit)	1111 (Exponent bits)	111 (significand bits)

This gives 01111111 which represents,

$(1.111)_2 \times 2^{15-7}$= $(1.111)_2 \times 2^8$. The decimal equivalent of this number is $(480)_{10}$.

Thus, the 8-bit floating-point format (with 4 exponent bits and 3 significand

bits) can represent positive numbers from about $(0.0078)_{10}$ to $(480)_{10}$. In contrast, the 8-bit 2's-complement representation can only represent positive numbers in the range from 1 to 127. But the floating-point representation cannot represent all the numbers in this range, since 8 bits can represent only 2^8(256) distinct values, and there are infinitely many real numbers in the range to represent (even within a small range of 0.0 to 0.1). If the number of bits in the exponent is increased, the range of expressible numbers will get expanded. But because only a fixed number of different values can be expressed, the density of those numbers will be reduced and therefore the precision. The only way to increase both range and precision is to use more bits. Thus, most computers offer 32-bit single-precision numbers and 64 bit double-precision numbers.



CHECK YOUR PROGRESS

1. The decimal numbers represented in the computer are called as floating point numbers, as the decimal point floats through the number.
   (a) True (c) cannot say anything
   (b) False (d) none of the above
2. If the decimal point is placed to the right of the first significant digit, then the number is called as
   (a) Orthogonal (c) Determinate
   (b) Normalized (d) none of the above
3. _____ constitute the representation of the floating number.
   (a) Sign (c) Significant digits
   (b) Scale factor (d) All of the above
4. The sign followed by the string of digits is called as _____
   (a) Significant (c) Mantissa
   (b) Determinant (d) Exponent

7.5 OVERFLOW AND UNDERFLOW

An AND gate may have two or more inputs and a single output. In order to have a HIGH (1) output, all the inputs of the AND gate must be HIGH (1), otherwise the output is LOW. Figure 6.3 in unit 6 shows graphic symbols used to represent an AND gate.

Overflow occurs when an arithmetic operation results in a magnitude greater than the one that can be expressed in the given floating point notation. Underflow occurs if the number is not zero but too small to be represented in the given floating point representation. For example, Figure 7.2 shows the range of numbers that can be represented in a 32-bit word.

- Negative numbers can be represented between-$(2-2^{-23}) \times 2^{128}$ and -2^{-127}.
- Positive numbers can be represented between 2^{-127} and $(2-2^{-23}) \times 2^{128}$.

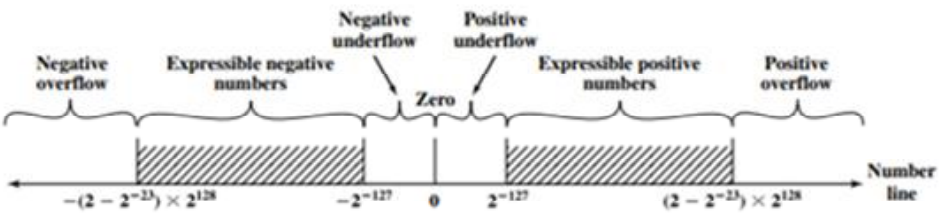


Figure 7.2: Expressible Numbers in Typical 32-Bit Formats

So, in this case overflow will occur if the magnitude of the result of an arithmetic operation is greater than the number that can be expressed with an exponent of 128 e.g. $2^{120} \times 2^{100} = 2^{220}$. Underflow will occur if the fractional magnitude is too small e.g. $2^{-120} \times 2^{-100} = 2^{-220}$.

7.6 IEEE STANDARD FOR FLOATING POINT REPRESENTATION

Floating point numbers are an important data type used in almost all computer hardware. The most important floating-point representation is defined in IEEE Standard 754, was adopted in 1985 by all computer manufacturers. This standard defines both a 32-bit single precision format and a 64-bit double precision format for representing a floating point number. The implied base is 2.

32-bits Double Precision IEEE 754 Standard

In 32-bit single-precision floating-point representation:

- The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.
- The following 8 bits represent exponent (E).
- The remaining 23 bits represents fraction (F).

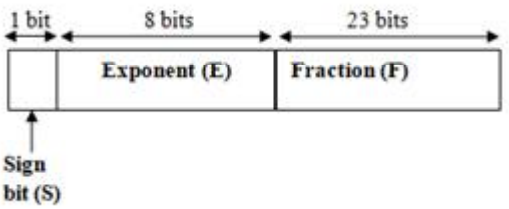


Figure 7.3: IEEE 754 representation of 32 bits floating point number

Figure 7.3 shows a typical 32-bit floating-point format. Exponent (E) can be both positive and negative numbers. So, instead of using a separate sign bit for the exponent, the standard uses a biased representation. Here, since the exponent field is of 8 bits, it can have numbers from 0 to 255. A bias value of 127 (2^7-1) is added to the true exponent to be stored in the exponent field. The exponent is biased, so that the range of exponents is from -126 to +127. So for exponent values in the range of 1 to 254, the value N = $\pm(1.F)\times 2^{E-127}$ or $(-1)^s \times (1.F)\times 2^{E-127}$ is the normalized nonzero floating-point number.

To increase the precision of the significand, the IEEE 754 standard uses a normalized significand. The MSB of the significand is always one and it is not necessary to store this bit. This is an implicit bit hidden in the significand. Thus in the single precision IEEE Standard, the significand field is 24 bits long of which 23 bits of the significand is stored in the memory and the MSB is an implied 1. The extra bit increases the number of significant digits in the significand.

For exponent (E) =0, the numbers are in the de-normalized form. Unlike normalized form in the de-normalized form an implicit leading 0 is used for the significand and the actual exponent is always -126. For E=0, the value N = $(-1)^s (0.F)\times 2^{-126}$. Hence, de-normalized form can be used to

represent the number zero with E=0 and F=0. An exponent of all ones (i.e. E=255) together with a fraction (significand) of zero represents positive or negative infinity, depending on the sign bit. Magnitude of numbers that can be represented in this 32-bits single precision format is approximately in the range of 1.8×10^{-38} to 3.40×10^{38}. An exponent of all ones (E=255) together with a nonzero fraction gives the value NaN, which means Not a Number. When an operation is performed by a computer on a pair of operands, the result may not be mathematically defined. For example, if zero is divided by zero, the result is indeterminate. Such a result is called Not a Number (NaN) in the IEEE Standard.

Example 7. 3: Convert the decimal number $(0.75)_{10}$ to a 32-bits single precision floating point binary number.

Solution: Given decimal number is $(0.75)_{10}$

Convert the decimal number to binary:

$(0.11)_2 = (1.1)_2 \times 2^{-1}$

Sign bit is equals to 0 since the number is positive. The MSB of the number $(1.1)_2 \times 2^{-1}$ will not occupy a bit position because it is always a 1 in normalized number. Therefore, the significand is the fractional 23-bits binary number 10000000000000000000000 and the biased exponent is:

$-1+127= (126)_{10} = (01111110)_2$

Therefore, the complete floating point number is

0	01111110	10000000000000000000000

Example 7. 4: Determine the decimal value of the following floating point binary number:

1 10010001 10001110001000000000000

Solution:

1. The number is negative, since the sign bit is 1.
2. The exponent bits represent $(10010001)_2 = (145)_{10}$. This is 127 more than the actual exponent, and so the actual exponent must be 145-127=18. Significand is 10001110001000000000000.

3. The general approach to determine the value of a normalised floating point number is expressed by the formula: $(-1)^{S}\times(1.F)\times2^{E-127}$
   $= (-1)^{1}\times(1.10001110001)\times2^{145-127} = (-1)^{1}\times(1.10001110001)\times2^{18}$
4. Convert this to binary number: - $(1.1001110001)_2\times2^{18}$= - $(1100011100010000000)_2$.
5. Convert the binary number into decimal number: - $(1100011100010000000)_2$= - $(407680)_{10}$.

64-bits Double Precision IEEE 754 Standard

In 64-bit double-precision floating-point representation:

- The most significant bit is the sign bit (S), with 0 for positive numbers and 1 for negative numbers.
- The following 11 bits represent exponent (E).
- The remaining 52 bits represents fraction (F).

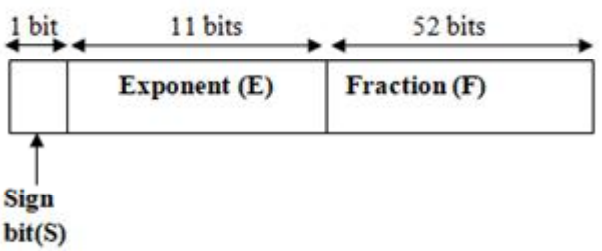


Figure 7.4: IEEE 754 representation of 64 bits floating point number

Figure 7.4 shows a typical 64-bit floating-point format. Similar to 32-bits representation, in this format also Exponent (E) can be both positive and negative numbers. So, instead of using a separate sign bit for the exponent, the standard uses a biased representation. Here, since the exponent field is of 11 bits, it can have numbers from 0 to 2048. A bias value of 1023 (2^{10}-1) is added to the true exponent to be stored in the exponent field. The exponent is biased, so that the range of exponents is from -1022 to +1023. So for exponent values in the range of 1 to 2046, the value N = $\pm(1.F)\times2^{E-1023}$ or $(-1)^{S}\times(1.F)\times2^{E-1023}$ is the normalized nonzero floating-point number. Here, the significand field is 53 bits long of which 52 bits of the significand is

stored in the memory and the MSB is an implied 1. The extra bit increases the number of significant digits in the significand.

For exponent (E) =0, the numbers are in the de-normalized form. Unlike normalized form in the de-normalized form an implicit leading 0 is used for the fraction and the actual exponent is always -1022. For E=0, the value N = $(-1)^S \times (0.F) \times 2^{-1022}$. Hence, de-normalized form can be used to represent the number zero with E=0 and F=0. An exponent of all ones (i.e. E=2047) together with a fraction of zero represents positive or negative infinity, depending on the sign bit. Magnitude of numbers that can be represented in this 64-bits double precision format is approximately in the range of 2.23×10^{-308} to 1.8×10^{308}. An exponent of all ones (E=2047) together with a nonzero fraction gives the value NaN, which means Not a Number.

7.7 FLOATING-POINT ARITHMETIC

For addition and subtraction, both the operands need to have the same exponent value. So, shifting of the radix point on one of the operands may be required to achieve proper alignment. Table 7.1 summarizes the basic operations for floating-point arithmetic. A floating-point operation may produce one of the following conditions:

(a) If a positive exponent exceeds the maximum possible exponent value, Exponent overflow occurs.

(b) If a negative exponent is less than the minimum possible exponent value then exponent underflow occurs. This means that the number is too small to be represented, and it may be expressed as zero.

(c) In the process of aligning the significands, digits may get shifted to the right end of the significand and some form of rounding operation may require. Significand underflow occurs in this situation.

(d) The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment and significand overflow may occur.

Table 7.1: Arithmetic Operations in floating point numbers

Floating Point Numbers	Arithmetic Operations
$X=X_s \times B^{Xe}$ $Y=Y_s \times B^{Ye}$ Where, X_s and Y_s are the significand of X and Y respectively X_e and Y_e are the exponent of X and Y respectively B= base	$X+Y=(X_s \times B^{X_e-Y_e}+Y_s)\times B^{Y_e}$ if $X_e \le Y_e$
	$X-Y=(X_s \times B^{X_e-Y_e}-Y_s)\times B^{Y_e}$ if $X_e \le Y_e$
	$X \times Y=(X_s \times Y_s)\times B^{X_e+Y_e}$
	$\frac{X}{Y}=\left(\frac{X_s}{Y_s}\right)\times B^{X_e-Y_e}$

7.7.1 Addition and Subtraction

Addition and subtraction are identical operation except for a sign change. So, in subtraction operation the sign of the subtrahend is changed.

There are four basic steps of the algorithm for addition and subtraction:

1. **Check for zero:** If either operand is 0, the other operand is expressed as the result.
2. **Align the significands:** The alignment is achieved by repeatedly shifting the magnitude portion of the significand of the smaller number to the right by 1 digit and incrementing the exponent until the two exponents are equal. If this process results in a 0 value for the significand, then the other number is expressed as the result.
3. **Add or subtract the significands:** The two significands are added together, taking into account their signs.
4. **Normalization of result:** If result of the operation is not in normalized form, normalization is done by left shifting the significand until MSB is nonzero and decrementing its exponent value or by right shifting the result and incrementing its exponent value. It may cause exponent underflow if exponent value is smaller than minimum exponent allowed. Exponent overflow may occur if its value is larger than maximum exponent allowed in the representation. Finally, the result must be rounded off.

Here, negative significands are first converted to 2's complement form and then the addition is performed. The result is converted back to sign-magnitude form. When adding numbers of opposite sign, cancellation may occur, resulting in an arbitrarily small sum or even zero if the numbers are equal in magnitude. Normalization in this case may require shifting by the total number of bits in the significand results in a large loss of accuracy.

Example 7. 5: Convert the decimal number $(-0.75)_{10}$ to a 64-bits double precision floating point binary number.

Solution: Given decimal number is $(-0.75)_{10}$

Converting the decimal number to binary:

$(-0.11)_2 = -(1.1)_2 \times 2^{-1}$

Sign bit is equals to 1 since the number is negative. The MSB of the number $-(1.1)_2 \times 2^{-1}$ will not occupy a bit position because it is always a 1 in normalized number. Therefore, the significand is the fractional 52-bits binary number 1000 0000 0000 0000 0000 000.....0 and the biased exponent is:

$-1+1023 = (1022)_{10} = (01111111110)_2$

Therefore, the complete floating point number is

1	**01111111110**	**10000000000000000000000.....0**



CHECK YOUR PROGRESS

5. In IEEE 32-bit representations, the mantissa of the fraction is said to occupy _____ bits.

   (a) 24 (b) 23

   (c) 20 (d) 16

6. In double precision format the size of the mantissa is _____ bits.

   (a)32 (b) 52

   (c) 64 (d) 72

7. A machine stores floating point numbers in 7-bit word. The first bit is used for the sign of the number, the next three for the biased exponent

and the next three for the magnitude of the mantissa. The number $(0010110)_2$ represented in base-10 is

(a) 0.375 (b) 0.875

(c) 1.5 (d) 3.5

8. A machine stores floating point numbers in 7-bit words. The first bit is stored for the sign of the number, the next three for the biased exponent and the next three for the magnitude of the mantissa. You are asked to represent 33.35 in the above word. The error you will get in this case would be

(a) Underflow (b) Overflow

(c) NaN (d) No error will be registered.

Example 7.6: Convert the following decimal number into IEEE 754 single precision format. Perform floating-point addition. Show the results in normalized form X = $(2345.125)_{10}$ and Y= $(0 .75)_{10}$.

Solution: 32- bit single precision representation of X and Y is obtained using the method discussed in section 7.4.1.

X = $(2345.125)_{10}$ is represented as:

0 (sign bit)	**10001010 (Exponent bits)**	**00100101001001000000000 (significand bits)**

Y= $(0 .75)_{10}$ is represented as:

0 (sign bit)	**01111110 (Exponent bits)**	**10000000000000000000 (significand bits)**

Addition operation is done using the formula shown in Table 7.1, i.e. $X+Y=(X_s \times B^{X_e - Y_e} + Y_s) \times B^{Y_e}$

Steps:

1. Here, X = $(2345.125)_{10}$= $(1.00100101001001)_2 \times 2^{11}$ and Y= $(0 .75)_{10}$ = $(1.1)_2 \times 2^{-1}$

Xe= 10001010, Xs= 00100101001001000000000, Ye= 01111110 and Ys= 10000000000000000000

So since Ye $\leq$ Xe , Y is a smaller number. Shifting the significand of Y until its exponent matches the exponent of X :

$Ys \times 2^{11}$ = 0.00000000000110000000000

2. Adding the significand of X and Y:

   Xs + Ys× 2^{11} = 1.00100101001001000000000 + 0.00000000000110000000000

   = 1. 00100101001111000000000

3. So, the result of addition is already a nonzero normalized number.
4. Therefore, X+Y is equal to :

0	10001010	00100101001111000000000
(sign bit)	**(Exponent bits)**	**(significand bits)**

7.7.2 Multiplication

Floating-point multiplication algorithm is as follows:

1. If either operand is zero, result of the operation is expressed as zero. The next step is to add the exponents.
2. If the exponents are stored in biased form, the exponent sum would have doubled the bias value. Thus, the bias value must be subtracted from the sum. The result could be either an exponent overflow or underflow, which is marked by ending the algorithm.
3. If the exponent of the product is within the proper range, the next step is to multiply the significands, taking into account their signs. The multiplication is performed in the same way as for integers.
4. The result of the multiplication will be double the length of the multiplier and multiplicand. The extra bits will be truncated.
5. The result is then normalized and rounded off like in addition and subtraction operations. The normalization could result in exponent underflow.

Example 7. 7: Convert the following decimal number into IEEE 754 single precision format. Perform floating-point multiplication. Show the results in normalized form X = $(-18)_{10}$ and Y= $(9.5)_{10}$.

Solution: 32- bit single precision representation of X and Y is obtained using the method discussed in section 7.4.1.

$X = (-18)_{10}$ is represented as:

1 **(sign bit)**	**10000011** **(Exponent bits)**	**00100000000000000000000** **(significand bits)**

$Y= (9.5)_{10}$ is represented as:

0 **(sign bit)**	**10000010** **(Exponent bits)**	**00110000000000000000000** **(significand bits)**

Multiplication operation is done using the formula shown in Table 7.1, i.e.

$$X \times Y = (X_s \times Y_s) \times B^{X_e + Y_e}$$

Steps:

1. Here, $X = (-18)_{10} = (-1.0010)_2 \times 2^4$ and $Y = (9.5)_{10} = (1.0011)_2 \times 2^3$
   Xe=10000011, Xs= 00100000000000000000000,
   Ye= 10000010 and
   Ys= 00110000000000000000000
   Sign of X is negative and Y is positive, so sign of the product will be negative.
2. Multiply the significand of X and Y: The product of the 24 bits significands will be of 48 bits with two bits to the left of the binary point: (01).0101011000000….000000
   So, truncate the product to 24 bits: (1).01010110000000000000000
3. Compute exponent of the result:
   Xe + Ye - 12710 = 1000 0011 + 1000 0010 - 0111111 = 1000 0110
4. So, the result of multiplication is already a nonzero normalized number.
5. Therefore, X×Y is equal to :

1 **(sign bit)**	10000110 **(Exponent bits)**	01010101100000000000000 **(significand bits)**

7.7.3. Division

Floating-point division algorithm is as follows:

1. If the divisor is zero, the result is set to infinity. A dividend of zero results in zero. If both divisor and dividend is zero, (i.e. 0/0) the result is NaN.
2. The divisor exponent is subtracted from the dividend exponent to remove the bias, which must be added back in the exponent of the result.
3. Check result exponent for overflow or underflow. Overflow occurs if result exponent is larger than maximum exponent allowed and underflow occurs if result exponent is smaller than minimum exponent allowed in the given representation.
4. Divide the significand. The result of division will be double the length of the divisor and dividend. The extra bits will be truncated.
5. The result is then normalized and rounded off.



7.8 LET US SUM UP

- The floating point number system is used to represent large fraction numbers for scientific and computational purposes using two segments namely manias and exponent.
- A normalized number is defined to have the most significant digit of the significand a nonzero value.
- Overflow occurs when an arithmetic operation results in a magnitude greater than the one that can be expressed in the given floating point notation.
- Underflow occurs if the number is not zero but too small to be represented in the given floating point representation.
- Floating point numbers are an important data type used in almost all computer hardware.
- The most important floating-point representation is defined in IEEE Standard 754, was adopted in1985 by all computer manufacturers.

- To increase the precision of the significand, the IEEE 754 Standard uses a normalized significand.
- Addition and subtraction are identical operation except for a sign change. So, in subtraction operation the sign of the subtrahend is changed.



7.9 FURTHER READING

- Overtorn M., (2001), Numerical Computing with IEEE Floating Point Anithmetic.
- Mano, Moorris. (2007), Digital Logic and Computer Design. Pearson Education.



7.10 ANSWERS TO CHECK YOUR PROGRESS

Ans 1: (a)

Ans 2: (b)

Ans 3: (d)

Ans 4: (c)

Ans 5: (b)

Ans 6: (b)

Ans 7: (b)

Steps:

1. The number is positive, since the sign bit is 0.
2. The exponent bits represent $(010)_2 = (2)_{10}$. This is 3 more than the actual exponent, and so the actual exponent must be $2^{-3}= -1$. Significand is 110.Thus, in binary scientific notation, we have $(1.110)_2 \times 2^{-1}$.
3. Convert this to binary number: $(1.110)_2 \times 2^{-1} = (0.1110)_2$.
4. Convert the binary number into decimal number: $(0.1110)_2 = (0.875)_{10}$.
5. Therefore, the resultant decimal number is $(0.875)_{10}$.

Ans 8: (b)

Binary representation of decimal number $(33.35)_{10}$ is :

$(33.35)_{10} \approx (100001.01)_2$

Its binary scientific notation is $(1.0000101)_2 \times 2^5$.

The biased exponent has 3 bits, so the biased exponent varies from $(000)_2$ to $(111)_2$ i.e. from 0 to $(7)_{10}$. So the exponent can vary from -3 to

Hence the number $(33.35)_{10}$ which has an exponent of 5 would overflow.



7.11 MODEL QUESTIONS

Q.1. Convert the decimal number $(-2345.125)_{10}$ to a 32-bits single precision floating point binary number.

Q.2. Consider a 7-bit floating-point representation with 3 bits for the excess-3 exponent and 3 bits for the mantissa.

(a) How would 0.375 be represented in this 7-bit representation?

(b) What decimal value do 0110110 represent?

Q.3. Convert the decimal number $(1460.125)_{10}$ to a 64-bits double precision floating point binary number.

Q.4. The following numbers use the IEEE 32-bit floating-point format. What is the equivalent decimal value?

(a) 1 10000011 11000000000000000000000

(b) 1 00000000 00000000000000000000001

Q.5. Compute the largest and smallest positive numbers that can be represented in the 32-bit normalized form.

****

UNIT 8 : LOGIC FAMILY

UNIT STRUCTURE

8.1 Learning Objectives
8.2 Introduction
8.3 Logic Family
8.4 Resistor Transistor Logic (RTL)
8.5 Integrated Injection Logic (I2L)
8.6 Diode-Transistor Logic (DTL)
8.7 Emitter-Coupled Logic (ECL)
8.8 Transistor-Transistor Logic (TTL)
8.9 Tri State Logic
8.10 Metal Oxide Semiconductor Field-Effect Transistor (MOSFET)
8.11 Let Us Sum Up
8.12 Further Reading
8.13 Answers to Check Your Progress
8.14 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define logic family
- describe the different types of logic family
- describe RTL, I2L, DTL, ECL, TTL and MOSFET
- describe tri-state logic

8.2 INTRODUCTION

In the previous units, we have learnt about logic gates and floating point number representation. Different types of logic gates and their conversion has been discussed in the previous units. We have also learnt to perform arithmetic functions on floating point numbers.

In this unit, we will learn about the concepts of logic family. Different types of logic families are covered in this unit like register transistor logic, integrated injection logic, diode transistor logic among others. Tri State logic and MOSFET technology is also covered in this unit.

8.3 LOGIC FAMILY

Logic family of digital integrated circuits (IC) devices is an assembly of logic gates built by using a number of different designs, methods, components and processes intended to generate binary output. Individual components are formed from various logic families, containing some interrelated basic logical functions. All these can be used as building-blocks to generate systems or as an interconnected structure to form more complex ICs. A logic family can also be referred to as a set of techniques that can be used to execute logic within VLSI ICs. Some of these logic families use static techniques to reduce the design complexities such as memories, central processors etc. Some other like domino logic, uses dynamic techniques to decrease the delay, size and power consumption. Prior to the extensive use of ICs, different vacuum-tubes and solid-state logic systems were used but these were not at all as interoperable and standardized as the IC devices.

ICs are basically used to perform low power circuit operation since they cannot handle large current and voltages. They are fabricated using different technologies like DTL, RTL, TTL, I2L, ECL etc. Some of the terms used in digital IC specifications are discussed below.

Threshold Voltage: It is defined as the voltage of the gate input that causes a change in the state from one logic level to the other at the output.

Propagation Delay: It defined as the average of the signal delay time of the output going from logic 1 to logic 0 and logic 0 to logic 1.

Power Dissipation: It is defined as the power required by a logic cycle to execute with 50% duty cycle at a particular frequency.

Fan-in: It is defined as the number of inputs, a logic gate can handle.

Fan-out: It is defined as the number of loads that the output of the logic gate can drive without damaging its regular operation.

Noise margin: The circuit's capacity to bear noise signals is known as the noise immunity, a quantitative measure of it is the noise margin.

Speed power product: It is defined as the product of gate power dissipation and propagation delay of the gate.

The main types of logic families are :-

A. Resistor Transistor Logic (RTL)

B. Integrated Injection Logic (I2L)

C. Diode-Transistor Logic (DTL)

D. Emitter-Coupled Logic (ECL)

E. Transistor-Transistor Logic (TTL)

F. Metal Oxide Semiconductor Field- Effect Transistor (MOSFET)

A comparison of the above families are given in Table 8.1.

Table 8.1: Comparing the parameters of different logic families

Parameter	RTL	I^2L	DTL	ECL	TTL	MOSFET
Basic Gate	NOR	NOR	NAND	OR-NOR	NAND	NAND
Fan out	5	Depends on Injector Current	8	25	20	20
Power dissipation	12mW	6 to 70 nW	8-12 mW	40-55 mW	10mW	0.2 – 10 mW
Noise immunity	Nominal	Poor	Good	Poor	Very good	Good
Propagation delay	12 nSec	25-30 nSec	30 nSec	1 nSec	10 nSec	300 nSec

8.4 Resistor Transistor Logic (RTL)

The resistor transistor logic (RTL) is a category of digital circuits that uses resistors at the input and bipolar junction transistors (BJTs) as the switching devices. These circuits were previously constructed with discrete components, but later on it became the first digital logic family to be produced, as the IC.

RTL inverter: A simple RTL inverter is shown in Fig 8.1. It consists of a common-emitter (CE) stage and a base resistor that is connected in between the input voltage source and the base. The base resistor's work is to increase the negligible transistor's input voltage range from around 0.7 V to about 3.5 V, by altering the input voltage to current. The job of the collector resistor is to change the collector current into voltage.

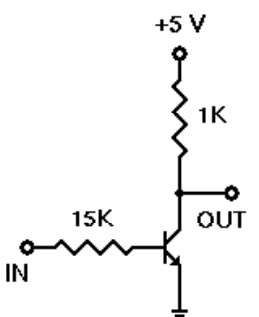


Figure 8.1: RTL inverter

RTL NOR gate: Using two or more base resistors (RC and R4) rather than one, it becomes a two input RTL NOR gate, shown in Fig 8.2. By applying the two arithmetic operations comparison and addition simultaneously, the logical operation OR is performed. The equivalent resistance of all the resistors connected to high state and that of the low state forms the two legs of a voltage divider circuit. The number of the inputs and the base resistances are chosen in such a way that only one high state is enough to make the base-emitter voltage greater than the threshold value and as such the transistor is in saturation mode. The transistor is cut-off, when all the input voltages are low. The pull down resistor (R1) biases the transistor to the proper on-off threshold value. Since the collector-emitter voltage of transistor (Q1) is taken as output, so it is inverted and when the inputs are low, it is high. Thus, it performs the logic function of a NOR gate.

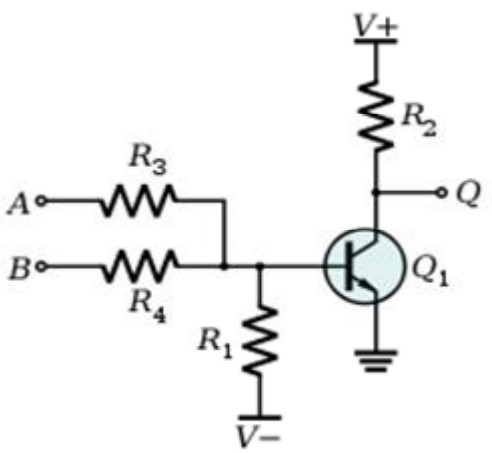


Figure 8. 2: RTL NOR gate with one transistor

8.5 INTEGRATED INJECTION LOGIC (I^2L)

The I^2L is a class of digital circuits that is built with several collector BJTs. In the initial stage, the I^2L had a speed that was almost equivalent to TTL, low power as CMOS, making it the best logic family for use in VLSI ICs. Though, the logic voltage levels are very close i.e. high value of 0.7V and low value of around 0.2V, it has a high noise immunity since it operates by current rather than by voltage. Sometimes, it is also termed as merged transistor logic. Figure 8.3 shows an I^2L circuit.

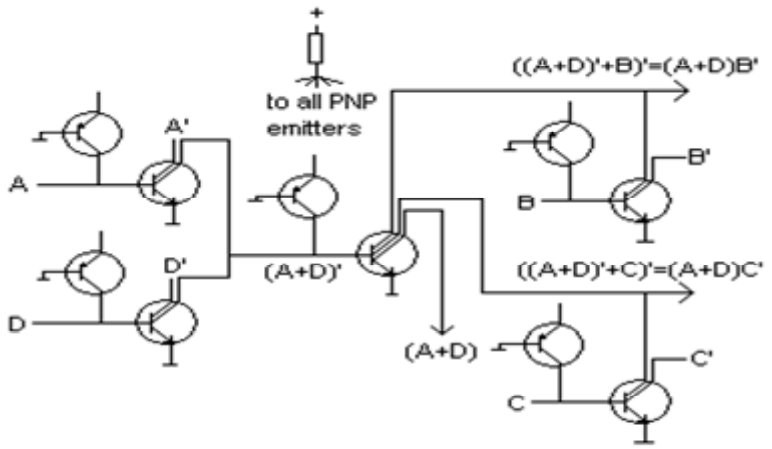


Figure 8.3: An I^2L logic circuit

The main part of an I^2L circuit is the common emitter open collector inverter. It consists of an NPN transistor with the base biased with a forward current and the emitter connected to ground. At the base, the input is applied either as a high-z floating condition or as a current sink. The output is taken from the collector. So, it is either high-z floating condition representing high logic level or a current sink showing the low logic level. If the bias current is grounded, the transistor is in off state and the collector is in floating condition. But, if the bias current is not connected to the ground, then the bias current will flow through the transistor to the emitter, turning the transistor on and this makes the collector to sink current i.e. low logic level. Since, the output can sink current instead of sourcing the current so it is better to connect the outputs of several inverters together, forming a wired AND gate (shown in Figure 8.4). It is a two-input NOR gate, when the outputs of two inverters are wired together.

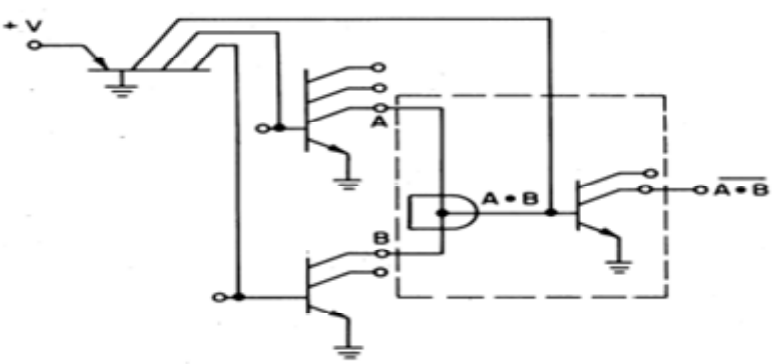


Figure 8.4: A wired AND gate followed by an inversion

8.6 DIODE-TRANSISTOR LOGIC (DTL)

It is a category of digital circuits that is the direct predecessor of transistor transistor logic. It is called so since the ANDing operation is carried out by a diode network and the transistor performs the amplifying part.

The DTL circuit, shown in Figure 8.5, consists of three stages. First is the input diode logic stage (R1, D1 and D2). Second is an intermediate level shifting part (RC and R4). Finally, an output common emitter amplifier stage (R2 and Q1). When both the inputs A and B are at logic 1, the diodes (D1 and D2) are reversed biased. The resistors (R1 and RC) will then provide sufficient current to turn on the transistor (Q1) along with the current required by the resistor (R4). A small voltage on the base (VBE = 0.3 V for Ge and 0.6 V for Si) of the transistor (Q1) will be generated. The Q1 transistor's collector current will then produce low value at the output (Q). If both the inputs or either of them is low, then one of the two input diodes conduct and pull the voltage less than around 2 volts. Resistors (RC and R4) behave as a voltage divider which makes transistor Q1's base voltage to be negative and turns it off. Transistor Q1's collector current will be around zero, so resistor (R2) will give high value at the output (Q).

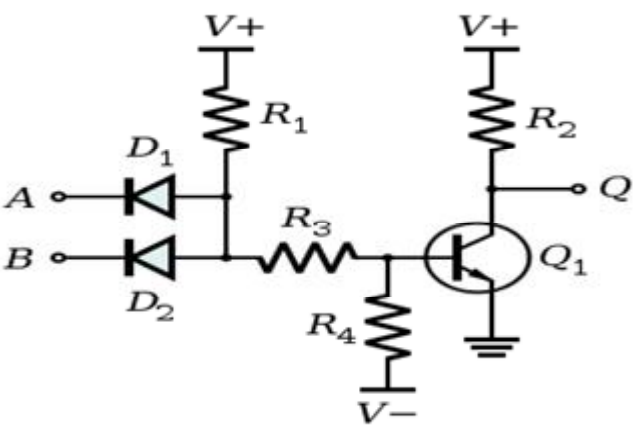


Figure 8.5: DTL logic

8.7 EMITTER-COUPLED LOGIC (ECL)

The ECL is a high speed IC BJT logic family. It uses a BJT differential amplifier having single ended input and low emitter current to avoid the

transistor going to its saturation state and also its slow turn-off characteristics. Since the current is steered in between the two legs of an emitter coupled pairs so it is sometimes called current steering logic (CSL), current switch emitter-follower (CSEF) logic or current mode logic (CML). In case of ECL, the transistors never reach the saturation stage, the input impedance is high, the output resistance is low and the input-output voltages have a small swing (around 0.8 V). So, the transistors change its states fast, fanout capability is high and gate delays are low.

The major limitation of an ECL is that each gate always draws the current, so it consumes more power than the other logic families. The equivalent of an ECL is made out of FETs known as the source-coupled logic (SCFL).

Figure 8.6 shows a NOR-OR gate using ECL logic. Here, A and B are the two inputs. When either one of the two inputs is high then the NOR output is low and the OR output is high. If both the inputs are low then the NOR is high and OR is low. Transistor Q3's base voltage is set at a level where there is a sufficient base current to make Q3 to conduct. Thus, Q3's collector brings the OR output low. Transistor Q3's emitter is high enough in comparison to Q2's base such that Q2 cannot conduct, so Q2's collector is grounded. This makes the NOR's output high. When either of the two inputs is high, then the subsequent transistor conducts brings Q1 or Q2's collector to be low causing the NOR's output to be low and OR's output to be high.

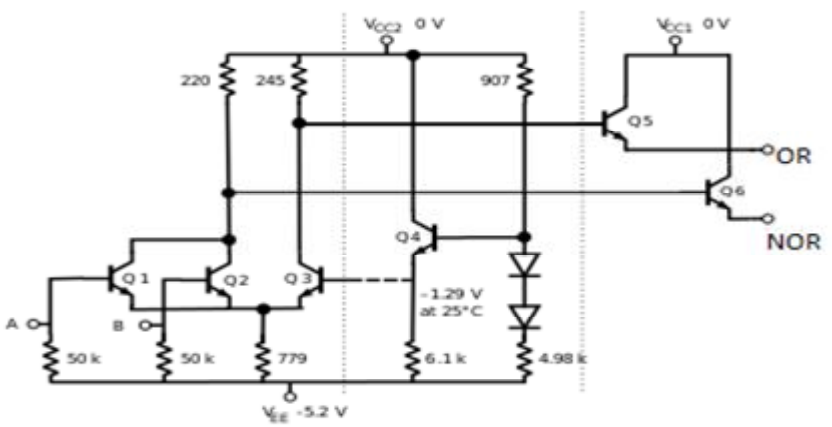


Figure 8.6: Two I/P ECL OR/NOR gate

8.8 TRANSISTOR-TRANSISTOR LOGIC (TTL)

It is a category of digital circuits built using BJTs and resistors. In this case, the transistors perform both the ANDing operation and the amplifying function. The TTL circuits were widely used in applications such as industrial controls, computers, instrumentation, consumer electronics etc. TTL inputs are the emitters of a multiple-emitter transistor. The TTL IC structure is functionally the same as that of the multiple transistors where the collector and bases are tied together. A common emitter amplifier buffers the output. To eliminate the problem with the high output resistance, totem-pole (push-pull) arrangement is used. It consists of the two n-p-n transistors (T_C and T_D), lifting diode (D) and the current-limiting resistor R_C, as shown in figure 8.7. It works by applying the current steering principle.

When both T_B and T_D are off, transistor T_C operates in an active region like a voltage follower producing a high output. When T_B is on, it activates T_D, driving low voltage to the output. The series combination of T_B's CE junction and T_D's BE junction is in parallel with the series of T_C's BE junction, T_D CE junction and D's anode cathode junction. The second series combination has the larger threshold voltage, so no current flows through it and transistor T_C is in off state. In between the transition, the resistor RC lowers the current that flows directly through the series connected transistors T_C, T_D and diode D. The efficiency of the gate may be increased by removing the pull-up and pull-down resistors from the output stage, without affecting the power consumption.

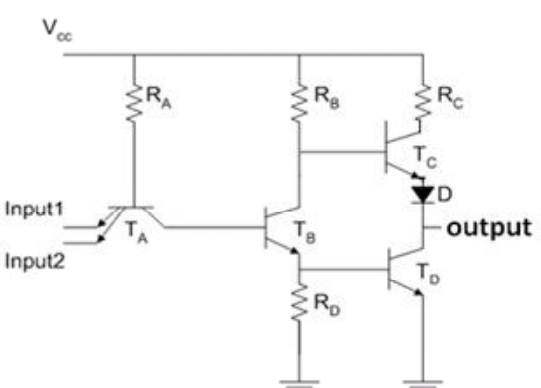


Figure 8.7: TTL logic inverter

TTL NAND

Figure 8.8 shows a NAND gate implemented using TTL logic. If both inputs are at logic 1, the two transistors are in reverse active mode. A current flows through the first resistor through the collector and base of these transistors and then on the right through the base of the transistor, thus saturating it and the output down is in low state. When both inputs are low, the simplest path to ground through the first resistor is through the base of the transistors. Thus the collector voltages are low, keeping that transistor off and bringing the output to logic 1 state. If only one of the inputs is at a logic 0, the input gives the easiest path to ground, keeping the transistor on the right switched off state.

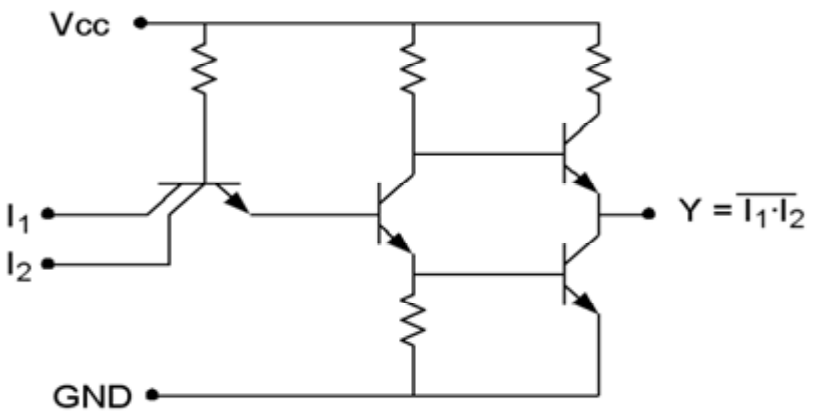


Figure 8.8: TTL NAND gate circuit

8.9 TRI STATE LOGIC

In digital electronics, tri state or three state logic allows an output port to assume a 0 and 1 logic levels in addition to high impedance state. This permits multiple circuits to distribute the same output lines. They are found applicable in many flip-flops, bus, drivers and registers. Some of the other uses are peripherals, computer memory and internal and external buses. Some of the devices are controlled by an active-low input called Output Enable (OE) which gives the information whether the outputs should be held in a logic1 state or a high-impedance state. It uses the advantage of high speed operation of wire ANDing of open collector configuration and totem-pole arrangement.

In the tri state logic, the high impedance state works as a selector which keeps away the circuits that are not being used. Keeping one device on a high-impedance state, prevents the design from a short circuit. Figure 8.9 shows the tri state logic. In this case, when B =1, the circuit operates as a normal inverter and when B=0,the circuit goes to high impedance state.

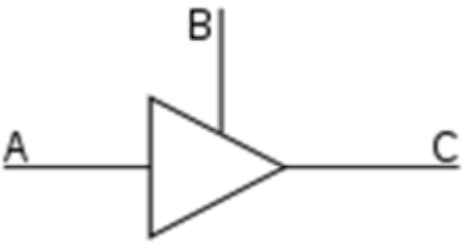


Figure 8.9: Tristate logic buffer

As shown in Fig 8.10, the low value at E forward biases the BE junction of Q1, and turns off Q2 by shunting the current in R1, making Q4 also off. The first diode shunts away from the base of Q3, at a low value of E, thus turning it off.

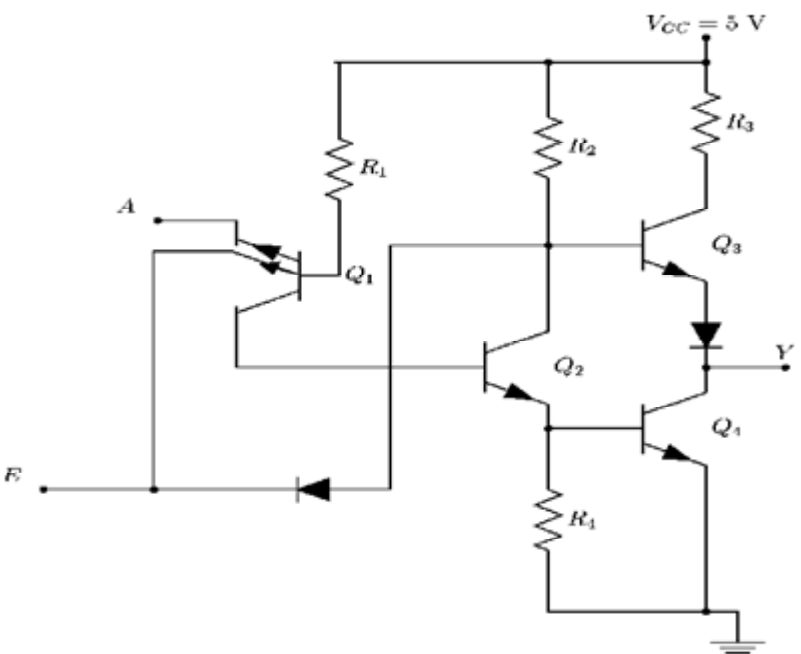


Figure 8.10: Tristate TTL inverter

8.10 METAL OXIDE SEMICONDUCTOR FIELD-EFFECT TRANSISTOR (MOSFET)

MOSFET is a FET that is usually fabricated by the controlled oxidation of Si. The conductivity of the device is determined by the insulated gate's voltage. This capacity to change the conductivity with respect to the amount of applied voltage can be used for switching or amplifying electronic signals. The main advantage of a MOSFET is that it needs almost zero input current to control the load current, when compared to BJTs. They can be classified under two types. One is the enhancement channel MOSFET and other is the depletion channel MOSFET. In case of the enhancement one, the conductivity of the device is increased by the voltage applied to the gate terminal increases. In the depletion mode MOSFET, the conductivity is decreased by applying voltage at the gate. The gate material used in MOSFET is a layer of polysilicon. Here, different dielectric materials are used in the oxide part. MOSFETs can be built with either p-type or n-type semiconductors or using complementary pairs of MOS transistors i.e. with CMOS logic. Figure 8.11 shows the p and n channel, enhancement and depletion type MOSFET.

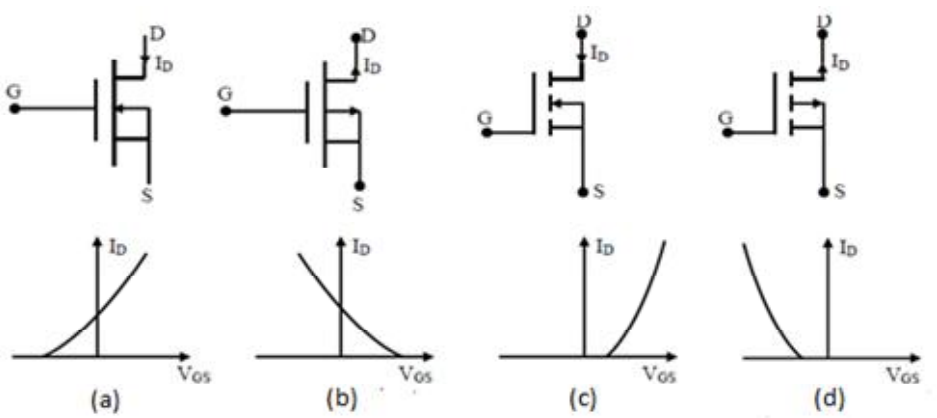


Figure 8.11: (a) n-channel depletion type MOSFET (b) p-channel depletion type MOSFET (c) n-channel enhancement type MOSFET (d) p-channel enhancement type MOSFET

Logic gates with MOSFET's:

Several logic gates such as AND, NAND, OR, and NOR using CMOS technology are discussed below. Figure 8.12 shows a two input NAND gate using CMOS logic. The transistors Q1 and Q3 are controlled by the input signal ($Input_A$), where the upper transistor is turned off and the lower transistor is turned on when the input is at logic 1. Q2 and Q4 are controlled by the same input signal ($Input_B$). The upper transistors Q1 and Q2 have their source and drain terminals parallelly connected while the transistors Q3 and Q4 are series-connected. The output will go high if either of the upper transistors saturates, and output will be low only if both lower transistors saturates.

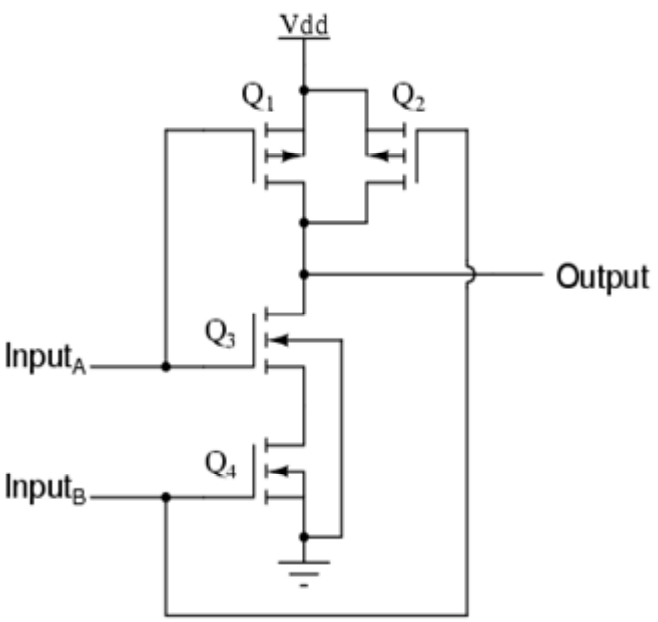


Figure 8.12: NAND gate using CMOS logic

As shown in Figure 8.13, the CMOS AND gate is created in the same way as the CMOS NAND gate but by only inverting the terminal at the output.

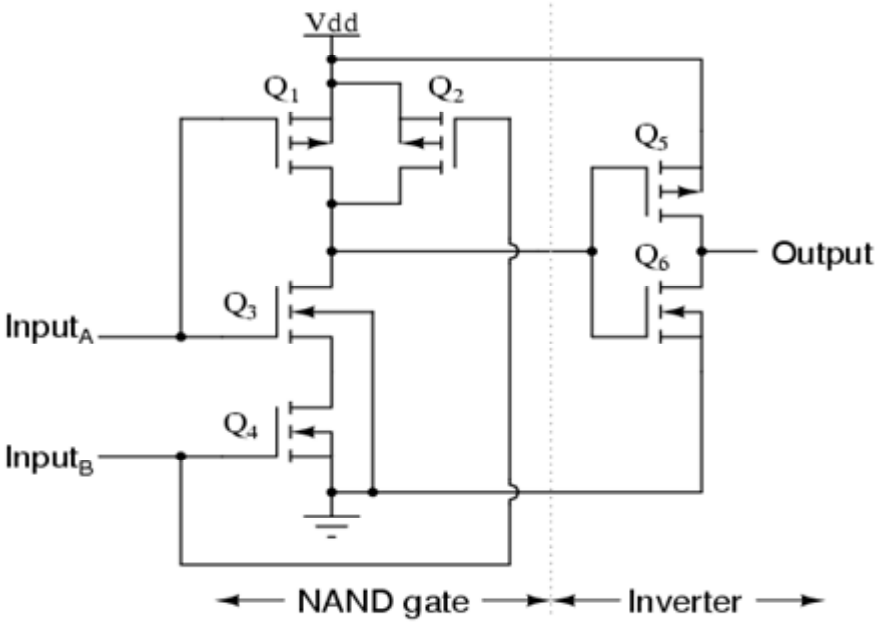


Figure 8.13: AND gate using CMOS logic

A CMOS NOR gate circuit also uses four MOSFETs alike that of the NAND gate, except the arrangements of the transistors. Here, in place of the two parallelly connected upper transistors and two series connected transistors lower ones, the NOR gate uses two two parallel-connected sinking transistors and two series-connected sourcing transistors, as shown in figure 8.14.

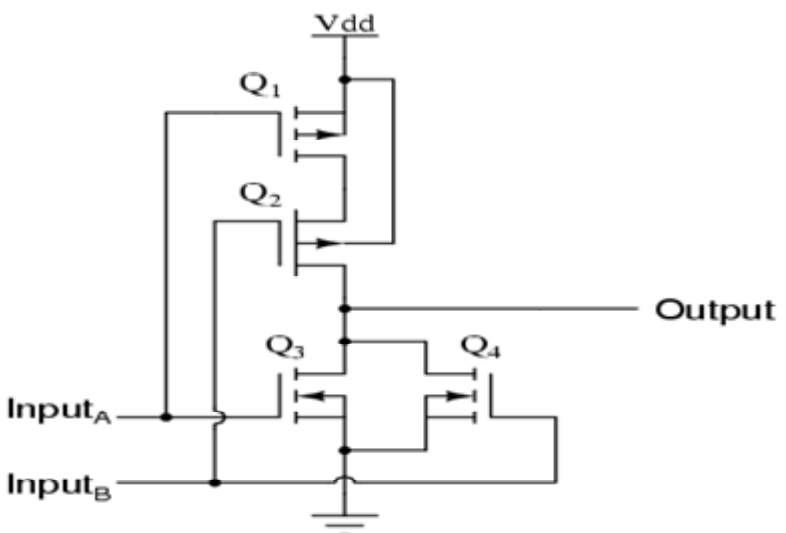


Figure 8.14: NOR gate using CMOS logic

The OR gate is made up from the basic NOR gate with an inverter stage on the output, as shown in Figure 8.15.

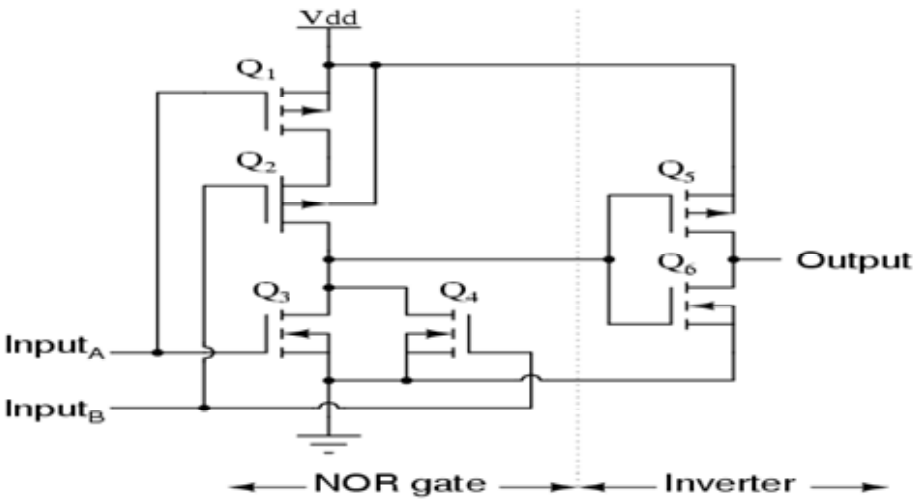


Figure 8.15: OR gate using CMOS logic



CHECK YOUR PROGRESS

1. What is a totem pole output?
2. State the advantages of CMOS logic.
3. Write a note on tri-state gates.
4. What is the significance of high impedance state in tri-state gates?
5. Define the term fan out.
6. How to increase fan-in of an RTL NOR gate?
7. The DTL propagation delay is relatively _____.
8. In an ECL, from where is the output taken?
9. In ECL the fanout capability is _____.
10. Give one disadvantage of ECL.



8.11 LET US SUM UP

- Logic family of digital integrated circuits (IC) devices is an assembly of logic gates built by using a number of different designs, methods, components and processes intended to generate binary output.
- ICs are basically used to perform low power circuit operation since they cannot handle large current and voltages.
- The RTL is a category of digital circuits that uses resistors at the input and bipolar junction transistors (BJTs) as the switching devices.
- The I2L is a class of digital circuits that is built with several collector BJTs. Initially it had a speed that was almost equivalent to TTL, still had a low power as CMOS, making it the best logic family for use in VLSI ICs.
- The DTL is a category of digital circuits that is the direct predecessor of transistor transistor logic.
- The ECL is a high speed IC BJT logic family. It uses a BJT differential amplifier having single ended input and low emitter current to avoid the transistor going to its saturation state and also its slow turn-off characteristics.
- TTL is a category of digital circuits built using BJTs and resistors.
- In digital electronics, tri state or three state logic allows an output port to assume a 0 and 1 logic levels in addition to high impedance state.
- The main advantage of a MOSFET is that it needs almost zero input current to control the load current, when compared to BJTs.



8.12 FURTHER READING

- Mano M. M. (2017), marris, 2007, Digital Logic and computer design. Pearson Education India.
- Kumar, A. A., (2014), Fundamental of Digital Circuit, PHI Learning Pvt. Ltd.

8.13 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q.1: Totem pole output is a standard output of a TTL gate. It is designed to reduce the propagation delay in the circuit and to provide enough output power for high fan-out.

Answer to Q.2: The advantages of CMOS logic are :–

- Consumes less power.
- Fan-out is more.
- Better noise margin
- Can be operated at high voltages, resulting in improved noise immunity.

Answer to Q.3: It is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic1 and logic 0. The third state is high impedance state.

Answer to Q.4: High impedance state of a tri state gate provides a special feature not available in other gates. A larger number of three state gate output can be connected with wires to form a common line without endangering loading effects.

Answer to Q.5: It is the maximum number of inputs which have same family that the gate can drive maintaining its output's normal operation.

Answer to Q.6: Can be increased by adding more output transistors

Answer to Q.7: Large

Answer to Q.8: Collector

Answer to Q.9: High

Answer to Q.10: It requires more power



8.14 MODEL QUESTIONS

Q.1. What are the different types of logic families?

Q.2. Compare the different parameters of the main logic families.

Q.3. Describe the working of the RTL taking into account its primary gate.

Q.4. How does the DTL family work? Describe using its primary gate.

Q.5. Write briefly about how the I^2L family works.

Q.6. Write short notes on TTL.

Q.7. What are the advantages and disadvantages of the MOSEFT family?

****