NEW SYLLABUS 2020-21

University of Mumbai

(As per New Syllabus for Academic Year 2020-2021)

MU

Digital Logic & Computer Organization & Architecture

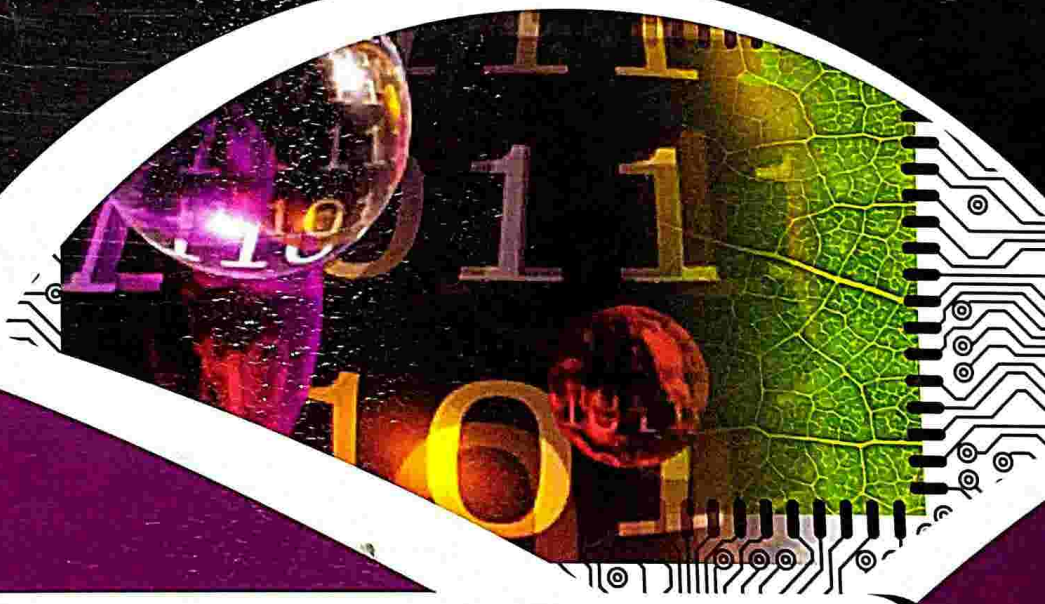
(Code : CSC304)

Semester 3 - Computer Engineering

Bharat Acharya

Shrikant Velankar

U. S. Shah



MCQ's from our Android App.

Books are available on Flipkart amazon

"Tech-Neo eBooks"

Download our Android APP!

SUBJECTS AUTHORS LEADERSHIP EXPERIENCE

OLD Meets NEW To Become BIGGER & BETTER With A Trusted Brand

TECH-NEO PUBLICATIONS

Where Authors Inspire Innovation

A Sachin Shah Venture

Digital Logic & Computer Organization and Architecture

(Code : CSC304)

Semester III - Computer Engineering

(University of Mumbai)

(As per New Syllabus for Academic Year 2020-2021)

Bharat Acharya
B.E. (Computers) Mumbai - 2000.
Founder | Bharat Acharya Education

Prof. Shrikant Velankar
M.E. (Electronics), F.I.E., F.I.E.T.E, C.E.
(Chartered Engineer) Professor,
Department of Electronics Engineering
Vidyalankar Institute of Technology,
Mumbai 400037

U. S. Shah
Formerly, Lecturer
Department of Electronics Engineering, Vishwakarma Institute of Technology (V.I.T.),
Pune. Maharashtra, India.

TECH-NEO PUBLICATIONS
Where Authors Inspire Innovation
A Sachin Shah Venture

B1258

Digital Logic & Computer Organization and Architecture

Bharat Acharya Prof. Shrikant Velankar, U. S. Shah

Semester III - Computer Engineering, (University of Mumbai)

▶ **Edition :** August 2020

Disclaimer

This book is presented solely for educational purposes. The Book is prepared as per the latest syllabus copy received by various Engineering Institutes affiliated to University of Mumbai. Due to Covid 19 Pandemic, online teaching has already started according to syllabus received. Although the Author and Publisher have made every effort to ensure that the information in this book was correct at printing time, the author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause. Any changes in latest syllabus copy will be notified on our website. And supplement regarding the same will be provided/made available on our Website. Tech-Neo Publications is not associated with any University.

ISBN 978-81-947354-9-6

Published by	**Printed at**
Mr. Sachin S. Shah & Mrs. Nayana S. Shah **Permanent Address** Tech-Neo Publications LLP Sr. No. 38/1, Behind Pari Company, Khedekar Industrial Estate, Narhe, Maharashtra, Pune-411041.	**Image Offset (Mr. Rahul Shah)** Dugane Ind. Area, Survey No. 28/25, Dhayari Near Pari Company, Pune - 411041. Maharashtra State, India. E-mail : rahulshahimage@gmail.com

Address For Correspondence

Tech-Neo Publications LLP

407-412, 4th floor, Decision Tower, Above Hotel Tiranga,

Nr. City Pride Theatre, Pune-Satara Road, Pune-411037. Maharashtra State, India.

Email : info@techneobooks.com

Website : www.techneobooks.com

Mobile : 9145531105 / 8668233261

Preface

Dear students,

We are extremely happy to come out with this book on **"Digital Logic & Computer Organization and Architecture"** for the students. This book has been strictly written as per the syllabus. We have divided the syllabus into small chapters so that the topics can be arranged and understood properly. The topics within the chapters have been arranged in a proper sequence to ensure smooth flow of the subject.

We are thankful to Shri. Sachin Shah for the encouragement and support that he has extended. We are also thankful to the staff members of Tech-Neo Publications and others for their efforts to make this book as good as it is. We have jointly made every possible efforts to eliminate all the errors in this book. However if you find any, please let us know, because that will help us to improve further.

We are also thankful to our family members and friends for their patience and encouragement.

Special Thanks to,

Anuja Gote, Ashish Shekhar, Mohit Gujar

- Authors

□□□

SYLLABUS

Course Code	Course Name	Credit
CSC304	Digital Logic & Computer Organization and Architecture	3

Pre-requisite: Knowledge on number systems

Course Objective :

1. To have the rough understanding of the basic structure and operation of basic digital circuits and digital computer.
2. To discuss in detail arithmetic operations in digital system.
3. To discuss generation of control signals and different ways of communication with I/O devices.
4. To study the hierarchical memory and principles of advanced computing.

Course Outcome :

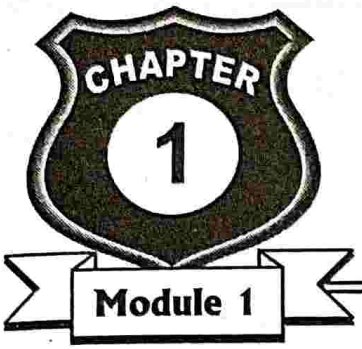
1. To learn different number systems and basic structure of computer system.
2. To demonstrate the arithmetic algorithms.
3. To understand the basic concepts of digital components and processor organization.
4. To understand the generation of control signals of computer.
5. To demonstrate the memory organization.
6. To describe the concepts of parallel processing and different Buses.

Module		Detailed Content	Hours
1		**Computer Fundamentals**	5
	1.1	Introduction to Number System and Codes	
	1.2	**Number Systems :** Binary, Octal, Decimal, Hexadecimal,	
	1.3	**Codes :** Grey, BCD, Excess-3, ASCII, Boolean Algebra.	
	1.4	**Logic Gates :** AND, OR, NOT, NAND, NOR, EX-OR	
	1.5	Overview of computer organization and architecture.	
	1.6	Basic Organization of Computer and Block Level functional Units, Von- Neumann Model. **(Refer chapters 1, 2, 3 and 4)**	

Module		Detailed Content	Hours
2		**Data Representation and Arithmetic algorithms**	8
	2.1	**Binary Arithmetic :** Addition, Subtraction, Multiplication, Division using Sign Magnitude, 1's and 2's compliment, BCD and Hex Arithmetic Operation.	
	2.2	Booths Multiplication Algorithm, Restoring and Non-restoring Division Algorithm.	
	2.3	IEEE-754 Floating point Representation. **(Refer chapter 5)**	
3		**Processor Organization and Architecture**	6
	3.1	**Introduction :** Half adder, Full adder, MUX, DMUX, Encoder, Decoder(IC level).	
	3.2	**Introduction to Flip Flop:** SR, JK, D, T (Truth table).	
	3.3	Register Organization, Instruction Formats, Addressing modes, Instruction Cycle, Interpretation and sequencing. **(Refer chapters 6, 7 and 8)**	
4		**Control Unit Design**	6
	4.1	**Hardwired Control Unit :** State Table Method, Delay Element Methods.	
	4.2	**Microprogrammed Control Unit :** Micro Instruction-Format, Sequencing and execution, Micro operations, Examples of microprograms. **(Refer chapter 9)**	
5		**Memory Organization**	6
	5.1	Introduction and characteristics of memory, Types of RAM and ROM, Memory Hierarchy, 2-level Memory Characteristic	
	5.2	**Cache Memory :** Concept, locality of reference, Design problems based on mapping techniques, Cache coherence and write policies. Interleaved and Associative Memory. **(Refer chapter 10)**	
6		**Principles of Advanced Processor and Buses**	8
	6.1	Basic Pipelined Data path and control, data dependencies, data hazards, branch hazards, delayed branch, and branch prediction, Performance measures-CPI, Speedup, Efficiency, throughput, Amdhal's law.	
	6.2	Flynn's Classification, Introduction to multicore architecture.	
	6.3	Introduction to buses : ISA, PCI, USB. Bus Contention and Arbitration. **(Refer chapter 11)**	

❑❑❑

Index



Number Systems

University Prescribed Syllabus

Computer Fundamentals

1.1 Introduction to Number System and Codes

1.2 Number Systems : Binary, Octal, Decimal, Hexadecimal.

1.1 INTRODUCTION

Q. 1.1.1 What is a Number System ?

Ans. :

Definition : A number system is a method to represent the numbers.

- The number system is used for representing a quantity or information in digital electronics.
- A number system refers to a mathematical representation of the numbers by a set of digits or symbols. Every number is represented by the positional notation of the digits.
- A number system can be classified as :

(i) **Non-positional number system :** In a non-positional number system, a digit of a number does not indicate any significance of its position and weight; e.g. Roman number systems. Non-positional number systems are difficult to use.

(ii) **Positional number system :** In a positional number system, the position of every digit of a number indicates the significance or weight to be attached to that digit; e.g. decimal number system.

- Digital electronics systems are positional number systems. They are widely used in computers, microprocessors, data processing, medical instruments, RADAR, navigation etc.

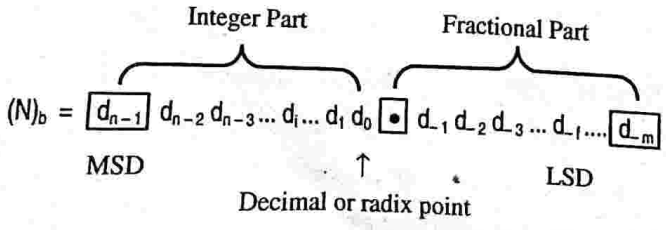
1.1.1 Representing a Number

- In any number system, every symbol in the number is called as a **digit**.
- The leftmost digit of the number has the greatest positional weight among the remaining digits in that number. It is called as the **Most Significant Digit (MSD)**.
- The rightmost digit of the number has the least positional weight among the digits in that number. It is called as the **Least Significant Digit (LSD)**.
- A number is formed by the collection of digits.
- A number has two parts :

(a) Integer part (b) Fractional part

- The fractional part is separated from the integer part with a **decimal point (•) or radix point.**

  The digits on the left side of the radix point form the integer part of the number and digits on the right side of the radix point form the fractional part of the number.
- Thus, a number can be represented as :



Where N = number decimal or radix point
b = base or radix of the number system
m = number of digits in fractional part
n = number of digits in integer part
d_{-m} = Least Significant Digit (LSD)
d_{n-1} = Most Significant Digit (MSD)

and

$0 \leq (d_i \text{ or } d_{-f}) \leq b - 1$

- The digits in the integer part will have weights that are positive powers of base (b) and the digits in the fractional part will have weights that are negative powers of base (b).

Definition : The base or radix (b) of the number system is the total number of digits used in that number system.

Example,

(i) If a number system represents digits 0 and 1 then the base of that system is 2.

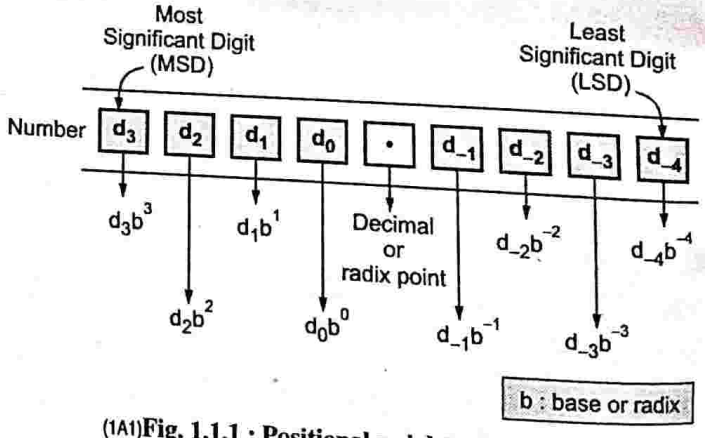
(ii) If number system represents digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 then the base of that system is 10.

(iii) In a number system, the largest value of a digit is one less than the base.

(iv) The value of a number is the sum of products of the digits of that number with their corresponding positional weights.

(v) The first digit to the left of the decimal point has a weight of unity or b^0. The second digit to the left of the decimal point has a weight of b^1 and the third digit to the left has a weight b^2 and so on. The first digit to the right of the decimal point has weight b^{-1}, the second digit to the right of the decimal point has weight b^{-2} and so on. Thus, each digit of the number represents a different multiple of the base.

Refer Fig. 1.1.1



(1A1) **Fig. 1.1.1 : Positional weights of a number**

The value of the number shown in Fig. 1.1.1 can be computed as,

$$N = (d_3 \times b^3) + (d_2 \times b^2) + (d_1 \times b^1) + (d_0 \times b^0) + (d_{-1} \times b^{-1}) + (d_{-2} \times b^{-2}) + (d_{-3} \times b^{-3}) + (d_{-4} \times b^{-4})$$

Thus, the value of each digit in a number can be determined using :

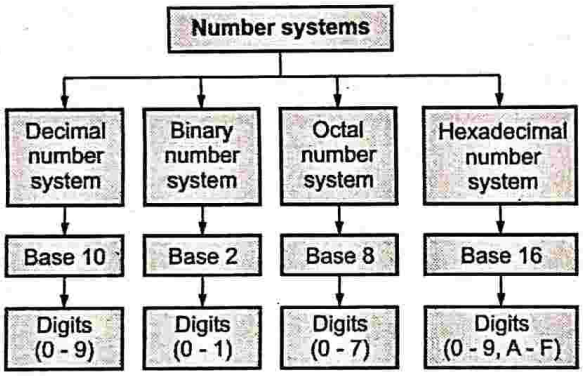
(i) The digit
(ii) Position of the digit
(iii) Base of the number system

1.2 TYPES OF NUMBER SYSTEMS

- The base or radix of the number system is the total number of digits used in that number system.

  e.g. if a number system represents the digits 0 and 1 then the base of the system is 2.
- Depending on the basis, the different **types of number systems** (Refer Fig. 1.2.1) are :

(i) Decimal number system
(ii) Binary number system
(iii) Octal number system
(iv) Hexadecimal number system



(1A2)**Fig. 1.2.1 : Types of number systems**

(I) Decimal number system

- The decimal number system has 10 symbols or digits : 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. As it comprises of ten digits it is called as decimal number system.
- The decimal number system represents ten digits (0 to 1), so the base of the system is 10.

(II) Binary number system

- The modern computers use binary number system for performing operations.
- A binary number system uses two symbols 0 and 1. Thus, the base of the binary number system is 2.

(III) Octal number system

- The octal number system has 8 digits : 0, 1, 2, 3, 4, 5, 6 and 7. Its base is eight (8).
- The octal number system was used in mini-computers.

(IV) Hexadecimal number system

- The Hexadecimal number system is used in microprocessor systems. The hexadecimal number system has base 16 (sixteen).
- It represents the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

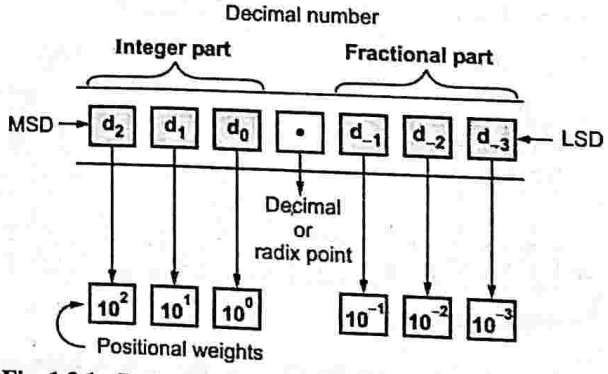
1.3 DECIMAL NUMBER SYSTEM

1. Decimal number system is the number system that we use in our day-to-day life.
2. Decimal numbers have ten digits : 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.
3. **Base :** The base of the decimal number system is **10.**
4. The decimal number digits and their values are shown in Table 1.3.1.

Table 1.3.1 : Decimal number digits and their values

Decimal digit	Value	Radix or base
0	0	10
1	1	10
2	2	10
3	3	10
4	4	10
5	5	10
6	6	10
7	7	10
8	8	10
9	9	10

5. The minimum value digit is 0 (zero) and the maximum value digit is 9 (nine).
6. The positional value of each digit can be determined by its relative position or place in the given number. This positional value of the digits is called as **weight.**
7. Thus the decimal number system is a **positional weighted system.** i.e. the position of the digit with reference to the decimal point determines the value/weight of that digit.
8. The number can be represented by the sum of the weighted digits.
9. Fig. 1.3.1 shows how the digits and their weights are expressed in the decimal number system as a power of 10.



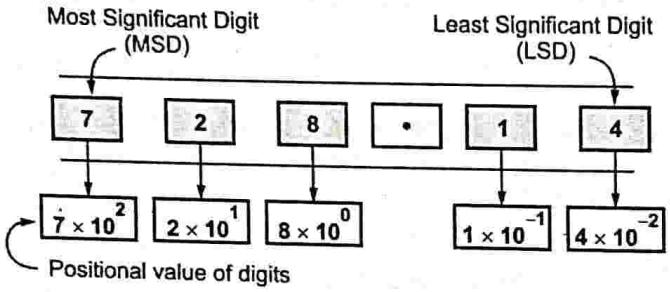
(1A3)**Fig. 1.3.1 : Positional weights for a decimal number system**

10. The leftmost digit has the greatest positional weight and is called as the **Most Significant Digit (MSD)** and the rightmost digit has the lowest positional weight and is called as the **Least Significant Digit (LSD)**.

Ex. 1.3.1

Represent the decimal number 728.14 in powers of 10.

☑ **Ans. :** (Refer Fig. Ex. 1.3.1)



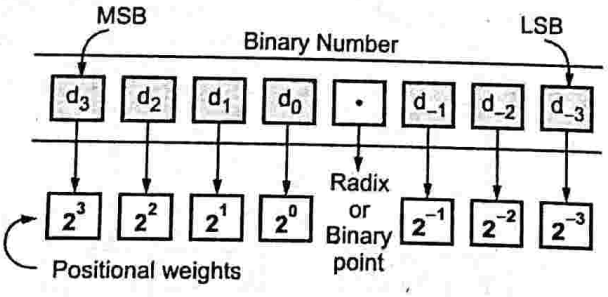
(1A7)**Fig. Ex. 1.3.1 : Representation of decimal number 728.14 in powers of 10**

Fig. Ex. 1.3.1 shows the representation of decimal number 728.14 in powers of 10.

Digit	Weight	Positional value = digit × weight	Number = sum of all weighted digits
7	$10^2 = 100$	$7 \times 10^2 = 700$	$700 + 20 + 8 + 0.1 + 0.04 = 728.14$
2	$10^1 = 10$	$2 \times 10 = 20$	
8	$10^0 = 1$	$8 \times 1 = 8$	
1	$10^{-1} = 0.1$	$1 \times 0.1 = 0.1$	
4	$10^{-2} = 0.01$	$4 \times 0.01 = 0.04$	

⏭ 1.4 BINARY NUMBER SYSTEM

1. Modern computers, microprocessors cannot work on the decimal numbers. They use the binary number system for performing various operations.
2. The binary number system uses 2 digits : 0 and 1. It allows the use of electronic devices like switches, transistors, diodes, etc with two states HIGH or LOW (ON or OFF).



(1A4)**Fig. 1.4.1 : Weights for different positions in a binary number system**

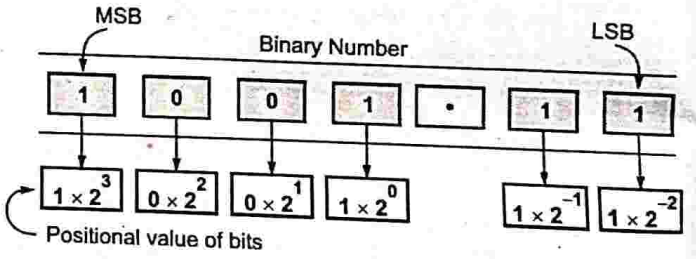
❐ *Definition : The number system with base (radix) two is called as binary number system. It uses two symbols or digits 0 and 1 called as binary digits.*

3. A binary number is a sequence of bits 0 and 1. The **radix point or binary point** separates the integer part from the fractional part.
4. **Base :** The base of the binary number system is 2.
5. It is a **positional weighted system** i.e. every digit position is assigned a weight in terms of powers of 2.
6. Fig. 1.4.1 shows the weights for different positions in a binary system.
7. As shown in Fig. 1.4.1, the first bit to the left of binary point has weight 2^0. The second bit to the left of the binary point has a weight 2^1. The third bit to the left has a weight 2^2 and the fourth bit to the left has a weight 2^3. i.e. the weight of every bit position to the left is one power of 2 greater than the weight of digit to its successive right.
8. The first bit to the right of the binary point has a weight 2^{-1}. The second bit to the right of the binary point has a weight 2^{-2}. The third bit to the right has a weight 2^{-3}.
9. The leftmost bit has the greatest positional weight and is called as the **Most Significant Bit (MSB)** and the rightmost bit has the lowest positional weight and is called as the **Least Significant Bit (LSB)**.

Ex. 1.4.1

Represent the binary number 1001.11 in powers of 2.

☑ **Ans. :** (Refer Fig. Ex. 1.4.1)



(1A5)**Fig. Ex. 1.4.1 : Representation of binary number 1001.11 in powers of 2**

Digit	Weight	Positional value = digit × weight	Number = sum of all weighted digits
1	$2^3 = 8$	$1 \times 8 = 8$	$8 + 0 + 0 + 1 + 0.5 + 0.25 = 9.75$
0	$2^2 = 4$	$0 \times 4 = 0$	
0	$2^1 = 2$	$0 \times 2 = 0$	
1	$2^0 = 1$	$1 \times 1 = 1$	
1	$2^{-1} = 0.5$	$1 \times 0.5 = 0.5$	
1	$2^{-2} = 0.25$	$1 \times 0.25 = 0.25$	

Uses of binary number system

1. The digital systems use binary numbers for their operation because the active devices like diodes, transistors operate as switches and have two stable states, 0 and 1, i.e. a switch can be OPEN or CLOSED. Binary 0 can be used to represent an OPEN switch and binary 1 can be used to represent a CLOSED switch.
2. Noise margin can be easily maintained in digital systems that operate on binary data.

Disadvantage of binary number system

– For representing a binary number we require a long string of **0's** and **1's**. e.g. consider number $(464)_{10}$. Its equivalent in binary is $(111010000)_2$. We require 9 bits to represent $(464)_{10}$. Thus for representing larger numbers, a long strong of **0's** and **1's** will be needed. Hence, the binary numbers are converted to any other number system like decimal, octal or hexadecimal number system.

1.4.1 Binary Number Formats

1. A binary digit is called as a bit e.g. 0, 1.
2. **Nibble :** In a binary number system, a **group of 4 bits** is called as a **nibble**, e.g. 0101, 0111.
3. **Byte :** In a binary number system, **a group of 8 bits** is called as a byte. A byte is a **group of 2 nibbles.** e.g. 1000 0000.
4. **Word :** In a binary number system, a group of 16 bits or 2 bytes or 4 nibbles is called a word. e.g. 0100 0110 0111 1000
5. **Double word :** In a binary number system, a **group of 32 bits or 2 words or 4 bytes or 8 nibbles** is called as a **double word.** e.g. 0110 0111 1010 1011 1000 1111 0011 0010.

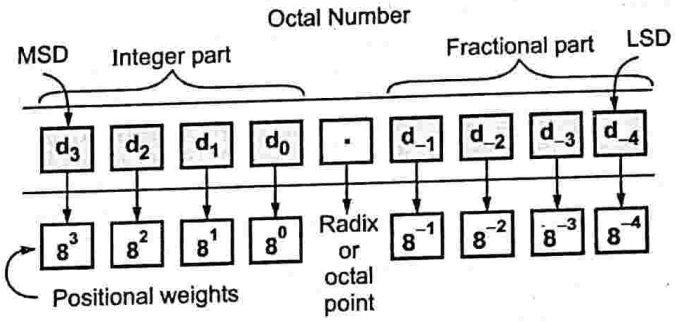
Table 1.4.1 : Data formats used in binary number system

Data format	Size	Example
Bit	1 bit	1
Nibble	4 bits	1010
Byte	8 bits / 2 nibbles	1101 1010
Word	16 bits / 2 bytes / 4 nibbles	1000 0110 1101 1010
Double word	32 bits / 2 words / 4 bytes / 8 nibbles	1011 1000 1110 1111 1001 0110 1101 1010

1.5 OCTAL NUMBER SYSTEM

Definition : The number system that uses 8 symbols 0, 1, 2, 3, 4, 5, 6, 7 to represent a number or information is called as octal number system.

1. **Base :** The base or radix of the octal number system is **8 (eight).**
2. The octal number system is a **positional weighted system.** Since $2^3 = 8$, we need three binary digits to represent an octal digit.
3. Fig. 1.5.1 shows the weights for different positions in an octal number system.



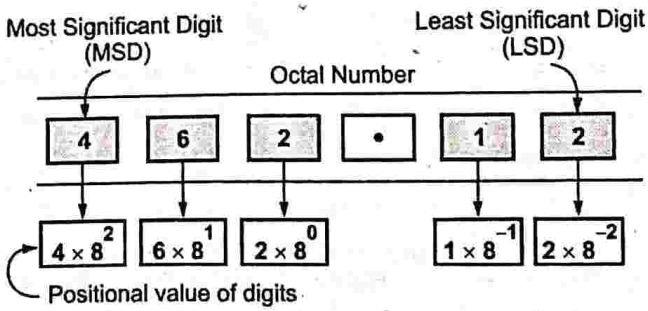
(1A8)**Fig. 1.5.1 : Weights for different positions in an octal number system**

4. As shown in Fig. 1.5.1, the first digit to the left of the octal point has weight 8^0.
5. The second digit to the left of the octal point has weight 8^1. The third digit to the left of the octal point has weight 8^2.
6. Thus, the weight of every position to the left is one power of 8 greater than the weight of the digit to its successive right.
7. The leftmost digit has the greatest positional weight and is called as the **Most Significant Digit (MSD).**
8. The **radix or octal point** separates the integer part from the fractional part.
9. The first digit to the right of the octal point has weight 8^{-1}. The second digit to the right of the octal point has weight 8^{-2} and so on. The rightmost digit has the lowest positional weight and is called as the **Least Significant Digit (LSD).**

Ex. 1.5.1

Express the octal number 462.12 in powers of 8.

☑ **Ans. :** (Refer Fig. Ex. 1.5.1)



(1A6)**Fig. Ex. 1.5.1 : Representation of octal number 462.12 in powers of 8**

Digit	Weight	Positional value = digit × weight	Number = sum of all weighted digits
4	$8^2 = 64$	$4 \times 64 = 256$	$256 + 48 + 2 + 0.125 + 0.3125 = 306.4375$
6	$8^1 = 8$	$6 \times 8 = 48$	
2	$8^0 = 1$	$2 \times 1 = 2$	
1	$8^{-1} = 0.125$	$1 \times 0.125 = 0.125$	
2	$8^{-2} = 0.015625$	$2 \times 0.015625 = 0.03125$	

- An octal number is $\frac{1}{3}^{rd}$ length of corresponding binary number.

Table Ex. 1.5.1(a) : Octal numbers and their equivalent binary numbers

Octal number (Base 8)	Equivalent binary number (Base 2)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Advantages

(i) The octal number system is an easier method to its equivalent group the binary numbers.

(ii) It is easy to convert a binary number to its equivalent octal number.

1.6 HEXADECIMAL NUMBER SYSTEM

- Hexadecimal number system is used in computers and microprocessor based systems.

Definition : The number system that uses 16 symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F to represent a number or information is called as the Hexadecimal number system.

OR

Definition : The number system that uses 16 symbol 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F for representation is called as Hexadecimal number system.

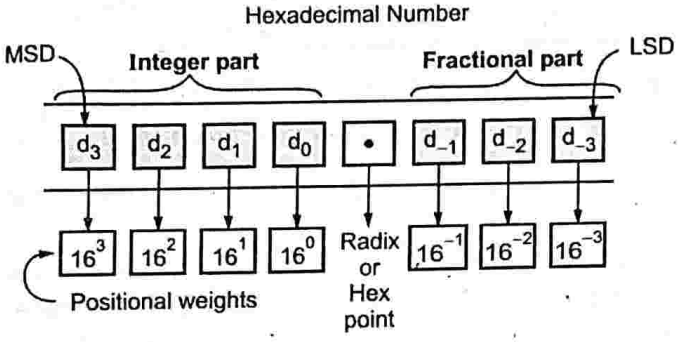
- **Base :** The radix or base of the hexadecimal number system is **16 (sixteen).**
- The hexadecimal number system is an **alphanumeric number system** as it uses both numeric digits and alphabets.
- The minimum value digit is 0(zero) and the maximum value digit is F(decimal 15).
- The hexadecimal number system is a **positional weighted system.** i.e. every digit position is assigned a weight in terms of powers of 16.
- Since $2^4 = 16$, we need 4 binary digits to represent a hexadecimal digit. Thus, a **hexadecimal number is $\frac{1}{4}^{th}$ the length of the corresponding binary number.**
- Computer operates with 8 bits, 16 bits, 32 bits, 64 bits data, i.e., multiples of 4 bits. Hence, it is easy to represent the numbers in hexadecimal.

Table 1.6.1 : Hexadecimal numbers and their equivalent binary numbers

Hexadecimal digit (Base = 16)	Decimal value (Base = 10)	Binary equivalent (Base = 2)
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
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

- Fig. 1.6.1 shows the weights for different positions in a hexadecimal number system.
- The radix point or hexadecimal (hex) point separates the integer part from the fractional part.
- As shown in Fig. 1.6.1, the first digit to the left of hex point has weight 16^0. The second digit to the left of the hex point has weight 16^1.
- The third digit to the left of the hex point has weight 16^2. Thus, the weight of every digit position to the left is one power of 16 greater than the weight of the digit to its successive right.

- The leftmost digit has the greatest positional weight and is called as the **Most Significant Digit (MSD).**
- The first digit to the right of hex point has weight 16^{-1}. The second digit to the right of the hex point has weight 16^{-2} and so on. The rightmost digit has the lowest positional weight and is called as the **Least Significant Digit (LSD).** (Refer Fig. 1.6.1)



(1A9)**Fig. 1.6.1 : Weights for different positions in a Hexadecimal number system**

1.6.1 Advantages of Hexadecimal Number System

1. It is very easy to represent a group of four bit binary numbers. Hence, hexadecimal number system is used for writing assembly language programs.
2. Hexadecimal numbers are compact (e.g. 1111 is binary can be expressed as $F(15)_{10}$).
3. Hexadecimal numbers are easy to recognise and interpret rather than long strings of binary numbers.
4. Hexadecimal numbers can be easily converted to binary and vice-versa.
5. Hexadecimal number system is more convenient to express a number as compared to binary and octal number systems.

1.7 RELATION BETWEEN DECIMAL, BINARY, OCTAL AND HEXADECIMAL NUMBER SYSTEMS

Table 1.7.1 shows the relation between decimal, binary, octal and hexadecimal number systems.

Table 1.7.1

Decimal number (base 10)	Binary number (base 2)	Octal number (base 8)	Hexadecimal number (base 16)
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

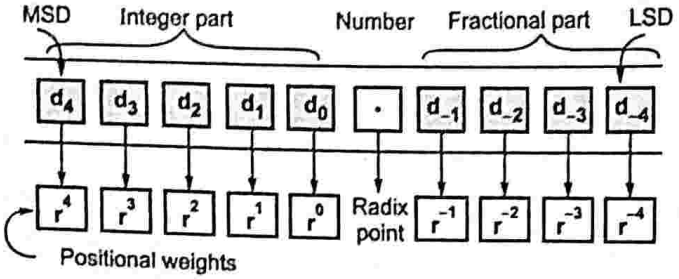
1.8 OTHER NUMBER SYSTEMS

- Many number systems are used for performing different operations.
- Table 1.8.1 shows the commonly used number systems along with their symbols / digits.

Table 1.8.1 : Number systems and their symbols

Number system	Base or Radix (r)	Digits/Symbols
Binary number system	2	0, 1
Ternary number system	3	0, 1, 2
Quaternary number system	4	0, 1, 2, 3
Quinary number system	5	0, 1, 2, 3, 4
Senary numbers system	6	0, 1, 2, 3, 4, 5
Septenary number system	7	0, 1, 2, 3, 4, 5, 6
Octal number system	8	0, 1, 2, 3, 4, 5, 6, 7
Decimal number system	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Undenary number system	11	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, α
Duodenary or duodecimal number system	12	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, α, β
Hexadecimal number system	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Fig. 1.8.1 shows the weights for different positions in a number system with radix r.



(1A10)**Fig. 1.8.1 : Weights for different positions in a number system with radix r**

- The weight of every position to the left is one power of r greater than the weight of the digit to its successive right.
- From Fig. 1.8.1, a number N can be represented as,

$$N = d_4 r^4 + d_3 r^3 + d_2 r^2 + d_1 r + d_0 r^0 + d_{-1} r^{-1} + d_{-2} r^{-2} + d_{-3} r^{-3} + d_{-4} r^{-4}$$

1.9 NUMBER SYSTEM CONVERSION

1. The decimal number system is important because it is universally used to represent numbers and quantities.
2. The digital systems operate with binary data. Hence, it is essential to convert the decimal number to binary before it is applied to a digital system and convert the binary number to decimal for displaying the result on the output devices.
3. If there are many binary numbers of large bits then it is convenient to express the binary numbers in terms of octal or hexadecimal numbers.
4. Hence, we need to convert the binary numbers to octal and vice-versa or we need to convert the binary numbers to hexadecimal and vice-versa.

1.10 BINARY-TO-DECIMAL CONVERSION

To convert a binary number to a decimal number, there are two methods :

(i) Positional notation method
(ii) Doubling method

1.10.1 Binary to Decimal Conversion by Positional Notation Method

- Positional notation method is a way in which the position of each digit has a place value or positional value. The number is expressed as the sum of products of the digits of that number with their corresponding positional values or weights.
- **Steps to be followed** to convert a binary number to an equivalent decimal number are as follows :

Step I : Write the given number.

▼

Step II : Write the positional weights for each digit.

▼

Step III : Multiply each digit in the given number with its corresponding weight to get product of digits or positional value.

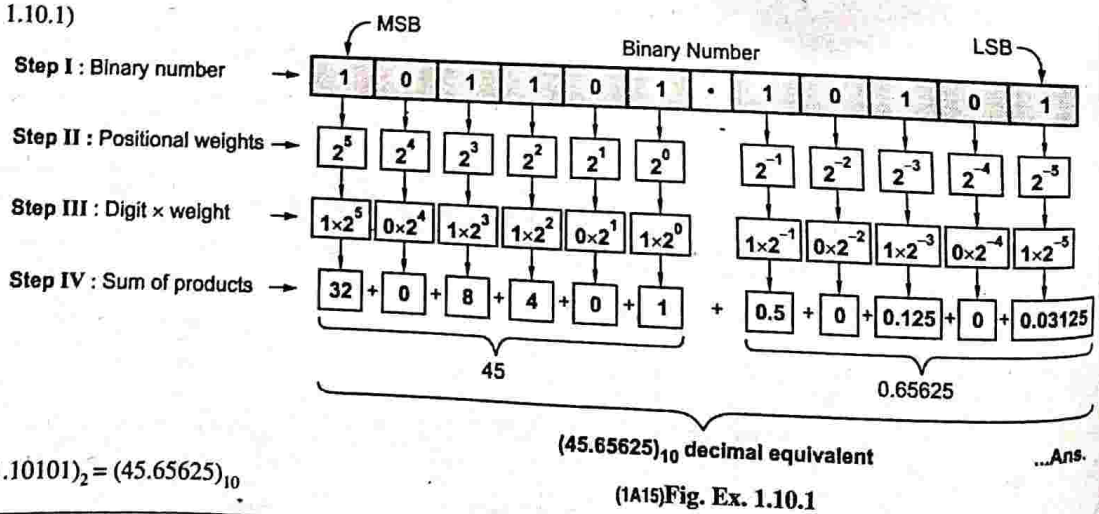
▼

Step IV : Find the sum of products to obtain the decimal equivalent of the given binary number.

Ex. 1.10.1

Convert the following number. Show all steps $(101101.10101)_2 = (\ )_{10}$.

☑ **Ans. :** (Refer Fig. Ex. 1.10.1)



Thus, $(101101.10101)_2 = (45.65625)_{10}$ ...Ans.

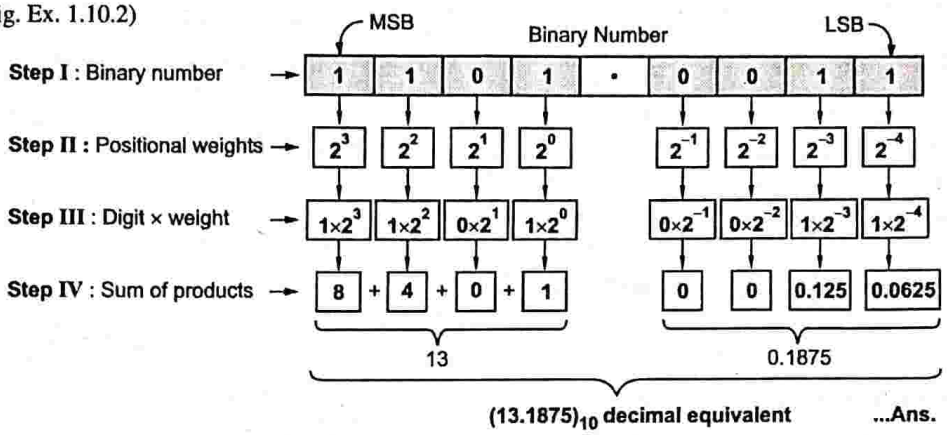
(1A15)**Fig. Ex. 1.10.1**

Module 1

Ex. 1.10.2

Do the following conversion $(1101.0011)_2 = (\ )_{10}$

☑ **Ans. :** (Refer Fig. Ex. 1.10.2)



(1A16)**Fig. Ex. 1.10.2**

Thus, $(1101.0011)_2 = (13.1875)_{10}$

1.10.2 Binary to Decimal Conversion by Doubling Method

- **Doubling method** is simpler than positional notation method. It can be used for converting large binary numbers. The given number is a base of 2, hence doubling can be used.
- **Steps to be followed** to convert a given binary number to decimal using doubling method are as follows :

Step I : Write the binary number.

▼

Step II : Beginning with MSB, multiply the bit by 2 and add the total to the next bit to the right.

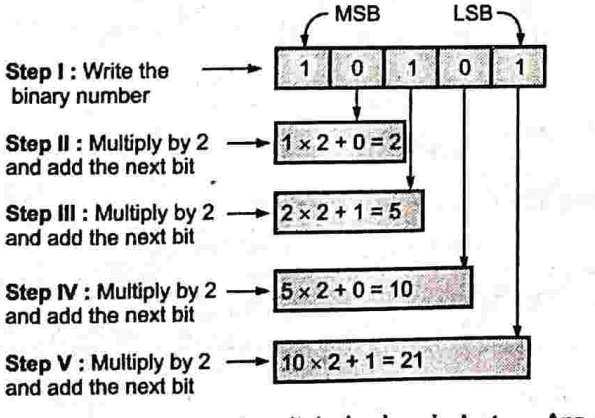
▼

Step III : Repeat step II till all bits are done.

Ex. 1.10.3

Convert $(10101)_2 = (\ )_{10}$ to decimal.

☑ **Ans. :** (Refer Fig. Ex. 1.10.3)



(1A21)**Fig. Ex. 1.10.3**

Thus, $(10101)_2 = (21)_{10}$

1.11 DECIMAL TO BINARY CONVERSION

1. This method of decimal to binary conversion is also called as **Double Dabble method** or **Dibble Dabble method.**
2. In this method, the integer part of the decimal number is converted to binary using successive division by 2 and the fractional part of the decimal number is converted to binary using successive multiplication by 2.
3. **In the successive division by 2,** the integer part is divided by 2 till the quotient reaches zero. The remainder to division is the MSB. To obtain the equivalent binary integer, the remainders are read from bottom to top.
4. **In successive multiplication by 2,** the fractional part of the decimal number is multiplied by 2, till the fractional part of the product is 0.
5. The equivalent binary fractional part is obtained by reading the integers from top to bottom, i.e., MSB to LSB. Finally, the result is combined to obtain the equivalent binary number.

1.11.1 Steps for Decimal to Binary Conversion for Integer Part

Step I : Write the decimal number.

▼

Step II : Divide the decimal number by 2. The remainder obtained is the LSB of the binary number.

▼

Step III : Divide the quotient obtained from step II. The remainder obtained is the second LSB of the binary number.

▼

Step IV : Repeat division by 2 till quotient becomes zero. i.e. quotient is no longer divisible by 2.

▼

Step V : The last remainder obtained from the division is the MSB of the binary number. The equivalent binary number is read from bottom to top.

Example,

Base	Quotient	Remainder	
2	26		
2	13	0 ←	First remainder (LSB)
2	06	1 ←	Second remainder
2	03	0 ←	Third remainder
2	01	1 ←	Fourth remainder
		1 ←	Fifth remainder (MSB)

Top

Binary number

Bottom Read up

$(26)_{10} = (11010)_2$

1.11.2 Steps for Decimal to Binary Conversion for Fractional Part

Step I : Write the fractional decimal number.

▼

Step II : Multiply the fractional decimal number by 2 and write the carry in integer part. The first carry is the MSB.

▼

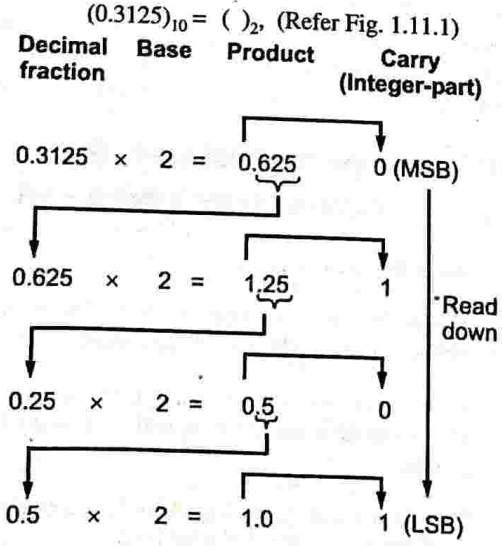
Step III : Multiply the fractional part of the product obtained in step II and record the carry.

▼

Step IV : Repeat steps II and III till the fractional part of the product is zero 0. The last carry is the LSB of the equivalent binary number. The equivalent binary fractional part is obtained by reading the integers from top to bottom i.e. MSB to LSB.

Example,

$(0.3125)_{10} = (\ )_2$, (Refer Fig. 1.11.1)



(1A24)**Fig. 1.11.1**

$(0.3125)_{10} = (0.0101)_2$

Ex. 1.11.1

Convert decimal number 199.375 into binary.

☑ **Ans. :**

➪ **Step I :** Decimal to binary conversion of integer part by successive division by 2.

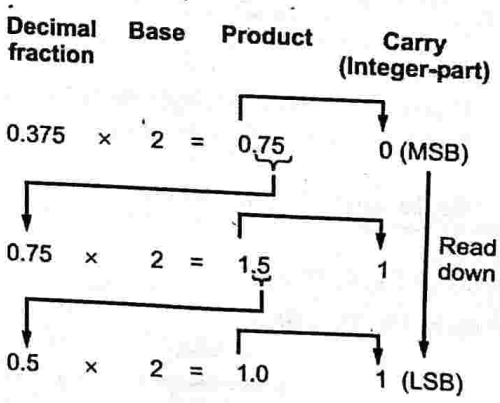
Base	Quotient	Remainder
2	199	
2	99	1 (LSB)
2	49	1
2	24	1
2	12	0
2	6	0
2	3	0
2	1	1
		1 (MSB)

Thus $(199)_{10} = (11000111)_2$

Read up

➪ **Step II :** Decimal to binary conversion of fractional part by successive multiplication by 2.

(Refer Fig. Ex. 1.11.1)



(1A25)**Fig. Ex. 1.11.1**

$(0.375)_{10} = (0.011)_2$

➪ **Step III :** Combine the answers obtained in step I and II.

$(199.375)_{10} = (11000111.011)_2$

UEx. 1.11.2 MU - Q. 1(a), Dec. 16, 4 Marks

Convert decimal number 155.33 into binary.

☑ **Ans. :**

➪ **Step I :** Decimal to binary conversion of integer part by successive division by 2.

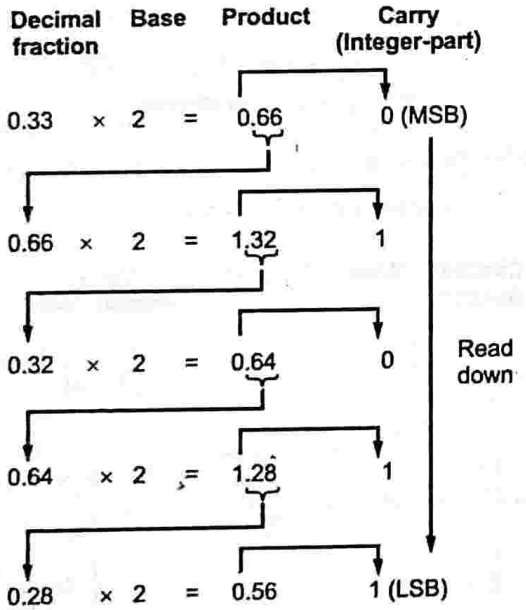
Base	Quotient	Remainder
2	155	
2	77	1 (LSB)
2	38	1
2	19	0
2	9	1
2	4	1
2	2	0
2	1	0
		1 (MSB)

Read up

$(155)_{10} = (10011011)_2$

➪ **Step II :** Decimal to binary conversion of fractional part by successive multiplication by 2.

(Refer Fig. Ex. 1.11.2)



(1A26)**Fig. Ex. 1.11.2**

$(0.33)_{10} = (0.01011)_2$

➪ **Step III :** Combine the answers obtained in step I and II.

$(155.33)_{10} = (10011011.01011)_2$

UEx. 1.11.3 MU - Q. 1(a), May 18, 1 Mark

Convert $(1473.45)_{10}$ into binary.

☑ **Ans. :**

➪ **Step I :** Decimal to binary conversion of integer part by successive division by 2.

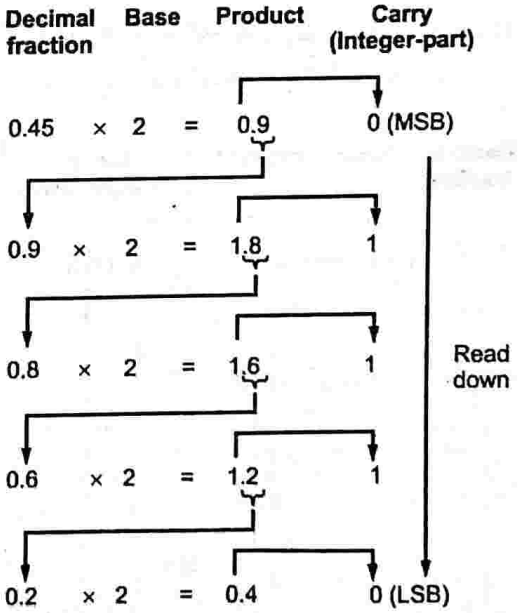
Base	Quotient	Remainder
2	1473	
2	736	1 (LSB)
2	368	0
2	184	0
2	92	0
2	46	0
2	23	0
2	11	1
2	05	1
2	02	1
2	1	0
		1 (MSB)

Read up

$(1473)_{10} = (10111000001)_2$

➪ **Step II :** Decimal to binary conversion of fractional part by successive multiplication by 2.

(Refer Fig. Ex. 1.11.3)



(1A27)**Fig. Ex. 1.11.3**

$(0.45)_{10} = (0.01110)_2$

➪ **Step III :** Combine the answers obtained in step I and step II.

$(1473.45)_{10} = (10111000001.01110)_2$

UEx. 1.11.4 **MU - Q. 1(b), May 15, 2 Marks**

Convert $(214.32)_{10}$ to binary.

Ans. :

Step I : Decimal to binary conversion of integer part by successive division by 2.

Base	Quotient	Remainder	
2	214		
2	107	0	(LSB)
2	53	1	
2	26	1	
2	13	0	
2	6	1	
2	3	0	Read up
2	1	1	
		1	(MSB)

$(214)_{10} = (11010110)_2$

Step II : Decimal to binary conversion of fractional part by successive multiplication by 2.

Decimal fraction	Base	Product	Carry (Integer-part)	
0.32	× 2 =	0.64	0	(MSB)
0.64	× 2 =	1.28	1	
0.28	× 2 =	0.56	0	
0.56	× 2 =	1.12	1	(LSB)

(2A1)**Fig. Ex. 1.11.4**

$(0.32)_{10} = (0.0101)_2$

Step III : Combine the answers obtained in step I and step II.

$(214.32)_{10} = (11010110.0101)_2$

UEx. 1.11.5 **MU - Q. 1(a), Dec. 18, 1 Mark**

Convert decimal number 576.24 into binary.

Ans. :

Step I : Decimal to binary conversion of integer part by successive division by 2.

Base	Quotient	Remainder	
2	576		
2	288	0	(LSB)
2	144	0	
2	72	0	
2	36	0	
2	18	0	
2	9	0	
2	4	1	
2	2	0	
2	1	0	
		1	(MSB)

$(576)_{10} = (1001000000)_2$

Step II : Decimal to binary conversion of fractional part by successive multiplication by 2.

Decimal fraction	Base	Product	Carry (Integer-part)	
0.24	× 2 =	0.48	0	(MSB)
0.48	× 2 =	0.96	0	
0.96	× 2 =	1.92	1	Read down
0.92	× 2 =	1.84	1	(LSB)

(2A2)**Fig. Ex. 1.11.5**

$(0.24)_{10} = (0.0011)_2$

Step III : Combine the answers obtained in step I and step II.

$(576.24)_{10} = (1001000000.0011)_2$

1.12 OCTAL TO DECIMAL CONVERSION

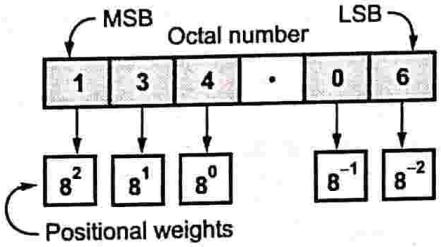
Step I : Write the given number

For e.g. 134.06

▼

Step II : Write the positional weights for each digit (Refer Fig. 1.12.1)

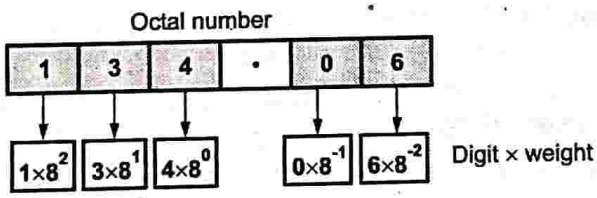
▼



(1A38)**Fig. 1.12.1**

Step III : Multiply each digit in the given number with its corresponding weight to get product of digits or positional value. (Refer Fig. 1.12.2)

▼



(1A39)**Fig. 1.12.2**

▼

Step IV : Find the sum of products to obtain the decimal equivalent of the given octal number.

$$(134.06)_8 = \underbrace{(1\times8^2)+(3\times8^1)+(4\times8^0)+(0\times8^{-1})+(6\times8^{-2})}_{\text{Sum of products}}$$

$$(134.06)_8 = 64+24+4+0+0.09375$$

$$(134.06)_8 = (92.09375)_1$$

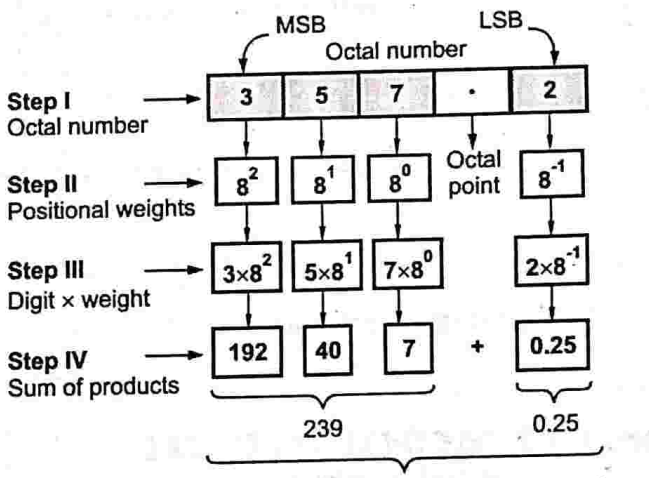
Ex. 1.12.1

Convert the following number into its equivalent decimal number (show step by step process of conversion).

(1) $(357.2)_8$ (2) $(458.54)_8$

☑ **Ans. :**

(1) $(357.2)_8$: (Refer Fig. Ex. 1.12.1(a))

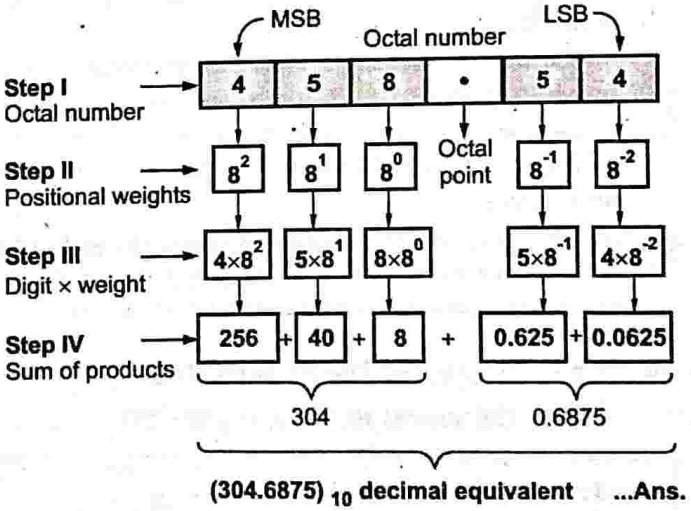


(1A42)**Fig. Ex. 1.12.1(a)**

Thus, $(357.2)_8 = (239.25)_{10}$

(2) $(458.54)_8$

(Refer Fig. Ex. 1.12.1(b))



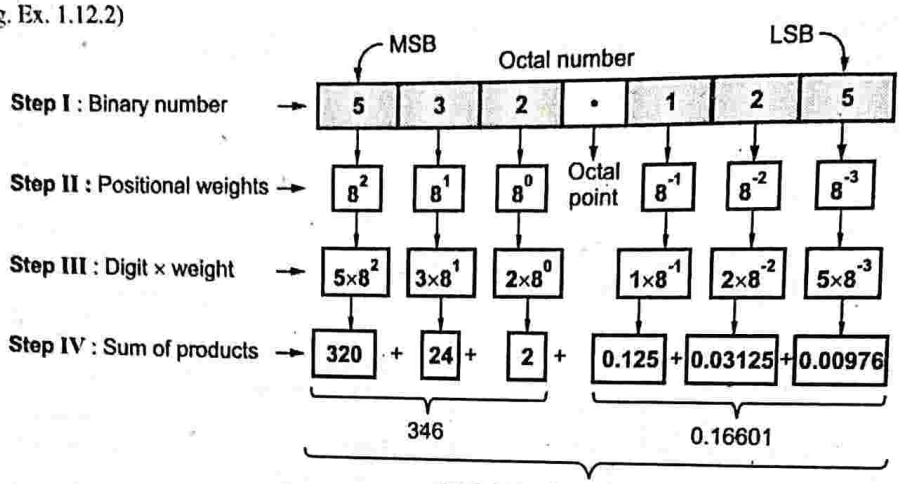
(1A43)**Fig. Ex. 1.12.1(b)**

Thus, $(458.54)_8 = (304.6875)_{10}$

UEx. 1.12.2 **MU - Q. 1(a), May 16, 1 Mark**

Convert $(532.125)_8$ into decimal.

Ans. : (Refer Fig. Ex. 1.12.2)



Thus, $(532.125)_8 = (346.16601)_{10}$ $(346.16601)_{10}$ decimal equivalent ...Ans.

(1A41)Fig. Ex. 1.12.2

1.13 DECIMAL TO OCTAL CONVERSION

- To convert a decimal number to an equivalent octal, the integer part of the decimal number is converted to octal by successive division by 8 and the fractional part of the decimal number is converted to octal by using successive multiplication by 8.
- **In the successive division by 8,** the integer part is divided by 8, till the quotient reaches zero. The remainder to the division is the MSB.
- To obtain the equivalent octal integer, the remainders are read from bottom to top.
- **In successive multiplication by 8,** the fractional part of the decimal number is multiplied by 8, till the fractional part of the product is zero.
- The equivalent octal fractional part is obtained by reading the integers from top to bottom. i.e. MSB to LSB. Finally, the result is combined to obtain the equivalent octal number.

1.13.1 Steps for Decimal to Octal Conversion for Integer Part

Step I : Write the decimal number.

▼

Step II : Divide the decimal number by 8. The remainder obtained is the LSB of the octal number.

▼

Step III : Divide the quotient obtained from step II. The remainder obtained is the second LSB of the octal number.

▼

Step IV : Repeat division by 8 till quotient becomes zero, i.e., quotient is no longer divisible by 8.

▼

Step V : The last remainder obtained from the division is the MSB of the octal number. The equivalent octal number is read from bottom to top.

Example : $(128)_{10}$

Base	Quotient	Remainder	
8	128		
8	16	0	(LSB) first remainder
8	2	0	Read up
		2	(MSB)

Octal number

Thus, $(128)_{10} = (200)_8$

1.13.2 Steps for Decimal to Octal Conversion of Fractional Part

Step I : Write the fractional decimal number

▼

Step II : Multiply the fractional decimal number by 8 and write the carry in integer part. The first carry is the MSB.

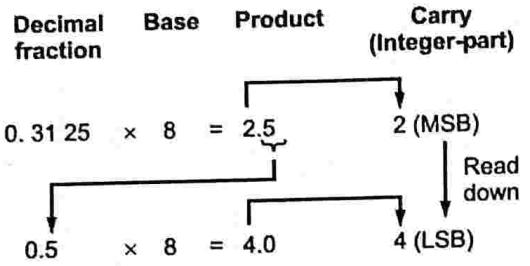
▼

Step III : Multiply the fractional part of the product obtained in step II and record the carry.

▼

Step IV : Repeat step II and III till the fractional part of the product is zero. The last carry is the LSB of the equivalent octal number. The equivalent octal fractional part is obtained by reading the integers from top to bottom i.e. MSB to LSB.

Example : $(0.3125)_{10} = (\ ?\ )_8$ (Refer Fig. 1.13.1)



(1A48)**Fig. 1.13.1**

Thus, $(0.3125)_{10} = (0.24)_8$

UEx. 1.13.1 MU - Q. 1(f), May 15, 1 Mark

Convert $(126)_{10}$ to octal

Ans. :

Step I : Decimal to octal conversion of given number by successive division by 8.

Base	Quotient	Remainder	
8	126		
8	15	6	(LSB)
8	1	7	Read up
		1	(MSB)

Equivalent octal Number

Thus, $(126)_{10} = (176)_8$

Ex. 1.13.2

Convert the following number, show all the steps : $(247)_{10} = (?)_8$

Ans. :

Step I : Decimal to octal number conversion of given number by successive division by 8.

Base	Quotient	Remainder	
8	247		
8	30	7	(LSB) Read up
8	3	6	
		3	(MSB)

$(247)_{10} = (367)_8$

UEx. 1.13.3 MU - Q. 1(a), Dec. 15, 1 Mark

Convert decimal number 199.375 into octal.

Ans. :

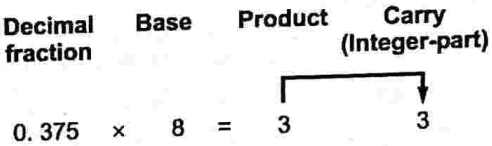
Step I : Decimal to octal conversion of integer part of given number by successive division by 8.

Base	Quotient	Remainder	
8	199		
8	24	7	(LSB)
8	3	0	Read up
		3	(MSB)

Equivalent octal Number

$(199)_{10} = (307)_8$

Step II : Decimal to octal conversion of fractional part of given number by successive multiplication by 2. (Refer Fig. Ex. 1.13.3)



(1A49)**Fig. Ex. 1.13.3**

$(0.375)_{10} = (0.3)_8$

Step III : Combine the answers obtained in step I and II.

$(199.375)_{10} = (307.3)_8$

UEx. 1.13.4 MU - Q. 1(a), Dec. 16, 1 Mark

Convert decimal number 151.33 into octal.

Ans. :

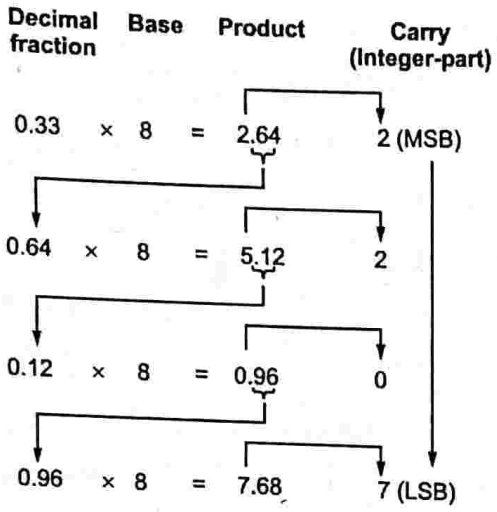
Step I : Decimal to octal conversion of integer part by successive division by 8.

Base	Quotient	Remainder	
8	151		
8	18	7	(LSB)
8	2	2	Read up
		?	(MSB)

Equivalent octal Number

$(151)_{10} = (227)_8$

Step II : Decimal to octal conversion of fractional part by successive multiplication by 8.



(2A3)Fig. Ex. 1.13.4

$(0.33)_{10} = (0.2207)_8$

Step III : Combine the answers obtained in step I and II.

$(151.33)_{10} = (227.2207)_8$

UEx. 1.13.5 MU - Q. 1(a). Dec. 18, 1 Mark

Convert decimal number 576.24 into octal number.

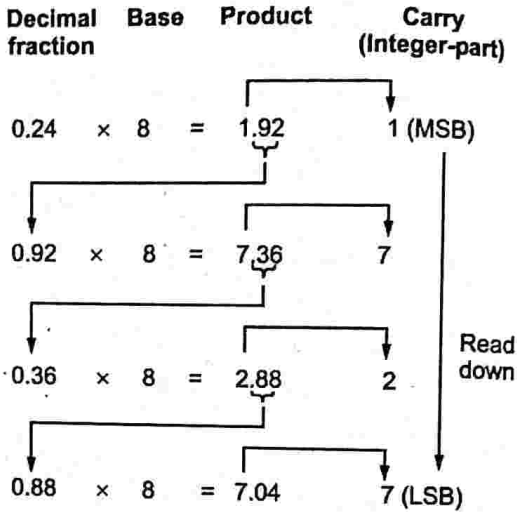
Ans. :

Step I : Decimal to octal conversion of integer part by successive division by 8.

Base	Quotient	Remainder	
8	576		
8	72	0	(LSB)
8	9	0	(Read up)
8	1	1	
		1	(MSB)

$(576)_{10} = (1100)_8$

Step II : Decimal to octal conversion of fractional part by successive multiplication by 8.



(2A4)Fig. Ex. 1.13.5

$(0.24)_{10} = (0.1727)_8$

Step III : Combine the answers obtained in step I and II.

$(576.24)_{10} = (1100.1727)_8$

UEx. 1.13.6 MU - Q. 1(a). May 18, 1 Mark

Convert $(1473.45)_{10}$ into octal.

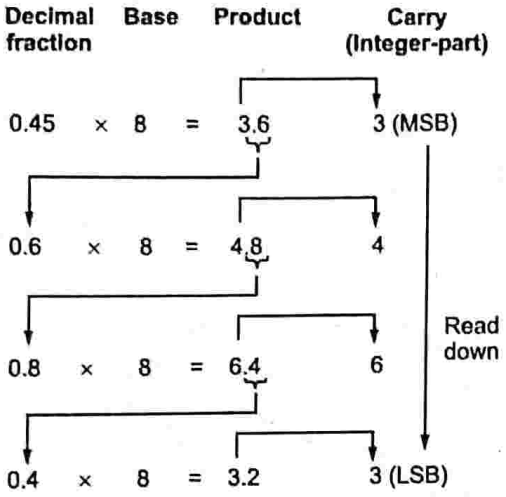
Ans. :

Step I : Decimal to octal conversion of integer part by successive division by 8.

Base	Quotient	Remainder	
8	1473		
8	184	1	(LSB)
8	23	0	(Read up)
8	2	7	
		2	(MSB)

$\therefore (1473)_{10} = (2701)_8$

➪ **Step II :** Decimal to octal conversion of fractional part by successive multiplication by 8.



(2A5)**Fig. Ex. 1.13.6**

$(0.45)_{10} = (0.3463)_8$

➪ **Step III :** Combine the answers obtained in step I and II.

$(1473.45)_{10} = (2701.3463)_8$

⏭ 1.14 OCTAL TO BINARY CONVERSION

Steps to be followed (Refer Fig. 1.14.1)

Step I : Write the given octal number.

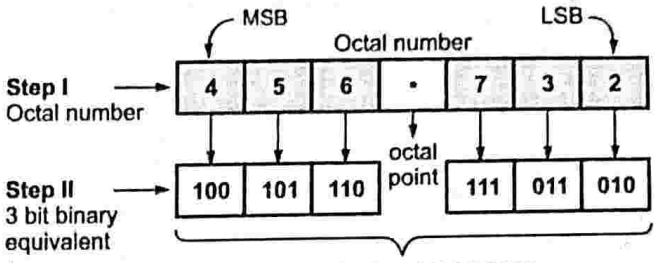
e.g. 456.732

▼

Step II : Replace each octal digit by its 3-bit binary equivalent.

Octal numbers and their binary equivalent numbers

Octal Digit (Base 8)	3 Bit equivalent binary number (Base 2)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

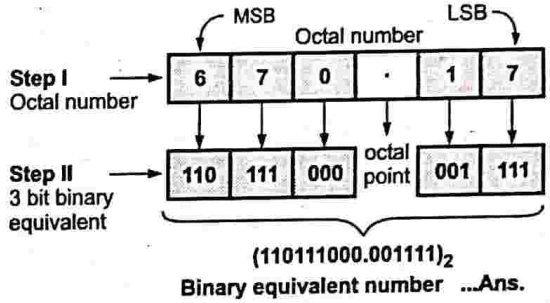


(1A52)**Fig. 1.14.1**

Ex. 1.14.1

Convert $(670.17)_8$ into binary.

☑ **Ans. :** (Refer Fig. Ex. 1.14.1)



(1A53)**Fig. Ex. 1.14.1**

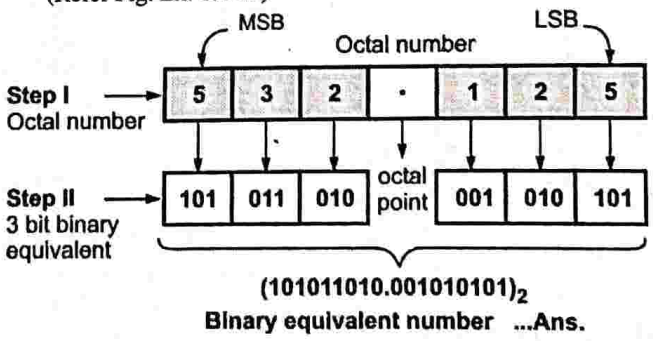
Thus, $(670.17)_8 = (110111000.001111)_2$

UEx. 1.14.2 MU - Q. 1(a), May 16, 1 Mark

Convert $(532.125)_8$ into binary.

☑ **Ans. :**

(Refer Fig. Ex. 1.14.2)



(1A54)**Fig. Ex. 1.14.2**

Thus, $(532.125)_8 = (101011010.001010101)_2$

1.15 BINARY TO OCTAL CONVERSION

Step I : Write the given binary number.

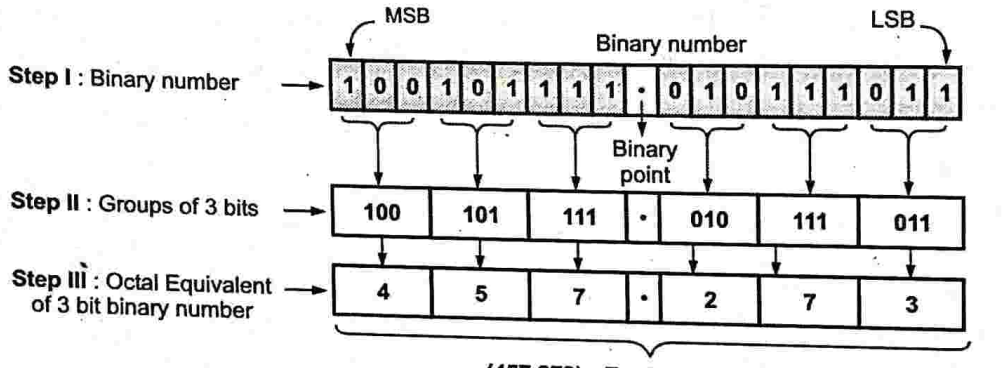
▼

Step II : Starting from either side of the binary point, form groups of 3 bits.

▼

Step III : Replace each 3 bit binary group with its equivalent octal number.

Example : $(100101111.010111011)_2$ (Refer Fig. 1.15.1)



$(457.273)_8$ **Equivalent octal number** ...Ans.

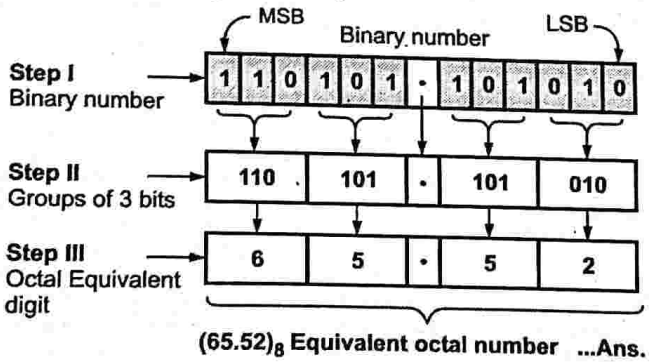
(1A56)**Fig. 1.15.1**

$(100101111.010111011)_2 = (457.273)_8$

Ex. 1.15.1

Convert $(110101.101010)_2$ to octal.

☑ **Ans. :** (Refer Fig. Ex. 1.15.1)



$(65.52)_8$ **Equivalent octal number** ...Ans.

(1A57)**Fig. Ex. 1.15.1**

Thus, $(110101.101010)_2 = (65.52)_8$

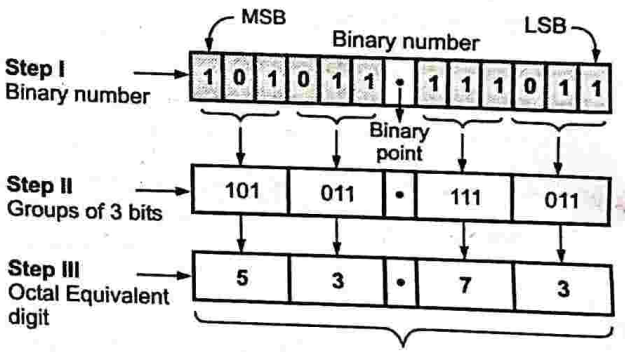
Ex. 1.15.2

Convert the following numbers, show all steps.

$(101011.111011)_2 = (?)_8$

☑ **Ans. :**

(Refer Fig. Ex. 1.15.2)



$(53.73)_8$ **Equivalent octal number** ...An

(1A58)**Fig. Ex. 1.15.2**

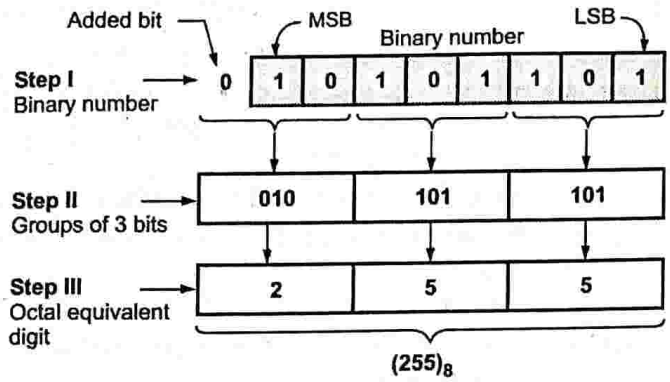
Thus, $(101011.111011)_2 = (53.73)_8$

Ex. 1.15.3

Convert $(10101101)_2 = (\_)_8$

☑ **Ans. :**

(Refer Fig. Ex. 1.15.3)



(1A59)**Fig. Ex. 1.15.3**

Thus, $(10101101)_2 = (255)_8$

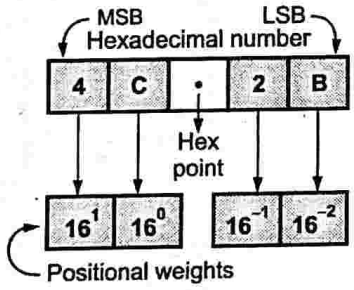
1.16 HEXADECIMAL TO DECIMAL CONVERSION

To convert a hexadecimal number to an equivalent decimal number, the steps to be followed are :

Step I : Write the given number.
e.g. $(4C.2B)_{16}$

▼

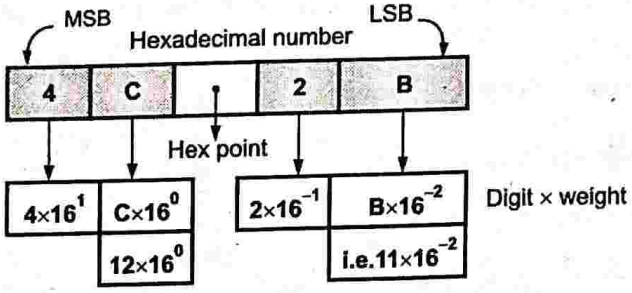
Step II : Write the positional weights for each digit. (Refer Fig. 1.16.1)



(1A60)**Fig. 1.16.1**

▼

Step III : Multiply each digit in the given number with its corresponding weight to get product of digits or positional value. (Refer Fig. 1.16.2)



(1A61)**Fig. 1.16.2**

▼

Step IV : Find the sum of products to obtain the decimal equivalent of the given octal number.

$(4C \cdot 2B)_{16} = (4 \times 16^1) + (12 \times 16^0) + (2 \times 16^{-1}) + (11 \times 16^{-2})$

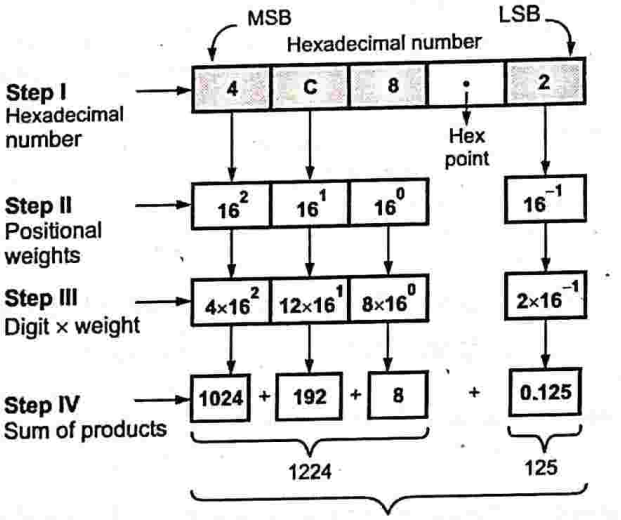
$(4C \cdot 2B)_{16} = (64) + (12) + (0.125) + (0.0429)$

$(4C \cdot 2B)_{16} = (76.1679)_{10}$

Ex. 1.16.1

Do the following conversion :
$(4C8 \cdot 2)_{16} \rightarrow (\ )_{10}$

☑ **Ans. :** (Refer Fig. Ex. 1.16.1)



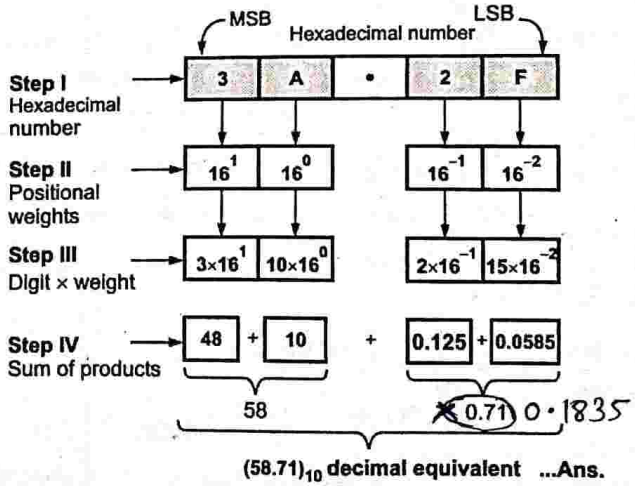
(1A62)**Fig. Ex. 1.16.1**

Thus, $(4C8.2)_{16} = (1224.125)_{10}$

Ex. 1.16.2

Do the following conversion : $(3A \cdot 2F)_{16} = (?)_{10}$

☑ **Ans. :** (Refer Fig. Ex. 1.16.2)



(1A63)**Fig. Ex. 1.16.2**

Thus, $(3A \cdot 2F)_{16} = (58.71)_{10}$

1.17 DECIMAL TO HEXADECIMAL CONVERSION

- To convert a decimal number to an equivalent hexadecimal number, the integer part of the decimal number is converted to hexadecimal by successive division by 16 and the fractional part of the decimal number is converted to decimal by using successive multiplication by 16.
- **In the successive division by 16,** the integer part is divided by 16, till the quotient reaches zero. The remainder to the division is the MSB. To obtain the equivalent hexadecimal integer, the remainders are read from bottom to top.
- **In successive multiplication by 16,** the fractional part of the decimal number is multiplied by 16, till the fractional part of the product is zero.
- The equivalent hexadecimal part is obtained by reading the integers from top to bottom, i.e., MSB to LSB. Finally, the result is combined to obtain the equivalent hexadecimal number.

1.17.1 Steps for Decimal to Hexadecimal Conversion for Integer Part

Step I : Write the decimal number.

▼

Step II : Divide the decimal number by 16. The remainder obtained is the LSB of the hexadecimal number.

▼

Step III : Divide the quotient obtained from Step II. The remainder obtained is the second LSB of the hexadecimal number.

▼

Step IV : Repeat division by 16, till the quotient becomes zero. i.e. quotient is no longer divisible by 16.

▼

Step V : The last remainder obtained from the division is the MSB of the hexadecimal number. The equivalent hexadecimal number is read from bottom to top.

Example : $(2156)_{10}$

Base	Quotient	Remainder	
16	2156		
16	134	C	first remainder (LSB)
16	8	6	Second remainder
	8		(MSB)

Read up

$(2156)_{10} = (86C)_{16}$

1.17.2 Steps for Decimal to Hexadecimal Conversion of Fractional Part

Step I : Write the fractional decimal number.

▼

Step II : Multiply the fractional decimal part by 8 and write the carry in the integer part. The first carry is the MSB.

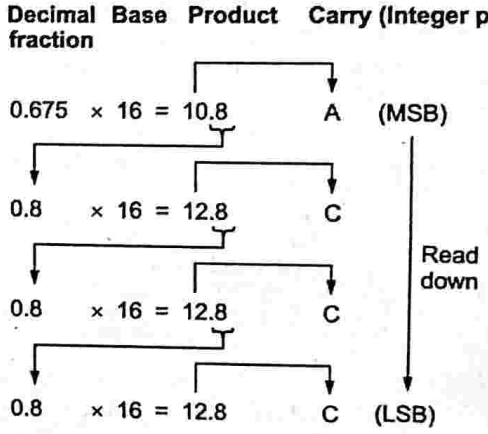
▼

Step III : Multiply the fractional part of the product obtained in step II and record the carry.

▼

Step IV : Repeat steps II and III till the fractional part of the product is zero. The last carry is the LSB of the equivalent hexadecimal number. The equivalent hexadecimal fractional part is obtained by reading the integers from top to bottom, i.e., MSB to LSB.

Example : $(0.675)_{10}$ = ? (Refer Fig. 1.17.1)



(1A64)**Fig. 1.17.1**

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

UEx. 1.17.1 **MU - Q. 1(f), May 15, 1 Mark**

Convert $(126)_{10}$ into hexadecimal number.

Ans. :

Step I : Decimal to hexadecimal conversion of given number by successive division by 16.

Base	Quotient	Remainder	
16	126		
16	7	14	(E)
	7		

(LSB) ↑ (MSB) Read up

Thus, $(126)_{10} = (7E)_{16}$

UEx. 1.17.2 **MU - Q. 1(a), Dec. 15, 1 Mark**

Convert decimal number 199.375 into hexadecimal system.

Ans. :

Step I : Decimal to hexadecimal conversion of integer part by successive division by 16.

Base	Quotient	Remainder	
16	199		(LSB)
16	12	7	Read up
	12		(MSB)

Thus, $(199)_{10} = (C7)_{16}$

Step II : Decimal to hexadecimal conversion of fractional part by successive multiplication by 16.

(Refer Fig. Ex. 1.17.2)

Decimal fraction	Base	Product	Carry (Integer part)
0.375	× 16 =	6.00	6

(1A65)**Fig. Ex. 1.17.2**

$(0.375)_{10} = (0.6)_{16}$

Step III : Combine the answers obtained in step I and step II.

$(199.375)_{10} = (C7.6)_{16}$

UEx. 1.17.3 **MU - Q. 1(a), May 18, 1 Mark**

Convert $(1473.45)_{10}$ into hexadecimal.

Ans. :

Step I : Decimal to Hexadecimal conversion of integer part by successive division by 16.

Base	Quotient	Remainder	
16	1473		(LSB)
16	92	1	
16	5	C	Read up
	5		(MSB)

Thus, $(1473)_{10} = (5C1)_{16}$

Step II : Decimal to Hexadecimal conversion of fractional part by successive multiplication by 16. (Refer Fig. Ex. 1.17.3)

Decimal fraction	Base	Product	Carry (Integer part)	
0.45	× 16 =	7.2	7	(MSB)
0.2	× 16 =	3.2	3	Read down
0.2	× 16 =	3.2	3	
0.2	× 16 =	3.2	3	(LSB)

(1A66)**Fig. Ex. 1.17.3**

$(0.45)_{10} = (0.7333)_{16}$

Step III : Combine the answers obtained in step I and step II.

$(1473.45)_{10} = (5C1.7333)_{16}$

UEx. 1.17.4 **MU - Q. 1(a), Dec. 16, 1 Mark**

Convert decimal number 151.33 into hexadecimal system.

Ans. :

Step I : Decimal to hexadecimal conversion of integer part by successive division by 16.

Base	Quotient	Remainder	
16	151		(LSB)
16	9	7	Read up
	9		(MSB)

$(151)_{10} = (97)_{16}$

Step II : Decimal to hexadecimal conversion of fractional part by successive multiplication by 16. (Refer Fig. Ex. 1.17.4)

Decimal fraction	Base	Product	Carry (Integer part)	
0.33	× 16 =	5.28	5	(MSB)
0.28	× 16 =	4.48	4	
0.48	× 16 =	7.68	7	Read down
0.68	× 16 =	10.88	A	
0.88	× 16 =	14.08	E	(LSB)

(1A67)**Fig. Ex. 1.17.4**

$(0.33)_{10} = (0.547AE)_{16}$

Step III : Combine the answers obtained in Step I and Step II.

$(151.33)_{10} = (97.547AE)_{16}$

UEx. 1.17.5 **MU - Q. 1(a), Dec. 18, 1 Mark**

Convert decimal number 576.24 into hexadecimal system.

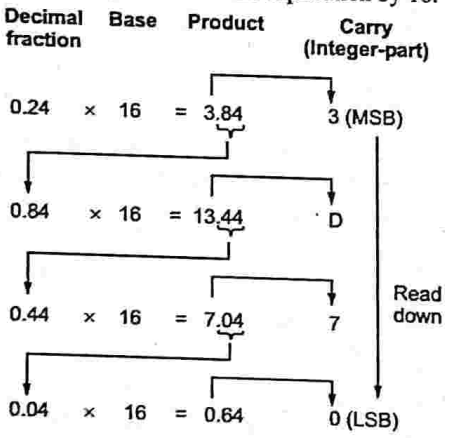
Ans. :

Step I : Decimal to hexadecimal conversion of integer part by successive division by 16.

Base	Quotient	Remainder	
16	576		(LSB)
16	36	0	↑ Read up
16	2	4	(MSB)
		2	

$(576)_{10} = (240)_H$

Step II : Decimal to hexadecimal conversion of fractional part by successive multiplication by 16.



(2A6)Fig. Ex. 1.17.5

$\therefore (0.24)_{10} = (0.3D70)_H$

Step III : Combine the answers obtained in Step I and Step II.

$(576.24)_{10} = (240.3D70)_H$

1.18 BINARY TO HEXADECIMAL CONVERSION

Step I : Write the given binary number.

▼

Step II : Starting from either side of the binary point, form groups of 4 bits.

▼

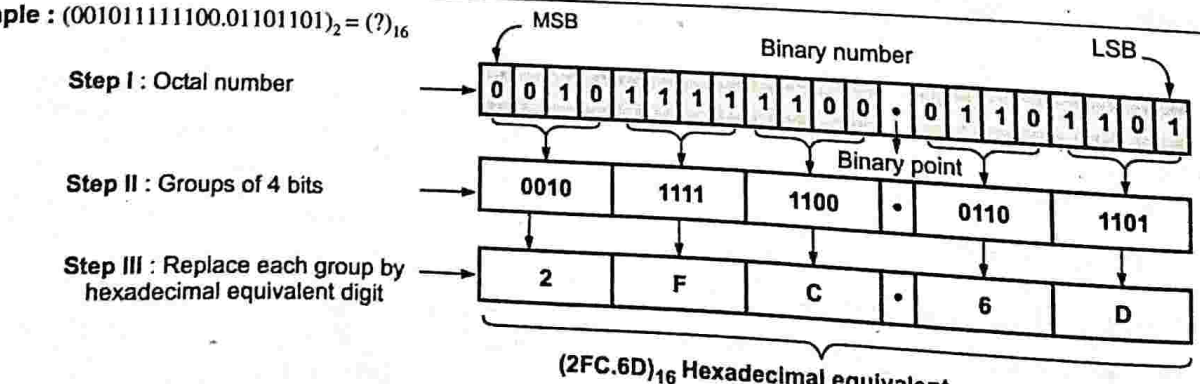
Step III : Replace each 4 bit binary group with its hexadecimal equivalent.

Hexadecimal numbers and their binary equivalent.

▼

Hexadecimal digit (Base 16)	4 bit binary equivalent (Base 2)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Example : $(001011111100.01101101)_2 = (?)_{16}$



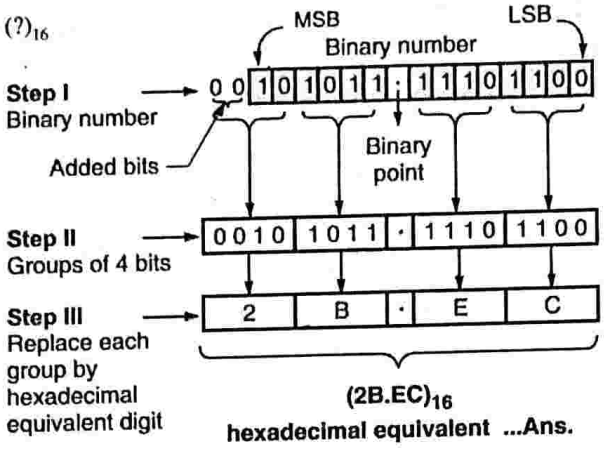
$(001011111100.01101101)_2 = (2FC.6D)_{16}$ ...Ans.

(1A68)Fig. 1.18.1

Ex. 1.18.1

Do the following : $(101011.111011)_2 = (?)_{16}$

☑ **Ans. :** (Refer Fig. Ex. 1.18.1)



(1A70)**Fig. Ex. 1.18.1**

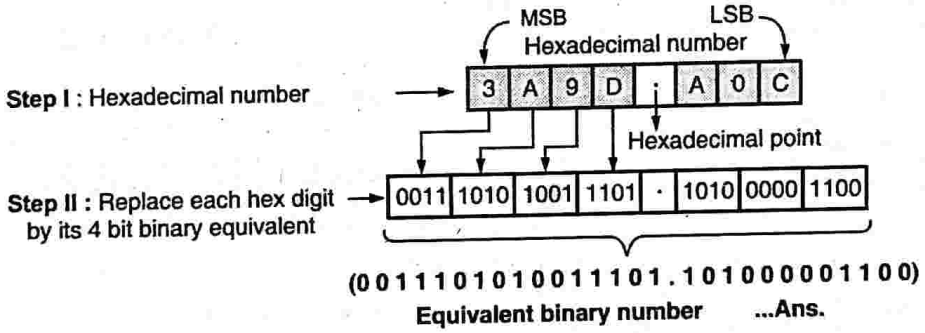
Thus, $(101011.111011)_2 = (2B.EC)_{16}$

1.19 HEXADECIMAL TO BINARY CONVERSION

Steps to be followed : (Refer Fig. 1.19.1)

➪ **Step I : Write the given hexadecimal number. e.g. 3A9D.A0C**

➪ **Step II : Replace each hexadecimal digit by its 4 bit binary equivalent.**



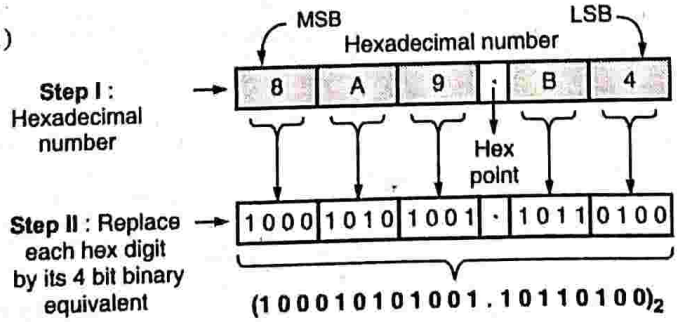
(1A71)**Fig. 1.19.1**

$(3A9D.A0C)_{16} = (0011101010011101.101000001100)_2$

Ex. 1.19.1

Convert $(8A9.B4)_{16}$ to binary.

☑ **Ans. :** (Refer Fig. Ex. 1.19.1)



(1A72)**Fig. Ex. 1.19.1**

Thus, $(8A9.B4)_{16} = (100010101001.10110100)_2$

1.20 OCTAL TO HEXADECIMAL CONVERSION

For converting an octal number to a hexadecimal number, we first convert the octal number to binary equivalent and then we convert the binary number to equivalent hexadecimal number.

Steps to be followed : (Refer Fig. 1.20.1)

Step I : Write the given octal number.
e.g. 423.613

▼

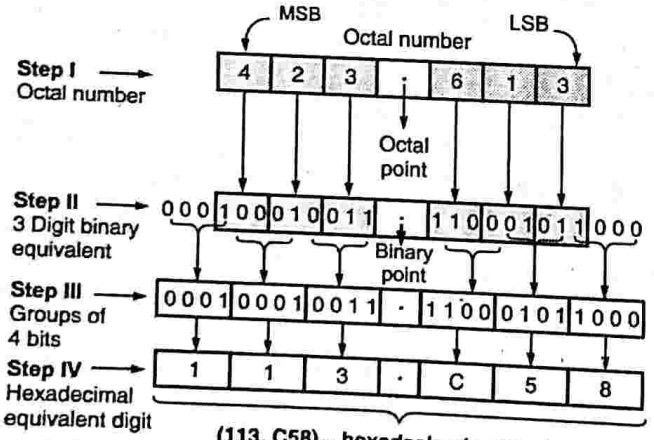
Step II : Replace each octal digit by its equivalent 3 bit binary number.

▼

Step III : Starting from either side of the binary point, form group of 4 bits.

▼

Step IV : Replace each 4 bit binary group with its hexadecimal equivalent.



$(113.C58)_{16}$ hexadecimal equivalent number ...Ans.

(1A75)**Fig. 1.20.1**

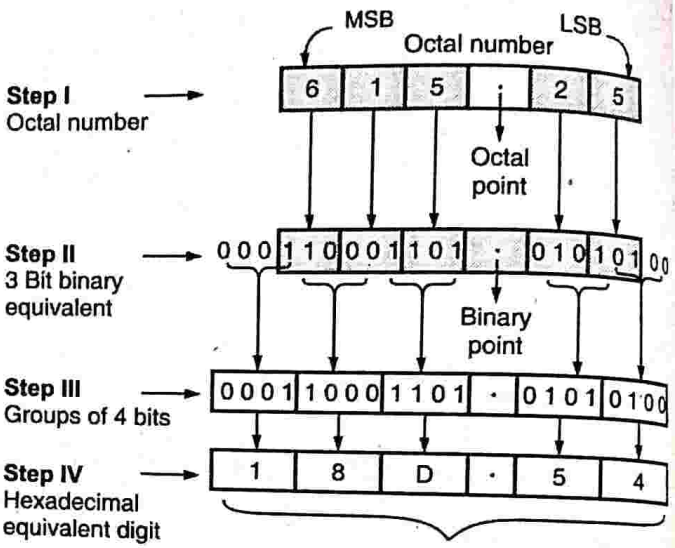
$(423.613)_8 = (113.C58)_{16}$

Ex. 1.20.1

Convert $(615.25)_8$ to hexadecimal.

☑ **Ans. :**

(Refer Fig. Ex. 1.20.1)



$(18D.54)_{16}$ hexadecimal equivalent number ...Ans.

(1A76)**Fig. Ex. 1.20.1**

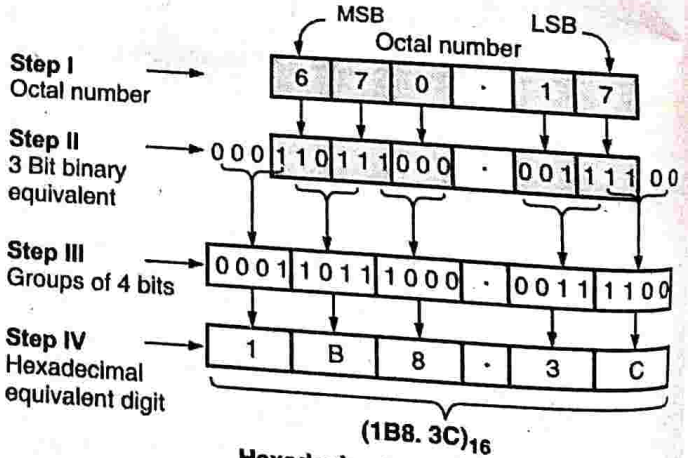
Thus, $(615.25)_8 = (18D.54)_{16}$

UEx. 1.20.2 **MU - Q. 1(a), Dec. 14, 2 Marks**

Convert $(670.17)_8$ into hexadecimal.

☑ **Ans. :**

(Refer Fig. Ex. 1.20.2)



$(1B8.3C)_{16}$ Hexadecimal equivalent number ...Ans.

(1A77)**Fig. Ex. 1.20.2**

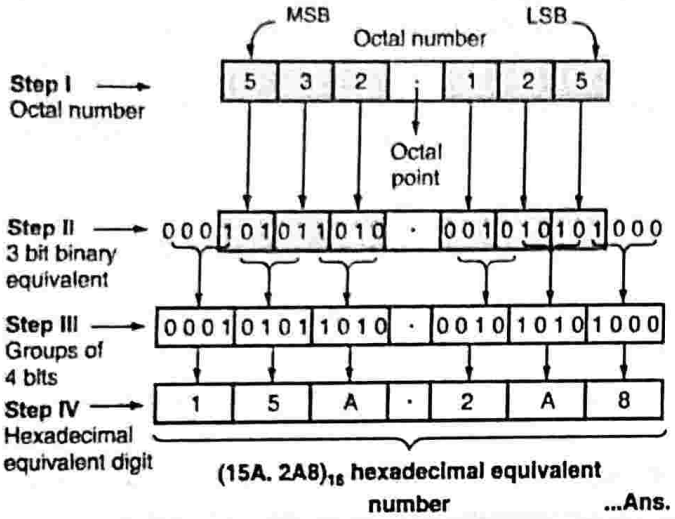
Thus, $(670.17)_8 = (1B8.3C)_{16}$

UEx. 1.20.3 MU - Q. 1(a), May 16, 1 Mark

Convert $(532.125)_8$ to hexadecimal.

Ans. :

(Refer Fig. Ex. 1.20.3)



(2A7)**Fig. Ex. 1.20.3 : Hexadecimal equivalent number**

Thus, $(532.125)_8 = (15a.2A8)_{16}$

1.21 HEXADECIMAL TO OCTAL CONVERSION

For converting a hexadecimal number to an octal number, we first convert the hexadecimal number to equivalent binary number and then we convert the binary number to equivalent octal number.

Steps to be followed : (Refer Fig. 1.21.1)

Step I : Write the given hexadecimal number

▼

Step II : Replace each hexadecimal digit by its equivalent 4 bit binary number.

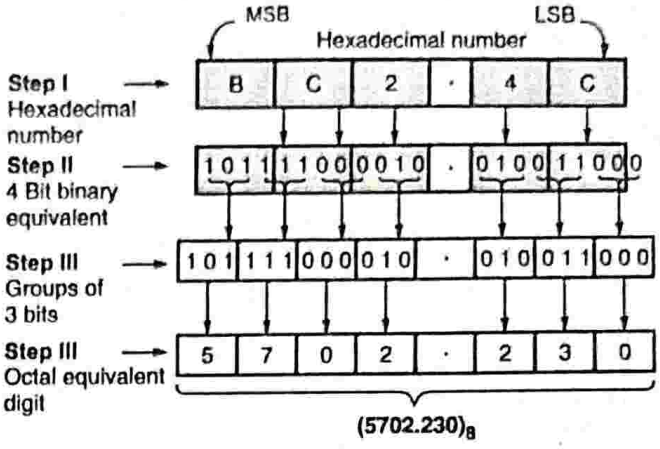
▼

Step III : Starting from either side of the binary point, form groups of 3 bits.

▼

Step IV : Replace each 3 bit group with its octal equivalent digit to obtain the octal equivalent number.

Example : BC2.4C



(1A78)**Fig. 1.21.1**

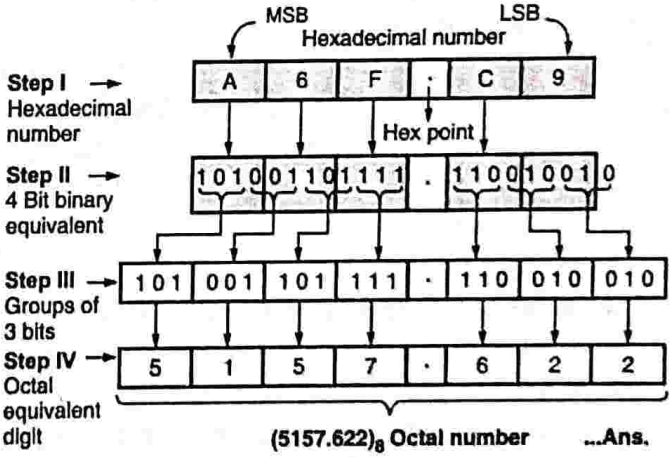
$(BC2.4C)_{16} = (5702.230)_8$

Ex. 1.21.1

Convert $(A6F.C9)_{16}$ into octal number.

Ans. :

(Refer Fig. Ex. 1.21.1)



(1A80)**Fig. Ex. 1.21.1**

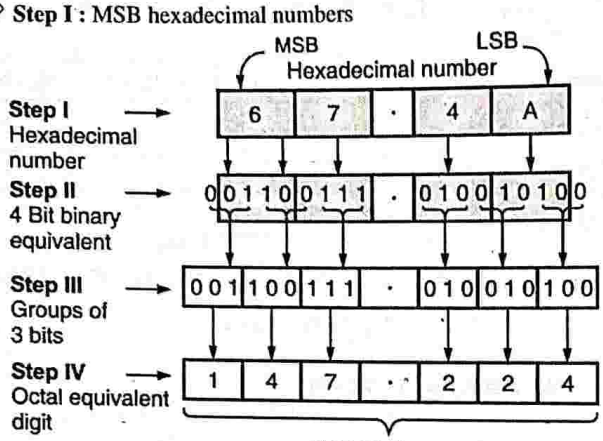
Thus, $(A6F.C9)_{16} = (5157.622)_8$

UEx. 1.21.2 **MU - Q. 1(b), Dec. 14, 2 Marks**

Convert the following hexadecimal number $(67.4A)_{16}$ into equivalent octal number.

Ans. :

Step I : MSB hexadecimal numbers



$(147.224)_8$

Octal equivalent number ...Ans.

(2A8)**Fig. Ex. 1.21.2**

Thus, $(67.4A)_{16} = (147.224)_8$

1.22 DECIMAL INTO RADIX R CONVERSION

- To convert a decimal number to an equivalent radix r, the integer part of the decimal number is converted to radix r by successive division by r and the fractional part of the decimal number is converted to radix r by using successive multiplication by r.
- **In the successive division by r,** the integer part is divided by r, till the quotient reaches zero. The remainder to the division is the MSB.
- To obtain the equivalent octal integer, the remainders are read from bottom to top.
- **In successive multiplication by r,** the fractional part of the decimal number is multiplied by r, till the fractional part of the product is zero.
- The equivalent radix r fractional part is obtained by reading the integers from top to bottom. i.e. MSB to LSB. Finally, the result is combined to obtain the equivalent radix r number.

Steps for Decimal to radix r Conversion for Integer part

Step I : Write the decimal number.

▼

Step II : Divide the decimal number by r. The remainder obtained is the LSB of the radix r number.

▼

Step III : Divide the quotient obtained from step II. The remainder obtained is the second LSB of the radix r number.

▼

Step IV : Repeat division by r till quotient becomes zero, i.e., quotient is no longer divisible by r.

▼

Step V : The last remainder obtained from the division is the MSB of the radix r number. The equivalent radix r number is read from bottom to top.

UEx. 1.22.1

MU - Q. 1(c), Dec. 14, 4 M, Q. 1(b), May 17, 2 M

Convert decimal (214.32) into base 7.

Ans. :

Step I : Decimal to base 7 conversion of integer part by successive division by 7.

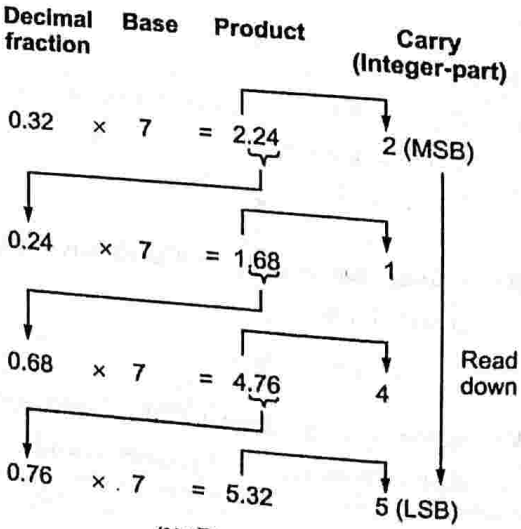
Base	Quotient	Remainder
7	214	
7	30	4
7	4	2
		4

(LSB) ↑ Read up (MSB)

$(214)_{10} = (424)_7$

Step II : Decimal to base 7 conversion of fractional part by successive multiplication by 7.

(Refer Fig. Ex. 1.22.1)



(2A9)**Fig. Ex. 1.22.1**

$\therefore (0.32)_{10} = (0.2145)_7$

➪ **Step III :** Combine the answers obtained in Step I and Step II.

$(214.32)_{10} = (424.2145)_7$

UEx. 1.22.2 MU - Q. 1(a), Dec. 18, 1 Mark

Convert decimal number 576.24 into base 9 system.

☑ **Ans. :**

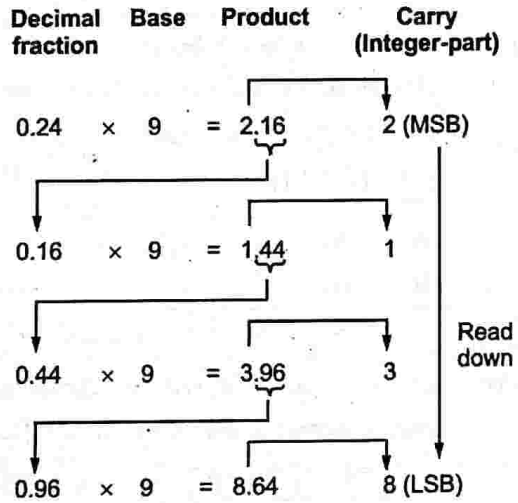
➪ **Step I :** Decimal to base 9 conversion of integer part by successive division by 9.

Base	Quotient	Remainder	
9	576		(LSB)
9	64	0	
9	7	1	Read up
		7	(MSB)

$(576)_{10} = (710)_9$

➪ **Step II :** Decimal to base 9 conversion of fractional part by successive multiplication by 9.

(Refer Fig. Ex. 1.22.2)



(2A10)**Fig. Ex. 1.22.2**

$\therefore (0.24)_{10} = (0.2138)_9$

➪ **Step III :** Combine the answers obtained in Step I and Step II.

$(576.24)_{10} = (710.2138)_9$

UEx. 1.22.3 MU - Q. 1(a), Dec. 16, 1 Mark

Convert decimal number 151.33 into base 4 system.

☑ **Ans. :**

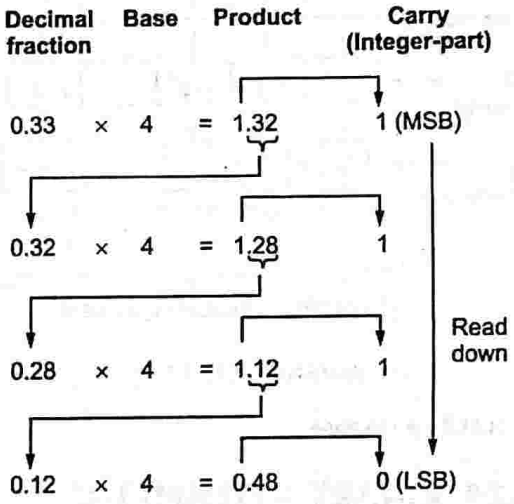
➪ **Step I :** Decimal to base 4 conversion of integer part by successive division by 4.

Base	Quotient	Remainder	
4	151		(LSB)
4	37	3	
4	9	1	
4	2	1	Read up
		2	(MSB)

$\therefore (151)_{10} = (2113)_4$

➪ **Step II :** Decimal to base 4 conversion of fractional part by successive multiplication by 4.

(Refer Fig. Ex. 1.22.3)



(2A11)**Fig. Ex. 1.22.3**

$\therefore (0.33)_{10} = (0.1110)_4$

➪ **Step III :** Combine the answers obtained in Step I and Step II.

$\therefore (151.33)_{10} = (2113.1110)_4$

⏭ 1.23 RADIX R TO DECIMAL CONVERSION

– **Steps to be followed** to convert a Radix r number to an equivalent decimal number are as follows :

Step I : Write the given number.

▼

Step II : Write the positional weights for each digit.

▼

Step III : Multiply each digit in the given number with its corresponding weight to get product of digits or positional value.

▼

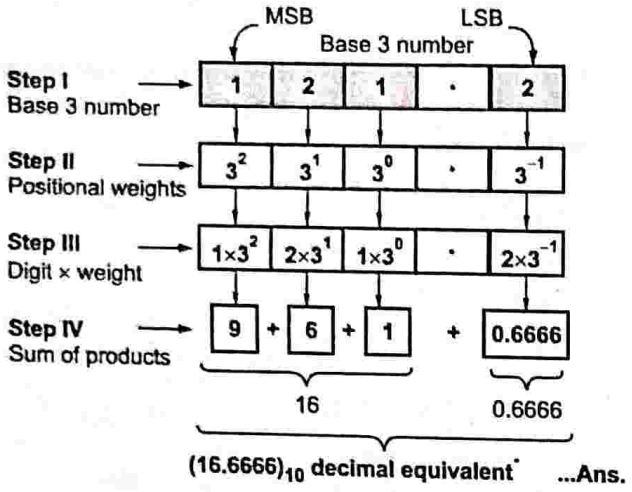
Step IV : Find the sum of products to obtain the decimal equivalent of the given radix r number.

UEx. 1.23.1 **MU - Q. 1(a), May 15, 2 Marks**

Convert $(121.2)_3$ into base 10.

Ans. :

- Refer Fig. Ex. 1.23.1



...Ans.

(2A12)Fig. Ex. 1.23.1

$\therefore (121.2)_3 = (16.6666)_{10}$

1.24 BINARY ARITHMETIC

- We will study the arithmetic operations like addition, subtraction, multiplication and division on binary numbers.
- Binary arithmetic is required for digital systems and in digital computers.
- As binary arithmetic comprises of two digits 0 and 1, it is much simple in comparison to decimal arithmetic.

1.24.1 Binary Addition

Table 1.24.1 shows the rules of binary addition.

Table 1.24.1 : Rules of binary addition

B_1		B_0	Sum	Carry
0	+	0	0	0
0	+	1	1	0
1	+	0	1	0
1	+	1	0	1

In the first three cases, there is no carry. The addition is similar to the decimal addition. In the fourth case, the addition of $1 + 1 = (2)_{10} = (10)_2$ i.e. the addition results in sum = 0 and carry = 1

Suppose we want to add two 4 bit binary numbers $(A_3\ A_2\ A_1\ A_0)$ and $(B_3\ B_2\ B_1\ B_0)$, where A_3 and B_3 are the most significant bits of the numbers and A_0 and B_0 are the least significant bits of the numbers. The addition of the two numbers is done as follows :

Step I : Add the LSBs A_0 and B_0. Record the sum as S_0 and carry as C_0 if any, is forwarded to the next column.

C_0 ← Carry

$A_3\ A_2\ A_1\ A_0$

$+\ B_3\ B_2\ B_1\ B_0$

S_0 ← Sum

From LSB addition i.e. addition of A_0 and B_0

Step II : Add the bits A_1 and B_1 and carry C_0 from previous addition. Record the sum as S_1 and carry C_1 if any, is forwarded to next column.

$C_1\ C_0$ ← Carry

$A_3\ A_2\ A_1\ A_0$

$+\ B_3\ B_2\ B_1\ B_0$

$S_1\ S_0$ ← Sum

Step III : Add the bits A_2 and B_2 and carry C_1 from previous addition. Record the sum as S_2 and carry C_2 is forwarded to the next column.

$C_2\ C_1\ C_0$ ← Carry

$A_3\ A_2\ A_1\ A_0$

$+\ B_3\ B_2\ B_1\ B_0$

$S_2\ S_1\ S_0$ ← Sum

Step IV : Add the bits A_3 and B_3 and carry C_2 from previous addition. Record the sum as S_3 and carry C_3.

$C_2\ C_1\ C_0$ ← Carry

$A_3\ A_2\ A_1\ A_0$

$+\ B_3\ B_2\ B_1\ B_0$

Carry → C_3 $S_3\ S_2\ S_1\ S_0$ ← Sum

Example : Add $(5)_{10}$ and $(3)_{10}$

$(5)_{10} \rightarrow (0101)_2$

$(3)_{10} \rightarrow (0011)_2$

[1] [1] [1] ←

$0\ 1\ 0\ 1$

$+\ 0\ 0\ 1\ 1$

[0] $1\ 0\ 0\ 0$ ← C_3

$C_2\ C_1\ C_0$

$A_3\ A_2\ A_1\ A_0$

$+\ B_3\ B_2\ B_1\ B_0$

$S_3\ S_2\ S_1\ S_0$

$(5)_{10} + (3)_{10} = (1000)_2 = (8)_{10}$

Ex. 1.24.1

Add the two binary numbers $(1101.101)_2$ and $(0101.100)_2$

Ans. :

		[1]		[1]	[1]				← Carry
		1	1	0	1	.	1	0	1 ← Number 1
	+	0	1	0	1	.	1	0	0 ← Number 2
Final Carry →	[1]	0	0	1	1	.	0	0	1 ← Sum

Final Sum

Thus, $(1101.101)_2 + (0101.100)_2 = (10011.001)_2$

Note

(1) In a column, every pair of 1's results in a carry that is forwarded to the next higher bit column.

(2) If the number of 1's that are to be added in a column is odd then the sum bit is one (1) while if the number of 1's to be added is even then the sum bit is zero (0).

Ex. 1.24.2

Perform the binary arithmetic $(11011.11)_2 + (11011.01)_2 = (?)_2$

Ans. :

Carry	→	[1]		[1]	[1]	[1]		[1]	
Number 1	→	1	1	0	1	1	.	1	1
Number 2	→	1	1	0	1	1	.	0	1
Final Sum →	[1]	1	0	1	1	1	.	0	0

↑ Final Carry

Thus, $(11011.11)_2 + (11011.01)_2 = (110111.00)_2$

Ex. 1.24.3

Perform the addition of given binary numbers 1000011 + 1110001

Ans. :

Carry	→					[1]	[1]	
Number 1	→		1	0	0	0	1	1
Number 2	→	+	1	1	1	0	0	0	1
Final Sum →	[1]		0	1	1	0	1	0	0

↑ Final Carry

Thus, $(1000011)_2 + (1110001)_2 = (10110100)_2$

Ex. 1.24.4

Perform the addition (1100010 + 1010001)

Ans. :

Carry	→							
Number 1	→		1	1	0	0	0	1	0
Number 2	→	+	1	0	1	0	0	0	1
Final Sum →	[1]		0	1	1	0	0	1	1

↑ Final Carry

Thus, $(1100010 + 1010001) = (10110011)_2$

Ex. 1.24.5

Solve

(i) $(1011011)_2 + (10010)_2 = (?)_2$

(ii) $(101101)_2 + (11001)_2 = (?)_2$

(iii) $(01000100.1100)_2 + (11100100.0111)_2 = (?)_2$

Ans. :

Carry	→			[1]			[1]		
Number 1	→		1	0	1	1	0	1	1
Number 2	→	+	0	0	1	0	0	1	0
Final Sum →	[0]		1	1	0	1	1	0	1

↑ Final Carry

Thus, $(1011011)_2 + (10010)_2 = (1101101)_2$

(ii) $(101101)_2 + (11001)_2 = (?)_2$

Carry	→		[1]	[1]			[1]	
Number 1	→		1	0	1	1	0	1
Number 2	→	+	0	1	1	0	0	1
Final Sum →	[1]		0	0	0	1	1	0

↑ Final carry

Thus, $(101101)_2 + (11001)_2 = (1000110)_2$

(iii) $(01000100.1100)_2 + (11100100.0111)_2 = (?)_2$

Carry →		[1]				[1]			[1]		[1]		
Number 1 →		0	1	0	0	0	1	0	0	.	1	1	0	0
Number 2 →	+	1	1	1	0	0	1	0	0	.	0	1	1	1
Final Sum →	[1]	0	0	1	0	1	0	0	1	.	0	0	1	1

↑ Final carry

Thus, $(01000100.1100)_2 + (11100100.0111)_2 = (100101001.0011)_2$

1.24.2 Binary Subtraction

- The binary subtraction is same as decimal subtraction. Table 1.24.2 lists the rules of binary subtraction.

Table 1.24.2 : Rules of Binary Subtraction

B_1		B_0	Difference	Borrow
0	–	0	0	0
0	–	1	1	1
1	–	0	1	0
1	–	1	0	0

- The outputs of binary subtraction are difference and borrow.
- Consider **first case :** $B_1 = 0, B_0 = 0$

  $\therefore 0 - 0 = 0$ i.e. difference = 0, borrow = 0
- Consider **second case :** $B_1 = 0, B_0 = 1$

  $\because$ Digit $B_0 > B_1$, we cannot subtract $(0 - 1)$.

Therefore, we have to borrow 2, i.e., $(10)_2$ as the base of binary numbers is 2.

It generates borrow = 1 and difference $(10)_2 - (1)_2 = (1)$ i.e. difference = 1, borrow = 1

- Consider **third case :** $B_1 = 1, B_0 = 1$

  $\therefore 1 - 0 = 1$ i.e. difference = 1, borrow = 0
- Consider **fourth case :** $B_1 = 1, B_0 = 1$

  $\therefore 1 - 1 = 0$ i.e. difference = 0, borrow = 0

Steps for performing binary subtraction for large numbers are as follows :

Step I : In the LSB column, start subtraction by subtracting the bit in the lower row from the bit in the upper row.

▼

Step II : With the help of binary subtraction rules perform the subtraction. If the bit in the upper row is less than the bit in the lower row, borrow 1 from the upper row of column on the left side. The result of subtraction of two bits is recorded as the difference.

▼

Step III : Repeat step II for every column till the MSB bit.

Examples

(1) Subtract $(7)_{10} - (5)_{10} = (0111)_2 - (0101)_2$

➪ **Step I :** Perform column by column subtraction from LSB to MSB using binary subtraction rules.

		MSB			LSB
Borrow →		[0]	[0]	[0]	
Number 1 →		0	1	1	1
Number 2 →	–	0	1	0	1
Difference →	[0]	0	0	1	0

↑ Final borrow

$(7)_{10} - (5)_{10} = (00010)_2 = (2)_{10}$

(2) Subtract $(0101)_2$ from $(0110)_2$

➪ **Step I :** Perform column by column subtraction from LSB to MSB using binary subtraction rules.

		MSB			LSB
Borrow →					[1]
Number 1 →		0	1	1	0
Number 2 →	–	0	1	0	1
				[–1]	
Difference →	[0]	0	0	0	1

↑ Final borrow

$(0110)_2 - (0101)_2 = (00001)_2$

Ex. 1.24.6

Perform $(1011.010)_2 - (0110.101)_2$

☑ **Ans. :**

➪ **Step I :** Perform column by column subtraction from LSB to MSB using binary subtraction rules.

		MSB							LSB
Borrow →			[1]				[1]		[1]
Number 1 →		1	0	1	1	.	0	1	0
Number 2 →	–	0	1	1	0	.	1	0	1
		[–1]			[–1]			[–1]	
Difference →	[0]	0	1	0	0	.	1	0	1

↑ Final borrow

Thus, $(1011.010)_2 - (0110.101)_2 = (00100.101)_2$

Ex. 1.24.7

Perform $(10101.100)_2 - (00111.000)_2$

☑ **Ans. :**

➪ **Step I :** Perform column by column subtraction from LSB to MSB using binary subtraction rules.

		MSB							LSB
Borrow →			1	1	1				
Number 1 →		1	0	1	0	1	· 1	0	0
Number 2 →	–	0	0	1	1	1	· 0	0	0
		–1	–1	–1					
Difference →	0	0	1	1	1	0	· 1	0	0
	↑ Final borrow								

Thus, $(10101.100)_2 - (00111.000)_2 = (001110.100)_2$

⏭ 1.25 REPRESENTATION OF SIGNED NUMBERS

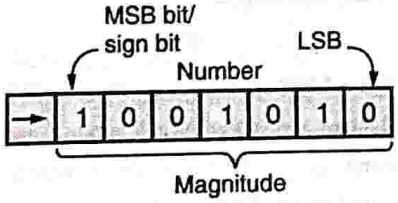
- A binary number can be positive or negative. The symbol "+" is used to represent positive numbers and the symbol "–" is used to represent negative numbers.
- In digital computers the sign of a binary number needs to be represented using 0 and 1.
- For **representing signed numbers** there are two methods :

(1)	Sign magnitude form	
(2)	Complement form	
	(a)	1's complement
	(b)	2's complement

1.25.1 Sign Magnitude Form

- An n-bit signed binary number comprises of two parts :
  - (i) Sign bit
  - (ii) Magnitude

Refer Fig. 1.25.1



If bit = 1, Positive number
If bit = 0, Negative number

(1A81)**Fig. 1.25.1 : n-bit signed binary number**

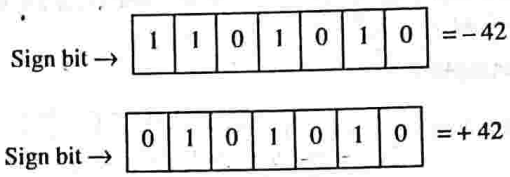
If sign bit = 0, the number is positive

If sign bit = 1, the number is negative

The remaining bits represent the magnitude of the number.

- An 8- bit signed binary number can represent data in the range – 127 to + 127. The MSB bit is the sign bit.

Example



Ex. 1.25.1

Convert $(– 124)_{10}$ to its equivalent sign magnitude form.

☑ **Ans. :**

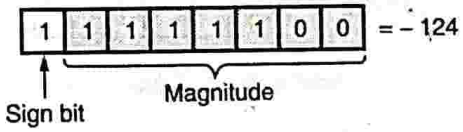
➪ **Step I :** To find the binary equivalent of the given number.

Base		Quotient	Remainder
2	124		
2	62	0	(LSB)
2	31	0	
2	15	1	Read up
2	7	1	
2	3	1	
2	1	1	
		1	(MSB)

$(124)_{10} = (1111100)_2$

➪ **Step II :** To represent $(– 124)_{10}$ in sign magnitude form.

∵ The number is negative, we will add a sign bit '1' to the leftmost bit of the number. (Refer Fig. Ex. 1.25.1)



(1A82)**Fig. Ex. 1.25.1 : – 124 in sign-magnitude form**

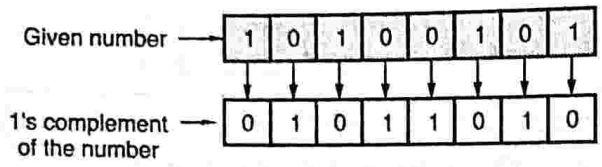
1.25.2 Representation of Signed Numbers using 1's Complement or 2's Complement Method

- For performing binary subtraction, the digital computers use 1's and 2's complement method.
- The main advantage of using complement method is the **reduction in hardware** and secondly, they allow the representation of negative numbers.

1.25.2(A) 1's Complement of a Binary Number

- The 1's complement of a binary number is obtained by **complementing every bit of the number i.e. a 0 is changed to 1 and vice-versa.** (Refer Fig. 1.25.2)

Example :



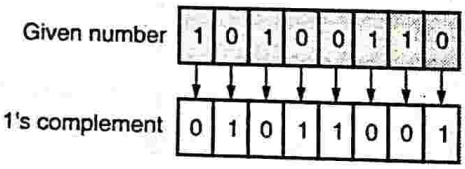
(1A83)**Fig. 1.25.2 : 1's complement of a number**

- It is called as **1's complement** because if we subtract the given number from 11111111 to get the result, it is same as complementing the number.

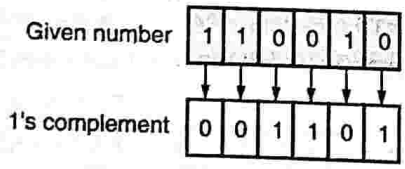
Ex. 1.25.2

Obtain the 1's complement of the following numbers.

☑ **Ans. :** (Refer Fig. Ex. 1.25.2(a) and (b))



(1A84)**Fig. Ex. 1.25.2(a)**

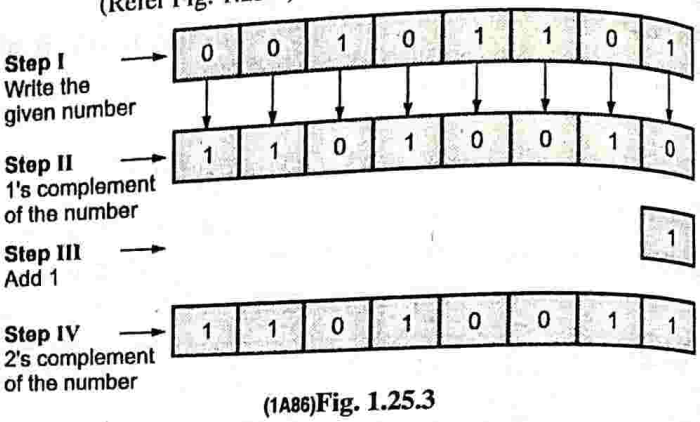


(1A85)**Fig. Ex. 1.25.2(b)**

1.25.2(B) 2's Complement of a Binary Number

- 2's complement arithmetic is used in computers for handling negative numbers. Instead of using separate circuits for binary addition and subtraction, only adding circuits, i.e., adders are used. The complement of the subtrahend is added to the minuend rather than subtracting the number,
- The 2's complement of a binary number can be obtained by :
  (1) Adding 1 to the LSB of 1's complement of that number.
  2's complement = 1's complement + 1
  (2) Beginning from the LSB, write down each bit up to and including the first one and then complement the remaining bits.
  (3) Subtracting the given n-bit number from 2^n.

Example : To find 2's complement of 00101101 (Refer Fig. 1.25.3)



(1A86)**Fig. 1.25.3**

1.25.3 Binary Subtraction using 1's Complement Method

- In binary subtraction using 1's complement method, we add 1's complement of the subtrahend to the minuend. If the addition results in a carry then the carry is added to the LSB of the number to get final result. If the MSB bit is 0, then the result is positive and in true binary form.
- If the MSB bit is 1, then the result is negative and in 1's complement form. Convert it to true binary form by taking 1's complement.

Steps to be followed for binary subtraction using 1's complement are as follows

Step I : Obtain the 1's complement of the subtrahend.

▼

Step II : Add minuend to the 1's complement of subtrahend using rules of binary addition.

▼

Step III : If carry is 1, then add it to LSB to get the result of binary subtraction.

▼

Step IV : If the MSB = 0, then result obtained in step III is positive and in true binary form. If the MSB = 1, the result obtained in step III is negative and in 1's complement form. Take its 1's complement to convert it to true binary form.

Ex. 1.25.3

Perform $(5)_{10} - (2)_{10}$ using 1's complement method.

Ans. :

Step I : Obtain 1's complement of subtrahend.

$(2)_{10} = (0010)_2$

1's complement of $(2)_{10} = (1101)_2$

Step II : Add $(5)_{10}$ to 1's complement of $(2)_{10}$

			[1]		[1]	
$(5)_{10}$	=		0	1	0	1
1's complement of $(2)_{10}$	=	+	1	1	0	1
		[1]	0	0	1	0

↑

End around carry

Step III : Add carry to the result obtained in step II.

0 0 1 0

\+ 1 (End around carry)

Step IV : Check MSB 0 0 1 1 ← Answer is positive and in true binary form

$(3)_{10}$

Thus, $(5)_{10} - (2)_{10} = (3)_{10}$

Ex. 1.25.4

Perform $(2)_{10} - (5)_{10}$ using 1's complement method.

Ans. :

Step I : To obtain 1's complement of the subtrahend.

$(5)_{10} = (0101)_2$

1's complement of $(5)_{10} = (1010)_2$

Step II : Add $(2)_{10}$ to 1's complement of $(5)_{10}$

		[1]			
$(2)_{10}$		0	0	1	0
1's complement of $(5)_{10}$	+	1	0	1	0
Result :		1	1	0	0

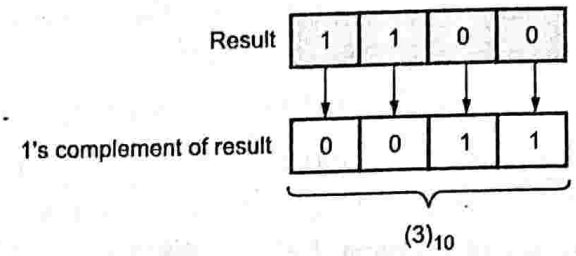
↑

MSB

The MSB = 1. Hence, the answer is negative and in 1's complement form.

Step III : Take 1's complement of the result to obtain the answer in true binary form.

(Refer Fig. Ex. 1.25.4)



(1A87)**Fig. Ex. 1.25.4**

Thus, $\therefore (2)_{10} = (5)_{10} = (-3)_{10}$

Ex. 1.25.5

Subtract $(33)_{10} - (44)_{10}$ using one's complement.

Ans. :

Step I : Obtain 1's complement of $(44)_{10}$

$(44)_{10} = (101100)_2$

1's complement of $(44)_{10} = (010011)_2$

Step II : Add $(33)_{10}$ to the 1's complement of $(44)_{10}$

				[1]	[1]		
$(33)_{10}$	=	1	0	0	0	0	1
	+						
+ 1's complement of $(44)_{10}$	=	0	1	0	0	1	1
Result	=	1	1	0	1	0	0

↑

MSB

MSB = 1, Hence, the answer is negative and in 1's complement form.

Step III : Take 1's complement of the result to obtain the answer in true binary form.

Result : 1 1 0 1 0 0

1's complement : $(0\ 0\ 1\ 0\ 1\ 1)_2$

$(11)_{10}$

Thus, $(33)_{10} - (44)_{10} = (-11)_{10}$

Ex. 1.25.6

Find the one's complement and two's complement of $(57)_{10}$

Ans. :

Step I : To obtain 1's complement of $(57)_{10}$

$(57)_{10} = (111001)_2$

1's complement of $(57)_{10} = (000110)_2$

➪ **Step II :** To find 2's complement of $(57)_{10}$

1's complement of $(57)_{10}$	=	0 0 0 1 1 0
Add $(1)_{10}$	+	1
2's complement of $(57)_{10}$	→	0 0 0 1 1 1

Thus, **2's Complement of $(57)_{10}$ = $(000111)_2$**

1.25.4 Binary Subtraction using 2's Complement Method

- The 2's complement method is used for representing negative numbers.
- In binary subtraction using 2's complement method, we add the minuend with 2's complement of the subtrahend. If carry is generated from the addition, **always ignore the carry.**
- If the MSB bit of the result is zero then the result is positive and in true binary form. If the MSB bit of the result is one (1), then the result is negative and in 2's complement form.
- To obtain the result in true binary form, take the 2's complement of the number.

Steps to be followed

Step I :	Add minuend to the 2's complement of the subtrahend.
▼	
Step II :	Ignore carry, if any. If MSB = 0, result is positive and in true binary form.
▼	
Step III :	If MSB = 1, result is negative and in 2's complement form. Compute 2's complement to obtain the result in true binary form.

Ex. 1.25.7

Perform the subtraction using 2's complement.

$(27)_{10} - (32)_{10}$

☑ **Ans. :**

➪ **Step I :** Convert the minuend and subtrahend to binary.

Base	Quotient	Remainder	
2	27		
2	13	1	(LSB)
2	6	1	
2	3	0	Read up
2	1	1	
		1	(MSB)

$(27)_{10}$ = $(11011)_2$

Base	Quotient	Remainder	
2	32		
2	16	0	(LSB)
2	8	0	
2	4	0	Read up
2	2	0	
2	1	0	
		1	(MSB)

$(32)_{10}$ = $(100000)_2$

➪ **Step II :** Obtain 2's complement of $(32)_{10}$

$(32)_{10}$ =	1	0	0	0	0	0
	1	1	1	1	1	
1's complement of $(32)_{10}$ +	0	1	1	1	1	1
Add 1 +						1
2's complement of $(32)_{10}$	1	0	0	0	0	0

➪ **Step III :** Add $(27)_{10}$ to 2's complement of $(32)_{10}$

$(27)_{10}$	→	0 1 1 0 1 1
2's complement of $(32)_{10}$ →	+	1 0 0 0 0 0
Result :		1 1 1 0 1 1 (No carry)

MSB

MSB = 1 indicates that result is negative and in 2's complement form.

➪ **Step IV :** Take 2's complement of the result.

Result :		1 1 1 0 1 1
1's complement Add 1	+	0 0 0 1 0 0
		1
Final Result :		0 0 0 1 0 1 = $(5)_{10}$

Thus, $(27)_{10} - (32)_{10} = (-5)_{10}$

UEx. 1.25.8 **MU - Q. 1(c), May 16, 4 Marks**

Perform the subtraction $(52)_{10} - (65)_{10}$ using 2's complement method.

Ans. :

Step I : To find the binary equivalent of $(52)_{10}$ and $(65)_{10}$

Base	Quotient	Remainder	
2	52		
2	26	0	(LSB)
2	13	0	
2	6	1	Read up
2	3	0	
2	1	1	
		1	(MSB)

$(52)_{10} = (110100)_2$

Base	Quotient	Remainder	
2	65		
2	32	1	(LSB)
2	16	0	
2	8	0	
2	4	0	Read up
2	2	0	
2	1	0	
		1	(MSB)

$(65)_{10} = (1000001)_2$

Step II : To obtain 2's complement of $(65)_{10}$

$(65)_{10} = 1\ 0\ 0\ 0\ 0\ 0\ 1$

1's complement of $(65)_{10} = 0\ 1\ 1\ 1\ 1\ 1\ 0$

Add 1 + 1

2's complement of $(65)_{10}$ $0\ 1\ 1\ 1\ 1\ 1\ 1$

Step III : Add $(52)_{10}$ to the 2's complement of $(65)_{10}$

Carry [1] [1] [1] [1]

$(52)_{10}$ + 0 1 1 0 1 0 0

2's complement of $(65)_{10}$ 0 1 1 1 1 1 1

Result : [0] [1] 1 1 0 0 1 1

no carry ← [0] ; [1] ← MSB

∵ MSB = 1, result is negative and in 2's complement form.

Step IV : Compute 2's complement to get the result is true binary form.

1's complement of result : 0 0 0 1 1 0 0

Add 1 + 1

2's complement of result : 0 0 0 1 1 0 1 $= (13)_{10}$

Thus, $(52)_{10} - (65)_{10} = (-13)_{10}$

UEx. 1.25.9 **MU - Q. 1(b), Dec. 13, 2.5 Marks**

Subtract the following using 2's complement $(11)_{10} - (22)_{10}$

Ans. :

Step I : Find the binary equivalent of $(11)_{10}$ and $(22)_{10}$

$(11)_{10} = (1011)_2$

$(22)_{10} = (10110)_2$

Base	Quotient	Remainder	
2	22		
2	11	0	(LSB)
2	5	1	
2	2	1	Read up
2	1	0	
		1	(MSB)

Step II : To find the 2's complement of $(22)_{10}$

$(22)_{10} = (1\ 0\ 1\ 1\ 0)_2$

[1]

1's complement of $(22)_{10} = (0\ 1\ 0\ 0\ 1)$

Add 1 + 1

2's complement of $(22)_{10} \rightarrow 0\ 1\ 0\ 1\ 0$

Step III : Add $(11)_{10}$ to the 2's complement of $(22)_{10}$

Carry → [1] [1]

$(11)_{10} \rightarrow$ 0 1 0 1 1

2's complement $(22)_{10} \rightarrow$ + 0 1 0 1 0

Result → [0] [1] 0 1 0 1

No Carry MSB

∵ MSB = 1, result is negative and in 2's complement form.

➪ **Step IV :** Take 2's complement of the result

		1	0	1	0	1
Result :						
1's complement of Result :	0	1	0	1	0	
Add 1 +					1	
Final Result :	0	1	0	1	1	$= (11)_{10}$

$\therefore (11)_{10} - (22)_{10} = (-11)_{10}$

UEx. 1.25.10 **MU - Q. 1(i), May 15, 4 Marks**

Perform binary subtraction using 2's complement for $(62)_{10}$ and $(99)_{10}$

☑ **Ans. :**

➪ **Step I :** Find the binary equivalent of $(62)_{10}$ and $(99)_{10}$

Base	Quotient	Remainder	
2	62		
2	31	0	(LSB)
2	15	1	
2	7	1	
2	3	1	Read up
2	1	1	
		1	(MSB)

$(62)_{10} = (111110)_2$

Base	Quotient	Remainder	
2	99		
2	49	1	(LSB)
2	24	1	
2	12	0	
2	6	0	Read up
2	3	0	
2	1	1	
		1	(MSB)

$(99)_{10} = (1100011)_2$

➪ **Step II :** To find 2's complement of $(99)_{10}$

$(99)_{10} \rightarrow$	1	1	0	0	0	1	1
1's complement of $(99)_{10} \rightarrow$	0	0	1	1	1	0	0
Add $(1)_{10}$ +							1
2's complement of $(99)_{10} \rightarrow$	0	0	1	1	1	0	1

➪ **Step III :** Add $(62)_{10}$ to 2's complement of $(99)_{10}$

	[1]	[1]	[1]	[1]			
Carry →							
$(62)_{10} \rightarrow$	0	1	1	1	1	1	0
2's complement of $(99)_{10}$ +	0	0	1	1	1	0	1
Result → [0]	[1]	0	1	1	0	1	1

↑

No carry MSB

$\because$ MSB = 1, result is negative and in 2's complement form.

➪ **Step IV :** Convert the answer in true binary form by taking 2's complement of the result.

Result →	1	0	1	1	0	1	1
1's complement of result →	0	1	0	0	1	0	0
Add 1 +							1
Final Result →	0	1	0	0	1	0	1

$= (37)_{10}$

$\therefore (62)_{10} - (99)_{10} = (-37)_{10}$

UEx. 1.25.11 **MU - Q. 1(c), May 16, 4 Marks**

Subtract using 1's and 2's complement method.

$(56)_{10} - (76)_{10}$

☑ **Ans. :**

➪ **Step I :** To obtain the binary equivalent of $(56)_{10}$ and $(76)_{10}$

Base	Quotient	Remainder	
2	56		
2	28	0	(LSB)
2	14	0	
2	7	0	Read up
2	3	1	
2	1	1	
		1	(MSB)

$(56)_{10} = (111000)_2$

Base	Quotient	Remainder
2	76	
2	38	0
2	19	0
2	9	1
2	4	1
2	2	0
2	1	0
		1

(LSB) ↑ Read up ↑ (MSB)

$(76)_{10} = (1001100)_2$

➪ **Step II :** To obtain 1's complement and 2's complement of $(76)_{10}$

$(76)_{10} \rightarrow$ 1 0 0 1 1 0 0

[1] [1]

1's complement of $(76)_{10} \rightarrow$ 0 1 1 0 0 1 1

Add 1 + 1

2's complement of $(76)_{10} \rightarrow$ 0 1 1 0 1 0 0

➪ **Step III :** Add $(56)_{10}$ to 1's complement of $(76)_{10}$

Carry → [1] [1]

$(56)_{10} \rightarrow$ 0 1 1 1 0 0 0

1's complement of $(76)_{10}$ + 0 1 1 0 0 1 1

Result → [0] [1] 1 0 1 0 1 1

↑ ↑

No carry MSB

Since MSB = 1, result is negative and in 1's complement form. To get result in true binary form, take 1's complement of the result.

Result : 1 1 0 1 0 1 1

1's complement of result : 0 0 1 0 1 0 0 $= (20)_{10}$

Thus,

$(56)_{10} - (76)_{10} = (-20)_{10}$ using 1's complement method.

➪ **Step IV :** Add $(56)_{10}$ to 2's complement of $(76)_{10}$

Carry → [1] [1]

$(56)_{10} \rightarrow$ 0 1 1 1 0 0 0

1's complement of $(76)_{10}$ + 0 1 1 0 1 0 0

Result → [0] [1] 1 0 1 1 0 0

↑ ↑

No carry MSB = 1

Since, MSB = 1, result is negative and in 2's complement form. To obtain the result in true binary form, take 2's complement of the result.

Result : 1 1 0 1 1 0 0

[1] [1]

1's complement of Result : 0 0 1 0 0 1 1

Add 1 + 1

2's complement of Result → 0 0 1 0 1 0 0 $= (20)_{10}$

Thus,

$(56)_{10} - (76)_{10} = (-20)_{10}$ using 2's complement method.

UEx. 1.25.12 MU - Q. 1(d), May 18, 4 Marks

Subtract using 1's and 2's method : $(15)_{10} - (21)_{10}$

☑ **Ans. :**

➪ **Step I :** To obtain the binary equivalent of $(15)_{10}$ and $(21)_{10}$

$(15)_{10} = (1111)_2$

Base	Quotient	Remainder
2	21	
2	10	1
2	5	0
2	2	1
2	1	0
		1

(LSB) ↑ Read up ↑ (MSB)

$(21)_{10} = (10101)_2$

➪ **Step II :** To find the 1's and 2's complement of $(21)_{10}$

$(21)_{10}$: 1 0 1 0 1

1's complement of $(21)_{10}$: 0 1 0 1 0

+

Add 1 : 1

2's complement of $(21)_{10} \rightarrow$ 0 1 0 1 1

➪ **Step III :** Add $(15)_{10}$ to 1's complement of $(21)_{10}$

Carry → [1] [1] [1]

$(15)_{10} \rightarrow$ 0 1 1 1 1

1's complement of $(21)_{10} \rightarrow$ + 0 1 0 1 0

Result → [0] [1] 1 0 0 1

No carry ↑ MSB

Since MSB = 1, result is negative and in 1's complement form. Take 1's complement of the result to obtain result in true binary form.

Result : 1 1 0 0 1

1's complement of Result : 0 0 1 1 0 = $(6)_{10}$

Thus,

$(15)_{10} - (21)_{10} = (-6)_{10}$ using 1's complement method.

➪ **Step IV :** Add $(15)_{10}$ to 2's complement of $(21)_{10}$

		1	1	1	1
Carry ⟶					
$(15)_{10}$ ⟶		0	1	1	1	1
2's complement of $(21)_{10}$ ⟶	+	0	1	0	1	1
Result ⟶	0	1	1	0	1	0
	↑ No carry	↑ MSB				

Since MSB = 1, result is negative and in 2's complement form. To obtain the result in true binary form, take the 2's complement.

Result : 1 1 0 1 0

1

1's complement of Result : 0 0 1 0 1

Add 1 + 1

2's complement of Result : 0 0 1 1 0 = $(6)_{10}$

$\therefore (15)_{10} - (21)_{10} = (-6)_{10}$ using 2's complement method.

UEx. 1.25.13 MU - Q. 1(c), Dec. 18, 4 Marks

Convert $(-89)_{10}$ to its equivalent sign magnitude 1's complement and 2's complement form.

☑ **Ans. :**

➪ **Step I :** To convert $(89)_{10}$ into binary by successive division by 2.

Base	Quotient	Remainder	
2	89		
2	44	1	(LSB)
2	22	0	
2	11	0	
2	5	1	Read up
2	2	1	
2	1	0	
	↓	1	(MSB)

$(89)_{10} = (1011001)_2$

$(-89)_{10} = (11011001)_2$

↑

Sign bit

➪ **Step II :** To represent $(-89)_{10}$ in its equivalent sign magnitude, 1's complement and 2's complement form.

Number	Sign magnitude representation	1's complement representation	2's complement representation
– 89	11011001	10100110	10100111

1.26 BINARY MULTIPLICATION

The rules of binary multiplication are :

(i) $0 \times 0 = 0$

(ii) $0 \times 1 = 0$

(iii) $1 \times 0 = 0$

(iv) $1 \times 1 = 1$

Ex. 1.26.1

Perform the following binary multiplication.

(i) 1011×0101 (ii) 1001.001×1010.101

(iii) 101.01×11.01

☑ **Ans. :**

(i) 1011×0101

```
        1 0 1 1
      × 0 1 0 1
      ---------
        1 0 1 1
      0 0 0 0
    1 0 1 1
0 0 0 0
---------------
0 1 1 0 1 1 1
---------------
```

(ii) 1001.001×1010.101

```
          1 0 0 1 . 0 0 1
        × 1 0 1 0 . 1 0 1
        -----------------
          1 0 0 1 0 0 1
        0 0 0 0 0 0 0
      1 0 0 1 0 0 1
    0 0 0 0 0 0 0
  1 0 0 1 0 0 1
0 0 0 0 0 0 0
1 0 0 1 0 0 1
-------------------------
1 1 0 0 0 0 0 . 1 1 1 1 0 1
-------------------------
```

(iii) 101.01 × 11.01

$$\begin{array}{rrrrrrrrr} & & 1 & 0 & 1 & \cdot & 0 & 1 & \\ & \times & & 1 & 1 & \cdot & 0 & 1 & \\ \hline & & & 1 & 0 & 1 & 0 & 1 & \\ & & 0 & 0 & 0 & 0 & 0 & & \\ & 1 & 0 & 1 & 0 & 1 & & & \\ 1 & 0 & 1 & 0 & 1 & & & & \\ \hline 1 & 0 & 0 & 0 & 1 & \cdot 0 & 0 & 0 & 1 \\ \hline \end{array}$$

1.26.1 Binary Division

The rules of binary division are :

(i) $0 \div 1 = 0$ (ii) $1 \div 1 = 1$

Ex. 1.26.2

Perform the following binary division.

(1) $1001 \div 10$ (2) $(11.11)_2 \div (11)_2$

(3) $1100 \div 11$

Ans. :

(1) $1001 \div 10$

$$\begin{array}{r|l} & 100.1 \\ \hline 10 & 1001 \\ & -10 \\ & \hline & 0010 \\ & -10 \\ & \hline & 000 \end{array}$$

$1001 \div 10 = 100.1$

(2) $(11.11)_2 \div (11)_2$

$$\begin{array}{r|l} & 1.01 \\ \hline 11 & 11.11 \\ & -11 \\ & \hline & 0011 \\ & -11 \\ & \hline & 00 \end{array}$$

$11.11 \div 11 = 1.01$

(3) $1100 \div 11$

$$\begin{array}{r|l} & 100 \\ \hline 11 & 1100 \\ & -11 \\ & \hline & 0000 \end{array}$$

$1100 \div 11 = 100$

1.27 OCTAL ARITHMETIC

1.27.1 Octal Addition

- Octal addition is same as the decimal or binary addition.
- If the sum exceeds 7, while performing the addition, a carry is generated.

Ex. 1.27.1

Perform the octal addition of $(5)_8 + (6)_8$.

Ans. :

Step I : Add the numbers by assuming them to be decimal

$\therefore (5) + (6) = (11)_{10}$

Step II : The result is greater than 8.

$\therefore$ We will subtract 8 from the result $(11)_{10} - (8)_{10} = 3$ and generate carry = 1

$$\begin{array}{rrl} & (5)_8 & \\ \therefore & + \quad (8)_8 & \\ \hline \text{Carry} \rightarrow & \boxed{1} \quad 3 & \textbf{Result} \\ \hline \end{array}$$

$\therefore (5)_8 + (6)_8 = (13)_8$

UEx. 1.27.2 MU - Q. 1(f)(i), May 16, 2 Marks

Perform the following operations without changing the base.

(i) $(314)_8 + (737)_8$

Ans. :

Whenever column addition is greater than or equal to 8, subtract 8 and generate carry.

$$\begin{array}{rrrrl} & 3 & 1 & 4 & \\ + & 7 & 3 & 7 & \\ \text{Carry} & & \boxed{1} & & \\ \hline & 10 & 4 & 11 & \\ - & 8 & & 8 & \text{Subtract 8 and generate carry} \\ \hline \text{Carry} \rightarrow \boxed{1} & 2 & 5 & 3 & \textbf{Result} \\ \hline \end{array}$$

$\therefore (314)_8 + (737)_8 = (1253)_8$

1.27.2 Octal Subtraction

- Octal subtraction is same as decimal or binary subtraction.
- Octal subtraction can also be done by converting the octal number to binary and then using rules of binary subtraction perform the subtraction.
- Octal subtraction can also be done using 7's complement and 8's complement method.

1.27.3 Octal Subtraction using 7'S Complement Method

- The 7's complement of an octal number is obtained by subtracting each digit from 7.
- The steps for octal subtraction using 7's complement method are as follows :

Step I : Find 7's complement of the subtrahend

▼

Step II : Add two octal numbers.

▼

Step III : If the addition generates a carry, the result is positive. Add the carry in the least significant bit of the sum. If the addition does not generate carry. Take the 7's complement of the result to obtain the result in true form. The result is negative and in 7's complement form.

Ex. 1.27.3

Perform the octal subtraction of

(i) $(73)_8 - (65)_8$ (ii) $(653)_8 - (177)_8$

(iii) $(25)_8 - (73)_8$

☑ **Ans. :**

(i) $(73)_8 - (65)_8$

➪ **Step I :** Find 7's complement of subtrahend.

```
  77
- 65
----
  12   7's complement of (65)8
```

➪ **Step II :** Add the two octal numbers

➪ **Step III :** Add carry to LSB bit of sum.

```
      7 3
    +
      1 2    7's complement of (65)8
    -------
      8 5
    -
      8      <-- (∵ Column addition is greater than 8,
    -------       we subtract 8 and generate carry)
Carry -> 1 0 5
        +
           1   Add carry to LSB bit of sum
    -------
        0 6    Result
```

$\therefore (73)_8 - (65)_8 = (06)_8$

(ii) $(653)_8 - (177)_8$

➪ **Step I :** Find 7's complement of the subtrahend.

```
  777
- 177
-----
  600   7's complement of (177)8
```

➪ **Step II :** Add the two octal numbers

```
      6 5 3
    +          7's complement of 600
      6 0 0
    -------
    1 2 5 3
    -
        8      <-- (∵ Column addition is greater than 8,
    -------       we subtract 8 and generate carry)
Carry -> 1 4 5 3  Result
```

➪ **Step III :** Add carry to the LSB bit of the sum.

```
      4 5 3
    +
        1      <-- carry
    -------
      4 5 4    Result
```

$\therefore (653)_8 - (177)_8 = (454)_8$

(iii) $(25)_8 - (73)_8$

➪ **Step I :** Find 7's complement of the subtrahend.

```
  77
- 73
----
  04   7's complement of (73)8
```

➪ **Step II :** Add the two octal numbers.

```
    2 5
  + 0 4
  -----
    1
```

```
    2 9   (∵ column addition is greater than 8,
  -   8   subtract 8 and generate carry)
  -----
    3 1
```

➪ **Step III :** As there no carry, result is negative and in 7's complement form. We need to find the 7's complement of the result, to get answer in true form.

```
  77
- 31
----
  46   Result
```

$\therefore (25)_8 - (73)_8 = (-46)_8$

1.27.4 Octal Subtraction using 8's Complement Method

- The 8's complement of an octal number is obtained by adding 1 to the 7's complement of the octal number.
- The steps for performing octal subtraction using 8's complement method are :

Step I : Find the 8's complement of the subtrahend.

▼

Step II : Add the two octal numbers.

▼

Step III : If carry is produced, it indicates that the result is positive and true form. Discard the carry. If carry is not produced, the result is negative and in 8's complement form. Find the 8's complement of the result to express the result in true form.

Ex. 1.27.4

Use 8's complement method to compute $(360)_8 - (715)_8$

Ans. :

Step I : To find 8's complement of the subtrahend.

```
  777
 -715
 ----
  062   7's complement of subtrahend
+   1
 ----
  063   8's complement of (715)8
```

Step II : Add the two octal numbers.

```
    3  6  0
+      6  3
 ----------
  1
    3 12  3
-      8
 ----------
    4  4  3
```

(∵ column addition is greater than 8, subtract 8 and generate carry)

Step III : As there is no carry, result is negative and in 8's complement form. Find 8's complement of the result.

```
  777
 -443
 ----
  334   7's complement of the result.
+   1
 ----
  335   8's complement of the result.
```

$\therefore (360)_8 - (715)_8 = (-335)_8$

Ex. 1.27.5

Use 8's complement method to compute $(243)_8 - (153)_8$.

Ans. :

Step I : To find 8's complement of $(153)_8$

```
  777
 -153
 ----
  624   7's complement of (153)8
+   1
 ----
  625   8's complement of (153)8
```

Step II : Add the two octal number.

```
    2  4  3
+   6  2  5
 ----------
  1
    8  6  8
-      8  8   <-- (∵ Column addition is equal or greater than 8, subtract 8 or generate carry)
 ----------
  1  0  7  0   Result
Discard carry
```

Step III : As carry is generated, result is positive and true form. Discard carry.

$\therefore$ **Result** $= (243)_8 - (153)_8 = (70)_8$

1.27.5 Octal Multiplication

UEx. 1.27.6 **MU - Q. 2(a) Dec. 13, 2.5 Marks**

Perform the following directly without converting to any base.

(i) $(63)_8 * (21)_8$

Ans. :

```
          6  3
      ×   2  1
 ----------------
+         6  3
     12   6
 ----------------
  1
     12  12  3
-     8   8
 ----------------
  1   5   4  3   Result
```

∵ Column addition is greater than 8, subtract 8 and generate carry

$\therefore (63)_8 \times (21)_8 = (1543)_8$

UEx. 1.27.7 **MU - Q. 1(a) May 14, 2.5 Marks**

Perform the following without converting into other base.

(i) $(57)_8 * (24)_8$

Ans. :

```
          5   7
      ×   2   4
 ----------------
         20  28
     10  14
 ----------------
     4   3
         10  34  28
     -    8  32  24
 ----------------
Result 1     6   5   4
```

- We find the multiple closest to each digit, for 28 the closest multiple will be $8 \times 3 = 24$, so subtract 28 by 24 to get 4 and carry the 3 over to second digit.
- For 34, the closest multiple will be $8 \times 4 = 32$, so subtract 34 by 32 to get 2 plus previous carry 3 to get 5 and carry 4 over to third digit.
- For 10 the closest multiple is $8 \times 1 = 8$, so subtract 8 to get 2 plus previous carry 4 to get 6 and carry over 1 to the MSB digit.

$\therefore (57)_8 \times (24)_8 = (1654)_8$

1.28 HEXADECIMAL ARITHMETIC

1.28.1 Hexadecimal Addition

- Hexadecimal addition is similar to decimal, binary and octal addition.
- If the sum exceeds F i.e. $(15)_{10}$, a carry is generated for each digit position.

Ex. 1.28.1

Perform the hexadecimal addition of

(i) $(8)_{16} + (9)_{16}$.

Ans. :

Step I : $(8)_{16} + (9)_{16} = (17)_{10}$

As the result is greater, than 16, we need to perform two modifications,

(i) Subtract $(16)_{10}$ from the result : $(17)_{10} - (16)_{10} = (1)_{10}$

(ii) Generate carry

Step II :

$$\begin{array}{r} 9 \\ +\ 8 \\ \hline \boxed{1}\ 1 \end{array}$$

$\therefore (9)_{16} + (8)_{16} = (11)_{16}$

UEx. 1.28.2 MU - Q. 1(g) Dec. 14, 2 Marks

Add $(DDCC)_{16}$ and $(BBAA)_{16}$

Ans. :

D D C C →	$(13)_{10}$	$(13)_{10}$	$(12)_{10}$	$(12)_{10}$	
+ B B A A →	$(11)_{10}$	$(11)_{10}$	$(10)_{10}$	$(10)_{10}$	
	1	1	1		
Carry	$(24)_{10}$	$(24)_{10}$	$(22)_{10}$	$(22)_{10}$	
−	$(16)_{10}$	$(16)_{10}$	$(16)_{10}$	$(16)_{10}$	←
Carry →	1 9	9	7	6	Result

(∵ Sum greater than 16, subtract 16 and generate carry)

$\therefore (D D C C)_{16} + (B B A A)_{16} = (19976)_{16}$

UEx. 1.28.3 MU - Q. 1(b) Dec. 15, 2 Marks

Perform hexadecimal arithmetic operation : DADA + BABA

Ans. :

D A C A →	$(13)_{10}$	$(10)_{10}$	$(13)_{10}$	$(10)_{10}$	
+ B A B A →	$(11)_{10}$	$(10)_{10}$	$(11)_{10}$	$(10)_{10}$	
	1	1	1		
Carry	$(24)_{10}$	$(20)_{10}$	$(24)_{10}$	$(20)_{10}$	
−	$(16)_{10}$	$(16)_{10}$	$(16)_{10}$	$(16)_{10}$	←
Carry →	1 9	5	9	4	Result

(∵ Sum greater than 16, subtract 16 and generate carry)

$\therefore (D A C A)_{16} + (B A B A)_{16} = (19594)_{16}$

1.28.2 Hexadecimal Subtraction

- In the direct subtraction of hexadecimal numbers, if the minuend is less than the subtrahend we borrow (16) and return carry.

Ex. 1.28.4

Subtract $(73)_{16} - (1C)_{16}$

Ans. :

	16	}	$(19)_{10}$
7	3		
− 1	C	}	$(12)_{10}$
Borrow →	1		
5	7		Result

$\therefore (73)_{16} - (1C)_{16} = (57)_{16}$

Ex. 1.28.5

Perform $(BC5)_H - (A2B)_H$

Ans. :

Without converting to any other base.

			$(16)_{10}$	} $(21)_{10}$
B C 5 →	$(11)_{10}$	$(12)_{10}$	$(5)_{10}$	
− A 2 B →	$(10)_{10}$	$(02)_{10}$	$(11)_{10}$	
Borrow		1		
	1	9	A	Result

$\therefore (BC5)_H - (A2B)_H = (19A)_H$

Ex. 1.28.6

Perform the following directly without converting of any other base : $(D9)_H - (80)_H$

☑ **Ans. :**

D9 ⟶ $(13)_{10}$ $(9)_{10}$

−

80 ⟶ $-(8)_{10}$ $(0)_{10}$

Result 5 9

∴ $(D9)_{H} - (80)_{H} = (59)_{H}$

1.28.3 Hexadecimal subtraction using 15's complement

- The 15's complement of a hexadecimal number is obtained by subtracting each digit from 15.
- The steps to be followed for hexadecimal subtraction using 15's complement are as follows.

Step I : Find the 15's complement of the subtrahend.

▼

Step II : Add the two hexadecimal numbers.

▼

Step III : If carry is generated in the addition, the result is positive and is true form. Add carry to the LSB of the sum. If carry is not generated in the addition, the result is negative and in 15's complement form. Find 15's complement of the result to get the result in true form.

Ex. 1.28.7

Compute the hexadecimal subtraction of $(C2A)_{16} - (921)_{16}$.

☑ **Ans. :**

➪ **Step I :** To find 15's complement of the subtrahend.

15 15 13

− 9 2 1

6 13 14 15's complement of subtrahend

➪ **Step II :** Add the two hexadecimal numbers.

C 2 A $(12)_{10}$ $(02)_{10}$ $(10)_{10}$

6 D E $(06)_{10}$ $(13)_{10}$ $(14)_{10}$

Carry [1] [1]

$(18)_{10}$ $(15)_{10}$ $(24)_{10}$

$-(16)_{10}$ $(16)_{10}$ $(16)_{10}$ (∵ sum greater than 16, subtract 16 and generate carry)

[1] 3 0 8

↑ carry

➪ **Step III :** As carry is generated, result is positive and in true. Add carry to LSB digit of result.

3 0 8

\+ [1] ← carry

3 0 9 Result

∴ $(C2A)_{16} - (921)_{16} = (309)_{16}$

Ex. 1.28.8

Use 15's complement method to find $(69B)_{16} - (C14)_{16}$

☑ **Ans. :**

➪ **Step I :** To find 15's complement of $(C14)_{16}$.

15 15 15

\+ C 1 4

3 E B 15's complement of subtrahend

➪ **Step II :** Add the two hexadecimal numbers.

6 9 B ⟶ $(06)_{10}$ $(09)_{10}$ $(11)_{10}$

6 D E ⟶ $+(03)_{10}$ $(14)_{10}$ $(11)_{10}$

[1] [1]

$(09)_{10}$ $(23)_{10}$ $(22)_{10}$

− $(16)_{10}$ $(16)_{10}$ (∵ sum greater than 16, subtract 16 and generate carry)

A 8 6

➪ **Step III :** As carry is generated, result is negative and in 15's complement form. Find 15's complement of the result.

15 15 15

− A 8 6

5 7 9 Result

∴ $(69B)_{16} - (C1P4)_{16} = (-579)_{16}$

1.28.4 Hexadecimal subtraction using 16's complement

- The 16's complement of a hexadecimal number is obtained by adding 1 to the 15's complement of the number.
- The steps for performing subtraction using the 16's complement method are as follows :

Step I : Find the 16's complement of the subtrahend.

▼

Step II : Add the two hexadecimal numbers.

▼

Step III : If carry is generated from the result, it is discarded. If carry is not generated, it indicates that result is negative and in 16's complement form. Find 16's complement of the result.

Ex. 1.28.9

Perform the hexadecimal subtraction of $(D23)_H - (A13)_H$ using 16's complement method.

☑ **Ans. :**

➪ **Step I :** To obtain 16's complement of $(A13)_H$.

	15	15	15	
+	A	1	3	
	5	E	C	15's complement of $(A13)_H$
+			1	Add 1
	5	E	D	16's complement of $(A13)_H$

➪ **Step II :** Add the two hexadecimal numbers.

D 2 3 ⟶		$(13)_{10}$	$(02)_{10}$	$(03)_{10}$
5 E D ⟶		+	$(14)_{10}$	$(13)_{10}$
		$(05)_{10}$		
Carry		1	1	
		$(18)_{10}$	$(16)_{10}$	$(16)_{10}$
	−			
		$(16)_{10}$	$(16)_{10}$	$(16)_{10}$
Discard carry	1	3	1	0

(∵ sum is greater than 16, subtract 16 and generate carry)

➪ **Step III :** As carry is generated form addition, the result, is positive and in true form discard carry.

∴ $(D23)_H - (A13)_H = (310)_H$

Ex. 1.28.10

Perform the hexadecimal subtraction of $(69B)_{16} - (C14)_{16}$.

☑ **Ans. :**

➪ **Step I :** Find 16's complement of subtrahend.

	15	15	15	
−	C	1	4	
	3	E	B	15's complement of $(C14)_{16}$
+			1	Add 1
	3	E	C	16's complement of $(C14)_{16}$

➪ **Step II :** Add hexadecimal numbers.

6 9 B ⟶	$(06)_{10}$	$(09)_{10}$	$(11)_{10}$
3 E C ⟶	+ $(03)_{10}$	$(14)_{10}$	$(12)_{10}$
	1	1	
	$(09)_{10}$	$(23)_{10}$	$(23)_{10}$
	−	$(16)_{10}$	$(16)_{10}$
0	A	8	7

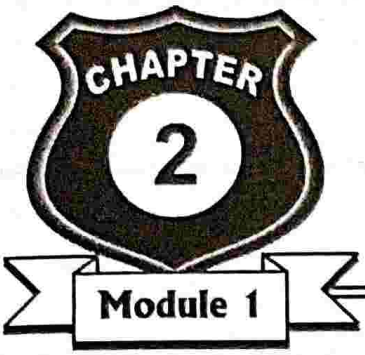
(∵ sum is greater than 16, subtract 16 and generate carry)

No carry generated ↑

➪ **Step III :** As no carry is generated result is negative and in 16's complement from. Find 16's complement of the result, to obtain the result in true form.

	15	15	15	
−	A	8	7	
	5	7	8	15's complement of result
+			1	Add 1
	5	7	9	16's complement of result

∴ $(69B)_{16} - (C14)_{16} = (-579)_{16}$



Codes

University Prescribed Syllabus

Computer Fundamentals

1.3 Codes : Grey, BCD, Excess-3, ASCII.

2.1 BINARY CODES

Q. 2.1.1 What is a code ? What are binary codes ? Give the advantages of binary codes.

Ans. :

- When the data or information is coded , the numbers, letters or words are represented by a specific group of symbols. The process of coding these symbols is called as **encoding the data** .
- The group of symbols is called as a **code**.
- The digital data is represented, stored and transmitted in the form of a stream (group) of binary **0**'s and **1**'s. This group of binary bits is also called as **binary code**.
- The binary codes can be represented by the numbers, letters of the alphabets and special characters.
- The binary codes are classified as numeric and alphanumeric codes. The numeric codes represent numbers.
- The alphanumeric codes represent alphabets and characters.

2.1.1 Advantages of Binary Codes

1. Binary codes are suitable for the computer applications.
2. Binary codes are used for the digital communication systems.
3. Binary codes use only 0 and 1. Thus, their implementation is very easy.
4. The digital circuits use binary codes for their analysis and design.

2.1.2 Classification of Binary Codes

Q. 2.1.2 Explain the classification of binary codes.

OR Explain in brief weighted and non-weighted codes with one example each.

MU - Q.1(g), Dec. 15, 2 Marks

Ans. :

- The binary codes are classified as

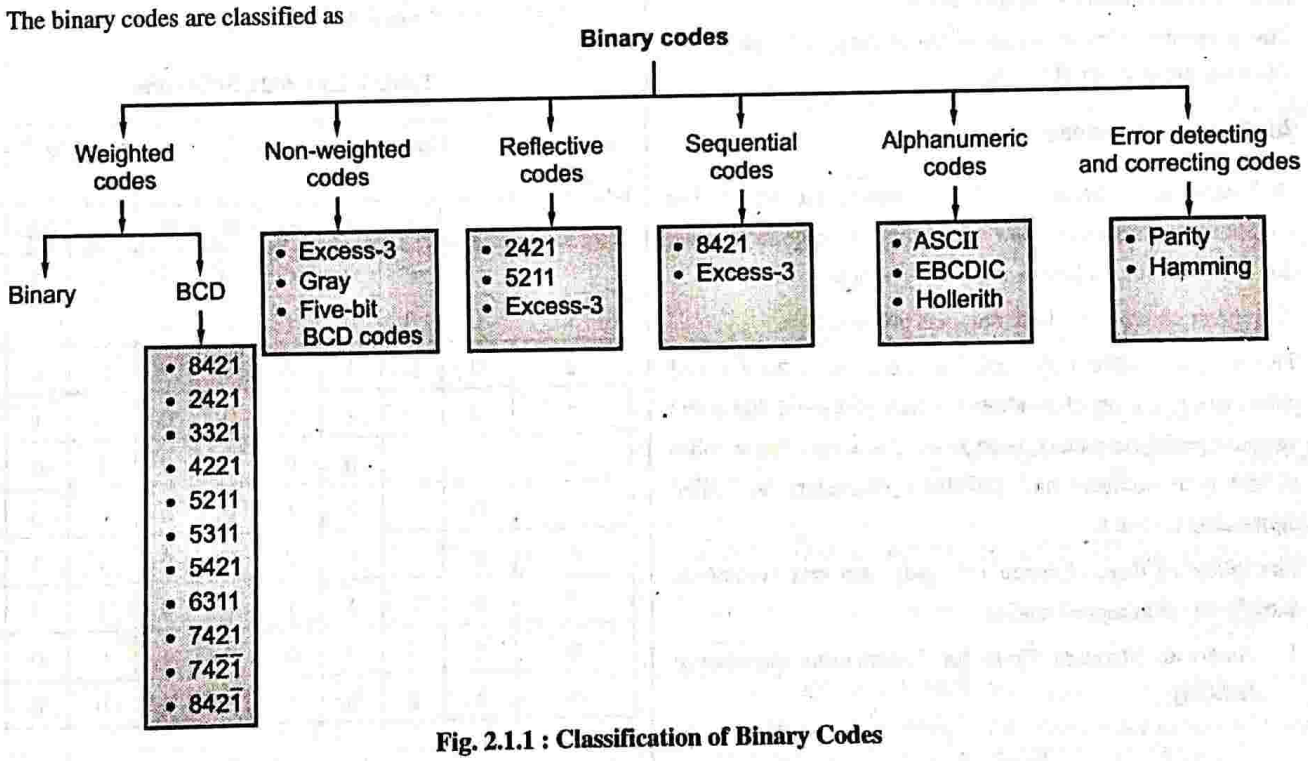


Fig. 2.1.1 : Classification of Binary Codes

1. Weighted codes

- Weighted codes are the codes that obey positional weights principle. i.e. in a weighted code every digit position of the number represents a specific weight.
- In weighted binary codes every digit has a weight 8, 4, 2 or 1. For example, in decimal code, if number is 423 then weight of 4 is 100, weight of 2 is 010 and weight of 3 is 001.
- The codes 8421, 2421 and 5211 are weighted codes.

2. Non-weighted codes

- Non-weighted codes are codes in which positional weights are not assigned to every digit position, i.e., every digit position within the number is not assigned fixed value.
- E.g. : Excess-3 and gray codes are the non-weighted codes.

3. Reflective Codes

- A code is reflective if the code is self complementing. i.e. the code for 9 is the complement the code for 0, the code for 8 is the complement for 1, the code for 7 is the complement for 2, the code for 6 is the complement for 3 and the code for 5 is the complement for 4.
- e .g. 2421 BCD, 5421 BCD and Excess-3 code are reflective codes.
- 8421 is not a reflective code.

4. Sequential Codes

- In sequential codes, every successive code is one binary number greater than its previous code.
- This property helps in manipulation of data. 8421 BCD and Excess-3 are sequential codes.

5. Alphanumeric codes

- A binary digit represents only two numbers 0 and 1. For communication between two computers, we require the two digits along with the letters and other symbols.
- Computer manipulates both numbers and symbols.
- The programs written by computer users are in the form of characters i.e., a set of symbols consists of letters, digits and various special characters, such as +, –, >, < etc. These codes consist both numbers and alphabetic characters are called alphanumeric codes.
- The following three alphanumeric codes are very commonly used for the data representation.
  1. American Standard Code for Information Interchange (ASCII).
  2. Extended Binary Coded Decimal Interchange Code (EBCDIC).
  3. Five bit Binary Code.
- ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers.

6. Error detecting and correcting codes

- The codes that allow error detection and correction are called **error detecting and correcting codes.**
- Hamming code is the mostly commonly used error detecting and correcting code.

⏭ 2.2 BCD CODE

1. In this code each decimal digit is represented by a 4-bit binary number.
2. BCD is a way to express each of the decimal digits with a binary code.
3. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111).
4. But in BCD code only first ten of these are used (0000 to 1001).
5. The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.
6. It is also called as 8-4-2-1 code because the weights of the 4 bits are 8-4-2-1 from left(MSB) to right (LSB).
7. Table 2.2.1 shows 8-4-2-1 BCD code.

Table 2.2.1 : 8421 BCD code

Decimal digit	Binary code				BCD code				
	B_3	B_2	B_1	B_0	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1
2	0	0	1	0	0	0	0	1	0
3	0	0	1	1	0	0	0	1	1
4	0	1	0	0	0	0	1	0	0
5	0	1	0	1	0	0	1	0	1
6	0	1	1	0	0	0	1	1	0
7	0	1	1	1	0	0	1	1	1
8	1	0	0	0	0	1	0	0	0
9	1	0	0	1	0	1	0	0	1

Ex. 2.2.1

Convert the following

$(124)_{10} = (\ )_{BCD}$?

Ans. :

$(124)_{10} = (0001\ 0010\ 0100)_{BCD}$

2.2.1 BCD Addition

The steps are followed to perform BCD addition :

- **Step 1 :** Add the two BCD numbers.
- **Step 2 :** If the sum is equal to or less than 9 (1001), it is a valid BCD number.
- **Step 3 :** If the sum is greater than 9, or if a carry is generated from the sum, the result is invalid BCD.
- **Step 4 :** Add 6 (0110) to the make the result valid BCD. If a carry is generated, when 6 is added, then add the carry to the next digit.

UEx. 2.2.2 **MU - Q. 1(e), Dec. 14, 2 Marks**

Add $(57)_{10}$ and $(26)_{10}$ in BCD.

Ans. :

```
Carry
                 1
  57     0101  0111
+ 26   + 0010  0110
       1 1 1 1
       0111  1101   Invalid BCD number
             0110   Add 6
       ------------
Result ->1000  0011   Valid BCD result
         8     3
```

(2B3)**Fig. Ex. 2.2.2**

$\therefore (57)_{10} + (26)_{10} = (83)_{10}$ **in BCD**

UEx. 2.2.3 **MU - Q. 1(c), May 17, 2 Marks**

Add $(7)_{10}$ and $(6)_{10}$ in BCD.

Ans. :

```
Carry
              1 1
   7     0111
+  6     0110
       ---------
         1
       + 1101   Invalid BCD number
         0110   Add 6
       ---------
       1 0011   Valid BCD result
         3
```

(2B4)**Fig. Ex. 2.2.3**

$\therefore (7)_{10} + (6)_{10} = (13)_{10}$ **in BCD**

Ex. 2.2.4

Add $(24)_{10}$ and $(18)_{10}$ is BCD

Ans. :

```
   24     0010 0100
+  18   + 0001 1000
       1 1 1 1
       0011 1100   Invalid BCD
            0110   Add 6
       ------------
       0100 0010   Valid BCD result
         4    2
```

(2B5)**Fig. Ex. 2.2.4**

$\therefore (24)_{10} + (18)_{10} = (42)_{10}$

2.2.2 BCD Subtraction Using 9's Complement Method

Following are the steps to be performed for BCD subtraction using 9's complement method.

- **Step I :** Find the 9's complement of the negative number.
- **Step II :** Using BCD addition, add the two numbers.
- **Step III :** If carry is produced, add carry to the result, else find the 9's complement.

2.2.3 BCD Subtraction Using 10's Complement Method

The steps to be followed for performing BCD subtraction using 10's complement method are as follows :

- **Step I :** Compute the 10's complement of the negative number. The 10's complement is obtained by taking 9's complement of number and adding '(1)' to it.
- **Step II :** Using BCD addition, add the two numbers.
- **Step III :** Determines the 10's complement of the number, if carry is not generated after BCD addition. If carry is generated, after the BCD addition, it is ignored.

2.2.4 Advantages of BCD Codes

1. It is similar to decimal system.
2. We need only the binary equivalent of decimal numbers 0 to 9 only.

2.2.5 Disadvantages of BCD Codes

1. The addition and subtraction of BCD numbers have different rules.
2. The BCD arithmetic is little more complicated.
3. BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

Ex. 2.2.5

Convert the decimal number 25 into BCD format.

Ans. :

$(25)_{10} = (0010\ 0101)_{BCD}$

2.3 EXCESS 3 CODE

Q. 2.3.1 *Write a note on Excess-3 code.* **MU - Q. 6(c), May 15, 7 Marks**

Ans. :

- The Excess-3 code is also called as XS-3 code.
- It is non-weighted code used to express decimal numbers.
- The Excess-3 code words are obtained from the 8421 BCD code words adding $(0011)_2$ or $(3)_{10}$ to each code word in 8421.
- The excess-3 codes are obtained as follows :

Decimal Number ⟶ 8421 BCD —(Add 0011)→ Excess-3

- Table 2.3.1 shows Excess-3 codes to represent a single decimal digit

Table 2.3.1 : Excess-3 code

Decimal number	BCD				Excess 3			
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Ex. 2.3.1

Convert $(340)_{10}$ to excess-3 code.

Ans. :

$(340)_{10} = (001101000000)_{BCD} = (0110\ 0111\ 0011)_{Excess-3}$

Ex. 2.3.2

Obtain excess-3 code for $(428)_{10}$.

Ans. :

$(428)_{10} = (0100\ 0010\ 1000)_{BCD} = (011101011011)_{Excess-3}$

Ex. 2.3.3

Obtain excess-3 code for $(25)_{10}$.

Ans. :

$(25)_{10} = (0010\ 0101)_{BCD} = (01011000)_{Excess-3}$

UEx. 2.3.4 **MU - Q. 1(a), Dec. 14, 1 Mark**

Represent the decimal number 29 into Excess 3 code.

Ans. :

$(29)_{10} = (00101001)_{BCD} = (0101\ 1100)_{Excess3}$

UEx. 2.3.5 **MU - Q. 1(b), May 15, 1 Mark**

Represent $(52)_{10}$ into excess – 3 code.

Ans. :

$(52)_{10} = (0101\ 0010)_{BCD} = (1000\ 0101)_{Excess-3}$.

2.3.1 Excess 3 Addition

The steps to be performed for XS-3 addition are as follows :

- **Step I :** Add two XS-3 numbers.
- **Step II :** If carry = 0, subtract 0011 (33) or add (1101) to the sum. If carry = 1, add 0011 (3) to the sum.

Ex. 1 Add 9 and 4

```
        [1]
         1  1  0  0   9 in XS -3
    +    0  1  1  1   4 in XS-3
   [1]   0  0  1  1   Result of addition
+  0  0  1  1  0  0  1  1   Add 3 as carry is 1
   0  1  0  0  0  1  1  0   Excess-3 for 13
```

Ex. 2 Add 2 and 4.

```
        [1]
         0  1  0  1   Excess 3 for 2
         0  1  1  1
        [1]
         1  1  0  0   Result of addition
         1  1  0  1   Subtract 3 as carry as 0
Ignore→[1]
Carry    1  0  0  1   Excess 9 for 6
```

2.3.2 Excess 3 Subtraction

To perform Excess 3 subtraction following are the steps to be performed.

Step I : Complement the subtrahend.

Step II : Add the complemented subtrahend to minuend. If carry = 0. Result is positive. Add end around carry. If carry = 0, Result is negative. Subtract 3 or add 1101 and take 1's complement of the result.

Ex. 1 Subtract 9 – 4.

	1	1	0	0	9 in XS-3
+	1	0	0	0	Complement of 4 in XS-3.
[1]	0	1	0	0	
+	0	0	1	1	Add 3
+	1	1	0	1	
				[1]	
	1	0	0	0	5 in XS-3 ← Result

Ex. 2 Subtract 4 – 9

	0	1	1	1	4 in XS-3
+	0	0	1	1	Complement of 9 in XS-3.
	1	0	1	0	No carry
+	1	1	0	1	Subtract 3 or Add 13
[1]	0	1	1	1	Complement the result
	1	0	0	0	(– 5) in XS in XS-3

2.4 GRAY CODE

Q. 2.4.1 Write note on : Gray code.

MU - Q. 6(c), May 15, 3.5 Marks

Ans. :

- It is the non-weighted code.
- It is not arithmetic code. i.e. there are no specific weights assigned to the bit position.
- It has a very special feature that, only one bit will change each time the decimal number is incremented. As only one bit changes at a time, the gray code is called as **a unit distance code.** The gray code is a cyclic code.
- Gray code cannot be used for arithmetic operation.

Table 2.4.1 : Gray code

Decimal	8421 code 4 bit binary code				Gray code			
	B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

2.4.1 Uses of Gray Code

Q. 2.4.2 Explain uses of gray code.

MU - Dec. 14, 4 Marks

Ans. :

1. The gray code is used in the transmission of digital signals as it minimizes the occurrence of errors.
2. The gray code is preferred over the binary code in angle-measuring devices. Use of the gray code almost eliminates the possibility of the angle misread.
3. The gray code is used for labelling the axes of Karnaugh maps, graphical technique used for minimization of Boolean expressions.
4. The use of gray code to address program memory in computers minimizes power consumption.
5. Gray codes are very useful in generating algorithms.

2.4.2 Gray Code to Binary Code Conversion

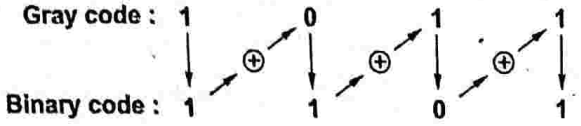
Step I : Start with the most significant bit (MSB) of the gray code number.

Step II : The second most significant bit, in the binary number is obtained by XORing the MSB in the binary number to the second MSB in the Gray code number.

Step III : The third MSB in the binary number is obtained by XORing the second MSB in the binary number to the third MSB in the Gray code number.

➪ **Step IV :** The process continues until we obtain the LSB of the binary number.

Example : Convert gray code 1011 to binary



Thus , $(1011)_{gray} = (1101)_2$

2.4.3 Binary to Gray Code Conversion

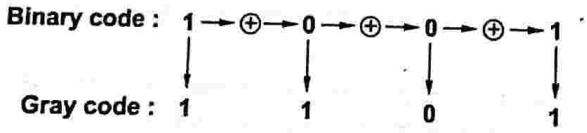
➪ **Step I :** Start with the most significant bit (MSB) of the binary number. The MSB of the gray code equivalent is the same as the MSB of the given binary number.

➪ **Step II :** The second most significant bit, adjacent the MSB, in the gray code number is obtained by XORing the MSB and the second MSB of the binary number that is if the MSB and the bit adjacent to it are both '1', then the corresponding gray code bit would be a '0'.

➪ **Step III :** The third most significant bit, adjacent to the second MSB, in the gray code number is obtained by XORing the second MSB and the third MSB in the binary number.

➪ **Step IV :** The process continues until we obtain the LSB of gray code.

Example convert 1000 binary to gray



Thus ,$(1000)_2 = (1101)_{gray}$

UEx. 2.4.1 **MU - Q. 1(b), May 16, 2 Marks**

Convert $(47.3)_7$ into BCD, excess 3.

☑ **Ans. :**

$$(47.3)_7 = 4\times7 + 7\times7^0 + 3\times7^{-1}$$
$$= 28 + 7 + \frac{3}{7}$$
$$= 35 + 0.4285$$
$$= 35.4285$$
$$\therefore (47.3)_7 = (35.4285)_{10}$$
$$\therefore (35.4285)_{10} = (0011\ 0101\cdot 0100\ 0010\ 1000\ 0101)_{BCD}$$
$$= (0110\ 1000\cdot 0111\ 0101\ 1011\ 1000)_{Excess-3}$$

UEx. 2.4.2 **MU - Q. 1(a), May 14, 1 Mark**

Represent $(29)_{10}$ into gray code.

☑ **Ans. :**

➪ **Step I :** To express $(29)_{10}$ into binary.

Base	Quotient	Remainder
2	29	
2	14	1
2	7	0
2	3	1
2	1	1
		1

(LSB) ↑ (MSB)

$\therefore (29)_{10} = (11101)_2$

➪ **Step II :** Convert binary to gray

Binary code : 1 → ⊕ → 1 → ⊕ → 1 → ⊕ → 0 → ⊕ → 1
Gray code : 1 0 0 1 1

Thus $(11101)_2 = (10011)_{gray}$

$\therefore (29)_{10} = (11101)_2 = (10011)_{gray}$

UEx. 2.4.3 **MU - Q. 1(a), May 14, 1 Mark**

Represent $(52)_{10}$ into Gray code.

☑ **Ans. :**

➪ **Step I :** To express $(52)_{10}$ into binary.

Base	Quotient	Remainder
2	52	
2	26	0
2	13	0
2	6	1
2	3	0
2	1	1
		1

(LSB) ↑ Read up ↑ (MSB)

$\therefore (52)_{10} = (110100)_2$

➪ **Step II :** Convert binary to gray

Binary code : 1 → ⊕ → 1 → ⊕ → 0 → ⊕ → 1 → ⊕ → 0 → ⊕ → 0
Gray code : 1 1 1 1 1 0

$(52)_{10} = (110100)_2 = (101110)_{gray}$

2.5 ASCII CODE

– It is a standard alphanumeric code . It is abbreviated as the American Standard Code for Information Interchange (ASCII) .

- It is a seven-bit code in which the decimal digits are represented by the BCD code preceded by 011. As it is a 7-bit code, it represents $2^7 = 128$ symbols .

Table 2.5.1 : ASCII Code

Alphabet	7-bit ASCII Code
A	100 0001
B	100 0010
C	100 0011
D	100 0100
E	100 0101
F	100 0110
G	100 0111
H	100 1000
I	100 1001
J	100 1010
K	100 1011
L	100 1100
M	100 1101
N	100 1110
O	100 1111
P	101 0000
Q	101 0001
R	101 0010
S	101 0011
T	101 0100
U	101 0101
V	101 0110
W	101 0111
X	101 1000
Y	101 1001
Z	101 1010
0	011 0000
1	011 0001
2	011 0010
3	011 0011
4	011 0100
5	011 0101
6	011 0110
7	011 0111
8	011 1000
9	011 1001
BLANK	010 0000
.	010 1110
(	010 1000
+	010 1011
$	010 0100
*	010 1010
)	010 1001
-	010 1101
/	010 1111
,	010 1100
=	011 1101

2.6 ERROR DETECTING AND CORRECTING CODES

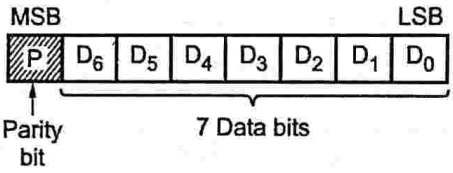
- When data is transmitted from one system to other system in binary format, an error may occur.
- This indicates that the signal at the receiver end may change a **0** to **1** or **1** to **0** because of noise in the communication channel.
- An additional extra bit is added to the data transmitted, in order to maintain the data integrity between transmitter and receiver.
- The extra bit allow the error detection and correction of the data.
- The data along with the extra bit/bits called **parity bits** forms a code.
- The codes that support only error detection are called **error detecting codes.**
- The codes that allow error detection and correction are called as **error detecting and correcting codes.**
- **Parity checking** is commonly used for detect the errors.
- **Hamming code** is the mostly commonly used error detecting and correcting code.

2.6.1 Parity Checking

Q. 2.6.1 *What is parity bit, even parity and odd parity ?*

Ans. :

- Parity bits are the extra bit/bits added to the data being transmitted for detecting the errors in data transmission.
- Fig 2.6.1 shows transmitted data with parity bit .
- If the number of 1's in the given word is even(2,4,6), the data word is said to have **Even parity** .
- If the number of 1's in the given word is odd (1,3,5), the data word is said to have **Odd parity** .

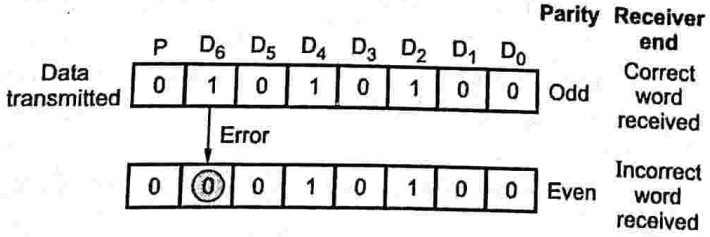


(2B6)**Fig. 2.6.1 : Transmitted data word with parity bit**

Q. 2.6.2 *How does parity checking detect errors? What are its drawbacks?*

Ans. :

- **Parity checking method** is used to detect errors at the receiver end.
- If the parity of the signal received is different than the assigned parity, then there is an error is the received signal. i.e. if the transmitted data is assigned odd parity and if the signal received has an even parity, then there is an error in the signal received as shown in Fig. 2.6.2.



(2B7)**Fig. 2.6.2 : Error Detection using parity checking.**

- If there is a single error during the data transmission, the parity of the transmitted word changes. At the receiver end, the party of the received word is checked. A change in parity bit indicates that there is an error in the received signal.
- If the receiver detects that there is an error in the transmitted signal, it ignores the received byte and requests the transmitter to retransmit the same data byte.

Drawbacks of parity checking method

(1) If the number of errors in the received codeword is double or even, the parity of the received data word remains the same. i.e. even number of errors are undetected by the receiver.

(2) If the number of errors in the received code word is odd, the parity of the received data word changes. However, the error cannot be corrected. The receiver needs to request the transmitter to retransmit the code word.

(3) It only indicates that error is there, but does not indicate which bit is in error.

2.6.2 Hamming Code

Q. 2.6.3 *Explain with example how hamming code is useful for detecting and correcting errors.*

OR *Prove that hamming code is an error detecting and correcting code.*

MU - Q. 2(b), May 14, 10 Marks

Ans. :

- **Hamming code** is used to overcome the drawback of parity checking method. i.e. it detects the error and also indicates which bit is in error.
- The incorrect bit can then be changed to correct bit, hence, the Hamming code is also called as a **self correcting code.**
- The Hamming code is used for correcting a single error on the transmitted 7 bit code word as shown in Fig. 2.6.3, assuming a four bit message.

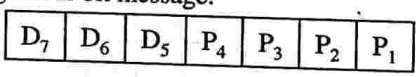


Fig. 2.6.3 : Bit code word for Hamming code

- D_7 is the most significant bit in the 4 bit word **(D_7, D_6, D_5, D_3).**
- The parity bits **P_4, P_2, P_1** are allocated values by making the following three parity relations considering four bit from the 7 bit code word.

(i) Group 1 considers P_1 ($2^0 = 1 = P_1$ position)

P_1 is selected 0 or 1, in order to set odd / even parity over bits, 1, 3, 5 and 7 i.e. bits P_1, D_3, D_5, D_7.

(ii) Group 2 considers P_2 ($2^1 = 2 = P_2$ position) : P_2 is selected to be 0 or 1, in order to set odd/ even parity over the bits 2, 3, 6 and 7 i.e. bits P_2, D_3, D_6 and D_7.

(iii) Group 3 consider P_4 ($2^2 = 4 = P_4$ position) P_4 is selected to be 0 or 1 to set odd/even parity over the bits 4, 5, 6 and 7. i.e. bits P_4, D_5, D_6 and D_7.

UEx. 2.6.1 MU - Q. 1(b), Dec. 17, 5 Mark

Assume that the data has been encoded in a 7 bit even parity Hamming code and the number 1000010. Correct if for any errors and extract 4 bit data.

Ans. :

Step I : Construct the 7-bit Hamming code.

D_7	D_6	D_5	P_4	D_3	P_2	P_1
1	0	0	0	0	1	0

Step II : Check for parity bits

For P_1 : P_1 checks 1, 3, 5 and 7

As there is one 1, the parity check for even parity is wrong.

$\therefore P_1 = 1$

For P_2 : P_2 checks 2, 3, 6 and 7.

There are two 1's in the group. Therefore parity checking for even parity is correct.

$\therefore P_2 = 0$

For P_4 : P_4 checks 4, 5, 6 and 7.

There is one 1. The parity check for even parity is wrong.

$\therefore P_4 = 1$

$\therefore$ The result word is $(P_4 P_2 P_1) = (101)$ It indicates that bit in **5^{th} position is in error.**

D_5 bit is 0, it should be a 1

$\therefore$ The correct code word is 1010010

$\therefore$ The 4 bit data ($D_7 D_6 D_5 D_3$) is 1010

This, it is proved that Hamming code is error detecting and correcting code.

UEx. 2.6.2 MU - Q. 2(b), Dec. 15, Q. 1(c), May 17, 2 Marks

Obtain even parity Hamming code for 1010.

Ans. :

Step I : Construct the 7-bit Hamming code.

D_7	D_6	D_5	P_4	D_3	P_2	P_1
1	0	1		0		

Step II : To determine the parity bits.

(1) For P_1 : P_1 checks bits 1, 3, 5, and 7.

There are two 1' is the group. Therefore parity checking for even parity is correct.

$\therefore P_1 = 0$

(2) For P_2 : P_2 checks bits 2, 3, 6 and 7

There is one 1 in the group. Therefore, parity checking for even parity is correct.

$\therefore P_2 = 1$

(3) For P_4 : P_4 checks bits 4, 5, 6 and 7

There are two 1's in the group. Therefore, parity checking for even parity is correct.

$\therefore P_4 = 0$

$\therefore$ The 7 bit code word is,

1 0 1 0 0 1 0 for even parity

UEx. 2.6.3 MU - Q. 1(e), May 15, Q. 1(d), May 16, 2 Marks

Obtain odd parity Hamming code for 1011.

Ans. :

Step I : Construct the 7-bit Hamming code.

D_7	D_6	D_5	P_4	D_3	P_2	P_1
1	0	1		1		

Step II : To determine parity bits

(1) For P_1 : P_1 checks bits 1, 3, 5, 7.

There are three 1's in the group. Therefore, odd parity checking is correct.

$\therefore P_1 = 0$

(2) For P_2 : P_2 check bits 2, 3, 6, 7

There are two 1's in the group. Therefore, odd parity checking is incorrect.

$\therefore P_2 = 1$

(3) For P_4 : P_4 checks bits 4, 5, 6, 7

There are two 1's in the group. Therefore, odd parity checking is incorrect.

$\therefore P_4 = 1$

$\therefore$ 7 bit Hamming codeword is 1011110

UEx. 2.6.4 MU - Q.1(e), May 17, 2 Marks

Construct Hamming code for BCD 0110. Use even parity.

Ans. :

Step I : To construct 7 bit Hamming code

D_7	D_6	D_5	P_4	D_3	P_2	P_1
1	0	1		0		

Step II : To determine parity bits.

(1) For P_1 : P_1 checks bits 1, 3, 5, and 7

There are one 1' in the group. Therefore, even parity checking is wrong.

$\therefore P_1 = 1$

(2) For P_2 : P_2 checks bits 2, 3, 6 and 7

There is one 1 in the group. Therefore, even parity checking is wrong.

$\therefore P_2 = 1$

3) For P_4 : P_4 checks bits 4, 5, 6 and 7

There are two 1's in the group. Therefore, parity checking for even parity is correct.

$\therefore P_4 = 0$

∴ 7 bit Hamming code word is, 0 110011

UEx. 2.6.5 MU - Q. 1(e), May 18, 2 Marks

Encode the data bits 0101 into a seven bit even parity Hamming code.

Ans. :

Step I : Construct the 7 bit Hamming code.

D_7	D_6	D_5	P_4	D_3	P_2	P_1
0	1	0		1		

Step II : To determine parity bits.

(1) For P_1 : P_1 checks bits 1, 3, 5, 7

There is one 1 is the group. Therefore even parity checking for group is incorrect

$\therefore P_1 = 1$

(2) For P_2 : P_2 checks bits 2, 3, 6, 7

There are two 1's is the group. Therefore, parity checks for even parity is correct.

$\therefore P_2 = 0$

(3) For P_4 : P_4 checks bits 4, 5, 6, 7

There is one 1 is the group. Therefore, parity checks for even parity is incorrect.

$\therefore P_4 = 1$

∴ 7 bit Hamming codeword is 0101101

UEx. 2.6.6 MU - Q. 1(b), Dec. 18, 4 Marks

Construct Hamming code for 1010 using odd parity.

Ans. :

Step I : Construct 7 bit Hamming code.

D_7	D_6	D_5	P_4	D_3	P_2	P_1
1	0	1		0		

Step II : To determine the parity bits

(1) For P_1 checks bits 1, 3, 5, 7

There are two 1's in the group. Therefore parity checks for odd parity is incorrect.

$\therefore P_1 = 0$

(2) For P_2 : P_2 checks bits 2, 3, 6, 7

There is one 1 in the group. Therefore parity checks for odd parity is correct.

$\therefore P_2 = 1$

(3) For P_4 : P_4 checks bits 4, 5, 6, 7

There are two 1's in the group. Therefore parity checks for odd parity is incorrect.

$\therefore P_4 = 1$

∴ 7 bit Hamming code for odd parity is 1011001

UEx. 2.6.7 MU - Q. 1(e), Dec. 19, 4 Marks

If the 7 bit hamming code word received by receiver is 1011011, assuming the even parity, state whether the received code word is correct or wrong? If wrong locate the bit having error and extract corrected data.

Ans. :

Step I : Construct the 7-bit Hamming code.

D_7	D_6	D_5	P_4	D_3	P_2	P_1
1	0	1	1	0	1	1

Step II : Check for parity bits

For P_1 : P_1 checks 1, 3, 5 and 7

As there are three 1's , the parity check for even parity is wrong.

$\therefore P_1 = 0$

For P_2 : P_2 checks 2, 3, 6 and 7.

There are two 1's in the group. Therefore parity checking for even parity is correct.

$\therefore P_2 = 0$

For P_4 : P_4 checks 4, 5, 6 and 7.

There are three 1's. The parity check for even parity is wrong.

$\therefore P_4 = 0$

∴ The result word is $(P_4\ P_2\ P_1) = (000)$ It indicates that bit in **0^{th} position is in error.**

D_0 bit is 1, it should be a 0

∴ The correct code word is 1011010

∴ The 4 bit data ($D_7\ D_6\ D_5\ D_3$) is 1010

This, it is proved that Hamming code is error detecting and correcting code.

...Chapter Ends

❑❑❑

CHAPTER 3

Module 1

Boolean Algebra and Logic Gates

University Prescribed Syllabus

Computer Fundamentals

1.4 Boolean Algebra, Logic Gates : AND, OR, NOT, NAND, NOR, EX-OR.

Previous University Paper Questions

3.1 LOGIC GATES

Q. 3.1.1 What do you mean by logic gates? Give its characteristics.

Ans. :

Definition of Logic Gates : The electronics circuits that are used for implementing the logical or boolean expressions are called as logic gates.

- A logic gate is a digital circuit with one or more inputs and only one output.

3.1.1 Characteristics of Logic Gates

1. The relationship between the input and output is such that every logic gate is designed to perform a specific **logical expression.**
2. The logic gates are the basic building blocks of digital systems.
3. The logic gates are made up of a number of components and electronic devices.

Q. 3.1.2 Which gates are called as basic gates ?

Ans. : The three most basic types of gates are

1. AND 2. OR 3. NOT

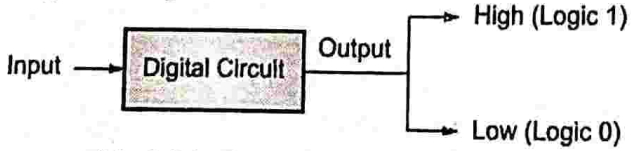
- A digital circuit of any complexity can be constructed with the help of these 3 gates.
- Hence, they are called as **basic gates** or **basic building blocks of a digital circuits.**

3.2 POSITIVE AND NEGATIVE LOGIC

Q. 3.2.1 What do you understand by positive and negative logic ?

Ans. :

- The digital circuits operate on two output levels : logic 0 and logic 1.

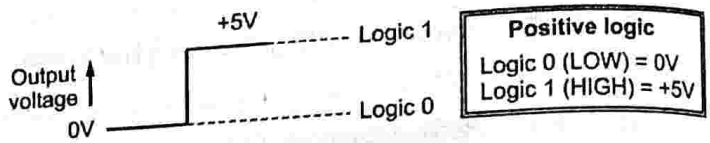


(1B1) **Fig. 3.2.1 : Output levels in a digital circuit**

- The output voltage levels (high and low) that are assigned to represent the logic levels 0 and 1 may be **positive logic or negative logic.**

3.2.1 Positive Logic

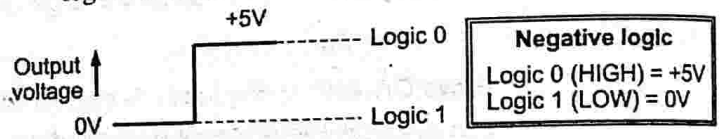
- A positive logic system is one in which a HIGH voltage level represents logic 1 and a LOW voltage level represents logic 0. E.g. : In positive logic 0V represents logic 0 state and + 5V represents logic 1 state.



(1B2) **Fig. 3.2.2 : Positive logic**

3.2.2 Negative Logic

- A negative logic system is one in which a low voltage level represents logic 1 and a HIGH voltage level represents logic 0. E.g., if 0V represents logic 1 state and + 5V represents logic 0 state, it is called as negative logic.



(1B3) **Fig. 3.2.3 : Negative logic**

3.3 TRUTH TABLE

Q. 3.3.1 What do you mean by truth table ?

Ans. :

Definition of Truth Table : A truth table is a table that shows all the input-output combinations of a logic circuit.

- In a truth table, all the input combinations of binary **0** and 1s are listed in numerical order.
- The output corresponding to every input combination is also listed, i.e., a truth table shows how a logic circuit's output will react to the different input combinations.

3.4 NOT GATE

Q. 3.4.1 What is a NOT gate ?
OR Describe NOT gate with its symbol and Boolean expression.

Ans. : Answer includes following points :

A. Definition and description of NOT Gate
B. Logic symbol of NOT Gate

C. Logical or Boolean expression
D. Logical operation
E. NOT gate using a switch
F. Truth table
G. Input and Output Waveforms

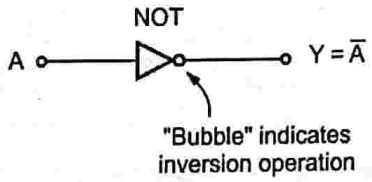
(A) Definition and description of NOT Gate

Definition : NOT gate is a logic circuit whose output is always complement of the input.

- A NOT gate is also called as **an inverter.**
- It has one input and one output.
- The NOT operation is also called as **inversion or complementation**; i.e., output of the NOT gate will be logic 1 when its input is logic 0 and vice-versa.

(B) Logic symbol of NOT Gate

The symbol for NOT operation is '–' (bar).



(1B4) **Fig. 3.4.1 : Logic symbol for NOT gate (inverter)**

- As shown in Fig. 3.4.1, a small circle, called **"bubble"**, indicates inversion operation in digital circuits.

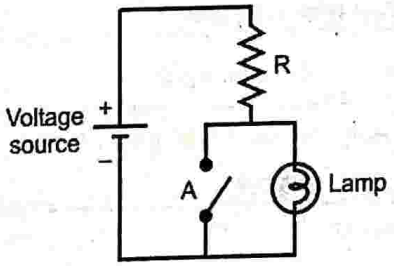
(C) Logical or Boolean expression of NOT Gate

$$Y = \text{NOT } A = \bar{A}$$

(D) Logical operation of NOT gate

Inversion

(E) NOT gate using a switch



(1B49) **Fig. 3.4.2 : NOT gate using switch**

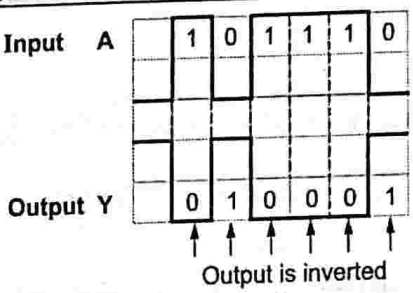
- When switch A is open the lamp is ON.
- When switch A is closed the lamp is OFF.
- In binary language when the input is LOW (logic 0) output is HIGH (logic 1) and vice versa.

(F) Truth table of NOT Gate

Table 3.4.1 : Truth table of NOT gate

Input A	Output Y
0	1
1	0

(G) Input and Output Waveforms of NOT Gate



(1B50) **Fig. 3.4.3 : Input and Output waveforms for NOT gate**

Q. 3.4.2 Which gates can be used as inverters in addition to the NOT gate ?

☑ **Ans. :** NAND, NOR, EX-OR, and EX-NOR gates can be used as inverters in addition to the NOT gate.

3.5 AND GATE

Q. 3.5.1 Describe the AND gate with the symbol, the logical statement, the Boolean expression and its logical diagram.

☑ **Ans. : Answer includes following points :**

A. Definition and description of AND Gate
B. Logic symbol of AND Gate
C. Logical or Boolean expression
D. Logical operation
E. AND gate using switches
F. Truth table
G. Input and Output Waveforms
H. Application of AND gate

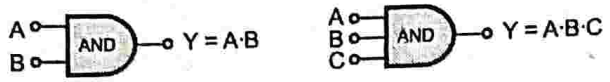
(A) Definition and Description of AND Gate

Definition : AND gate is a logic circuit whose output is HIGH when all its inputs are HIGH.

- An AND gate has two or more inputs and one output.
- An AND gate will have output 1 only if all its inputs are 1. The output of the AND gate will be logic 0, if any of the inputs are at logic 0. It is also called as **all gate** or **nothing gate.**

(B) Logic Symbol of AND Gate

The symbol for AND operation is '·' (dot).



(1B5)(a) Logic symbol for 2-input AND gate

(1B5A)(b) Logic symbol for 3-input AND gate

Fig. 3.5.1

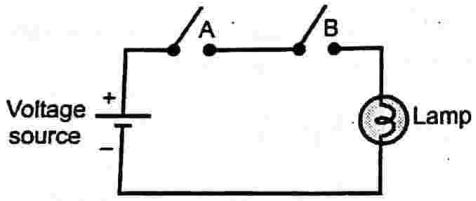
(C) Logical or Boolean Expression of AND Gate

Y = A AND B Y = A AND B AND C

Y = A · B Y = A · B · C

The AND operator (·) represents logical multiplication. Sometimes, dot is not used and we represent logical multiplication of A · B as AB and is read as "A and B".

(D) AND gate using Switches



(1B45)Fig. 3.5.2 : AND gate using switches

1. When both the switches A and B are open, the lamp is OFF.
2. When switch A is open and switch B is closed, the lamp is OFF.
3. When switch A is closed and switch B is open, the lamp is OFF.
4. When both the switches A and B are closed, the lamp is ON.

Table 3.5.1 : Operation of AND gate using switches

A	B	Lamp
Open	Open	OFF
Open	Closed	OFF
Closed	Open	OFF
Closed	Closed	ON

- Table 3.5.1, shows the 4 combinations of inputs A and B and its corresponding output.
- In binary language when any input is LOW the output is LOW.
- And when both the inputs are HIGH, the output is HIGH.

(E) Logical Operation of AND Gate

Logical multiplication

(F) Truth Table

Table 3.5.2 : Truth table of 2-input AND gate

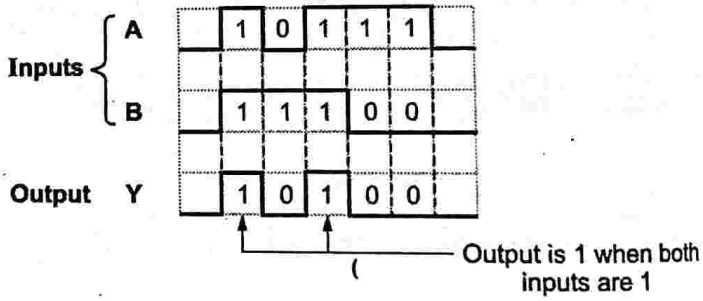
Input		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Output is 1 when all inputs are 1

Table 3.5.3 : Truth table of 3-input AND gate

Input			Output
A	B	C	Y = A · B · C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(G) Input and Output Waveforms of AND gate



1B46)Fig. 3.5.3 : Input and Output waveforms for 2 input AND gate

(H) Application of AND gate

1. Enable gate 2. Inhibit gate

Q. 3.5.2 Why is an AND gate called ALL or nothing gate ?

Ans. :

An AND gate is called **ALL** or **NOTHING gate** because it produces output HIGH when all its inputs are HIGH (logic 1), otherwise the output is LOW (logic 0).

Q. 3.5.3 *An AND gate in positive logic system is equivalent to which gate in negative logic system?*

☑ **Ans. :**

An AND gate in positive logic system is equivalent to **OR gate in negative logic system.**

3.6 OR GATE

Q. 3.6.1 *Draw symbol for 3 input OR gate with truth table.*

☑ **Ans. : Answer includes following points :**

A. Definition and description of OR Gate
B. Logic symbol of OR Gate
C. Logical or Boolean expression
D. Logical operation
E. OR gate using switches
F. Truth table
G. Input and Output Waveforms

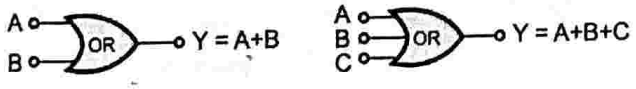
☞ (A) Definition and description of OR Gate

❐ *Definition : OR gate is a logic circuit whose output is HIGH when any of its inputs are HIGH.*

- An OR gate has two or more inputs and one output.
- The OR gate will have output 1 if one of its input is 1. The output of the OR gate is 0, if all of its inputs are in logic 0.
- It is called as **ALL gate** or **inclusive-OR gate.**

☞ (B) Logic symbol of OR Gate

The symbol for OR operation is **'+' (Plus).**



(1B6)(a) **Symbol for 2-input OR gate** (1B6A)(b) **Symbol for 3-input OR gate**

(1B6) **Fig. 3.6.1**

☞ (C) Logical or Boolean Expression of OR Gate

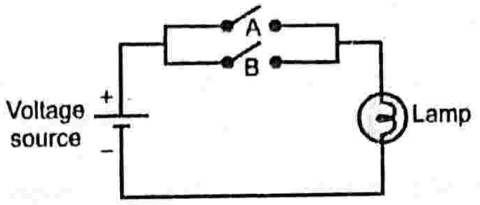
Y = A OR B = A + B **Y = A OR B OR C = A + B + C**

- The OR operator "+" represents logical addition of (A + B) and is read as "A OR B".

☞ (D) Logical operation of OR gate

Logical addition

☞ (E) OR Gate Using Switches



(1B47) **Fig. 3.6.2 : OR gate using switches**

1. When both the switches A and B are open, the lamp is OFF.
2. When switch A is open and switch B is closed, the lamp is ON.
3. When switch A is closed and switch B is open, the lamp is ON.
4. When both the switches A and B are closed, the lamp is ON.

Table 3.6.1 : OR gate using switches

A	B	Lamp
Open	Open	OFF
Open	Closed	ON
Closed	Open	ON
Closed	Closed	ON

- Table 3.6.1 shows the 4 combinations of inputs A and B and its corresponding output.
- In binary language when any input is HIGH the output is HIGH.
- And when both the inputs are LOW, output is LOW.

☞ (F) Truth Table of OR Gate

Table 3.6.2 : Truth table of 2-input OR gate

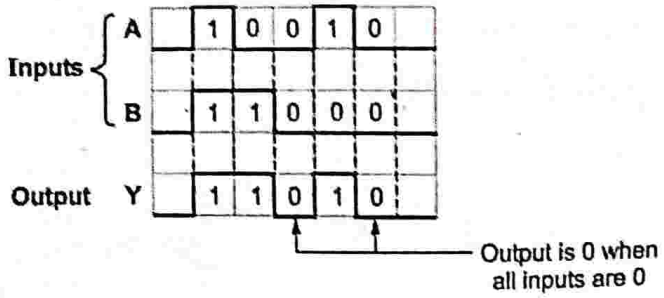
Input		Output
A	B	Y = A + B
0	**0**	**0**
0	1	1
1	0	1
1	1	1

Output is 0 when all inputs are 0

Table 3.6.3 : Truth table of 3-input OR gate

Input			Output
A	B	C	Y = A + B + C
0	**0**	**0**	**0**
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(G) Input and Output Waveforms of OR Gate



(1B48) **Fig. 3.6.3 : Input and Output waveforms for 2 input OR gate**

Q. 3.6.2 An OR gate in positive logic system is equivalent to which gate in negative logic system ?

Ans. : An OR gate in positive logic system is equivalent to AND gate in negative logic system.

3.7 UNIVERSAL GATES

Q. 3.7.1 Prove that NAND and NOR gates are Universal gates.

MU - Q. 1(a), Dec. 19, 4 Marks

Ans. :

- With the help of three basic gates AND, OR and NOT, we can realize any given Boolean expression.
- Two more gates : NAND gate and NOR gate have been constructed for implementing Boolean expressions.

Definition of Universal Logic Gate : A universal logic gate is a gate that can be used to implement any Boolean expression without using any other type of gate.

Advantage of Universal Gates

- **NAND and NOR** gates are called as **universal logic gates** because all other gates can be constructed using them; i.e., both the NAND and NOR gates can be used to perform the three basic logic functions : AND, OR and NOT.

Q. 3.7.2 What is AOI and OAI logic ?

Ans. :

- **AND-OR-INVERT (AOI) logic** is two level logical or Boolean expression constructed from the combination of one or more AND gates followed by a NOR gate.
- **OR-AND-INVERT (OAI)** logic is a two level Boolean expression constructed from the combination of one or more OR gates followed by a NAND gate.
- Thus, **AOI or OAI logic can be converted to NAND logic or NOR logic.**

Q. 3.7.3 NAND gate followed by an inverter is equivalent to which gate ?

Ans. :

NAND gate followed by an inverter is equivalent to **AND gate.**

Q. 3.7.4 NOR gate followed by an inverter is equivalent to which gate ?

Ans. :

NOR gate followed by an inverter is equivalent to **OR gate.**

Q. 3.7.5 How can NAND and NOR gate be used as an inverter ?

Ans. : When both the inputs of NAND or NOR gates are tied together, and input is applied to the common terminal the NAND or NOR gate functions as **an inverter.**

3.8 NAND GATE

Q. 3.8.1 Draw symbol for 3-input NAND gate with truth table.

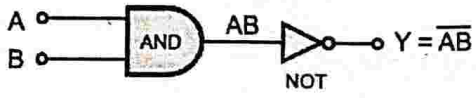
Ans. : Answer includes following points :

A. Definition and description of NAND Gate
B. Logic symbol of NAND Gate
C. Logical or Boolean expression
D. Logical operation
E. Truth table
F. Input and Output Waveforms

(A) Definition and Description NAND Gate

Definition : NAND gate is a logic circuit whose output is LOW when all is inputs are HIGH.

- NAND gate means **NOT-AND operation.** It is a combination of AND and NOT gate. Fig. 3.8.1 shows NAND operation as NOT-AND operation.

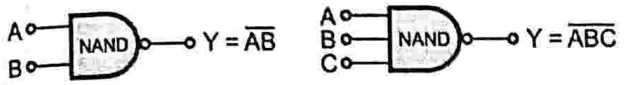


(1B7) **Fig. 3.8.1 : NAND operation as NOT-AND operation**

- A NAND gate has two or more inputs and one output.

- A NAND gate will have output low (0) when all its inputs are high (1).
- The output of the NAND gate will be logic 1, if any of its inputs are at logic 0.
- **It is called as universal logic gate as it can be used to implement any Boolean function. It is also called an active low OR gate.**

(B) Logic Symbol of NAND Gate



(1B8)(a) Symbol for 2-input NAND gate

(1B8A)(b) Symbol for 3-input NAND gate

(1B8) Fig. 3.8.2

- Bubble on the NAND gate represents NOT operation or complementation or inversion.

(C) Boolean Expression of NAND Gate

$Y = \overline{A \cdot B} = \overline{AB}$

[A · B represents AND operation and the bar represents NOT operation]

$Y = \overline{A \cdot B \cdot C} = \overline{ABC}$

[A · B · C represents AND operation and the bar represents NOT operation]

(D) Logical Operation of NAND Gate

NOT AND or NAND

(E) Truth Table of NAND Gate

Table 3.8.1 : Truth table of 2-input NAND gate

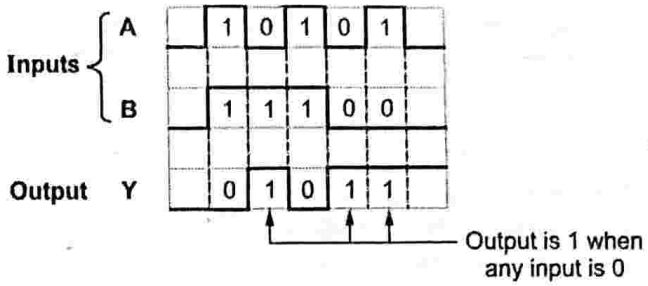
Input		Output
A	B	$Y = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

Output is 0 when all inputs are 1

Table 3.8.2 : Truth table of 3-input NAND gate

Input			Output
A	B	C	$Y = \overline{A \cdot B \cdot C}$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(F) Input and Output Waveforms of NAND Gate



(1B51)Fig. 3.8.3 : Input and Output waveforms for 2-input NAND gate

3.9 NOR GATE

Q. 3.9.1 Draw symbol for 3 input NOR gate with truth table.
OR Draw truth table for NOR gate.

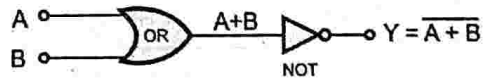
☑ **Ans. : Answer includes following points :**

A. Definition and description of NOR Gate
B. Logic symbol of NOR Gate
C. Logical or Boolean expression
D. Logical operation
E. Truth table
F. Input and Output Waveforms

(A) Definition and Description of NOR Gate

❐ *Definition : NOR gate is a logic circuit whose output is HIGH when all its inputs are LOW.*

- NOR gate means **NOT-OR operation.**
- It is a combination of NOT and OR gate. Fig. 3.9.1 shows NOR operation as NOT-OR operation.



(1B9) Fig. 3.9.1 : NOR operation as NOT-OR operation

- NOR gate has two or more inputs and one output.
- A NOR gate will have output 1 only if all its inputs are 0.
- The output of the NOR gate will be logic 0, if any of its inputs are at logic 1.
- It is called **universal logic gate** as it can be used to implement any Boolean function.

(B) Logic symbol of NOR Gate

(1B10)(a) Symbol for a 2-input NOR gate

(1B10A)(b) Symbol for a 3-input NOR gate

Fig. 3.9.2

- Bubble on the NOR gate represents NOT operation or inversion or complementation.

(C) Logical or Boolean Expression of NOR Gate

$Y = \overline{A + B}$ $Y = \overline{A + B + C}$

[A + B represents OR operation and the bar represents NOT operation]

It is read as **"Y equals NOT (A OR B)"**

(D) Logical Operation

NOT OR or NOR

(E) Truth table of NOR Gate

Table 3.9.1 : Truth table of 2-input NOR gate

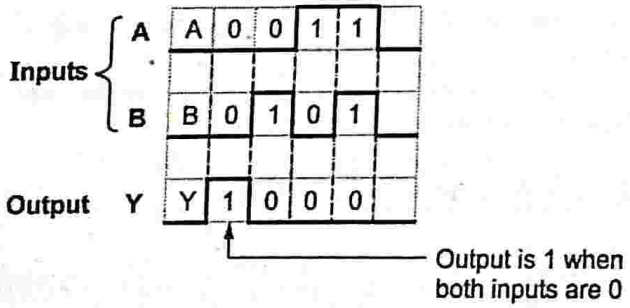
Input		Output
A	B	$Y = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Output is 1 when all inputs are 0

Table 3.9.2 : Truth table of 3-input NOR gate

Input			Output
A	B	C	$Y = \overline{A + B + C}$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(F) Input and Output Waveforms of NOR Gate



(1B52)**Fig. 3.9.3 : Input and Output waveforms for 2 input NOR gate**

3.10 EX-OR GATE

Q. 3.10.1 Which gates are also known as controlled NOT gate?

Ans. :

X-OR gate is also called as controlled NOT gate because if one input of the gate is tied to 1 and the other is X then the output is $\overline{X}$).

(1B55)**Fig. 3.10.1 : XOR as controlled NOT gate**

Q. 3.10.2 Draw symbol for 3 input EX-OR gate with truth table.

Ans. : Answer includes following points :

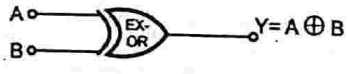
A. Definition and description of EX-OR Gate
B. Logic symbol of EX-OR Gate
C. Logical or Boolean expression
D. Truth table
E. Input and Output Waveforms
F. Applications of EX-OR gate

(A) Definition and Description of EX-OR Gate

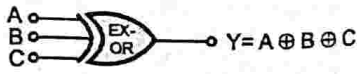
Definition : An EX-OR gate is a logic circuit whose output is high when both the inputs are not same.

- EX-OR gate is a gate that has two or more inputs and one output.
- An EX-OR gate will have output "1" when the inputs are not equal.
- The output of the EX-OR gate is 0 when both the inputs are same, i.e., A = B = 0 or A = B = 1.
- It is also called as **inequality detector** or **anti-coincidence gate.**
- It is used in many digital circuits. The output of an EX-OR gate is modulo-sum of the two inputs.

(B) Logic Symbol of EX-OR Gate



(1B11)(a) **Symbol for a 2-input EX-OR gate**



(1B11A)(b) **Symbol for a 3-input EX-OR gate**

Fig. 3.10.2

(C) Logical or Boolean Expression of EX-OR gate

$Y = A \text{ EX-OR } B = A \oplus B = A\bar{B} + \bar{A}B$

$Y = A \text{ EX-OR } B \text{ EX-OR } C = A \oplus B \oplus C$

(D) Truth Table of EX-OR Gate

Table 3.10.1 : Truth table of 2-input EX-OR gate

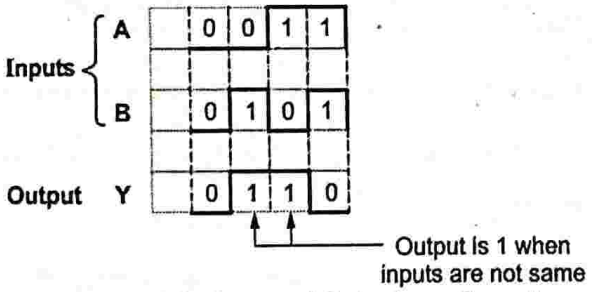
Input		Output
A	B	Y = A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.10.2 : Truth table of 3-input EX-OR gate

Input			Output
A	B	C	Y = A ⊕ B ⊕ C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Output is 1 when inputs are not equal

(E) Input and Output Waveforms of EX-OR Gate



(1B53)Fig. 3.10.3 : Input and Output waveforms for 2 input EX-OR gate

(F) Applications of EX-OR Gate

1. Magnitude comparator
2. Binary-to-gray converter/Gray-to-binary converter
3. Parity generator/checker
4. Modulo-2-adder
5. Adder and subtractor circuits
6. Combinational logic circuit minimization

3.11 EX-NOR GATE

Module 1

Q. 3.11.1 Draw symbol for 3 input EX-NOR gate with truth table.

Ans. : Answer includes following points :

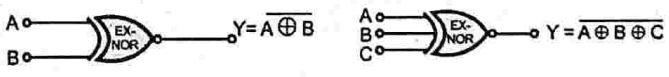
A. Definition and description of EX-NOR Gate
B. Logic symbol of EX-NOR Gate
C. Logical or Boolean expression
D. Truth table
E. Input and Output Waveforms
F. Application of EX-NOR gate

(A) Definition and description of EX-NOR Gate

Definition : EX-NOR gate is a logic circuit whose output is HIGH when both its inputs are equal.

- An EX-NOR gate is NOT-EX-OR operation. It is a combination of NOT and EX-OR gate. An EX-NOR gate has two inputs and one output.
- The output of the EX-NOR gate will be high (logic 1) when both the inputs are same, i.e., A = B = 0 or A = B = 1.
- The output of the EX-NOR gate will be low (logic 0) when both the inputs are not equal.
- It is also called as **coincidence gate** or **equality detector.**

(B) Logical Symbol of the EX-NOR Gate



(1B12)(a) Symbol for a 2-input EX-NOR gate

(1B12A)(b) Symbol for a 3-input EX-NOR gate

Fig. 3.11.1

(C) Logical or Boolean Expression of EX-NOR Gate

$Y = A \text{ EX-NOR } B = A \odot B = AB + \bar{A}\bar{B} = \overline{A \oplus B} = \overline{A\bar{B} + \bar{A}B}$

$Y = A \text{ EX-NOR } B \text{ EX-NOR } C = A \odot B \odot C = \overline{A \oplus B \oplus C}$

(D) Truth Table of EX-NOR Gate

Table 3.11.1 : Truth table of 2-input EX-NOR gate

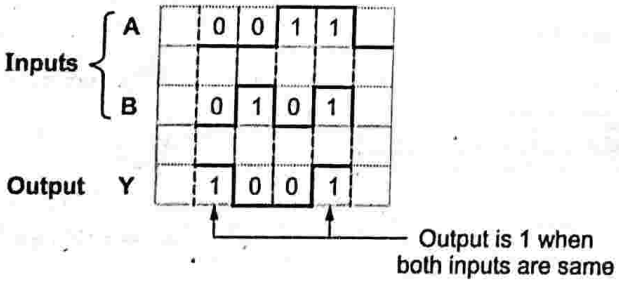
Input		Output
A	B	Y = A ⊙ B
0	0	1
0	1	0
1	0	0
1	1	1

Table 3.11.2 : Truth table of 3-input EX-NOR gate

Input			Output
A	B	C	Y = A ⊙ B ⊙ C
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Output is 1 when both inputs are same

(E) Input and Output Waveforms of EX-NOR Gate



(1B54)**Fig. 3.11.2 : Input and Output waveforms for 2 input EX-NOR gate**

(F) Applications of EX-NOR Gate

1. In error detecting circuits to detect even or odd parity bits in digital data transmission circuits.
2. Arithmetic and encryption circuits.
3. Comparators.

Q. 3.11.2 How will you realize EX-OR and XNOR gates with three or more inputs ?

Ans. :

- Three or more input EX-OR and X-NOR gates do not exist.
- A number of two input EX-OR and EX-NOR gates can be used to realize EX-OR and EX-NOR gates with three or more inputs.

3.12 BOOLEAN ALGEBRA

Q. 3.12.1 What is Boolean algebra ? Explain its characteristics.

Ans. :

Definition of Boolean Algebra : Boolean algebra is the mathematics used for analyzing the digital gates and circuits.

Characteristics of Boolean algebra

1. It was developed in 1854 by an Irish mathematician George Boole.
2. Boolean algebra is a method to algebraically express the logic function. It is also called as **"Binary algebra"** or **"logical algebra"**.
3. Any complex logical statement can be expressed with the help of a Boolean function.
4. It can also be used for the simplification of complex logic statements with the help of Karnaugh maps or Quine-McCluskey method.
5. Boolean algebra uses a set of rules and laws in order to define the operation of a digital circuit.

Q. 3.12.2 Differentiate between ordinary algebra and Boolean algebra.

Ans. :

– Boolean algebra is simpler than ordinary algebra. The differences between the two are as follows :

Sr. No	Ordinary algebra	Boolean algebra
1.	The letter symbols used in ordinary algebra can take any number of values.	In Boolean algebra, there are only two values, 0 and 1, for any symbol or variable.
2.	In ordinary algebra, a variable has numerical significance. E.g., $A \cdot A = A^2$.	In Boolean algebra, a variable has only logical value, it doesn't have any numerical significance. E.g., $A \cdot A = A$.
3.	In ordinary algebra '·' represents multiplication, '+' represents addition, '–' represents subtraction.	Arithmetic operations like addition, subtraction, multiplication, division are not performed in Boolean algebra. There are no negative numbers, fractions, square, square root, imaginary numbers in Boolean algebra. In Boolean algebra '·' indicates AND operation whereas '+' indicates OR operation. AND, OR and NOT are the three basic functions or operations performed in Boolean algebra. NAND, NOR, X-OR and X-NOR are derived operations performed in Boolean algebra.

3.12.1 Variables, Literals and Terms in Boolean Expressions

Q. 3.12.3 Define Variables, Literals and Terms in Boolean Expressions.

Ans. :

❒ *Definition of Variables : Variables are symbols (generally letters) used in Boolean expressions.*

– Available represented in complemented or uncomplemented form is called a literal.

– Binary vvariables have only two values, **logic 1 (high)** and **logic 0 (low).** E.g., A, B are two variables.

❒ *Definition of Literals : Every occurrence of a variable or its complement is called as a literal.*

e.g. $\bar{A} + A \cdot \bar{B} + ABC$ ...(3.12.1)

– In the above expression, there are 6 (six) literals. The complement of a variable is not considered as a separate variable.

❒ *Definition of Terms : A term is defined as the expression formed by operations and literals at one level.*

– In equation (3.12.1), there are 4 terms including three AND terms and the OR term that combines the first level AND terms.

3.12.2 Rules in Boolean Algebra

Q. 3.12.4 List the rules for using Boolean algebra.

Ans. :

Following are the set of rules to be performed while using Boolean algebra.

1. Variables or symbols have only two values : logic 1 for high and logic 0 for low.
2. The complement of a variable can be represented by a bar (–).
   E.g., the complement of A is $\bar{A}$.
   If $A = 0$ then $\bar{A} = 1$ and
   If $A = 1$ then $\bar{A} = 0$.
3. The ORing of variables can be represented by a (+) sign.
   E.g., $A + B$ is read as A OR B.
4. The ANDing of variables can be represented by (•) sign.
   E.g., $A \cdot B$ is read as A AND B.
5. In Boolean algebra any Boolean function can be proved by **perfect induction method.** In this method, the Boolean function is verified for every possible combination of values of the variables with the help of a truth table.

3.13 THEOREMS AND PROPERTIES OF BOOLEAN ALGEBRA

Q. 3.13.1 What are axioms or postulates in Boolean algebra ? List the postulates.

Ans. :

Definition of Axioms or Postulates In Boolean Algebra : In Boolean algebra, axioms or postulates are a group of logical expressions that we accept without any proof.

- Axioms are the basic definitions of the three basic logical operations : AND, OR and NOT. With the help of these axioms, we can build theorems.

Table 3.13.1 : Basic Axioms/Postulates of boolean algebra

Axiom/Postulate	Operation
$\bar{1} = 0$	NOT operation
$\bar{0} = 1$	
$0 \cdot 0 = 0$	AND operation
$0 \cdot 1 = 0$	
$1 \cdot 0 = 0$	
$1 \cdot 1 = 1$	
$0 + 0 = 0$	OR operation
$0 + 1 = 1$	
$1 + 0 = 1$	
$1 + 1 = 1$	

Postulates of Boolean Algebra

Table 3.13.2 : Lists the postulates of Boolean algebra

Postulate No.	Postulate	Comments
1	Result of operation is 1 or 0	$1, 0 \in B$
2a	$A + 0 = A$	Identity laws
2b	$A \cdot 1 = A$	
3a	$A + B = B + A$	Commutative laws
3b	$A \cdot B = B \cdot A$	
4a	$A (B + C) = AB + AC$	Distributive laws
4b	$A + BC = (A + B) (A + C)$	
5a	$A + \bar{A} = 1$	Complementation laws
5b	$A \cdot \bar{A} = 0$	

3.13.1 Laws of Boolean Algebra

Q. 3.13.2 What do you understand by laws of Boolean algebra ?

Ans. :

Definition of Laws of Boolean Algebra : A set of rules that help to reduce the number of logic gates required for performing a specific logic operation are called as Laws of Boolean Algebra.

Q. 3.13.3 State the Boolean algebra laws used in k-map simplification.

MU - Q. 2(a)(i), Dec. 14, 5 Marks

Ans. :

The basic laws of Boolean algebra are as follows :

1. Complementation laws
2. Commutative laws
3. Associative laws
4. Distributive laws

1. Complementation Laws

Complement means to invert or negate, i.e., change 0's to 1's or 1's to 0's.

Law 1 : If A = 0, then $\bar{A} = 1$

Law 2 : If A = 1, then $\bar{A} = 0$

2. Commutative Laws

Commutative law states that modifying the sequence of variables does not affect the output/result.

Law 1 : A + B = B + A

- It states that A OR B produces the same output as B OR A ; i.e., the order in which variables are ORed does not change the output/result.

Table 3.13.3 : Truth table for commutative law for OR gates

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

=

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

- From truth Table 3.13.3 it is proved that **A + B = B + A**

Law 2 : AB = BA

- It states that A AND B produces the same output as B AND A; i.e., the order in which variables are ANDed does not change the output/result.

Table 3.13.4 : Truth table for commutative law for AND gates

A	B	AB		B	A	BA
0	0	0	=	0	0	0
0	1	0		0	1	0
1	0	0		1	0	0
1	1	1		1	1	1

- From truth Table 3.13.4, it is proved that **AB = BA.**
- Commutative law can be extended to any number of variables.

e.g., (1) A + B + C = C + A + B = B + C + A = B + A + C
(2) A · B · C = B · C · A = B · A · C

☞ 3. Associative Laws

Associative law states that irrespective of grouping of variables, the result remains the same.

Law 1 : A + (B + C) = (A + B) + C

Law 2 : A · (BC) = (AB) C

☞ 4. Distributive Laws

Distributive laws support multiplying or factoring of expressions

Law 1 : A (B + C) = AB + AC

- It states that ORing of many variables and ANDing the result with a single variable is equivalent to ANDing that single variable with each variable and then ORing the remaining terms. The truth table for proving the result is shown in Table 3.13.5.

Table 3.13.5 : Truth table for distributive law of OR gates

A	B	C	(B + C)	A (B + C)		A	B	C	AB	AC	AB + AC
0	0	0	0	0		0	0	0	0	0	0
0	0	1	1	0		0	0	1	0	0	0
0	1	0	1	0		0	1	0	0	0	0
0	1	1	1	0	=	0	1	1	0	0	0
1	0	0	0	0		1	0	0	0	0	0
1	0	1	1	1		1	0	1	0	1	1
1	1	0	1	1		1	1	0	1	0	1
1	1	1	1	1		1	1	1	1	1	1

- From truth Table 3.13.5 it is proved that **A (B + C) = AB + AC.**

Law 2 : A + BC = (A + B) (A + C)

- It states that ANDing of many variables and ORing the result with a single variable is equivalent to ORing that single variable with each of several variables and then ANDing the sums. The truth table for proving the result is shown in Table 3.13.6.

Table 3.13.6 : Truth table for distributive law of AND gates

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

From truth Table 3.13.6 it is proved that

A + BC = (A + B) (A + C)

3.13.2 Duality Property

Q. 3.13.4 State and explain principle of duality.

☑ **Ans. :**

☞ Definition of Duality

- It states that every Boolean expression derived from the postulates of Boolean algebra is valid if the operators (OR (+) and AND (·)) and identity elements (0, 1) are interchanged.
- **Duality principle** is an important property of Boolean algebra.
- If we need to find the dual of an expression, then we interchange the AND and OR operators and replace 1's by 0's and vice-versa. The Huntington postulates are listed in pairs as seen in Table 3.13.2 Part (a) and (b).
  e.g., Dual of relation A · 0 = 0 is A + 1 = 1
  We can obtain the dual of an expression as follows :
  1. Replace OR by AND operation.
  2. Replace AND by OR operation.
  3. Replace 0 by 1.
  4. Replace 1 by 0.
- Using the duality theorem we can create new Boolean expressions from the given expression.
- Duality theorem can be verified by constructing truth tables for both size of the given expression.
- Table 3.13.7 lists the Boolean laws and theorems and their duals.

Table 3.13.7

Sr. No.	Expression	Dual
1.	$\bar{0} = 1$	$\bar{1} = 0$
2.	$0 \cdot 1 = 0$	$1 + 0 = 1$
3.	$0 \cdot 0 = 0$	$1 + 1 = 1$
4.	$1 \cdot 1 = 1$	$0 + 0 = 0$
5.	$A \cdot 0 = 0$	$A + 1 = 1$
6.	$A \cdot 1 = A$	$A + 0 = A$
7.	$A \cdot A = A$	$A + A = A$
8.	$A \cdot \bar{A} = 0$	$A + \bar{A} = 1$
9.	$A \cdot B = B \cdot A$	$A + B = B + A$
10.	$A \cdot (B \cdot C) = (A \cdot B) C$	$A + (B + C) = (A + B) + C$
11.	$A \cdot (B + C) = AB + AC$	$A + BC = (A + B)(A + C)$
12.	$A (A + B) = A$	$A + AB = A$
13.	$A \cdot (A \cdot B) = A \cdot B$	$A + A + B = A + B$
14.	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$
15.	$(A + B)(\bar{A} + C)(B + C) = (A + B)(\bar{A} + C)$	$AB + \bar{A}C + BC = AB + \bar{A}C$

3.13.3 Basic Theorems

Q. 3.13.5 Prove the basic theorems of Boolean algebra.

Ans. :

Idempotence Laws

Theorem 1(a) : A + A = A

If A = 0, then 0 + 0 = 0
If A = 1, then 1 + 1 = 1

(Theorem 1(a) and 1(b) are also called as Idempotence laws. Idempotence indicates same value, i.e., A + A = A)

Theorem 1(b) : A · A = A (Idempotence law)

If A = 0 $0 \cdot 0 = 0$
If A = 1 $1 \cdot 1 = 1$
} i.e., $A \cdot A = A$

Proof

$A + A = (A + A) \cdot 1$ ($\because A \cdot 1 = A$ postulate 2(b))

$A + A = (A + A) \cdot (A + \bar{A})$ ($\because A + \bar{A} = 1$ postulate 5(a))

$A + A = A + A\bar{A}$

($\because A + BC = (A + B)(A + C)$ postulate 4(b))

$A + A = A + 0$ ($\because A \cdot \bar{A} = 0$ postulate 5(b))

$\therefore A + A = A$ ($\because A + 0 = A$ postulate 2(a))

Thus, proved.

Theorem 1(b) : A · A = A (Idempotence law)

If A = 0 $0 \cdot 0 = 0$
If A = 1 $1 \cdot 1 = 1$
} i.e., $A \cdot A = A$

Proof

$A \cdot A = A \cdot A + 0$ ($\because A + 0 = A$ postulate 2(a))

$A \cdot A = A \cdot A + A\bar{A}$ ($\because A\bar{A} = 0$ postulate 5(b))

$A \cdot A = A(A + \bar{A})$ ($\because A(B + C) = AB + BC$ postulate 4(a))

$A \cdot A = A \cdot 1$ ($\because A + \bar{A} = 1$ postulate 5(a))

$A \cdot A = A$ ($\because A \cdot 1 = A$ postulate 2(b))

Thus, proved.

Identity laws

Theorem 2(a) : A + 1 = 1 (Identity law)

If A = 0, then 0 + 1 = 1
If A = 1, then 1 + 1 = 1
} A + 1 = 1

Proof

$A + 1 = 1 \cdot (A + 1)$ [$\because A \cdot 1 = 1$ postulate 2(b)]

$A + 1 = (A + \bar{A}) \cdot (A + 1)$ [$\because A + \bar{A} = 1$ postulates 5(a)]

$A + 1 = A + \bar{A} \cdot 1$

[$\because A + BC = (A + B)(A + C)$ postulate 4(b)]

$A + 1 = A + \bar{A}$ [$\because A \cdot 1 = 1$ postulate 2(b)]

$\therefore A + 1 = 1$ [$A + \bar{A} = 1$ postulate 5(a)]

Thus, proved.

Theorem 2(b) : A · 0 = 0 (Identity law)

If A = 0, then $0 \cdot 0 = 0$
If A = 1, then $1 \cdot 0 = 0$
} $A \cdot 0 = 0 \cdot A = 0$

Proof

$A \cdot 0 = 0$ by duality of theorem 2(a)

Involution laws

Theorem 3 : $\bar{\bar{A}} = A$ (Involution law)

Proof : Let $\bar{A}$ be complement of A and $\bar{\bar{A}}$ be complement of $\bar{A}$

$\bar{\bar{A}} = \bar{\bar{A}} + 0$ [$\because A + 0 = A$ postulate 2(a)]

$\bar{\bar{A}} = \bar{\bar{A}} + A\bar{A}$ [$\because A \cdot \bar{A} = 0$ postulate 5(b)]

$\bar{\bar{A}} = (\bar{\bar{A}} + A)(\bar{\bar{A}} + \bar{A})$

[$\because A + BC = (A + B)(A + C)$ postulate 4(b)]

$\bar{\bar{A}} = (\bar{\bar{A}} + A) \cdot 1$ ($\because A + \bar{A} = 1$ postulate 5(a))

$\bar{\bar{A}} = (\bar{\bar{A}} + A) \cdot (A + \bar{A})$ ($\because A + \bar{A} = 1$ postulate 5(a))

$\bar{\bar{A}} = A + \bar{A} \cdot \bar{\bar{A}}$

[$\because (A + B)(A + C) = A + BC$ postulate 4(b)]

$\bar{\bar{A}} = A + 0$ [$\because A\bar{A} = 0$ postulate 5(b)]

$\therefore \bar{\bar{A}} = A$ [$\because A + 0 = A$ postulate 2(a)]

Hence proved.

Associative law

Theorem 4(a) : (Associative law)

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

It states that A ORed with B OR C is same as A OR B ORed with C

Table 3.13.8 : Truth table for associative law of OR gates

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

- From truth Table 3.13.8 we can see that for all possible combinations of A, B, and C : $(A + B) + C = A + (B + C)$. Thus, **the associative law of OR gates is proved.**

Theorem 4(b) : (Associative law)

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

- It states that A ANDed with B AND C is same as A AND B ANDed with C.
- Table 3.13.9 shows truth table for associative law of AND gates.

Table 3.13.9 : Truth table for associative law of AND gates

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

- From truth Table 3.13.9 we can see that for all possible combinations of A, B, and C, $A \cdot (B \cdot C) = (A \cdot B) \cdot C$. Thus, **associative law of AND gates is proved.**

DeMorgan's Theorems

Q. 3.13.6 State and prove De Morgans theorem.

MU - Q. 1(a), Dec. 13, 5 M, Q. 1(g), May 16, Q. 1(g), May 16, 2 M, Q. 1(e), Dec. 18, Q. 1(d), May 19, 4 M

Ans. :

Theorem 5(a) : DeMorgan's Theorem

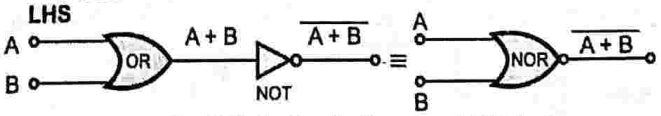
DeMorgan's theorem 1

Definition : DeMorgan's theorems are used in Boolean algebra. $\overline{A + B} = \bar{A}\,\bar{B}$. *This law states that the complement of sum of variables is equal to the product of their self complements.*

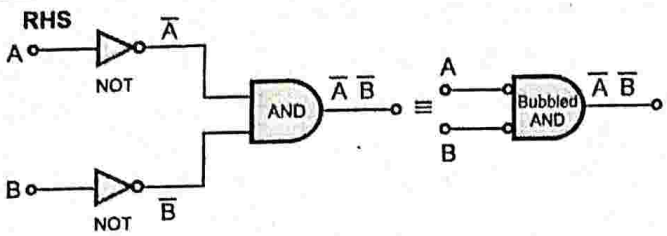
- The **LHS** of the expression represents **NOR** gate with inputs A and B. The **RHS** of the expression represents AND gate with inputs complemented. This AND gate is equivalent to **"Bubbled AND"** gate.

NOR = Bubbled AND ...(DeMorgan's theorem)

- We will prove it with the help of logic diagram and truth tables.



(1B13) **Fig. 3.13.1 : Logic diagram of LHS of DeMorgan's theorem**



(1B14) **Fig. 3.13.2 : Logic diagram of RHS of DeMorgan's theorem**

Table 3.13.10 : Truth table for DeMorgan's theorem

A	B	A + B	$\overline{A+B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

=

$\overline{A}$	$\overline{B}$	$\overline{A}\,\overline{B}$
1	1	1
1	0	0
0	1	0
0	0	0

Thus, $\overline{A+B} = \overline{A}\,\overline{B}$

Hence proved.

☞ Theorem 5(b) : DeMorgan's Theorem

☞ DeMorgan's theorem 2

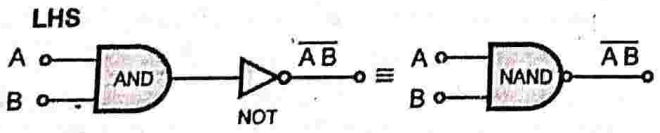
Definition : DeMorgan's theorems are used in Boolean algebra. $\overline{AB} = \overline{A} + \overline{B}$. This law states that the complement of the product of variables is equal to the sum of their self complements.

- The **LHS** of the theorem represents **NAND** gate with inputs A and B. The **RHS** of the theorem represents OR gate with complemented inputs. This OR gate is called as **"Bubbled OR"** gate.

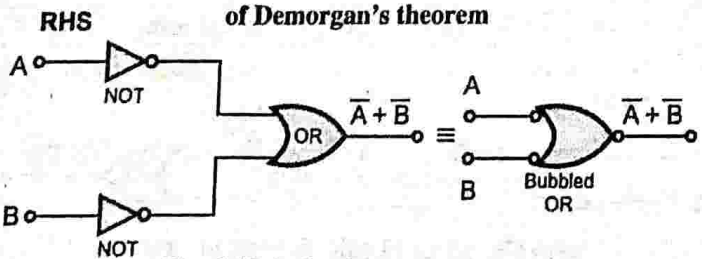
Thus,

NAND = Bubbled OR ...(DeMorgan's theorem)

We will prove this using logic diagram and DeMorgan's theorem.



(1B15) **Fig. 3.13.3 : Logic diagram of LHS of Demorgan's theorem**



(1B16) **Fig. 3.13.4 : Logic diagram of RHS of Demorgan's theorem**

Table 3.13.11 : Truth table for Demorgan's theorem

A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

=

A	B	$\overline{A}$	$\overline{B}$	$\overline{A}+\overline{B}$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

Thus, $\overline{AB} = \overline{A} + \overline{B}$

Hence proved.

☞ Absorption theorems

☞ Theorem 6(a) : Absorption

$$A + AB = A$$

Proof

$A + AB = A \cdot 1 + AB$ [$\because A \cdot 1 = A$ postulate 2(b)]

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

[$\because A(B + C) = AB + AC$ postulate 4(a)]

$A + AB = A(B + 1)$ [$\because A + B = B + A$ postulate 3(a)]

$A + AB = A \cdot 1$ [$\because A + 1 = 1$ theorem 2(a)]

$A + AB = A$ [$\because A \cdot 1 = A$ postulate 2(b)]

Hence proved.

☞ Theorem 6(b) : Absorption

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

Proof

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

[$\because A(B + C) = AB + AC$ postulate 4(a)]

$A(A + B) = A + AB$ [$\because A \cdot A = A$ theorem 1(b)]

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

[$\because A(B + C) = AB + AC$ postulate 4(a)]

$A(A + B) = A \cdot 1$ [$\because A + 1 = 1$ theorem 2(a)]

$A(A + B) = A$ [$\because A \cdot 1 = A$ postulate 2(b)]

Hence proved.

☞ Summary

While evaluating a **Boolean expression** the operator precedence is,

1. **Parentheses**
2. **NOT gate**
3. **AND gate**
4. **OR gate**

3.14 BOOLEAN FUNCTIONS

Q. 3.14.1 What is Boolean functions ? Explain how it can be represented using :
1. Algebraic equation
2. Truth table
3. Logic diagram

Ans. :

- Boolean algebra comprises of logic operations and binary variables. A binary variable takes values 0 and 1.

Definition of Boolean function : An expression that is formed using binary variables, and operators, NOT, AND, OR, parentheses and 'equal' to sign is called as a **Boolean function.**

- For a given value of variables, the value of a Boolean function can only be 0 or 1.

e.g. $F_1 = A\bar{B}C$ ...(3.14.1)

Boolean function **Boolean expression**

The function $F_1 = 1$ if A = 1, B = 1 and C = 1,

otherwise, $F_1 = 0$

- The Boolean function F_1 is an example of Boolean function represented as an **algebraic equation.**
- The Boolean function can also be represented with the help of a **truth table.** Depending on the number of variables in an expression, a truth table has 2^n possible combinations of inputs where n is the number of variables in a Boolean expression.
- e.g. For 3 variables there are $2^3 = 8$ possible combination of inputs (1's and 0's), i.e., there will be 8 rows in the truth table.
- The number of rows in truth table is equal to 2^n. For every possible combination of inputs, the value of Boolean function is either 0 or 1.

e.g. $F_2 = A + \bar{B}C$ ...(3.14.2)

- The truth table for Equation (3.14.2) is shown in Table 3.14.1.

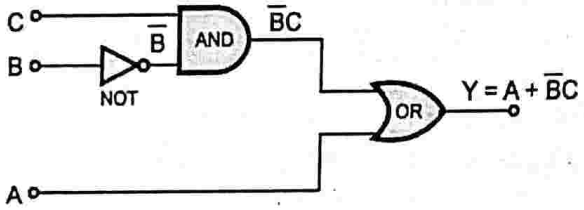
Table 3.14.1

$2^3 = 8$ Possible input combinations

A	B	C	$\bar{B}C$	$A + \bar{B}C$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	1	1
1	1	0	0	1
1	1	1	0	1

$F_2 = A + \bar{B}C$ Output function values corresponding to different input combinations

- A Boolean function can be translated / transformed from an algebraic expression into a logic diagram comprising of AND, OR, NOT gates.
- Fig. 3.14.1 shows the logic diagram for $F_2 = A + \bar{B}C$. The logic diagram comprises a NOT gate for every variable in complement form, an AND gate for every term and an OR gate for combining the terms in the Boolean expression.



(1B17) **Fig. 3.14.1 : Logic diagram for implementation of**

$$Y = A + \bar{B}C$$

3.15 BOOLEAN FUNCTION REDUCTION USING BOOLEAN LAWS

Ex. 3.15.1

Prove the following Boolean expressions.

1. $A + AB = A$
2. $A + \bar{A}B = A + B$
3. $(A + B)(A + C) = A + BC$

Ans. :

1. A + AB = A

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

$= A \cdot 1$ ($\because 1 + A = 1$ Theorem 2(a) Identity law)

$= A$ ($\because A \cdot 1 = A$ postulate 2(b))

= RHS. Hence proved

Thus, **A + AB = A**

2. $A + \bar{A}B = A + B$

LHS = $A + \bar{A}B$

= $A + \underline{AB} + \underline{\bar{A}B}$ ($\because A + AB = A$)

= $A + B(A + \bar{A})$

= $A + B$

($\because A + \bar{A} = 1$ postulate 5(a) Complementation law)

= RHS

Hence proved

Thus $A + \bar{A}B = A + B$

3. $(A + B)(A + C) = A + BC$

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

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

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

($\because A \cdot A = A$ Idempotence law Theorem 1(b))

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

= $A + AC + BC$

($\because A + 1 = 1$ Theorem 2(a) Identity law)

= $A(1 + C) + BC$

= $A + BC$

($\because A + 1 = 1$ Identity law Theorem 2(a))

= RHS

Hence proved.

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

Ex. 3.15.2

Apply DeMorgan's theorem to solve the following :

1. $\overline{(A + \bar{B}C)}(A\bar{B} + ABC) = 0$
2. $A[B + \bar{C}\,\overline{(AB + A\bar{C})}] = AB$

☑ **Ans. :**

1. $\overline{(A + \bar{B}C)}(A\bar{B} + ABC) = 0$

LHS = $\overline{(A + \bar{B}C)}(A\bar{B} + ABC)$

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

($\because \overline{A + B} = \bar{A} \cdot \bar{B}$ by DeMorgan's theorem)

= $\bar{A}BC(A\bar{B} + ABC)$ ($\because \bar{\bar{A}} = A$ Involution law)

= $\bar{A}BC \cdot A\bar{B} + \bar{A}BC \cdot ABC$

= $0 + 0$ ($\because A \cdot \bar{A} = 0$ postulate 5(b))

= 0

Thus, $\overline{(A + \bar{B}C)}(A\bar{B} + ABC) = 0$

2. $A[B + \bar{C}\,\overline{(AB + A\bar{C})}] = AB$

LHS = $A[B + \bar{C}\,\overline{(AB + A\bar{C})}]$

= $A[B + \bar{C}(\overline{AB} \cdot \overline{A\bar{C}})]$

($\because \overline{A + B} = \bar{A} \cdot \bar{B}$ DeMorgan's law)

$Y = A[B + \bar{C}(\bar{A} + \bar{B})(\bar{A} + \bar{\bar{C}})]$

($\because \overline{AB} = \bar{A} + \bar{B}$ DeMorgan's theorem)

$Y = A[B + \bar{C}(\bar{A} + \bar{B})(\bar{A} + C)]$ ($\because \bar{\bar{A}} = A$)

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

$Y = A[B + \bar{C}(\bar{A} + \bar{A}\bar{B} + \bar{A}C + \bar{B}C)]$

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

$Y = AB + A \cdot \bar{A}\bar{C} + A\bar{C} \cdot \bar{A}\bar{B} + A\bar{C} \cdot \bar{A}C + A\bar{C}\bar{B} \cdot C$

$Y = AB$ ($\because A \cdot \bar{A} = 0$ postulate 5(b))

UEx. 3.15.3 MU - Q. 2(b)(i), Dec. 13, 5 Marks

Simplify the following Boolean expression.

$Y = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

☑ **Ans. :**

$Y = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

As $A + A = A$ by Idempotence law we will add two more ABC terms

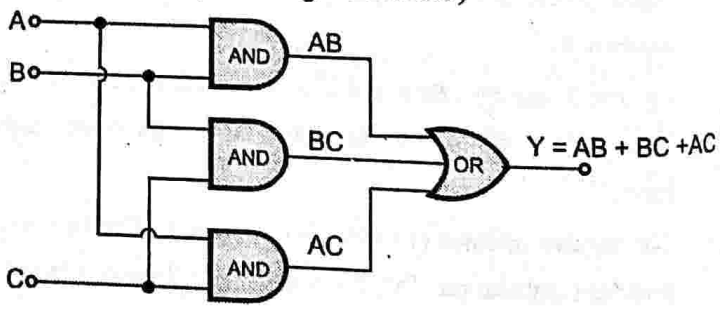
$Y = \underline{\bar{A}BC} + \underline{A\bar{B}C} + \underline{AB\bar{C}} + \underline{ABC} + \underline{ABC} + ABC$

$Y = \bar{A}BC + ABC + A\bar{B}C + ABC + AB\bar{C} + ABC$

$Y = BC(\bar{A} + A) + AC(\bar{B} + B) + AB(\bar{C} + C)$

$Y = BC + AC + AB$ ($\because A + \bar{A} = 1$ postulate 5(a))

Logic diagram (Refer Fig. Ex. 3.15.3)



(1B23) **Fig. Ex. 3.15.3 : Logic diagram $y = AB + BC + AC$**

UEx. 3.15.4 **MU - Q. 1(f), May 17, 2 Marks**

Prove that "A positive logic AND operation is equivalent to a negative logic OR operation".

Ans. : For proving "A positive logic AND operation is equivalent to a negative logic OR operation", we will use truth tables.

Truth table for positive logic AND operation

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

≡

Truth table for negative logic OR operation

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

Thus proved.

Similarly "A negative logic AND operation is equivalent to a positive logic OR operation."

UEx. 3.15.5 **MU - Q. 1(d), May 17, 2 Marks**

Simplify (B + A) (B + D) (A + C) (C + D)

Ans. :

$Y = (B + D)(B + A)(A + C)(C + D)$

$Y = [B \cdot B + B \cdot D + AB + AD][AC + C \cdot C + AD + CD]$

$Y = [B + BD + AB + AD][AC + C + AD + CD]$

$(\because A \cdot A = A$ Theorem 1(b))

$Y = [B(1 + D) + AB + AD][C(A + 1) + CD + AD]$

$Y = [B + AB + AD][C + CD + AD]$

$(\because A + 1 = 1$ Theorem 2(a) Identity law)

$Y = [B(1 + A) + AD][C(1 + D) + AD]$

$Y = (B + AD)(C + AD)$

$(\because A + 1 = A$ Theorem 2(a) Identity law)

$Y = BC + ACD + ABD + AD \cdot AD$

$Y = BC + ACD + ABD + AD$

$(\because A \cdot A = A$ Theorem 1(b))

$Y = BC + ACD + AD(1 + B)$

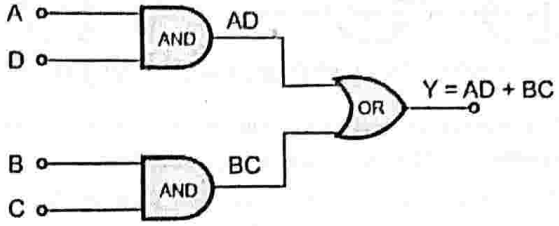
$Y = BC + ACD + AD$

$(\because A + 1 = 1$ Theorem 2(a) Identity law)

$Y = BC + AD(C + 1)$

$\mathbf{Y = BC + AD}$ $(\because A + 1 = 1$ Theorem 2(a) Identity law)

Logic diagram (Refer Fig. Ex. 3.15.5)



(1B24) **Fig. Ex. 3.15.5 : Logic diagram of (Ex. 3.15.5)**

3.16 NAND GATE AS UNIVERSAL GATE

Q. 3.16.1 Prove using Boolean algebra "NAND gate is universal gate".

MU - Q. 3(b), Dec. 16, 10 M.
Q. 1(a), Dec. 17, 2 M Q. 1(f), May 18, 2 M

Ans. :

The **NAND gate** is called as **"Universal Gate"** because any Boolean expression can be implemented with the help of NAND gates. We will implement all the basic gates with NAND gate.

3.16.1 NOT Gate (Inverter)

Q. 3.16.2 Implement NOT gates using NAND gates only.

Ans. :

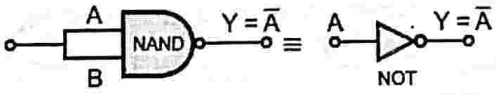
Fig. 3.16.1 shows NOT gate using NAND gate. A NAND gate can be used as a NOT gate by connecting all its inputs together. Both the inputs of the NAND gate are connected.

$\therefore$ Input = A = B = A

Output = $\overline{A \cdot B}$

= $\overline{A \cdot A}$ $(\because A = B)$

$Y = \overline{A}$ $(\because A \cdot A = A)$



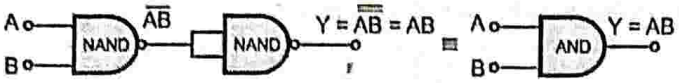
(1B32)**Fig. 3.16.1 : NOT gate using NAND gate**

3.16.2 AND Gate

Q. 3.16.3 Implement AND gates using NAND gates only.

Ans. :

Fig. 3.16.2 shows AND gate using NAND.



(1B33)**Fig. 3.16.2 : AND gate using NAND gate**

- To construct an AND gate from NAND gates, we need an inverter or NOT gate at the output of NAND gate.
- The double inversion cancels out and final result is AND function as shown in Fig. 3.16.2.
- The Boolean expression for AND gate is,

  $Y = AB$

  The output expression of Fig. 3.16.2

  $Y = \overline{\overline{AB}} = AB$ ($\because \overline{\overline{A}} = A$ Theorem 3 Involution)
- Thus, from output we can see that AND function is realized by NAND gates.

Truth table

Table 3.16.1

A	B	$\overline{A \cdot B}$	$Y = \overline{\overline{A \cdot B}}$	AB
0	0	1	0	0
0	1	1	0	0
1	0	1	0	0
1	1	0	1	1

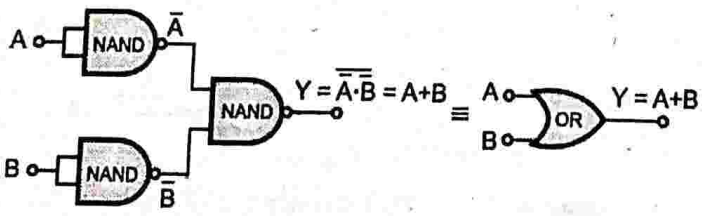
From the truth table also we can see that NAND gates can be connected to perform AND function.

3.16.3 OR Gate

Q. 3.16.4 Realize exclusive OR gate using NAND logic.

Ans. :

Fig. 3.16.3 shows the construction of an OR gate using three NAND gates.



(1B34)**Fig. 3.16.3 : OR gate using NAND gates**

- The first two NAND gates invert A and B inputs to produce outputs $\overline{A}$ and $\overline{B}$.
- The outputs are applied to inputs another NAND gate to obtain the OR function. Thus, by using three NAND gates we can realize an OR gate.

Boolean expression for the OR gate is

$Y = A + B$

Boolean expression for Fig. 3.16.3,

$Y = \overline{\overline{A} \cdot \overline{B}}$

$Y = \overline{\overline{A}} + \overline{\overline{B}}$ ($\because \overline{AB} = \overline{A} + \overline{B}$ DeMorgan's law)

$Y = A + B$ ($\because \overline{\overline{A}} = A$ Involution)

Truth table

Table 3.16.2 : Truth table for OR realization using NAND gates

A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$	$\overline{\overline{A} \cdot \overline{B}}$		A	B	A + B
0	0	1	1	1	0		0	0	0
0	1	1	0	0	1	≡	0	1	1
1	0	0	1	0	1		1	0	1
1	1	0	0	0	1		1	1	1

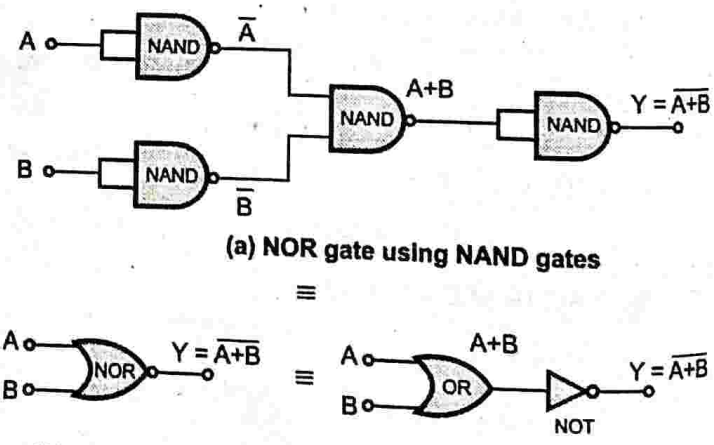
Thus, we can see that OR gate can be realized using 3 NAND gates.

3.16.4 NOR Gate

Q. 3.16.5 Implement NOR gates using NAND gates only.

Ans. :

- NOR gate can be realized using NAND gate by adding an inverter at the output of OR gate realized using NAND gates. Fig. 3.16.4 shows NOR gate using NAND gates.



(a) NOR gate using NAND gates

(b) NOR gate

(c) NOR gate using OR and NOT gate

(1B35)**Fig. 3.16.4**

- The Boolean expression for NOR gate is,

  $Y = \overline{A + B}$
- The Boolean expression for diagram shown in Fig. 3.16.4(a) is,

$$Y = \overline{\overline{A} \cdot \overline{B}}$$

$$Y = \overline{\overline{A} \cdot \overline{B}} \qquad (\because \overline{\overline{A}} = A)$$

$$Y = \overline{A + B} \qquad (\because \overline{A} \cdot \overline{B} = \overline{A + B}$$ DeMorgan's theorem)

Truth tables

Truth table for NOR gate		
A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

≡

Truth table for NOR gate using NAND gates						
A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$	$\overline{\overline{A} \cdot \overline{B}}$	$\overline{\overline{A} \cdot \overline{B}}$
0	0	1	1	1	0	1
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	1	0	0	0	1	0

Thus, we can realize NOR gate with NAND gates.

3.16.5 EX-OR Gate using NAND Gate

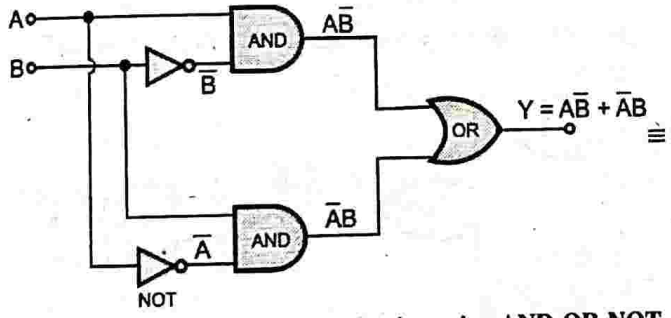
Q. 3.16.6 Implement EX-OR gates using NAND gates only.

Ans. :

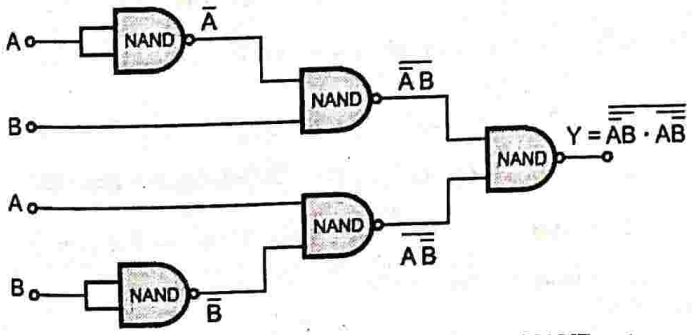
The Boolean expression for EX-OR gate is,

$$Y = A\overline{B} + \overline{A}B$$

Fig. 3.16.5 shows the implementation of EX-OR gate using basic gates and NAND gates.



(1B36) **Fig. 3.16.5(a) : EX-OR realization using AND-OR-NOT gates**



(1B37) **Fig. 3.16.5(b) : EX-OR realization using NAND gates**

The Boolean expression for circuit shown in Fig. 3.16.5(b) is,

$$Y = \overline{\overline{A}B \cdot A\overline{B}}$$

$$Y = \overline{\overline{A}B} + \overline{A\overline{B}}$$

$(\because \overline{A} \cdot \overline{B} = \overline{A + B}$ by DeMorgan's Theorem)

$\therefore Y = \overline{A}B + A\overline{B}$ ($\because \overline{\overline{A}} = A$ Involution law Theorem 3)

is the equation of EX-OR gate.

Truth tables

Truth table for X-OR gate		
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

≡

Truth table for X-OR gate using NAND gates						
A	B	$\overline{A}$	$\overline{B}$	$\overline{A\overline{B}}$	$\overline{\overline{A}B}$	$\overline{A\overline{B}} \cdot \overline{\overline{A}B}$
0	0	1	1	1	1	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	1	1	0

Thus, we can realize EX-OR gate using NAND gates.

3.16.6 EX-NOR Gate Using NAND Gate

Q. 3.16.7 Realize $y = AB + \overline{A} \cdot \overline{B}$ using NAND gates only.

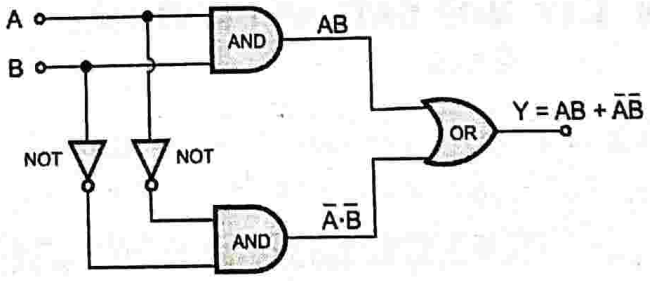
MU - Q. 1(d), May 15, 2 Marks

Ans. :

The Boolean expression for EX-NOR gate is,

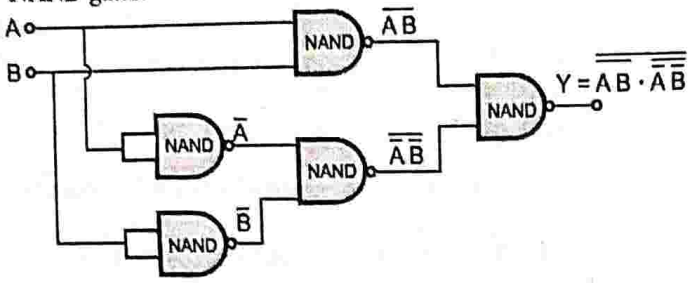
$$Y = AB + \overline{A}\,\overline{B}$$

Fig. 3.16.6(a) shows the implementation of EX-NOR gate using basic gates.



(1B38) **Fig. 3.16.6(a) : EX-NOR realization using AND-OR-NOT gates**

Fig. 3.16.6(b) shows implementation of EX-NOR gate using NAND gates.



(1B39) **Fig. 3.16.6(b) : EX-NOR realization using NAND gates**

The Boolean expression for circuit shown in Fig. 3.16.6(b) is,

$$Y = \overline{\overline{AB} \cdot \overline{\bar{A}\bar{B}}}$$

$$Y = \overline{\overline{AB}} + \overline{\overline{\bar{A}\bar{B}}}$$

(by DeMorgan's Theorem $\bar{A} \cdot \bar{B} = \overline{A + B}$)

$$Y = AB + \bar{A} \cdot \bar{B}$$

($\because \overline{\bar{A}} = A$ Involution law Theorem 3)

is the equation of EX-OR gate.

Truth tables

Truth table for EX-NOR gate		
A	B	A ⊙ B
0	0	1
0	1	0
1	0	0
1	1	1

≡

Truth table for EX-NOR gate using NAND gates						
A	B	$\bar{A}$	$\bar{B}$	$\overline{AB}$	$\overline{\bar{A}\bar{B}}$	$y = \overline{\overline{AB} \cdot \overline{\bar{A}\bar{B}}}$
0	0	1	1	1	0	1
0	1	1	0	1	1	0
1	0	0	1	1	1	0
1	1	0	0	0	1	1

Thus, we can realize X-NOR gate using NAND gates.

3.17 NOR GATE AS UNIVERSAL GATE

Q. 3.17.1 Prove OR-AND configuration is equivalent to NOR-NOR configuration.

MU - Q. 1(c), May 18, Q. 1(d), May 19, 4 Marks

☑ **Ans. :** NOR gate is called as **"Universal Gate"** because any Boolean expression can be implemented with the help of NOR gates.

3.17.1 NOT Gate (Inverter)

Q. 3.17.2 Construct NOT gate using the universal gates.

☑ **Ans. :**

Fig. 3.17.1 shows NOT gate using NOR gate. A NOR gate can be used as a NOT gate by connecting all its inputs together.

Both the inputs of the NOR gate are connected.

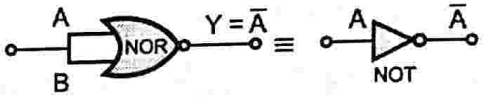
$\therefore$ Input $A = B = A$

$$\text{Output} = \overline{A + B}$$

$$Y = \overline{A + A}$$

$$Y = \bar{A} \quad (\because A + A = A \text{ Theorem 1(a)})$$

This is the expression for NOT gate. Fig. 3.17.1 shows NOT gate using NOR gate.



(1B40) **Fig. 3.17.1 : NOT gate using NOR gate**

3.17.2 AND Gate

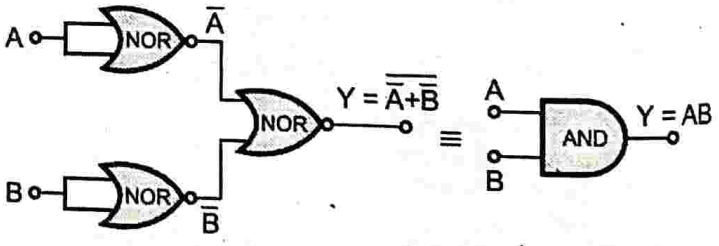
Q. 3.17.3 Construct AND gate using the universal gates.

☑ **Ans. :**

The Boolean expression for AND gate is,

$$Y = AB$$

Fig. 3.17.2 shows implementation of AND gate using NOR gates.



(1B41) **Fig. 3.17.2 : AND gate using NOR gate**

The Boolean expression for circuit shown in Fig. 3.17.2 is,

$$Y = \overline{\bar{A} + \bar{B}}$$

$$Y = \overline{\overline{AB}} \quad (\because \bar{A} + \bar{B} = \overline{AB} \text{ DeMorgan's Theorem})$$

$$Y = AB \quad (\because \overline{\bar{A}} = 1 \text{ Involution law Theorem 3})$$

is the equation of AND gate.

Truth tables

Truth table for AND gate

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

≡

Truth table for AND gate using NOR gates

A	B	$\bar{A}$	$\bar{B}$	$\bar{A}+\bar{B}$	$\overline{\bar{A}+\bar{B}}$
0	0	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	0	1

Thus, we can realize AND gate using NOR gates.

3.17.3 OR Gate

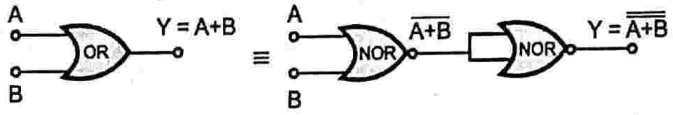
Q. 3.17.4 *Implement OR gates using NOR gate only.*

Ans. :

The Boolean expression for OR gate is,

$$Y = A+B$$

Fig. 3.17.3 shows the implementation of OR gate using NOR gates.



(1B42)**Fig. 3.17.3 : Implementation of OR gate using NOR gates**

The Boolean expression for circuit shown in Fig 3.17.3 is

$$Y = \overline{\overline{A+B}}$$

$$Y = A+B \qquad (\because \bar{\bar{A}} = A \text{ Involution law})$$

is the equation of OR gate.

Truth tables

Truth table for OR gate

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

≡

Truth table for OR gate using NOR gates

A	B	$\overline{A+B}$	$\overline{\overline{A+B}}$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

Thus, we can realize OR gate using NOR gates.

3.17.4 EX-OR Gate using NOR Gate

Q. 3.17.5 Implement EX-OR gates using NOR gates only.

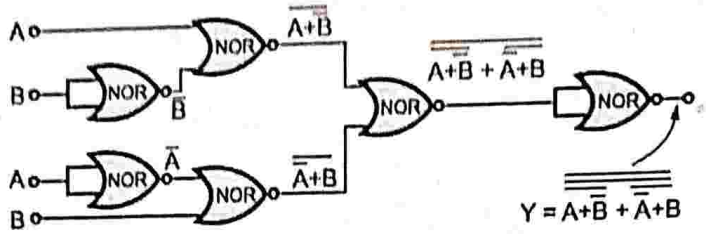
Ans. :

The Boolean expression for EX-OR gate is,

$$Y = A\bar{B} + \bar{A}B$$

Fig. 3.16.5(a) shows the implementation of EX-OR gate using basic gates.

Fig. 3.17.4 shows the implementation of EX-OR gate using NOR gates.



(1B43)**Fig. 3.17.4 : EX-OR realization using NOR gates**

The Boolean expression for logic diagram shown in Fig. 3.17.4 is,

$$Y = \overline{\overline{A+\bar{B}} + \overline{\bar{A}+B}}$$

$$Y = \overline{A+\bar{B}} + \overline{\bar{A}+B} \qquad (\because \bar{\bar{A}} = A \text{ Involution law})$$

$$Y = \bar{A}\cdot\bar{\bar{B}} + \bar{\bar{A}}\cdot\bar{B} \qquad (\because \overline{A+B} = \bar{A}\cdot\bar{B} \text{ DeMorgan's Theorem})$$

$$Y = \bar{A}\cdot B + A\cdot\bar{B} \qquad (\because \bar{\bar{A}} = A \text{ Involution law Theorem 3})$$

is the equation of EX-OR gate.

Truth tables

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

≡

A	B	$\bar{A}$	$\bar{B}$	$\overline{A+\bar{B}}$	$\overline{\bar{A}+B}$	$\overline{A+\bar{B}} + \overline{\bar{A}+B}$	$\overline{\overline{A+\bar{B}} + \overline{\bar{A}+B}}$
0	0	1	1	0	0	1	0
0	1	1	0	1	0	0	1
1	0	0	1	0	1	0	1
1	1	0	0	0	0	1	0

Thus, we can realize EX-OR gate using NOR gates.

3.17.5 EX-NOR Gate using NOR Gate

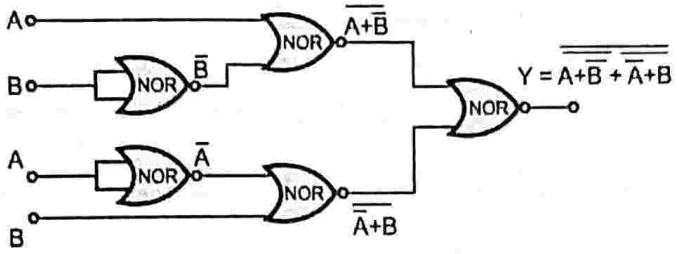
Q. 3.17.6 Implement EX-NOR gates using NOR gates only.

Ans. :

The Boolean expression for EX-NOR gate is,

$$Y = AB + \bar{A}\bar{B}$$

Fig. 3.16.6(a) shows the implementation of EX-NOR gate using basic gates. Fig. 3.17.5 shows the implementation of EX-NOR gate using NOR gates.



(1B44)**Fig. 3.17.5 : EX-OR gate using NOR gates**

The Boolean expression for logic diagram shown in Fig. 3.17.5 is

$$Y = \overline{\overline{A+\overline{B}} + \overline{\overline{A}+B}}$$

$$Y = \overline{A+\overline{B}} \cdot \overline{\overline{A}+B} \qquad (\because \overline{A+B} = \overline{A} \cdot \overline{B} \text{ DeMorgan's laws})$$

$$Y = (A+\overline{B}) \cdot (\overline{A}+B) \qquad (\because \overline{\overline{A}} = A \text{ Involution law Theorem 3})$$

$$Y = A\overline{A} + \overline{A} \cdot \overline{B} + AB + B \cdot \overline{B} = 0 + \overline{A} \cdot \overline{B} + AB + 0 \qquad (\because A \cdot \overline{A} = 0 \text{ postulate 5(b)})$$

$$Y = AB + \overline{A} \cdot \overline{B}$$

is the equation of EX-NOR gate.

Truth tables

Truth table for X-NOR gate

A	B	A ⊙ B
0	0	1
0	1	0
1	0	0
1	1	1

≡

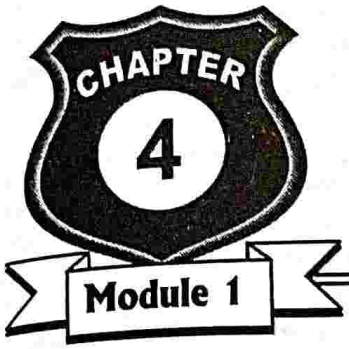
Truth table of X-NOR gate using NOR gates

A	B	$\overline{A}$	$\overline{B}$	$\overline{A+\overline{B}}$	$\overline{\overline{A}+B}$	$\overline{\overline{A+\overline{B}} + \overline{\overline{A}+B}}$
0	0	1	1	0	0	1
0	1	1	0	1	0	0
1	0	0	1	0	1	0
1	1	0	0	0	0	1

...Chapter Ends

❑❑❑

Note



Overview of Computer Organization and Architecture

University Prescribed Syllabus

Computer Fundamentals

1.5 Overview of computer organization and architecture.

1.6 Basic Organization of Computer and Block Level functional Units, Von-Neumann Model.

4.1 OVERVIEW OF COMPUTER ARCHITECTURE AND ORGANIZATION

4.1.1 Introduction

Q. 4.1.1 Define the terms computer organization and computer architecture.

The term computer architecture and computer organization is usually used interchangeably but there is some distinction between both the terms.

- **Computer Architecture :** This term is used from programmer perspective that is it describes logical execution of a program. Architecture of computer defines instruction format, registers, instruction and data memory. Usually computer architecture is also called as Instruction set architecture (ISA).
- **Computer Organization :** The computer organization describes operational units and their interconnections. For example control signals, interface between computer and peripheral and the memory technology.

UQ. 4.1.2 Differentiate between Computer Organization and Computer Architecture. MU - Q. 1(b), Dec. 15, Q. 1(e), May 16, Q. 1(c), Dec. 16, 5 Marks

Sr. No.	Computer Architecture	Computer Organization
1.	Computer Architecture deals with the way hardware components are connected together to form a computer system.	Computer Organization deals with the structure and behaviour of a computer system as seen by the user.
2.	It acts as the interaction between hardware and software.	It deals with the components level of a connection in a system.
3.	It provides the understanding of the functionalities at system level.	Computer Organization explains the exact arrangement of units in the system is and their interconnection.
4.	Architecture expresses in terms of instructions, addressing modes and registers.	Whereas Organization expresses the realization of architecture.
5.	While designing a computer system architecture is considered first.	An organization is done on the basis of architecture.

4.1.2 Basic Organization of Computer and Block Level Description of The Functional Units

- Complex machines like computers are described using hierarchical model. This approach is helpful for designing and their description. At each level there is some structure which shows the interconnections of components and functions of individual component.
- Let us understand each element of structure and function of computer.

Structure

The internal structure of computer is described with respect to number of processors. Let us first understand simple single processor structure and then proceeding with multicore structure.

Single core structure

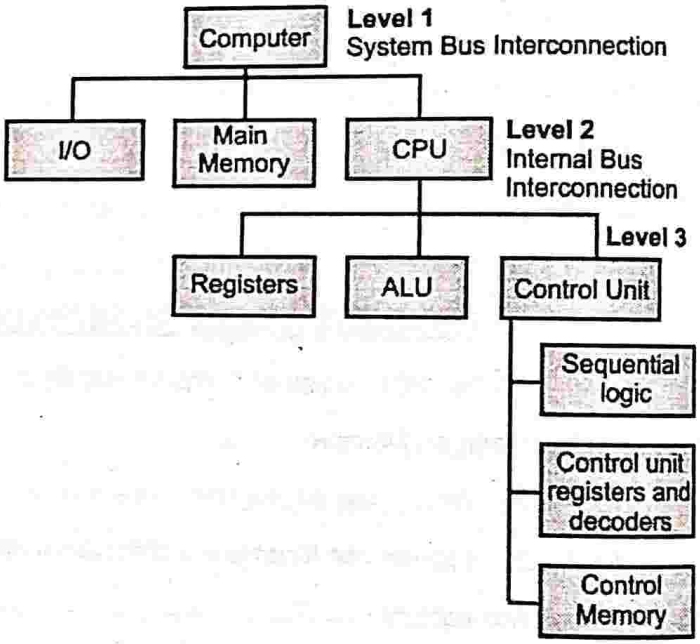


Fig. 4.1.1 : Single core structure

At level 1: Computer structure

There are main four structural components :

(i) **Central Processing Unit (CPU) :** It handles the data processing and control signals of the computer.

(ii) **Main Memory :** It is used to store program and data.

(iii) **Input/output (I/O) :** It is used to communicate between computer and external environment.

(iv) **System Interconnection :** All the above components i.e. CPU, memory and I/O are connected through conducting wires called as systems bus.

At level 2 : CPU structure

There are 4 main structural components :

(i) **Control Unit :** It controls all the components by sending out control signals.

(ii) **Arithmetic and logic unit :** It performs data processing functions.

(iii) **Registers :** These are temporary storage memory element inside the CPU.

(iv) **Internal bus :** The communication between control unit, ALU and registers happens through internal bus.

At Level 3 : Control Unit Structure

Generally control unit is implemented using microprogrammed approach. Control unit executes microinstructions which governs the functionality of the control unit.

Multicore Processor Structure

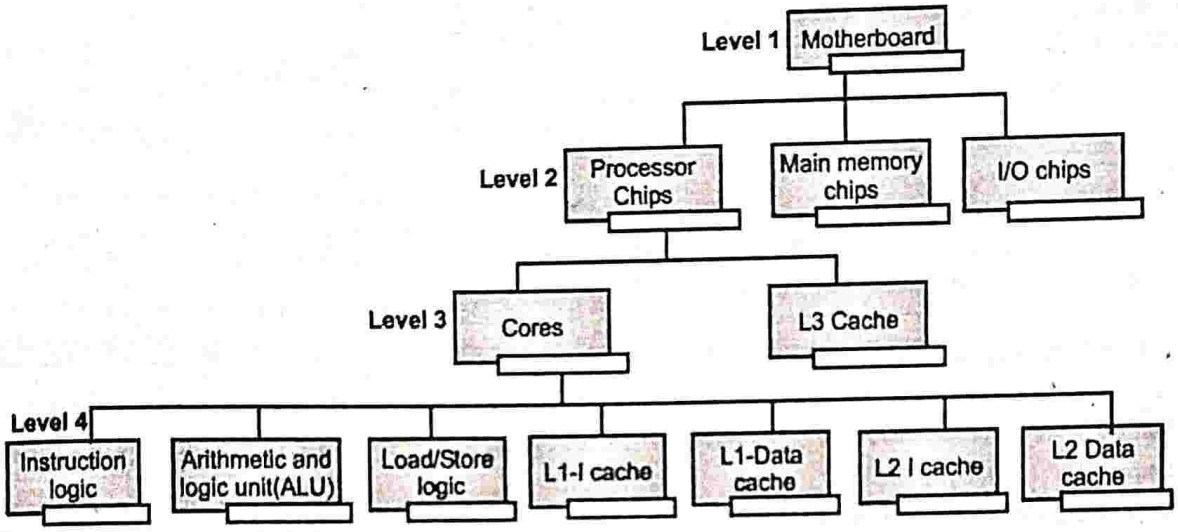


Fig. 4.1.2 : Multicore processor structure

When multiple processing units are implemented on single chip they are termed as multicore processing computer.

At Level-1 and 2 : Motherboard and its elements structure.

- **Motherboard** is a single flat rigid board which holds electronic components and their interconnections. It contains sockets and slots for memory chips, I/O controller chip, and processor chip.
- **Processor chip :** It consist of multiple cores (ALU, Control unit and registers) on a single chip.
- **Main Memory :** It is the main storage element.
- **I/O chip :** It is responsible to connect external peripherals to the computer.

At level 3 : Core Structure

- **Instruction logic :** This section in core is responsible to fetch instruction, decode instruction and calculate operand memory locations.
- **Arithmetic and logic unit :** This section operates on data according to the specified operation.
- **Load Store Instruction :** This section handles the transfer of data from and to main memory through cache
- **Cache Memory :** Usually small and fast memory is built in side core for better throughput. Cache memory is divided as instruction cache (I-cache) and data cache (Data-Cache). There are different levels of cache memories generally called as Level-1(L1), Level-2(L2) and Level-3(L3).
- **Functions :** There are majorly 4 basic functions that a computer performs :

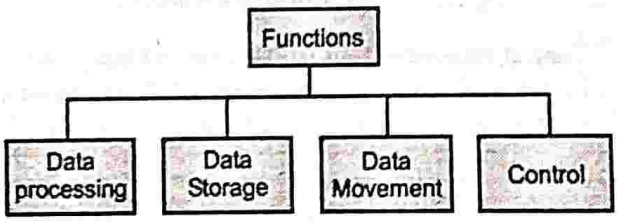


Fig. 4.1.3 : Functions

4.2 VON NEUMANN ARCHITECTURE

The Stored Program Concept

UQ. 4.2.1 What is stored program concept in digital computer ?

MU - Q. 1(b), May 15, 3 Marks

- The Stored Program Concept is the idea that the instructions i.e. program and data be stored together in a shared memory in a binary form in order to perform a variety of tasks in sequence or intermittently.
- The program is then fetched sequentially one instruction at a time from the memory and then executed using the Fetch-Decode-Execute cycle.
- The advantage of storing a program be electronically is that instructions could be modified by the computer as determined by intermediate computational results.
- An accumulator is therefore used in this concept for short term intermediate storage of arithmetic and logic data in a computer's CPU.
- The stored program concept gave rise to the Von Neumann Architecture.

UQ. 4.2.2 Explain Von Neumann architecture in detail. MU - Q. 1(a), Dec. 16, 5 Marks

- This architecture is named after the mathematician John Von Neumann. It works on the concept of the stored program. It was first implemented in the year 1940 at Princeton.
- This architecture relies on three key concepts:
  - Unified memory for data and program storage,
  - Sequential execution of program,
  - Data is stored in memory with unique address location.
- Fig. 4.2.1 shows the conceptual and basic structure of Von Neumann architecture

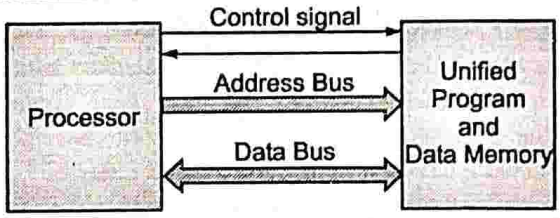


Fig. 4.2.1 : Von Neumann Architecture

- **Central Processing Unit (CPU) :** This element is the heart of the Von Neumann architecture, which has major two components viz. Control Unit (CU) and Arithmetic and logic unit (ALU).

4.3 HARVARD ARCHITECTURE

- The name of the architecture is based on world's largest electromagnetic relay based calculator Harvard Mark I.
- This architecture specifies two separate memory for instruction (program) and data.
- For better efficiency and higher bandwidth the resources such as buses and control signals are also separated. Due to separate fetch cycles this architecture is more suitable for pipelining and faster execution.

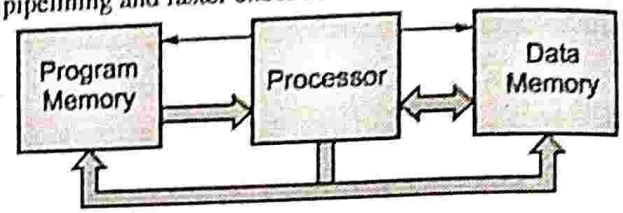
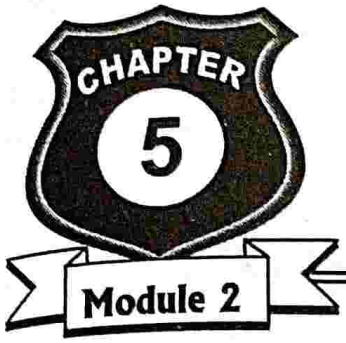


Fig. 4.3.1 : Harvard architecture

UQ. 4.3.1 Compare Von Neumann architecture and Harvard Architecture. MU - Q. 1(a), May 18, 5 Marks

Von Neumann Architecture	Harvard Architecture
In this architecture only one memory is used to store both data and program.	In this architecture two separate memories are used for storing data and program.
The data and code are fetched sequentially hence the execution is slower.	As the data and code can be fetched simultaneous the execution of program is faster.
This type of architecture is preferred in CISC machines.	This type of architecture is preferred in RISC machines.
It supports lower degree of parallelism.	It supports higher degree of parallelism.

...Chapter Ends



Data Representation and Arithmetic Algorithms

University Prescribed Syllabus

Data Representation and Arithmetic Algorithms

2.1 Binary Arithmetic : Addition, Subtraction, Multiplication, Division using Sign Magnitude, 1's and 2's compliment, BCD and Hex Arithmetic Operation.

2.2 Booths Multiplication Algorithm, Restoring and Non-restoring Division Algorithm.

2.3 IEEE-754 Floating point Representation.

5.1 DATA REPRESENTATION AND ARITHMETIC ALGORITHM

- This unit performs actual arithmetic and logical operation on data operands.
- It is an electronic component which is based on simple digital logic devices which are capable of storing binary digits and perform simple Boolean logic operations.

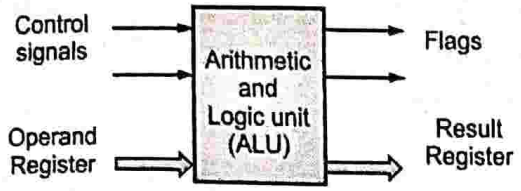


Fig. 5.1.1 : ALU

5.1.1 Integer Data Computation : Addition and Subtraction of Signed Numbers

- Let us consider single stage addition of an n bit number X and Y.
- The i^{th} stage bits are x_i, y_i, c_i, s_i, c_{i+1} the truth table is given as

x_i	y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- Therefore according to above output the logic for sum and carry is the circuit implementation is

$S_i = \bar{x}_i \bar{c}_i yi + \bar{c}_i \bar{y}_i\, x_i + x_i\, y_i\, c_i$ and $x_i + x_i\, c_i + x_i\, y_i$

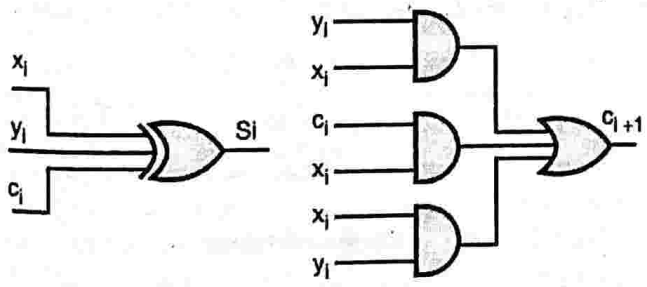


Fig. 5.1.2 : Circuit

The combined circuit is called as full adder

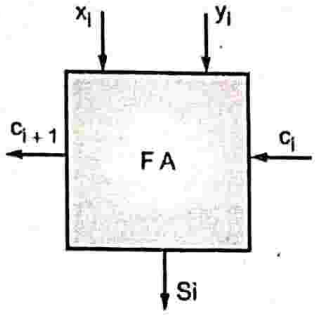


Fig. 5.1.3 : Logic

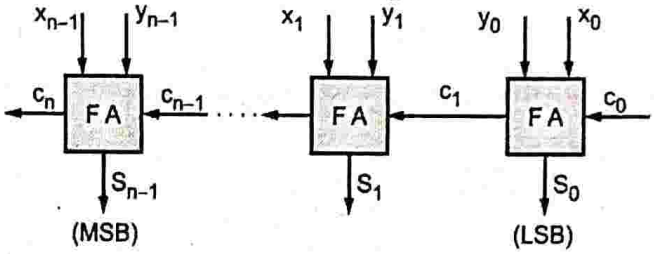


Fig. 5.1.4 : Cascade

- For n bit addition n stage cascade full adder is used
- In the Fig. 5.1.4 the carry of previous stage is forwarded to the next stage hence this configuration is called as n-bit ripple carry adder.
- The above circuit can also be used to add 2's complement numbers X and Y where x_{n-1} and y_{n-1} ae the sign bits.
- To perform X-Y, 2's complement of Y is obtained and added to X.
- The complete adder/subtraction logic circuit is shown in Fig. 5.1.5.
- The circuit has a add/sub input control line. The control signal is 0 the operation is addition and if the control signal is 1 the Y is 1's complemented by XOR gates and set c_0 is set to 1 to complete the 2's complement of Y.

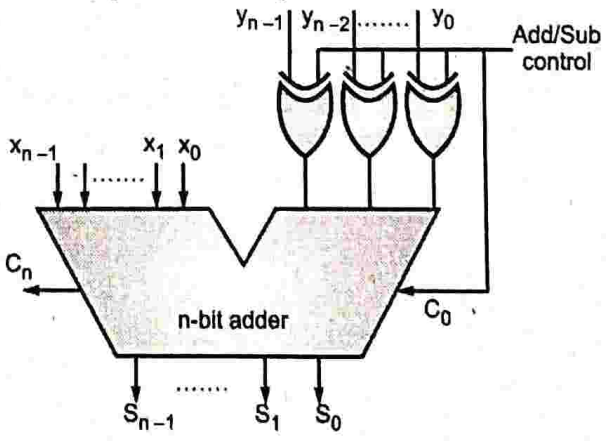


Fig. 5.1.5 : Add/Sub logic

5.1.1(A) Fast adder

- In the ripple carry adder the operation is completed when the carry bit from LSB is propagated to MSB position. It is the longest path of signal propagation.
- The time taken by the signal to propagate depends upon the delay produced by logic gates therefore the carry signal c_{n-1} is produced in 2 (n-1) gate delays whereas s_{n-1} signal is produced after 1 XOR gate delay.
- Hence all sum bits are obtained after 2n gate delays, including the delay through the XOR gates on the Y input. Therefore there is a need of faster adder which will propagate the carry much faster and generate the correct result faster.
- One such approach is carry look ahead addition.

5.1.1(B) Carry Look Ahead Addition

- For fast operation of adder circuit the carry signal must be generated which in turn generators the sum output.
- Looking at the expression.
  sum (S_i) and carry-out (C_i) are

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

$$\therefore c_{i+1} = G_i + P_i C_i$$

$$y_i = x_i y_i \quad \text{and} \quad P_i = x_i + y_i$$

  Where y_i is generate and P_i is propagate
- From above expression it can be seen that the carry c_{i+1} is 1 if y_i is 1. And y_i bit is 1 if x_i and y_i are 1. Therefore all generate and propagate functions are calculated parallel in one logic gate delay. Fig. 5.1.6 shows the implementation of look ahead addition.

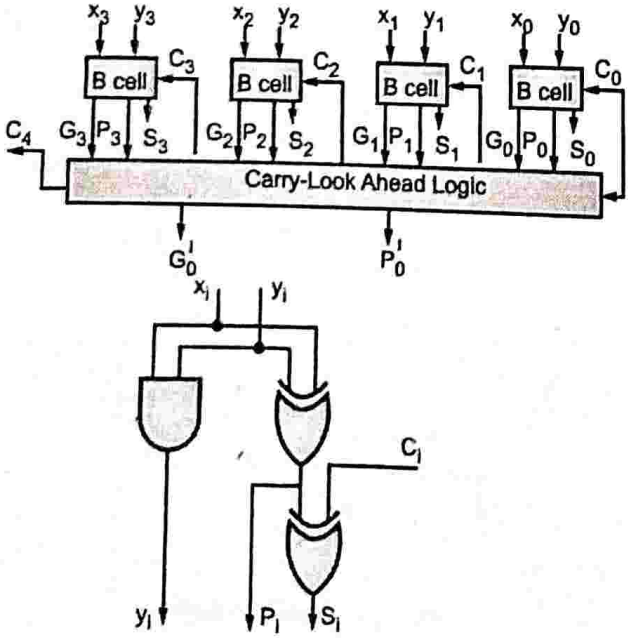


Fig. 5.1.6

- In above implementation the carries are generating three gate delays after the input signal X, Y and C_0 are applied.

5.1.2 Multiplication : Unsigned Multiplication

- Let us consider two n bit non negative (positive) numbers X and Y to be multiplied. The two n bit digits can produce a maximum of 2n bit result.
- The above algorithm is explained using example of 4 bit number as X = 1101 and Y = 1100

				X =	1	1	0	1
				Y =	1	1	0	0
			Partial Product-1		0	0	0	0
		Partial Product-2		0	0	0	0	X
	Partial Product-3		1	1	0	1	X	X
Partial Product-4		1	1	0	1	X	X	X
1	0	0	1	1	1	0	0	

Multiplication in binary is simple as it has basic three steps :

- **Step 1 :** If the multiplier is 0 replace the all multiplicand digits by 0 and if multiplier is 1 then write the multiplicand digits.
- **Step 2 :** For each multiplier bit position shift the multiplicand digits appropriately for e.g. for 3rd row in the partial product (partial product-3) the multiplicand number is shifted two bit position.
- **Step 3 :** Finally each all the partial products are added to generate the final product.

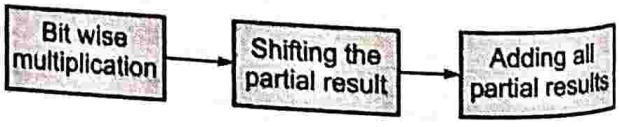
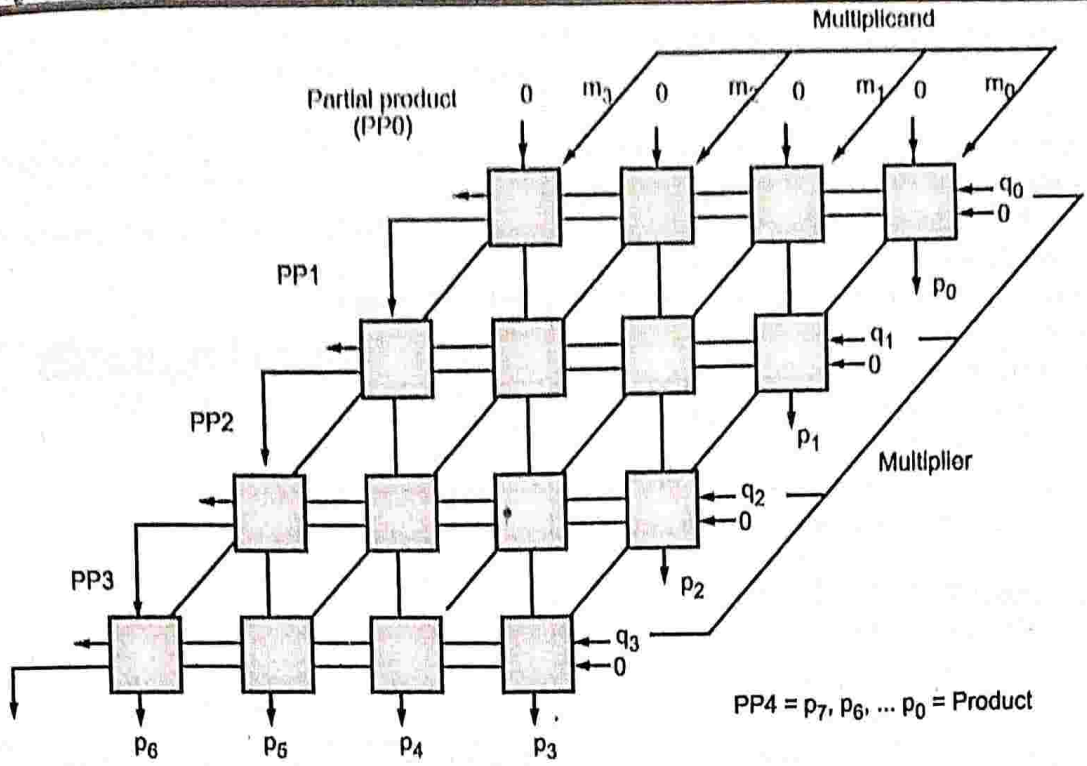


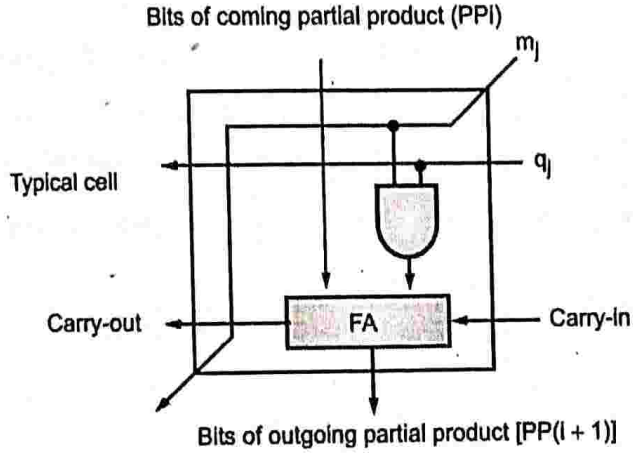
Fig. 5.1.7 shows the actual implementation of multiplication.



(a)

Fig. 5.1.7

Bit of incoming product (PPI)



(b) Array implementation

Fig. 5.1.7 : Array multiplier

- The basic components are full adder and an AND gate which determines whether a bit is to be forwarded or a 0 is to be forwarded.
- Each rows partial product is shifted to next level for addition if the multiplier is a 0 then partial product is vertically shifted downward by shifting 1 bit position at each level.
- Inside the processor the adder circuit is used for multiplication for a number of sequential steps.

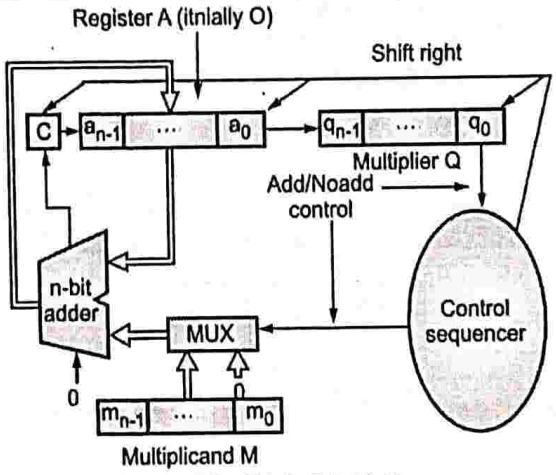


Fig. 5.1.8 : Multiplier

5.1.2(A) Booth's Algorithm

UQ. 5.1.1 *Explain Booths algorithm with an example.*

MU – Q. 2(a), May 18, Q. 1(b), Dec. 18, 5 Marks

UQ. 5.1.2 *Draw the flow chart for Booth's algorithm for two's complement multiplication.*

MU - Q. 2(a), May 14, Q. 1(d), Dec. 14, Q. 1(d), May 15, Q. 2(a)(i), Dec. 15, Q. 2(a)(i), Dec. 17, 4 Marks

- Booth's algorithm is widely used for multiplication and

division as it provides faster result compared to normal approach.

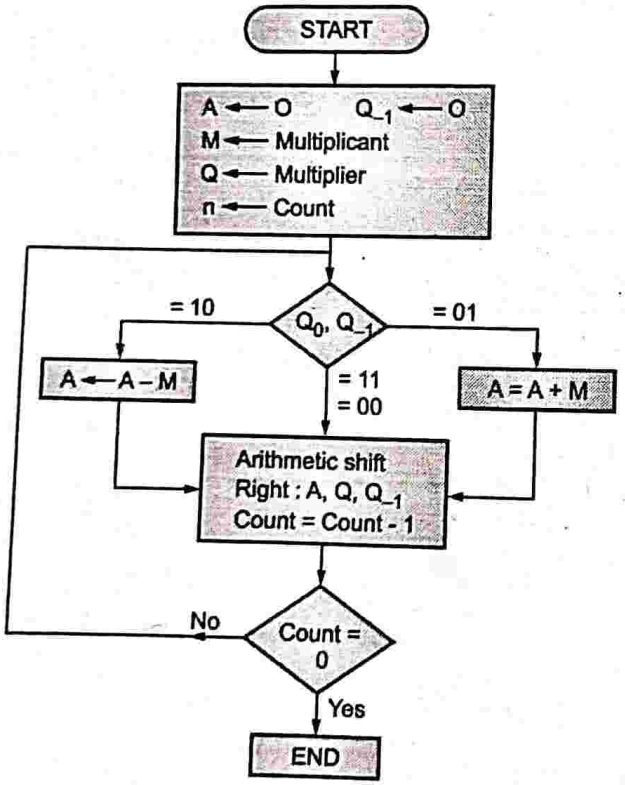
- Fig. 5.1.9 depicts the flow chart of the algorithm.

▶ **Step 1 :** Initialize the registers i.e. place the multiplier and multiplicand in the Q and M register respectively. Also A and Q_1 is initialized to zero.

▶ **Step 2 :** The control logic scans the bit of the multiplier. Each bit of the multiplier is examined with bit is its right.

Step 2.1 : If both bits are same i.e. [0 – 0 and 1 – 1] then all the bits are shifted to right of register A, Q and Q_1 [Arithmetic shift].

Step 2.2 : If the two bits are of the form. [0 – 1 or 1 – 0], then the multiplicant is subtracted from register A. After addition (or added). Or subtraction the bits of register A, Q and Q_1 are shifted right by 1 bit.



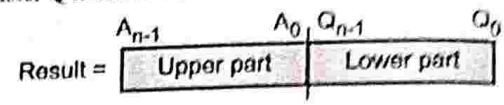
(1A1)**Fig. 5.1.9**

Note : The shifting process is such that the leftmost bit of A is retained and also shifted to next position. Such kind of shifting is known as Arithmetic shifting.

▶ **Step 3 :** Decrement count value and check if count equals zero.

Step 3.1 : If count ≠ 0. Repeat step 2.

If count = 0. Register A holds upper n bits including MSB and Register Q holds lower n bits of the result.



(1A2)**Fig. 5.1.10 : Result of multiplication**

Solved Examples

UEx. 5.1.1 **MU - Q. 3(a), May 17, 10 Marks**

A Multiply (– 5) and (2) using Booth's Algorithm.

☑ **Soln. :**

Multiplicant = $(-5)_{10} = (1011)_2$

Multiplier = $(2)_{10} = (0010)_2$

Initialize

$A = (0000)_2$ $Q = (0010)_2$

$q_{-1} = 0$ $n = 04$

Table Ex. 5.1.1

n	A	Q	q_{-1}	Action
4	0000	0010	0	Initialization
3	0000	0001	0	Arithmetic shift register
	0101	0001	0	A = A – M
2	0010	1000	1	Arithmetic shift register
	1101	1000	1	A = A + M
1	1110	1100	0	Arithmetic shift register
0	1111	0110	0	Arithmetic shift register
		Result		

As MSB = 1 the number is in 2's complement form

11110110

↓1's

00001001

↓2's

$(00001010)_2 \rightarrow (-10)_{10}$

UEx. 5.1.2 **MU - Q. 2(a)(ii), Dec. 17, 6 Marks**

Using Booth's algorithm multiply 14 times – 5.

☑ **Soln. :**

Multiplicant (M) = $(14)_{10} = (01110)_2$

Multiplier (Q) = $(-5)_{10} = (11011)_2$

Initialization

$A = (00000)_2$ $Q = (11011)_2$

$q_{-1} = 0$ $n = 05$

Table Ex. 5.1.2

n	A	Q	q_{-1}	Action
5	00000	11011	0	Initialization
	10010	11011	0	A = A – M
4	11001	01101	1	Arithmetic shift right
3	11100	10110	1	Arithmetic shift right
	01010	10110	1	A = A + M
2	00101	01011	0	Arithmetic shift right
	10111	01011	1	A = A – M
1	11011	10101	1	Arithmetic shift right
0	11101	11010	1	Arithmetic shift right
		Result		

As MSB = 1 the number is in 2's complement form

1110111010

↓1's

0001000101

↓2's

$(0001000110)_2 \rightarrow (-70)_{10}$

UEx. 5.1.3 MU - Q. 2(a), May 18, 5 Marks

Perform following multiplication using Booth's algorithm $M = (-9)_{10}$, $Q = (6)_{10}$.

Soln. :

Multiplicant = $(-9)_{10} = (10111)_2$

Multiplier = $(6)_{10} = (00110)_2$

Initialization

$A = (00000)_2$, $Q = (00110)_2$,

$q_{-1} = 0$, $n = 05$

Table Ex. 5.1.3

n	A	Q	q_{-1}	Action
5	00000	00110	0	Initialization
4	00000	00011	0	Arithmetic shift right
	01001	00011	0	A = A – M
3	00100	10001	1	Arithmetic shift right
2	00010	01000	1	Arithmetic shift right
	11001	01000	1	A = A + M
1	11100	10100	0	Arithmetic shift right
0	11110	01010	0	Arithmetic shift right
		Result		

As MSB = 1 the number is in 2's complement form

1111001010

↓1's

$0000110101 \xrightarrow{2's} (0000110110)_2 \rightarrow (-54)_{10}$

UEx. 5.1.4 MU - Q. 5(b), May 16, 10 Marks

Multiply (–10) and (– 4) using Booth's algorithm.

Soln. :

Multiplicant = $(M) = (-10)_{10} = (10110)_2$

Multiplier = $(Q) = (-4)_{10} = (11100)_2$

Initialization

$A = (00000)_2$, $Q = (11100)_2$,

$q_{-1} = 0$, $n = 05$

Table Ex. 5.1.4

n	A	Q	q_{-1}	Action
5	00000	11100	0	Initialization
4	00000	01110	0	Arithmetic shift right
3	00000	00111	0	Arithmetic shift right
	01010	00111	0	A = A – M
2	00101	00011	1	Arithmetic shift right
1	00010	10001	1	Arithmetic shift right
0	00001	01000	1	Arithmetic shift right
		Result		

As MSB = 0 the number is positive

$\therefore (0000101000)_2 \longrightarrow (40)_{10}$

UEx. 5.1.5 MU - Q. 5(b), Dec. 16, 10 Marks

Multiply (– 7) with (4) by using Booth's algorithm of Multiplication.

Soln. :

Multiplicant = $(M) = (-7)_{10} = (1001)_2$

Multiplier = $(Q) = (4)_{10} = (0100)_2$

Initialization

$A = (0000)_2$ $Q = (0100)_2$

$q_{-1} = 0$ $n = 04$

Module 2

Table Ex. 5.1.5

n	A	Q	q_{-1}	Action
4	0000	0100	0	Initialization
3	0000	0010	0	Arithmetic shift right
2	0000	0001	0	Arithmetic shift right
	0111	0001	0	A = A – M
1	0011	1000	1	Arithmetic shift right
	1001	1000	1	A = A + M
	1100	1000		
0	1110	0100		

As MSB = 1 the number is in 2's complement form

11100100

↓1's

00011011

↓2's

$(00011110)_2 \rightarrow (-28)_{10}$

The Answer is → –28

UEx. 5.1.6 MU - Q. 2(a)(ii). Dec. 15, 6 Marks

Using Booth's algorithm show the multiplication of – 3 * – 7.

☑ **Soln. :**

Multiplicant = (M) = $(-3)_{10} = (1101)_2$

Multiplier = (Q) = $(-7)_{10} = (1001)_2$

Initialization

$A = (0000)_2$ $Q = (1001)_2$

$q_{-1} = 0$ $n = 04$

Table Ex. 5.1.6

n	A	Q	q_{-1}	Action
4	0000	1001	0	Initialization
	0011	1001	0	A = A – M
3	0001	1100	1	Arithmetic shift right
	1110	1100	1	A = A + M
2	1111	0110	0	Arithmetic shift right
1	1111	1011	0	Arithmetic shift right
0	0010	1011	0	A = A – M
	0001	0101	1	Arithmetic shift right
		Result		

UEx. 5.1.7 MU - Q. 2(b). Dec. 14, 10 Marks

Multiply $(-2)_{10}$ and $(-5)_{10}$ using Booth's algorithm.

☑ **Soln. :**

Multiplicant = (M) = $(-2)_{10} = (1110)_2$

Multiplier = (Q) = $(-5)_{10} = (1011)_2$

Initialization

$A = (0000)_2$ $Q = (1011)_2$

$q_{-1} = 0$ $n = 04$

Table Ex. 5.1.7

n	A	Q	q_{-1}	Action
4	0000	1011	0	Initialization
	0010	1011	0	A = A – M
	0010	1011	0	
3	0001	0101	1	Arithmetic shift right
2	0000	1010	1	Arithmetic shift right
	1110	1010	1	A = A + M
	1110	1010	1	
1	1111	0101	0	Arithmetic shift right
	0010	0101	0	A = A – M
	0001	0101	0	
0	0000	1010	1	Arithmetic shift right

As MSB = 0 the number is positive

∴ $(00001010)_2 \longrightarrow (10)_{10}$

The Answer is → +10

UEx. 5.1.8 MU - Q. 2(b). May 14, 7 Marks

Using Booth's algorithm show the multiplication of 7 × 5.

☑ **Soln. :**

Multiplicant = (M) = $(7)_{10} = (0111)_2$

Multiplier = (Q) = $(5)_{10} = (0101)_2$

Initialization

$A = (0000)_2$ $Q = (0101)_2$

$q_{-1} = 0$ $n = 04$

Table Ex. 5.1.8

n	A	Q	q_{-1}	Action
4	0000	0101	0	Initialization
	1001	0101	0	A = A – M
3	1100	1010	1	Arithmetic shift right
	0011	1010	1	A = A + M
2	0001	1101	0	Arithmetic shift right
	1010	1101	0	A = A – M
1	1101	0110	1	Arithmetic shift right
	0100	0110	1	A = A + M
0	0010	0011	0	Arithmetic shift right
		Result		

As MSB = 0 the number is positive

$\therefore (00100011)_2 \longrightarrow (35)_{10}$

5.1.3 Integer Division

5.1.3(A) Restoring Division

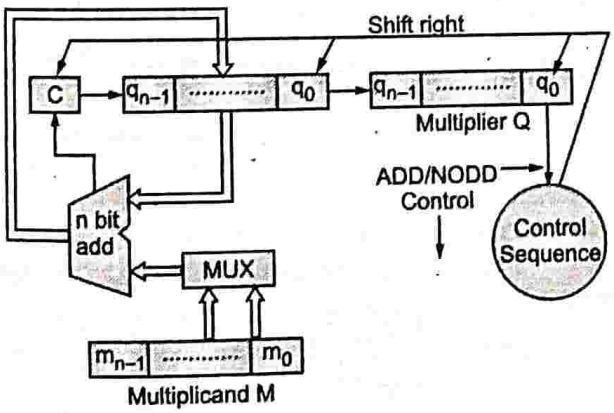
UQ. 5.1.3 Explain the restoring method of binary division with algorithm.

MU - Q. 2(b), May 18, 5 Marks

UQ. 5.1.4 Draw the flow chart for restore Division Algorithm.

MU - Q. 2(a)(i), May 17 , 5 Marks

- Fig. 5.1.11 shows the hardware implementation of restoring division and the flowchart of the process.



(1A23) **Fig. 5.1.11(a) : Hardware implementation**

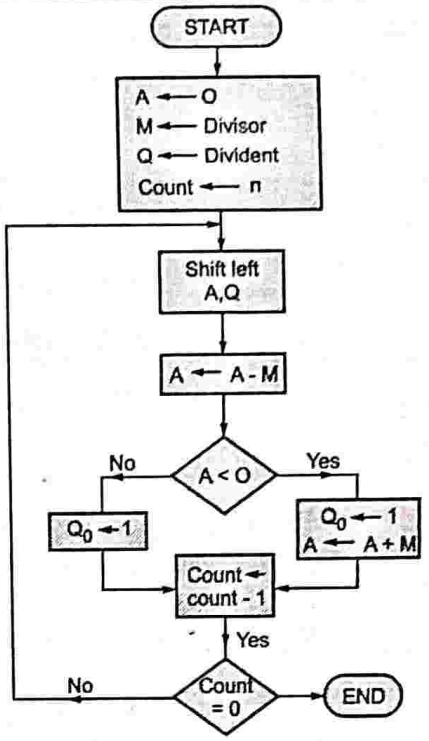


Fig. 5.1.11(b) : Flowchart of division (restoring)

- The procedure starts by loading the dividend and divisor values in register Q and M respectively.
- Register A is initialized to '0'. The result of division is placed n-bit Q register (Quotient) and n-bit A register (Reminder).
- Detailed stepwise procedure is as follows :

▶ **Step 1 :** Load register A with 0's, Q with dividend and M with 2's complement of divisor.

▶ **Step 2 :** Shift A, Q left by 1 bit position.

▶ **Step 3 :** Register A is subtracted from M (Multiplicant) and the result is stored in A.

▶ **Step 4 :** 1. If MSB of A register is '0', then set $Q_0 = 1$.
2. If MSB of A register is '1' then $Q_0 = 0$ and restore the value of previous A.

▶ **Step 5 :** Repeat the process as many times as there are bit positions in Q.

▶ **Step 6 :** The result of division is stored as
A = Remainder
Q = Quotient

UEx. 5.1.9

MU - Q. 2(b), May 15, Q. 2(a)(ii), May 17 , 6 Marks

Using unsigned binary division method, divide 7 by 3

OR Divide using restore division method 7/3.

☑ **Soln. :**

Divident = Q = $(7)_{10} = (0111)_2$

Divisor = M = $(3)_{10} = (0011)_2$

Module 2

▶ **Step 1 :** Obtain 2's complement of M

$0011 \xrightarrow{1's} 1100 \xrightarrow{2's} 1101$

▶ **Step 2 :** Table

n	A	Q	Action
4	0000	0111	Initialization
	0000	1110	Shift A and Q to left
	0000 1101 1101	1110	A ← A – M
	0000	1110	Set $Q_0 = 0$ Restore A
3	0001	1100	Shift A and Q to left
	0001 1101 1110	1100	A ← A – M
	0001	1100	Set $Q_0 = 0$ Restore A
2.	0011 0011 1101 0000	1000 1000	Shift A and Q to left A ← A – M
	0000	1001	Set $Q_0 = 1$
1	0001 0001 1101 1110	0010 0010	Shift A and Q to left A ← A – M
	0001	0010	Set $Q_0 = 0$ Restore A

Remainder = A = $(0001)_2 = (1)_{10}$

Quotient = Q = $(0010)_2 = (2)_{10}$

UEx. 5.1.10 MU - Q. 2(b), May 18, 5 Marks

Divide $(7)_{10}$ by $(4)_{10}$ using restoring method of binary division.

☑ **Soln. :**

Divident = Q = $(7)_{10} = (0111)_2$

Divisor = M = $(4)_{10} = (0100)_2$

▶ **Step 1 :** Obtain 2's complement of M

$(0100)_2 \xrightarrow{1's} (1011)_2 \xrightarrow{2's} 1100$

▶ **Step 2 :** Table

n	A	Q	Action
4	0000	0111	Initialization
	0000	1110	Shift A and Q to left
	0000 1100 1100	1110	A ← A – M
	0000	1110	Set $Q_0 = 0$ Restore A
3	0001	1100	Shift A and Q to left
	0001 1100 1101	1100	A ← A – M
	0001	1100	Set $Q_0 = 0$ Restore A
2.	0011	1000	Shift A and Q to left
	0011 1100 1111	1000	A ← A – M
	0011	1000	Set $Q_0 = 0$ Restore A
1	0111	0000	Shift A and Q left
	0111 1100 0011	0000	
	0011	0001	Set $Q_0 = 1$

Remainder = A = $(0011)_2 = (3)_{10}$

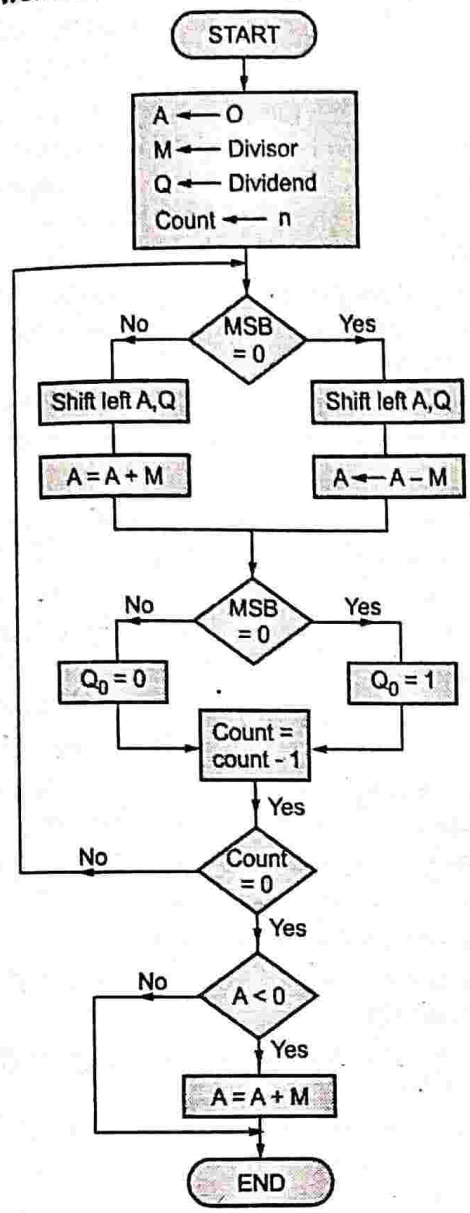
Quotient = Q = $(0001)_2 = (1)_{10}$

5.1.3(B) Non- Restoring Division

GQ. 5.1.5 Draw flow chart for non-restoring division.

In non restoring algorithm the need to restore A after unsuccessful subtraction is eliminated which provide improvement in calculation.

Flowchart of non-restoring division



(1A11)**Fig. 5.1.12**

Following steps are performed in non-restoring algorithm.

▶ **Step 1 :** 1. If MSB of A is 0, shift A and Q register 1 bit left and subtract M from A, otherwise shift A and Q left and add M to A.

2. If MSB of A is 0, set Q_0 to 1 otherwise, set Q_0 to 0.

Repeat step 1 for n times.

▶ **Step 2 :** After n times if MSB of A is 1 then add M to A for proper positive remainder in A.

E.g. : 1. Initialize the register contents

Divided = 1000 (Register Q)

Divisor = 00011 (Register M)

Partial = 00000 (Register M)

Result

2. MSB of register A = 0, Hence shift left and. Subtract A from M.

Action	Register A	Register Q	
Initially	0 0 0 0 0	1 0 0 0	
Shift left	0 0 0 0 1	0 0 0 □	First cycle
Subtract	1 1 1 0 1		
$Q_0 = 0$	①1 1 0 1	0 0 0 0	
Shift left	1 1 1 0 0	0 0 0 □	
Add	0 0 0 1 1		Second cycle
$Q_0 = 0$	①1 1 1 1	0 0 0 0	
Shift left	1 1 1 1 0	0 0 0 □	
Add	0 0 0 1 1		Third cycle
$Q_0 = 1$	⓪0 0 0 1	0 0 0 0	
Shift left	0 0 0 1 0	0 0 1 □	
Subtract	1 1 1 0 1		Fourth cycle
$Q_0 = 1$	①1 1 1 1	0 0 1 0	
Shift left	1 1 1 1 1		Restore Remainder
Subtract	0 0 0 1 1		
$Q_0 = 1$	0 0 0 1 0		

(1A23)**Fig. 5.1.13**

5.1.4 Floating Point Arithmetic

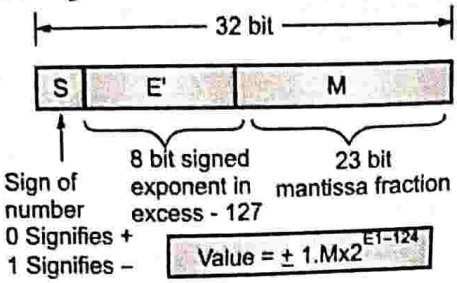
1. To represent large range of integers and small fraction a system must use numbers with variable position binary point numbers. Such numbers where the is called floating point numbers.
2. For hardware implementation simplicity the point position must be fixed e.g. the point is placed to the right of the first non-zero significant digits.
3. Such a number is called as normalized number

e.g. 6.0247×10^{23}, 6.6254×10^{-27}, -1.0341×10^{2}, -7.3000×10^{-14}.

4. From above e.g. we see that represent at has sign, the significant digit and the exponent in the scale factor.
5. Hence to represent a floating point number there elements must be used.
   (a) Mantissa [significant digits]
   (b) Exponent [with scale factor]
   (c) Sign [sign of significant digits]
6. A 32 bit representation of floating point number has been developed by IEEE. The choice of 32 bit is decided based on the standard computer word length.

Module 2

7. The 32 bits are divided as
   (a) 24 bit mantissa [provides 7 digit decimal number precision].
   (b) 8 bit – exponent [to base of 2]
   (c) 1 bit sign

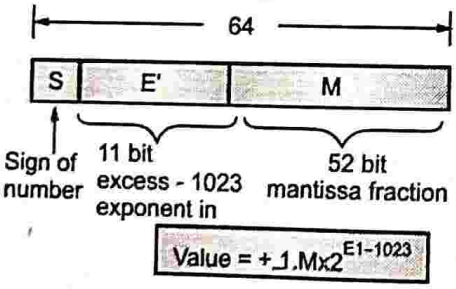


(1A17)Fig. 5.1.14

The above representation of number is called as single precision representation.

8. With the above scale factor the range of representation is approximately equal to 10 ± 38 [2^{-128} to 2^{+127}]. And 24 bit mantissa provides same precision as of 7 decimal digits.
9. For higher precision and range for floating point numbers. IEEE has increased the number of bits from 32 to 64 which is called as double precision number.
10. In double precision number the mantissa is represented using 53 bits, exponent of 11 bits and 1 sign bit.
11. With above bits the total range of floating
    (a) Scale factor range = $10^{\pm 308}$ [2^{-1022} to 2^{1023}]
    (b) Mantissa = 16 decimal digit precision.

Value = $\pm 1.M \times 2^{E-1023}$



(1A18)Fig. 5.1.15

5.1.4(A) IEEE-754 Standard

UQ. 5.1.6 Explain IEEE 754 floating point representation formats.
MU - Q. 1(b), May 18, 5 Marks

UQ. 5.1.7 Show IEEE-754 standards for Binary Floating Point Representation for 32 bit single format and 64 bit double format.
MU - Q. 1(b), May 14, 3 Marks, Q. 1(e), Dec. 15, 5 Marks, Q. 5(b) May. 17, 10 Marks

UQ. 5.1.8 Explain IEEE 754 standards for floating point number representation.
MU - Q.2(c), May 15, 6 Marks

- The IEEE-754 was published on 12th October 1985 and superseded in 1987, 2008 and 2019. This standard is the product of 754_WG-Working group for floating point arithmetic and sponsored by C/MSC-Microprocessor standard committee.
- The standard specifies interchange and arithmetic formats and methods for binary and decimal floating point arithmetic in a computer programming environment.
- The standard can be implemented entirely in software or in hardware or in any combination of software and hardware.
- This standard specifies :
  - Formats for binary and decimal floating-point data, for computation and data interchange.
  - Addition, subtraction, multiplication, division, fused multiply-add, square root, compare, and other operations.
  - Conversions between integer and floating-point formats.
  - Conversions between different floating-point formats.
  - Conversions between floating-point formats and external representations as character sequences.
  - Floating-point exceptions and their handling, including data that are not numbers (NaNs).
- The standard specifies five basic formats of
  - Three binary formats, with encodings in lengths of 32, 64, and 128 bits.
  - Two decimal formats, with encodings in lengths of 64 and 128 bits.
- Each number in both formats is represented by

$$n = [(-1)^s] \times [b^e] \times m$$

where

s is the sign bit which can take either 1 or 0

b is the base for binary b=2 and for decimal b=10

e is the exponent

m is the mantissa

- The format supports two infinities ($-\infty$, $+\infty$) and two NaN qNaN (quiet) and sNaN (signalling)
- The most commonly used precision in the format are:
  - Single precision (32 bit)
  - Double precision (64 bit)
- **Single Precision**
  - In single precision format, using binary base, the least significant 23 bits from bit D0 to D22 are used to store the fractional part i.e. Mantissa.
  - Next bits from D23 to D30 are used to store the 8-bit exponent.

- Bit 31 is used to store the sign bit.
- E.g.: the decimal number 0.15625_{10} represented in binary is 0.00101_2. In the single precision format, it will be represented as,

$$0.00101_2 = 1.01_2 \times 2^{-3}.$$

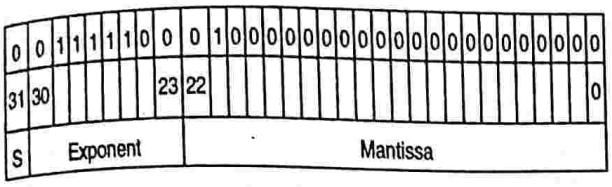


Fig. 5.1.16

- As illustrated in the Fig. 5.1.16, the three fields in the IEEE 754 representation of this number are:

sign = 0 (number is positive.)

biased exponent = exponent value + "bias"
(127 → single precision)
= −3 + 127
= 124
= 01111100
= in double precision, the bias is 1023, 1020.

fraction = $.01000\ldots_2$

- **Double Precision**
  - In double precision format, using binary base, the least significant 52 bits from bit D0 to D51 are used to store the fractional part i.e. Mantissa.
  - Next bits from D52 to D62 are used to store the 11-bit exponent.
  - Bit 63 is used to store the sign bit.
  - E.g.: the decimal number 0.15625_{10} represented in binary is 0.00101_2. In the single precision format, it will be represented as,

$$0.00101_2 = 1.01_2 \times 2^{-3}$$

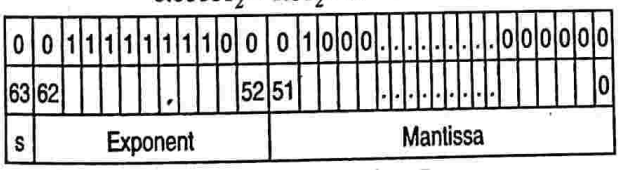


Fig. 5.1.17

- As illustrated in the Fig. 5.1.17 , the three fields in the IEEE 754 representation of this number are:

sign = 0 (number is positive.)

biased exponent = exponent value + "bias"
(1023 –> Double precision)
= −3 + 1023
= 1020
= 01111111100
= in double precision, the bias is 1023, 1020.

fraction = $.01000\ldots_2$

UQ. 5.1.9 In floating point representation how to identify sign of exponent?
MU- Q. 1(c), Dec. 17, 5 Marks

- In floating point number representation, Exponents are biased and therefore represented exponents are always unsigned i.e. positive numbers, due to this bias.
- For example, If the actual exponent is -3 Then the represented biased exponent in IEEE754 number representation,
- Single Precision would be E = E' +127 where E is biased exponent and E' is actual exponent. Therefore, we would get E = -3 + 127 = 124
- Double Precision would be E = E' +1023 where E is biased exponent and E' is actual exponent. Therefore, we would get E = -3 + 1023 = 1020
- It is evident that for Single precision numbers, actual exponent range is – 127 to +127 getting converted as 0 to 254 in the Biased exponent representation. Similarly, for Double precision numbers, actual exponent range is given as – 1023 to +1023 getting converted as 0 to 2047 in the Biased exponent representation.
- Therefore, the sign of the exponent can be determined from the represented biased exponent as :
  - **For Single Precision numbers** – Exponent is +Ve, if the biased exponent is having value equal or above 127.
  - **For Single Precision numbers** – Exponent is +Ve, if the biased exponent is having value equal or above 1023.

UEx. 5.1.11 MU.- Q. 1(b), May 18, 10 Marks

Explain IEEE 754 floating point representation formats and represent $(34.25)_{10}$ to single precision format.

☑ **Soln. :**

▶ **Step 1 :** Convert the number to binary

For integer part
$34 \rightarrow 0010\ 0010$
For fractional part
$0.25 \rightarrow 01$
Therefore the binary number is
$(34.25)_{10} \rightarrow (0010\ 0010.01)_2$

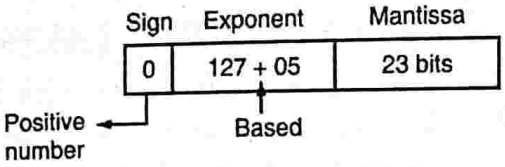
▶ **Step 2 :** Adjusting the number in mantissa and exponent

1.0001001×2^5

$\therefore$ Exponent → 05 and Mantissa → 0.0001001

Module 2

▶ **Step 3 :** IEEE – 754 Format



∴ Exponent 127 + 5 = 132

$(132)_{10} \rightarrow (1000\ 0100)_2$

Sign	Exponent	Mantissa
0	10000100	000000000000 ••• 0001001
1 bit	8 bits	23 bits

UEx. 5.1.12 **MU - Q. 1(b), May 16, 5 Marks**

Express $(35.25)_{10}$ in the IEEE single precision standard of floating point representation.

☑ **Soln. :**

▶ **Step 1 :** Convert the number into binary

For integer part

$(35)_{10} \rightarrow (100011)$

For fraction part

$(0.25)_{10} \rightarrow 01$

Therefore the binary number is

$(35.25)_{10} \rightarrow (100011.01)_2$

▶ **Step 2 :** Adjusting the number in exponent and Mantissa

1.0001101×2^5

Exponent – 05 Mantissa = 0.0001101

▶ **Step 3 :** IEEE 754 format

Exponent

$127 + 05 = 132$

$(132)_{10} \rightarrow (1000\ 0100)_2$

UEx. 5.1.13 **MU - Q. 2(a), Dec. 16, 10 Marks**

Convert number 127.125 to Single Precision and Double Precision IEEE-754 numbers.

☑ **Soln. :**

Single Precision

- Here 127 = 1111111 and 0.125 = 0.001 in Binary number system.
- Therefore, 127.125 = 1111111.001 in Binary number system.
- We normalize this number as → + 1.111111001 X 2^6; comparing with $(-1)^s$ 1.M X 2^{E^*}, we get -
- Normalized mantissa in 1.M format is 1.111111001 which means M = 111111001
- Further, the actual exponent E* = 6
- Therefore, biased exponent E = E*+127 = 6 + 127 = 133
- In Binary number system E = 10000101 and number is position, therefore, sign bit S=1
- Therefore, the IEEE-754 Single Precision number representation of **127.125** would be –

31	30 23	22 0
1	1 0 0 0 0 1 0 1	111 1 1 1 0 01 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- In Hexadecimal, the number would be given as →**C2FE 4000$_H$**

Double Precision

- Here 127 = 1111111 and 0.125 = 0.001 in Binary number system.
- Therefore, 127.125 = 1111111.001 in Binary number system.
- We normalize this number as → + 1.111111001 X 2^6; comparing with $(-1)^s$ 1.M X 2^{E^*}, we get -
- Normalized mantissa in 1.M format is 1.111111001 which means M = 111111001
- Further, the actual exponent E* = 6
- Therefore, biased exponent E = E*+1023 = 6 + 1023 = 1029
- In Binary number system E = 10000000101 and number is position, therefore, sign bit S=1
- Therefore, the IEEE-754 Single Precision number representation of **127.125** would be –

63	62 52	51 0
1	10000000101	1111110010000000000000000000000000000000

- In Hexadecimal, the number would be given as →**C051 FC80 0000 0000$_H$**

5.1.4(B) Arithmetic Operations

This section highlights the basic arithmetic operations which are performed on single precision floating point numbers. The section deals with floating point addition, subtraction, division and multiplication.

Procedure for addition/subtraction

UQ. 5.1.10 Draw and explain floating point addition subtraction algorithm.

MU - Q. 6(B), May 19, 10 Marks,
Q. 6(b), Dec. 18 , 10 Marks

▶ **Step 1 :** Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.

e.g. $(2.09400 \times 10^2) + (4.3100 \times 10^4)$

The smaller exponent is 2.9400×10^2

$\therefore 0.029400 \times 10^4$

The number is shifted right with incrementing the exponent.

▶ **Step 2 :** Set the exponent of the result equal to the large exponent.

i.e. 0.029400×10^4

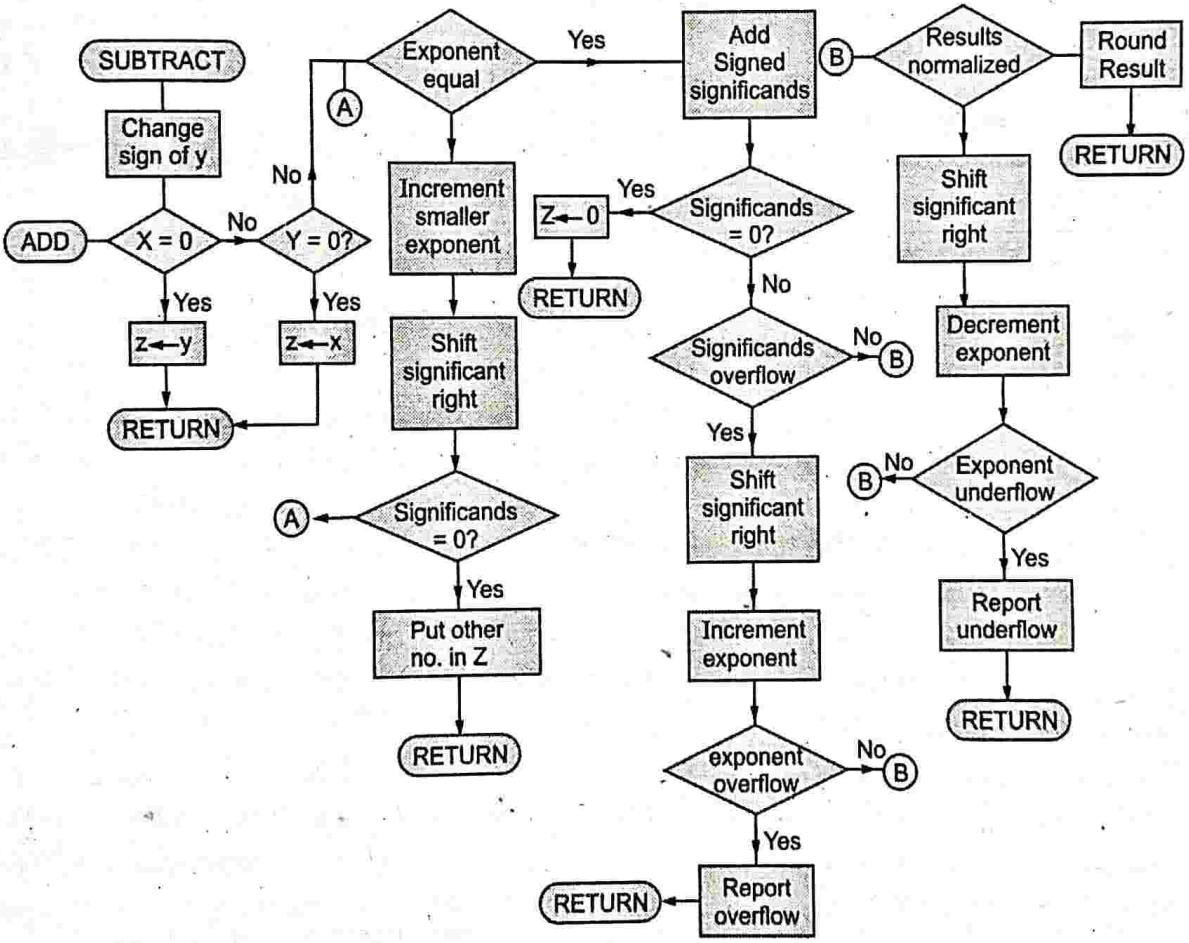
▶ **Step 3 :** Perform addition/subtraction on the mantissa and determine the sign of the result.

$$\begin{array}{r} 4.3100 \times 10^4 \\ + \quad 0.0294 \times 10^4 \\ \hline 4.3394 \times 10^4 \end{array}$$

▶ **Step 4 :** Normalize the resulting value, if necessary.

$4.3394 \times 10^4 \Rightarrow$ No need to normalize the number.

The following flowchart summarizes the complete process of addition/subtraction

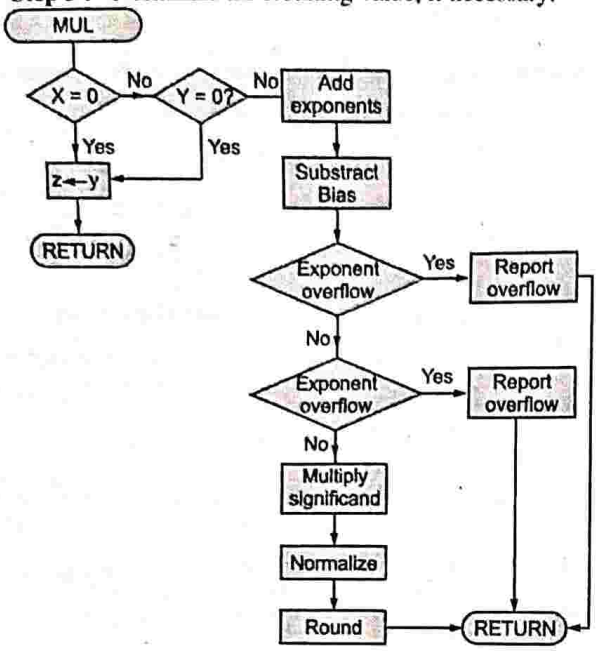


(1A19)**Fig. 5.1.18**

Module 2

Procedure for multiplication

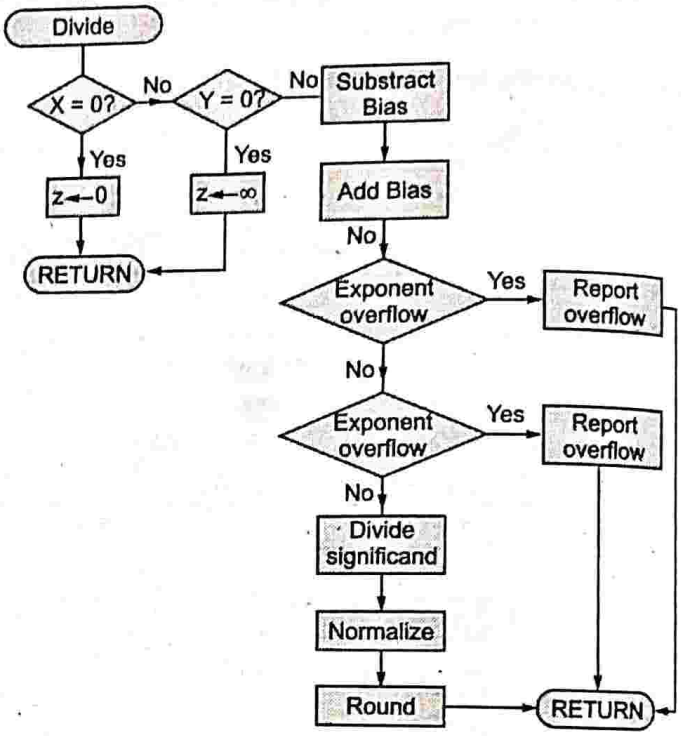
- **Step 1 :** Add the exponents and subtract 127.
- **Step 2 :** Multiply the mantissa and determine the sign of the result.
- **Step 3 :** Normalize the resulting value, if necessary.



(1A20)**Fig. 5.1.19**

Procedure for Division

- **Step 1 :** Subtract the exponents and add 127.
- **Step 2 :** Divide the mantissa and determine the sign of the result.
- **Step 3 :** Normalize the resulting value, if necessary.



(1A21)**Fig. 5.1.20**

...Chapter Ends

❑❑❑

Combinational Logic Design

University Prescribed Syllabus

Processor Organization and Architecture

3.1 Introduction : Half adder, Full adder, MUX, DMUX, Encoder, Decoder (IC level).

Previous University Paper Questions

Module 3

6.1 INTRODUCTION

Q. 6.1.1 *What do you understand by a digital logic circuit ? What are its different types ?*

Ans. :

Digital Logic Circuit

Definition : A digital logic circuit is a circuit formed when one or more gates are combined together to form a complex switching circuit.

- The digital logic circuits are also called as **switching circuits** as the voltage levels are assumed to be switched instantaneously from one value to another value.
- The **digital logic gates** are the **basic building blocks of a digital system**. They are used in digital electronic devices like mobile phones, calculators, computers etc.

Classification

- The digital logic circuits are **classified** as :
  1. Combinational Logic Circuits.
  2. Sequential Logic Circuits.

6.2 COMBINATIONAL CIRCUITS

Q. 6.2.1 *What is a combinational circuit ?*

Ans. :

(A) Combinational Logic Circuits

Definition : Combinational logic circuits are defined as the circuits whose output depends on its present inputs at any instant of time.

(B) Characteristics

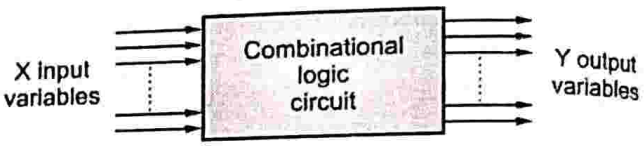
1. The logic gates are the basic building blocks of the combinational circuits.
2. The name combinational is given as it is made up of **combination of logic gates.**
3. A combinational circuit performs an operation that is logically assigned by :

   (a) Boolean expression (b) Truth table (c) Logic diagram
4. The examples of combinational logic circuits are

   (i) Half-Adders (ii) Full Adder (iii) Half Subtractor (iv) Full Subtractor (v) Binary Adder (vi) BCD Adder (vii) Multiplexer (viii) Demultiplexer
5. The combinational circuits do not need any memory as the outputs are dependent on the present inputs at any instant of time.

- Fig. 6.2.1 shows the block diagram of a combinational circuit.

(C) Block Diagram



(101)**Fig. 6.2.1 : Block diagram of a combinational logic circuit**

6. A combinational logic circuit can have x number of input variables and Y number of output variables. There are 2^X input combinations.
7. For every input there will be one output combination. Each output is expressed in terms of the x inputs.
8. Combinational logic circuits can be very simple or complex.

Q. 6.2.2 *Explain the procedure for designing combinational circuits.*

Ans. :

Procedure for Designing Combinational Circuits

A combinational circuit can be designed using the following steps :

- **Step I :** Define and understand the problem. Identify and determine the number of input variables available and the output variables required.
- **Step II :** Represent each input and output variable with the help of symbols.
- **Step III :** Derive the truth table for describing the relationship between the input and output variables.
- **Step IV :** Obtain the Boolean expression for every output variable in terms of the input variables. Simplify the Boolean expression for output variables.
- **Step V :** The logic diagram is drawn by implementing the minimized Boolean expressions.

Q. 6.2.3 Why are combinational logic circuits called as decision making circuits ?

Ans. :

- Combinational logic is a method of combining the logic gates, in-order to process the inputs to generate at least one output depending on the logic function of each logic gate.
- The combinational logic circuits are designed using individual logic gates. Hence, they are called **as decision making circuits.**

Q. 6.2.4 How are functions of combinational logic circuits speficified ?

Ans. : The functions of combinational logic circuits can be specified in three ways. They are :

1. Boolean expression
2. Truth table
3. Logic diagram

1. Boolean Expression

- The output of a combinational logic circuit can be expressed in the form of an expression using Boolean algebra.
- This expression is called as **Boolean expression.**
- The Boolean expression shows the operation of the combinational logic circuit for each input combination (0 or 1) that produces a HIGH (logic 1) output.

2. Truth Table

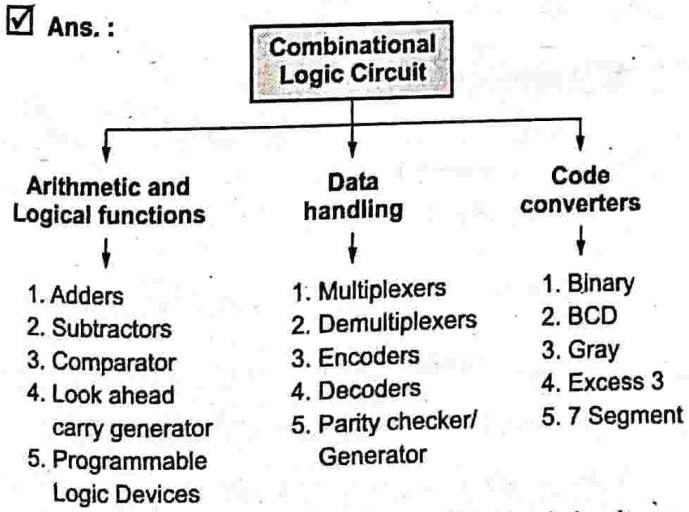
The **truth table** is a table that shows all possible combinations of inputs and the corresponding outputs.

3. Logic Diagram

The graphical representation of a combinational logic circuit using logic gates is called as a **logic diagram.**

Q. 6.2.5 Classify combinational circuits.

Ans. :



(1D151)**Fig. 6.2.2 : Classification of combinational circuits**

- The combinational logic circuits can be classified into different types depending on their use as shown in Fig. 6.2.2.
- There are 3 main types of combinational circuits. They are :

1. Arithmetic and logic functions

For implementing arithmetic and logical functions we use following combinational circuits :

1. Adders
2. Subtractors
3. Comparators
4. Look ahead carry generator

2. Data handling/data transmission and reception

The combinational logic circuits used for data handling are :

1. Multiplexer
2. Demultiplexer
3. Encoder
4. Decoders
5. Parity checker/generator

3. Code converters

Combinational logic circuits are used for implementing different code converters like binary to gray, BCD to excess-3 etc.

Module 3

6.3 BINARY ADDERS

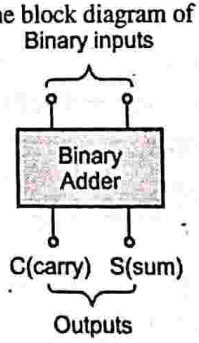
Q. 6.3.1 What do you understand by binary adders ? Give their classification.

Ans. :

Binary Adder

Definition : *Binary Adders are the combinational logic circuits used to perform the addition of the two binary digits.*

- Fig. 6.3.1 shows the block diagram of a binary adder.



(1D2)**Fig. 6.3.1 : Block diagram of binary adder**

- The addition operation has 4 possible combinations :
  1. $0 + 0 = 0$
  2. $0 + 1 = 1$
  3. $1 + 0 = 1$

  } produce one bit sum
  4. $1 + 1 = 10$ } produce two bit sum with carry =1 and sum = 0

- The operations (1), (2), (3) produce a sum whose length is one bit.
- In the operation (4), the binary sum consists of 2 digits.
- The MSB bit of the result is called as the **Carry** and the LSB bit of the result is called as **Sum**.
- The types of Binary Adders are :

1. Half Adder 2. Full Adder

6.4 HALF ADDER

Q. 6.4.1 What do you mean by half adder ?

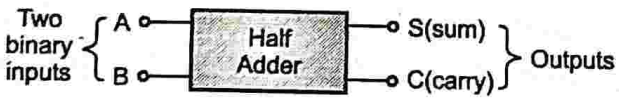
Ans. :

(A) Half Adder

Definition : A combinational logic circuit that computes the addition of two binary digits is called as a Half-Adder.

(B) Block Diagram

- A half adder has two inputs and two outputs
- Let A and B be the two inputs.
  Input A is called as the **augend.**
  Input B is called as the **addend.**
- The two outputs are **S (Sum)** and **C (Carry).**
- Fig. 6.4.1 shows the block diagram of a half adder.



(1D3)**Fig. 6.4.1 : Block diagram of half adder**

(C) Truth Table

Table 6.4.1 : Truth table of Half-Adder

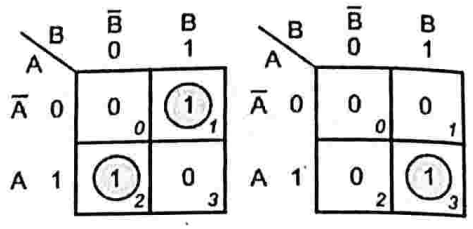
Inputs		Outputs	
A	B	S (Sum)	C (Carry)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(D) Operation

The half-adder is an arithmetic circuit that performs the addition of two binary digits using the rules of binary addition.

(E) K-maps for S(Sum) and C (Carry)

K-map for sum **K-map for carry**



$S = \bar{A}B + A\bar{B} = A \oplus B$ $C = AB$

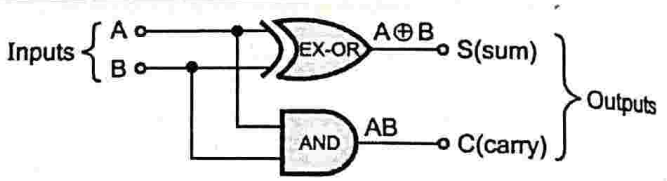
(1D4)(a) K-map for S (Sum) (1D5)(b) K-map for C (Carry)

Fig. 6.4.2

Expression for S (sum) = $A \oplus B = A\bar{B} + \bar{A}B$

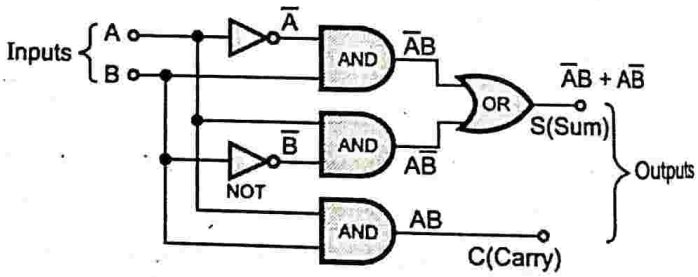
Expression for C(carry) = AB

(F) Logic diagram of Half Adder



(1D6)**Fig. 6.4.3 : Logic diagram realization of half adder**

(G) Logic Diagram of Half Adder Using Basic gates



(1D7)**Fig. 6.4.4 : Logic realization of half adder using basic gates**

(H) Drawback of Half-Adder

- If we want to add two 2-bit binary numbers A and B. Let $A = A_1 A_0$ and $B = B_1 B_0$

$$\begin{array}{rl} & A \quad\quad A_1\ A_0 \\ + & B \quad\quad B_1\ B_0 \\ \hline & \boxed{C_0} \text{ Carry from addition of } A_0 \text{ and } B_0 \\ \hline & \boxed{C_1}\ S_1\ S_0 \end{array}$$

(1D7A)**Fig. 6.4.5 : Two bit binary addition**

- When we add A_0 and B_0, the half adder produces outputs as sum S_0 and carry C_0.

- However, for the addition of the next bit we need to add three bits A_1,B_1 and previous carry C_0.
- It is not possible to add three bits with the help of a half-adder i.e **half-adders are not suitable for multi-byte addition.**
- Hence, half-adders are not used in practice. **Full Adders** are used in order to overcome this drawback of half-adders.

(I) Applications of Half Adder

1. The ALU (arithmetic logic unit) of a computer **uses half adder** to compute the binary addition operation on two bits.
2. Half adder is used to make full adder.

6.5 FULL ADDERS

Q. 6.5.1 What do you mean by full adder ?

UQ. 6.5.2 Design a full adder using half adder and additional gates.

MU - Q. 1(c), May 14, 5 Marks

Ans. :

(A) Full Adder

Definition : A combinational logic circuit that computes the addition of three binary digits is called as a Full-Adder.

(B) Block diagram

- A full-Adder has three inputs and two outputs.
- Let **A , B and C_{in} (previous carry)** be the **three inputs.**
- The **two outputs** are **S (Sum)** and **C_{out} (Carry)**. Fig. 6.5.1 shows the block diagram of a full adder.

(1D8)**Fig. 6.5.1 : Block diagram of a full adder**

(C) Truth Table

Table 6.5.1 : Truth table of Full-Adder

Inputs			Outputs	
A	B	C_{in}	S (Sum)	C_{out}(Carry)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(D) Operation

- The full-adder is an arithmetic circuit that performs the addition of three binary digits. i.e. the full adder adds the bits A and B along with previous carry C_{in} and produces outputs S (Sum) and C_{out} (carry).

(E) K-maps for S(Sum) and C_{out} (Carry)

K-map for S(Sum)

A \ BC_{in}	$\bar{B}\bar{C}_{in}$ 00	$\bar{B}C_{in}$ 01	BC_{in} 11	$B\bar{C}_{in}$ 10
$\bar{A}$ 0	0 (0)	1 (1)	0 (3)	1 (2)
A 1	1 (4)	0 (5)	1 (7)	0 (6)

$$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$$

$$S = C_{in}(\bar{A}\bar{B} + AB) + \bar{C}_{in}(\bar{A}B + A\bar{B})$$

$$S = C_{in}\overline{(A \oplus B)} + C_{in}(A \oplus B)$$

($\because \bar{A}\bar{B} + AB = \overline{(A \oplus B)}$ EX-NOR operation

$\bar{A}B + A\bar{B} = A \oplus B$)

$$S = A \oplus B \oplus C_{in}$$

(1D9) **Fig. 6.5.2(a) : K-map for Sum**

K-map for C_{out} (Carry)

A \ BC_{in}	$\bar{B}\bar{C}_{in}$ 00	$\bar{B}C_{in}$ 01	BC_{in} 11	$B\bar{C}_{in}$ 10
$\bar{A}$ 0	0 (0)	0 (1)	1 (3)	0 (2)
A 1	0 (4)	1 (5)	1 (7)	1 (6)

$$C_{out} = AB + AC_{in} + BC_{in}$$

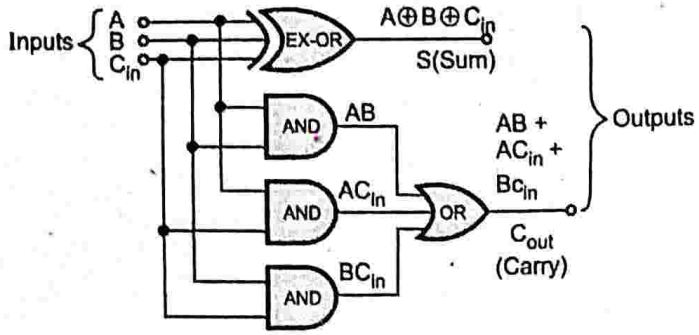
(1D10) **Fig. 6.5.2(b) : K-map for C_{out} (Carry)**

Expression for S (sum) = $A \oplus B \oplus C_{in}$

Expression for C_{out} (carry) = $AB + AC_{in} + BC_{in}$

Module 3

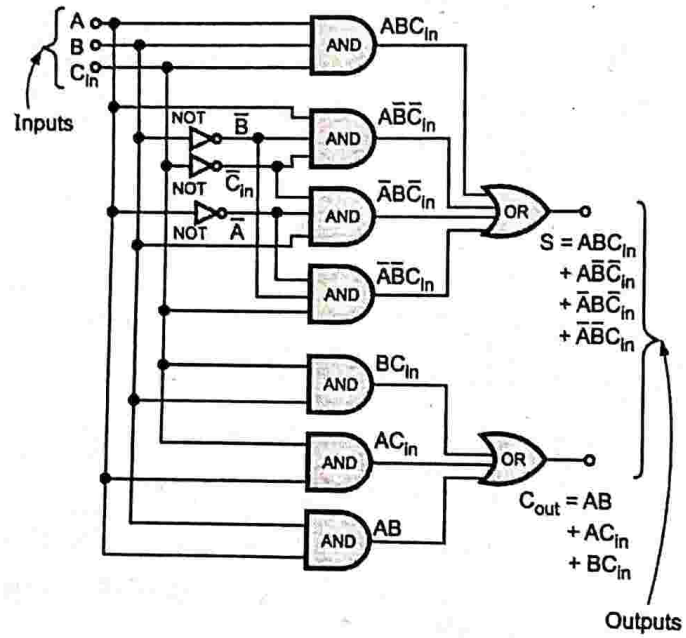
(F) Logic Diagram of Full Adder



(1D11)**Fig. 6.5.3 : Logic diagram of full adder**

(G) Logic Diagram of Full Adder using Basic Gates

Fig. 6.5.4 shows the implementation of full adder using basic gates.



(1D12)**Fig. 6.5.4 : Logic diagram of full adder using basic gates**

6.5.1 Advantages of Full Adder

- The addition of multiple bits can be obtained by increasing the number of the Full adders in a circuit.
- If we want to add two n-bit binary numbers, then the number of full adders required is equal to the number of bits want of each binary number.

6.5.2 Disadvantages of Full Adder

- When a full adder is implemented using two half-adders and an OR gate the bits need to propagate through many successive gates
- This increases the total propagation delay in comparison to the propagation delay of the full adder circuit using AOI logic.

(J) Applications of Full Adder

The applications of Full adder are as follows :

1. Arithmetic Logic Unit of a computer uses full adders to compute binary addition.
2. In CPU (central processing unit) and graphics processing unit (GPU) for graphics related applications, where there is a very much need of complex computations. The GPU uses optimized ALU which is made up of full adders.
3. Ripple ahead carry adder.
4. Digital calculators for performing arithmetic addition.
5. In microcontrollers, for arithmetic additions, to generate memory addresses, PC (program counter) to point the next instruction, the ALU uses full adder.
6. Digital signal processors (DSP) and networking systems.

Q. 6.5.3 What are the full adder's inputs that will produce each of the following outputs ?

(i) $\Sigma = 0, C_{out} = 0$

(ii) $\Sigma = 1, C_{out} = 0$

(iii) $\Sigma = 1, C_{out} = 1$

(iv) $\Sigma = 0, C_{out} = 1$.

☑ **Ans. :**

From the truth Table 6.5.1 of full adder

(i) $\sum = 0, C_{out} = 0$ is when

$A = 0, B = 0, C_{in} = 0$

(ii) $\sum = 1, C_{out} = 0$ is when

$A = 0, B = 0, C_{in} = 1$ or $A = 0, B = 1, C_{in} = 0$

or $A = 1, B = 0, C_{in} = 0$

(iii) $\sum = 1, C_{out} = 1$ is when

$A = 1, B = 1, C_{in} = 1$

(iv) $\sum = 0, C_{out} = 1$ is when

$A = 0, B = 1, C_{in} = 1$ or $A = 1, B = 0, C_{in} = 1$

or $A = 1, B = 1, C_{in} = 0$

6.5.3 Full Adder using NAND Gates

Q. 6.5.4 Design full adder using NAND gates.

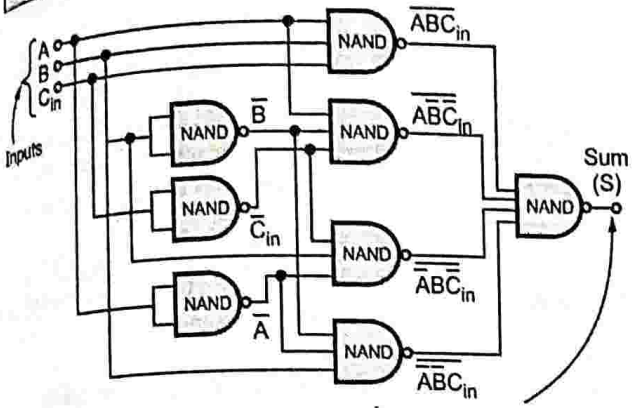
☑ **Ans. :**

$$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in} \text{ and}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Implementing the logic expressions in NAND-NAND logic, in order to realize full adder.

Logic diagram of full adder using NAND gates



$$S = \overline{\overline{ABC_{in}} \cdot \overline{A\overline{B}C_{in}} \cdot \overline{A\overline{B}\overline{C}_{in}} \cdot \overline{\overline{A}\overline{B}\overline{C}_{in}}}$$

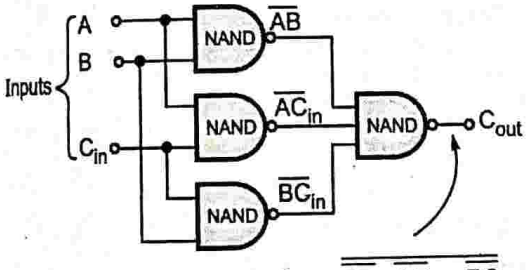
$$S = \overline{ABC_{in}} + \overline{A\overline{B}C_{in}} + \overline{A\overline{B}\overline{C}_{in}} + \overline{\overline{A}\overline{B}\overline{C}_{in}}$$

$(\because \overline{A \cdot B} = \overline{A} + \overline{B}$ DeMorgan's theorem)

$$S = ABC_{in} + A\overline{B}C_{in} + A\overline{B}\overline{C}_{in} + \overline{A}\overline{B}\overline{C}_{in}$$

$(\because \overline{\overline{A}} \cdot A)$

(1D13)(a)



$$C_{out} = \overline{\overline{AB} \cdot \overline{AC_{in}} \cdot \overline{BC_{in}}}$$

$$C_{out} = \overline{\overline{AB}} + \overline{\overline{AC_{in}}} + \overline{\overline{BC_{in}}}$$

$(\because \overline{A \cdot B} = \overline{A} + \overline{B}$ DeMorgan's theorem)

$$C_{out} = AB + AC_{in} + BC_{in}$$

$(\because \overline{\overline{A}} \cdot A)$

(b)

(1D14)**Fig. 6.5.5 : Implementing full adder using NAND gates**

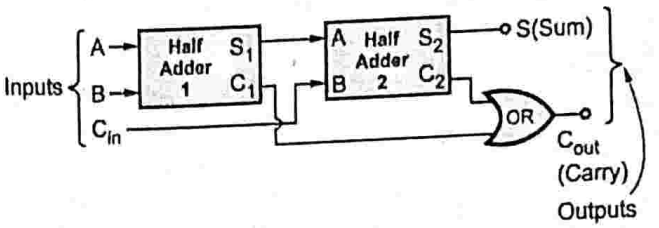
6.5.4 Full adder using Half Adders

UQ. 6.5.5 How will you implement full adder using half adder ? Draw the circuit diagram. MU - Q. 3(c), Dec. 19, 5 Marks

UQ. 6.5.6 Design a full adder using half adder and additional gates. MU - Q. 1(c), May 14, 5 Marks

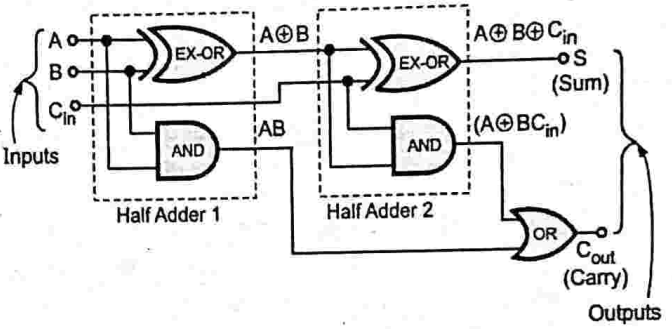
☑ **Ans. :**

- A full adder can be constructed using two half adders and an OR gate as shown in Fig. 6.5.6.



(1D15)**Fig. 6.5.6 : Full adder two using half adders and a OR gate**

- Fig. 6.5.7 shows the logic diagram of full adder using two half adders.



(1D16)**Fig. 6.5.7 : Full adder using two half adders**

Proof

From Fig. 6.5.7 the expression for S (sum) is,

$$\mathbf{S(sum)} = A \oplus B \oplus C_{in}$$

It is the expression of S (sum) for full adder.

$$C_{out}(\text{carry}) = A \oplus B\, C_{in} + AB$$

$$C_{out} = (\overline{A}B + A\overline{B})\, C_{in} + AB\,(1 + C_{in})$$

$(\because A + 1 = 1)$

$$C_{out} = \overline{A}B\, C_{in} + A\overline{B}\, C_{in} + AB + AB\, C_{in}$$

$$C_{out} = B\, C_{in}\,(\overline{A} + A) + A\overline{B}\, C_{in} + AB\,(1 + C_{in})$$

$$C_{out} = B\, C_{in} + A\, C_{in}\,(B + \overline{B}) + AB$$

$(\because A + \overline{A} = 1)$

$$C_{out} = B\, C_{in} + A\, C_{in} + AB \qquad (\because A + \overline{A} = 1)$$

- The expression for C_{out} is same as the expression for full adder.
- **Thus, two half adders and one OR gate can be used to implement a full adder.**

Module 3

6.6 DIFFERENCE BETWEEN HALF ADDER AND FULL ADDER

Q. 6.6.1 State the difference between half and full adder. MU - Dec. 18

Ans. :

Sr. No.	Parameter	Half Adder	Full Adder
1.	**Definition**	A combinational logic circuit that computes the addition of two binary digits is called as a Half-Adder.  (1D3)Fig. 6.6.1 : Block diagram of half adder	A combinational logic circuit that computes the addition of three binary digits is called as a Full-Adder.  (1D8)Fig. 6.6.2 : Block diagram of a full adder
2.	**Inputs**	A half adder has **two inputs (A and B).**	A Full adder has three inputs **A, B and C_{in}.**
3.	**Outputs**	A half adder has two **outputs S(Sum) and C(Carry).**	A Full adder has two **outputs S(Sum) and C_{out} (Carry).**
4.	**Carry input**	Half adder does not have carry input.	Full adder has carry input.
5.	**Components used**	Half adder uses one EX-OR and one AND gate for its implementation.	Full adder uses two EX-OR gates, two AND gates and one OR gate for its implementation.
6.	**Boolean expression**	S (sum) = $A \oplus B = A\bar{B} + \bar{A}B$ C(carry) = AB	S (sum) = $A \oplus B \oplus C_{in}$ C(carry) = $AB+AC_{in}+BC_{in}$
7.	**Logic gates used**	The number of logic gates used is less.	The number of logic gates used is more.
8.	**Applications**	1. Constructing full adders 2. Computer ALU	1. Arithmetic Logic Unit 2. Digital calculators 3. Digital computers 4. Digital signal processors

6.7 MULTIPLEXERS (MUX)

Q. 6.7.1 What is multiplexing ? Explain the need of multiplexers.

Ans. :

Definition : *Multiplexing is the method of selecting one of the available data inputs and connecting it to the output.*

- **Need of multiplexers :** In digital systems, digital data is present on multiple lines.
- In order to route this data over a single line, we need to select the data on a line.
- It is achieved by a device called **multiplexer.**
- A multiplexer selects a single data line from the multiple input lines and the data from that selected line is available on the output.
- As it selects one input and connects it to the output the multiplexer is called as **data selector.**

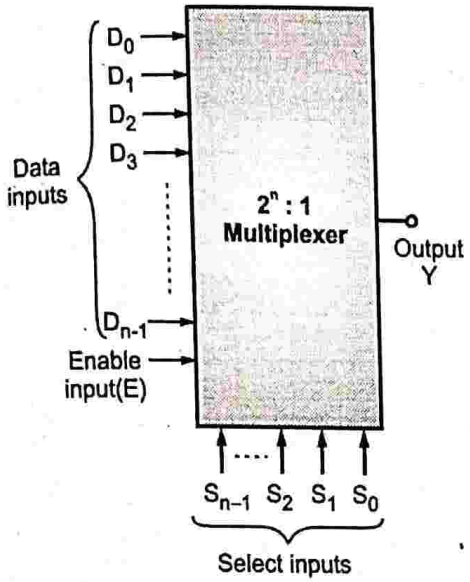
Q. 6.7.2 Discuss multiplexer with suitable diagram.

Ans. :

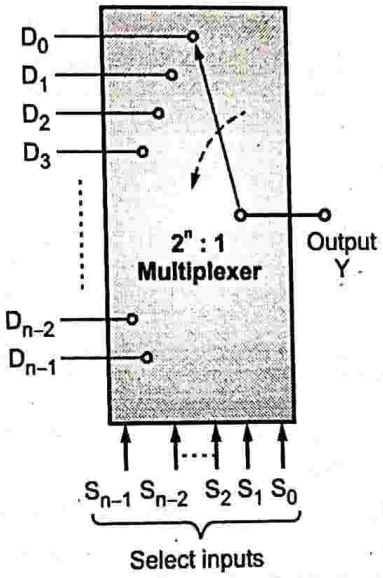
Definition : *Multiplexer a combinational logic circuit that selects one of the n data inputs. It routes the data from the selected data line to the output.*

- It is also called as **data selector.**
- Fig. 6.7.1 shows the functional diagram of a $2^n : 1$ multiplexer.

- A multiplexer has many data input lines and a single output line. The select inputs are responsible. For selecting a specific input line.



(1D29)**Fig. 6.7.1(a) : Block diagram of 2^n : 1 multiplexer**



(1D30)**Fig. 6.7.1(b) : Equivalent circuit**

- For a 2^n : 1 multiplexer there are n selection lines, 2^n data input lines and one output.
- The select inputs are also called as **address inputs.**
- Depending on the input combination applied to the select inputs, one of the many inputs is selected and connected to the output.
- Hence, a multiplexer is **"many into one"**. Thus, it acts as **digitally controlled multi-position switch.**
- Enable input (E) is active low input used for cascading.

Types of Multiplexers

The types of multiplexers are,

1. 2 : 1 multiplexer
2. 4 : 1 multiplexer
3. 8 : 1 multiplexer
4. 16 : 1 multiplexer
5. 32 : 1 multiplexer

6.7.1 Advantages of Multiplexers

Q. 6.7.3 Explain the advantages of multiplexers.

☑ **Ans. :**

1. It can be used to implement combinational circuits.
2. The logic design is simplified.
3. The logic expressions need not be simplified.
4. The transmission circuit is less complex and economical.
5. Heat dissipation is low.
6. The ability of MUX to switch digital signals can be extended to switch to audio signals, video signal etc.
7. It reduces the number of wires.

Module 3

6.7.2 Applications of Multiplexers

Q. 6.7.4 Explain the applications of multiplexers.

☑ **Ans. :** Applications of multiplexers are as follows :

1. **Data routing :** Multiplexer can be used to route the data from multiple data lines to a single destination.
2. Parallel to serial converter.
3. Operation sequencing.
4. They are used in time multiplexing systems.
5. They are used in telephone networks to integrate multiple audio signals on a single transmission line.
6. They are used to transmit data signals from the computer system of a satellite to the ground system through GSM communication.
7. They are used in computer memory.
8. They are used in data acquisition systems.
9. They are used in A/D and D/A converters.
10. They are used as logic function generators i.e. they can be used to implement the SOP expressions directly from the truth table without any simplification.

Q. 6.7.5 What is the minimum number of selection lines needed for selecting one of the 2^n input lines ?

☑ **Ans. :** The minimum number of selection lines n required for selecting one of the 2^n input line is,

$$n = \frac{\log_n}{\log_2}$$

Q. 6.7.6 *Why is time multiplexing used in display systems ?*

☑ **Ans. :** Time multiplexing is used in display systems to save power.

Q. 6.7.7 *Can the E 'enable' input of the multiplexer be used to increase its size ?*

☑ **Ans. :** The E **'enable' input** of the multiplexer can be used to increase its size.

Q. 6.7.8 *List the multiplexer ICs.*

☑ **Ans. :**

Multiplexer ICs

IC number	Description	Output
74150	16 : 1 multiplexer	Inverted input
74151	8 : 1 multiplexer	Complementary output
74151A	8 : 1 multiplexer	Complementary output
74152	8 : 1 multiplexer	Inverted input
74153	Dual 4 : 1 multiplexer	Same as input
74158	Quad 2 : 1 multiplexer	Inverted input
74157	Quad 2 : 1 multiplexer	Same as input

6.7.3 2 : 1 Multiplexer

Q. 6.7.9 *Explain the working of a 2 : 1 multiplexer.*

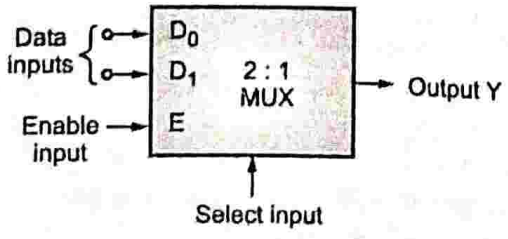
☑ **Ans. :**

(A) Function

A 2 : 1 multiplexer selects one of the two inputs and connects it to the output.

(B) Block Diagram

Fig. 6.7.2 shows the block diagram of a 2 : 1 MUX. It has two data inputs D_0 and D_1, one select input and one output.



(1D31)**Fig. 6.7.2 : Block diagram of a 2 : 1 MUX**

(C) Function Table

Table 6.7.1 : Function table of 2 : 1 MUX

E	S	Y
1	0	D_0
1	1	D_1
0	X	0

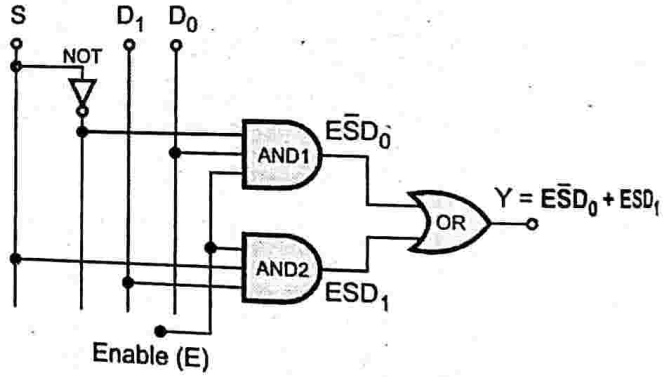
(D) Truth Table

Enable (E)	Select (S)	D_1	D_0	Output Y	
0	X	X	X	0	
1	0	X	0	0	
1	0	X	1	1	← $E\bar{S}D_0$
1	1	0	X	0	
1	1	1	X	1	← ESD_1

$Y = 1$ when $E\bar{S}D_0 = 1$ or $ESD_1 = 1$

$\therefore Y = E\bar{S}D_0 + ESD_1$

(E) Logic realization of 2 : 1 MUX



(1D32)**Fig. 6.7.3 : Logic realization of 2 : 1 MUX**

(F) Construction of 2 : 1 MUX

- D_0 is applied as one input to AND1 gate and D_1 is applied as one input to the AND2 gate.
- The Enable (E) input is applied to both the AND gates.
- To select (S) input is applied to AND2 gate and its complement ($\bar{S}$) is applied to the AND1 gate.
- The outputs from both the AND gates are applied to the OR gate to obtain the desired output Y.

(G) Working

1. When E = 0, irrespective of the select inputs the output Y = 0
2. When E = 1, S = 0

The AND1 gate is enabled and AND2 gate is disabled. The other input to the AND1 gate is D_0 its output $Y = D_0$.

3. When E = 1, S = 1

The AND1 gate is disabled and AND2 gate is enabled. The other input to the AND2 gate is D_1. $\therefore Y = D_1$

6.7.4 4 : 1 Multiplexer

Q. 6.7.10 Explain a 4 : 1 input Multiplexer.

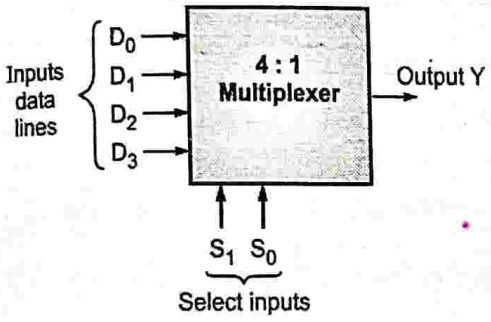
Ans. :

(A) Function

A 4 : 1 multiplexer selects one of the four inputs and connects it to the output.

(B) Block diagram

Fig. 6.7.4 shows the block diagram of a 4 : 1 MUX. It has four data inputs D_0, D_1, D_2 and D_3 two select lines S_0 and S_1 and one output.



(1D33)**Fig. 6.7.4 : Block diagram of 4 : 1 MUX**

(C) Function table of 4 : 1 MUX

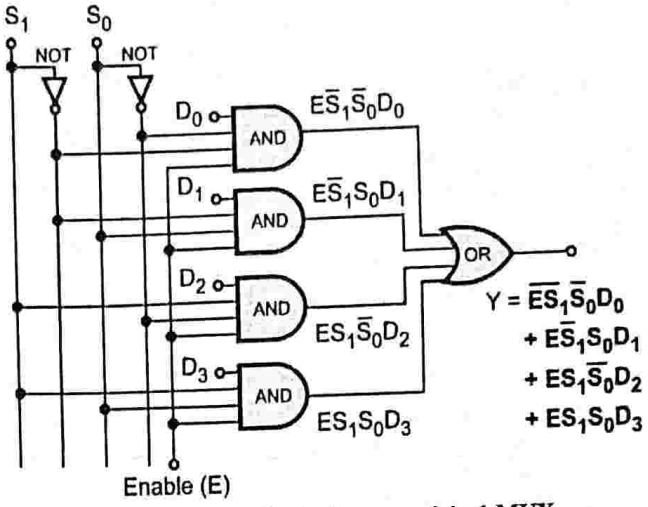
Function table of 4 : 1 MUX

E	S_1	S_0	Y	
0	X	X	0	
1	0	0	D_0	← $E\bar{S}_1\bar{S}_0D_0$
1	0	1	D_1	← $E\bar{S}_1S_0D_1$
1	1	0	D_2	← $ES_1\bar{S}_0D_2$
1	1	1	D_3	← $ES_1S_0D_3$

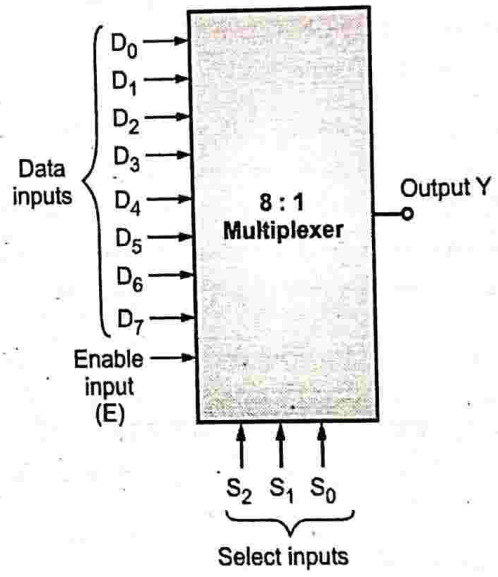
The expression for output is,

$$Y = E\bar{S}_1\bar{S}_0D_0 + E\bar{S}_1S_0D_1 + ES_1\bar{S}_0D_2 + ES_1S_0D_3$$

(D) Logic Diagram of 4 : 1 MUX



(1D34)**Fig. 6.7.5 : Logic diagram of 4 : 1 MUX**



(1D35)**Fig. 6.7.6 : Logic symbol of 8 : 1 MUX**

(E) Function table of 8 : 1 MUX

Enable (E)	Select lines			Output Y	
	S_2	S_1	S_0		
0	X	X	X	0	
1	0	0	1	D_0	← $E\bar{S}_2\bar{S}_1\bar{S}_0D_0$
1	0	0	1	D_1	← $E\bar{S}_2\bar{S}_1S_0D_1$
1	0	1	0	D_2	← $E\bar{S}_2S_1\bar{S}_0D_2$
1	0	1	1	D_3	← $E\bar{S}_2S_1S_0D_3$
1	1	0	0	D_4	← $ES_2\bar{S}_1\bar{S}_0D_4$

Module 3

Enable (E)	Select lines			Output Y	
	S_2	S_1	S_0		
1	1	0	1	D_5	← $ES_2\bar{S}_1S_0D_5$
1	1	1	0	D_6	← $ES_2S_1\bar{S}_0D_6$
1	1	1	1	D_7	← $ES_2S_1S_0D_7$

(F) Operation of a 4 : 1 MUX

- When E = 0, output Y = 0 irrespective of S_1 and S_0 inputs.
- The logic levels applied to the select inputs (S_1 S_0) determine which AND gate is enabled.
- The data input of that AND gate passes through the OR gate to the output e.g. if S_1S_0 = 10, the AND gate related to data input D_2 is enabled as E = 1, S = 1, $\bar{S}_0$ = 1.
- The output of the other three AND gates is 0 as atleast one input to three AND gates is zero (0).
- The output of OR gate is equal to D_2

$$Y = ES_1\bar{S}_0D_2 = 1.1.1\ D_2 = D_2$$

D_2 is connected to the output when S_1S_0 = 10

Similarly, D_0 is connected to the output when S_1S_0 = 00.

D_1 is connected to the output when S_1S_0 = 01

and D_3 is connected to the output when S_1S_0 = 11

6.7.5 8 : 1 Multiplexer

Q. 6.7.11 Draw logic diagram of 8:1 multiplexer. Write it's truth table.

Ans. :

(A) Function

A 8 : 1 multiplexer selects one of the eight inputs and connects (routes) it to the output.

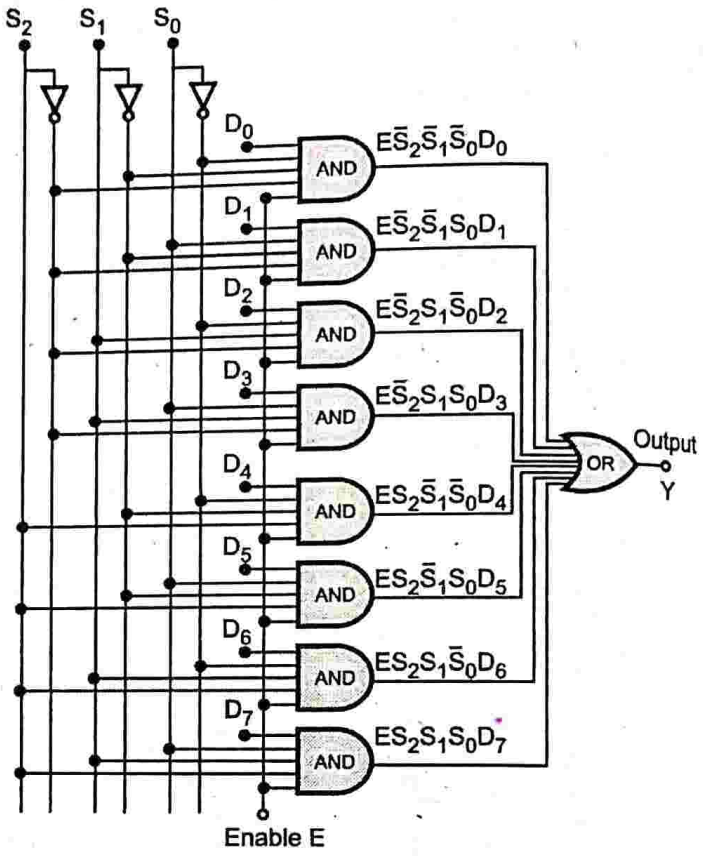
(B) Logic symbol of 8 : 1 MUX

- Fig. 6.7.6 shows the logic symbol of a 8 : 1 multiplexer. It has **eight data inputs** D_0, D_1, D_2, D_3, D_4, D_5, D_6 and D_7, **three select inputs** S_2, S_1, S_0 and **one output Y**.
- The logic levels applied to the select/address inputs (S_2, S_1, S_0) determine which AND gate is enabled.
- The data input of that AND gate is routed to the output.

The expression for Y is,

$$Y = E\bar{S}_2\bar{S}_1\bar{S}_0D_0 + E\bar{S}_2\bar{S}_1S_0D_1 + E\bar{S}_2S_1\bar{S}_0D_2 + E\bar{S}_2S_1S_0D_3 + ES_2\bar{S}_1\bar{S}_0D_4 + ES_2\bar{S}_1S_0D_5 + ES_2S_1\bar{S}_0D_6 + ES_2S_1S_0D_7$$

(C) Logic Diagram of 8 : 1 MUX



(1D36)**Fig. 6.7.7 : Logic diagram of 8 : 1 MUX**

(D) Operation

1. When E = 0, output Y = 0 irrespective to $S_2S_1S_0$ inputs.
2. When E = 1, $S_2S_1S_0$ = 000, the AND gate related to data input D_0 is enabled.

The output of remaining 7 AND gates is zero, as it at least one input to those AND gates is zero.

The output of OR gate is equal to D_0.

D_0 is connected to the output when $S_2S_1S_0$ = 000, E = 1.

Similarly, D_1 is connected to the output when $S_2S_1S_0$ = 001, E = 1.

D_2 is connected to the output when $S_2S_1S_0$ = 010, E = 1.

D_3 is connected to the output when $S_2S_1S_0$ = 011, E = 1.

D_4 is connected to the output when $S_2S_1S_0$ = 100, E = 1.

D_5 is connected to the output when $S_2S_1S_0$ = 101, E = 1.

D_6 is connected to the output when $S_2S_1S_0$ = 110, E = 1.

D_7 is connected to the output when $S_2S_1S_0$ = 111, E = 1.

6.7.6 16 : 1 Multiplexer

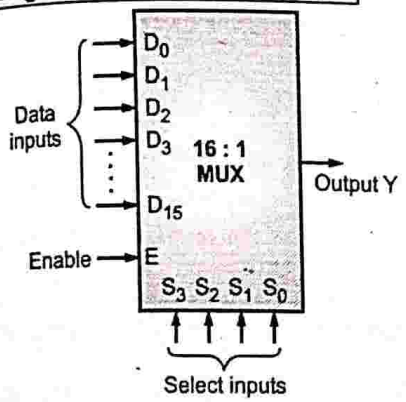
Q. 6.7.12 Explain the working of 16 : 1 multiplexer.

Ans. :

(A) Function

A 16 : 1 multiplexer, selects one of the 16 data inputs and routes it to the output.

(B) Logical symbol of 16 : 1 MUX



(1D37)**Fig. 6.7.8 : Logic symbol of 16 : 1 MUX**

It contains **16 data inputs** ($D_8 - D_{15}$), **four select inputs** S_3, S_2, S_1, S_0 and **one output Y.**

(C) Operation

- When E = 0, output Y = 0 irrespective of $S_3S_2S_1S_0$.
- When E = 1, the logic levels applied to the $S_3S_2S_1S_0$ inputs determine which AND gate is enabled. The data input of that AND gate is routed to the output.

6.7.7 Quadruple 2 to 1 Multiplexer (2 : 1 Multiplexer)

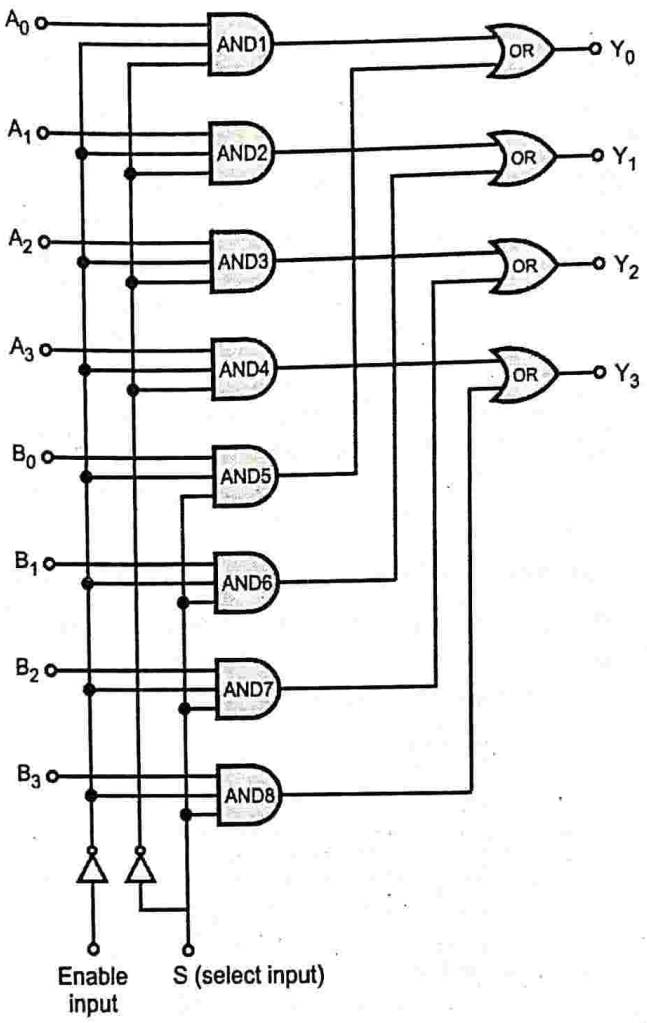
Q. 6.7.13 What is Quadruple multiplexer ? Explain with a neat diagram.

Ans. :

- When four 2 : 1 multiplexers are enclosed with in an integrated circuit the circuit is called as **Quadruple multiplexer.** A Quadruple 2 to 1 line multiplexer contains **four multiplexers.**
- Every multiplexer selects one of the two input lines. Fig. 6.7.9 shows a quadruple 2 to 1 line multiplexer.

Function Table

E	S	Output Y
1	X	0
0	0	Select A
0	1	Select B



(1D38)**Fig. 6.7.9 : 2 to 1 line quadruple multiplexer (Nibble multiplexer)**

Operation

1. When E = 1, output is zero. $Y_0 = 0$, $Y_1 = 0$, $Y_2 = 0$ and $Y_3 = 0$.
2. When E = 0, S = 0, the AND gates 1, 2, 3 and 4 are enabled. The AND gates 5, 6, 7 and 8 are disabled. Thus, the output will be $Y_0 = A_0$, $Y_1 = A_1$, $Y_2 = A_2$ and $Y_3 = A_3$.
3. When E = 0, S = 1, the AND gates 1, 2, 3 and 4 are disabled. The AND gats 5, 6, 7 and 8 are enabled.
   Thus, the outputs of multiplexer are
   $Y_0 = B_0$, $Y_1 = B_1$, $Y_2 = B_2$ and $Y_3 = B_3$

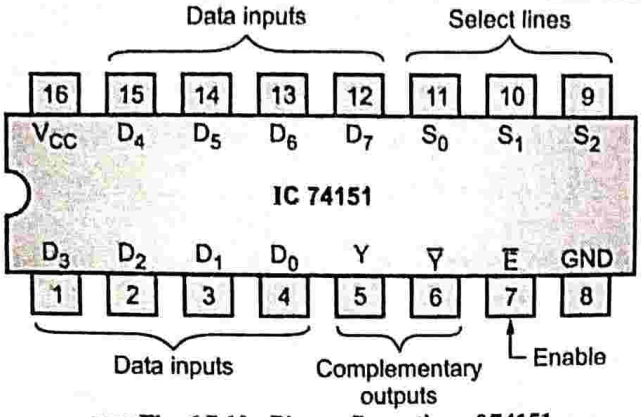
6.7.8 74151 Multiplexer

Q. 6.7.14 Write short on 74151 Multiplexer.

Ans. :

- 74151 is an 8 : 1 multiplexer that provides two complementary outputs Y and $\overline{Y}$. The output Y is same as the input selected, and output $\overline{Y}$ is its complement.

Module 3



(1D39)**Fig. 6.7.10 : Pin configuration of 74151**

- There are 8 data inputs (D_0, D_1, D_2, D_3, D_4, D_5, D_6 and D_7), three select lines (S_0, S_1, S_2) and two outputs (Y and $\bar{Y}$).
- The enable E is an active low pin. Hence, in-order to enable the multiplexer we have to apply a low level signal to the $\bar{E}$ pin.

Table 6.7.1 : Truth table of 74151

Input				Output	
E	S_2	S_1	S_0	Y	$\bar{Y}$
1	X	X	X	0	1
0	0	0	0	D_0	$\bar{D}_0$
0	0	0	1	D_1	$\bar{D}_1$
0	0	1	0	D_2	$\bar{D}_2$
0	0	1	1	D_3	$\bar{D}_3$
0	1	0	0	D_4	$\bar{D}_4$
0	1	0	1	D_5	$\bar{D}_5$
0	1	1	0	D_6	$\bar{D}_6$
0	1	1	1	D_7	$\bar{D}_7$

- Truth table for 74XX151 8 : 1 multiplexer.
- A multiplexer routes the input data from the selected input to the output.
- Hence, the two output columns indicate data D_n and $\bar{D}_n$.

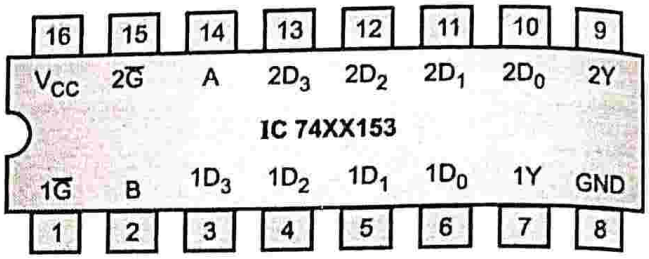
6.7.9 IC 74153 Multiplexer

Q. 6.7.15 Explain with the help of diagram IC 74153 Multiplexer.

☑ **Ans. :** Fig. 6.7.11 shows the logic symbol for IC 74153 dual 4 to 1 multiplexer.

Features

1. It contains two independent 4 : 1 multiplexers.
2. It is fully compatible with TTL circuits.
3. It has enable (strobe) line for cascading.
4. It allows multiplexing from 4 lines to 1 line.
5. It performs parallel to serial conversion.



(1D40)**Fig. 6.7.11 : IC74XX153**

$1\bar{G}$ and $2\bar{G}$ are the active low enable inputs to each multiplexer.

Truth table for 74XX153 dual 4 : 1 multiplexer

Table truth table for 74XX153 dual 4 to 1 multiplexer

Inputs				Outputs	
$1\bar{G}$	$2\bar{G}$	B	A	1Y	2Y
0	0	0	0	$1D_0$	$2D_0$
0	0	0	1	$1D_1$	$2D_1$
0	0	1	0	$1D_2$	$2D_2$
0	0	1	1	$1D_3$	$2D_3$
0	1	0	0	$1D_0$	0
0	1	0	1	$1D_1$	0
0	1	1	0	$1D_2$	0
0	1	1	1	$1D_3$	0
1	0	0	0	0	$2D_0$
1	0	0	1	0	$2D_1$
1	0	1	0	0	$2D_2$
1	0	1	1	0	$2D_3$
1	1	X	X	0	0

6.7.10 Multiplexer Tree

Q. 6.7.16 What do you mean by multiplexer tree ? Explain.

UQ. 6.7.17 What is multiplexer tree ?

MU - Q. 2(b), Dec. 15, 2 Marks

Ans. :

- 16 : 1 multiplexer is the largest available multiplexer IC. Hence, in-order to obtain a multiplexer with more number of inputs (e.g. 64 : 1 multiplexer), we need to cascade several multiplexers.

Definition : The multiplexers with more number of inputs can be obtained by cascading multiplexers with less number of inputs. Such a configuration is called as multiplexer tree.

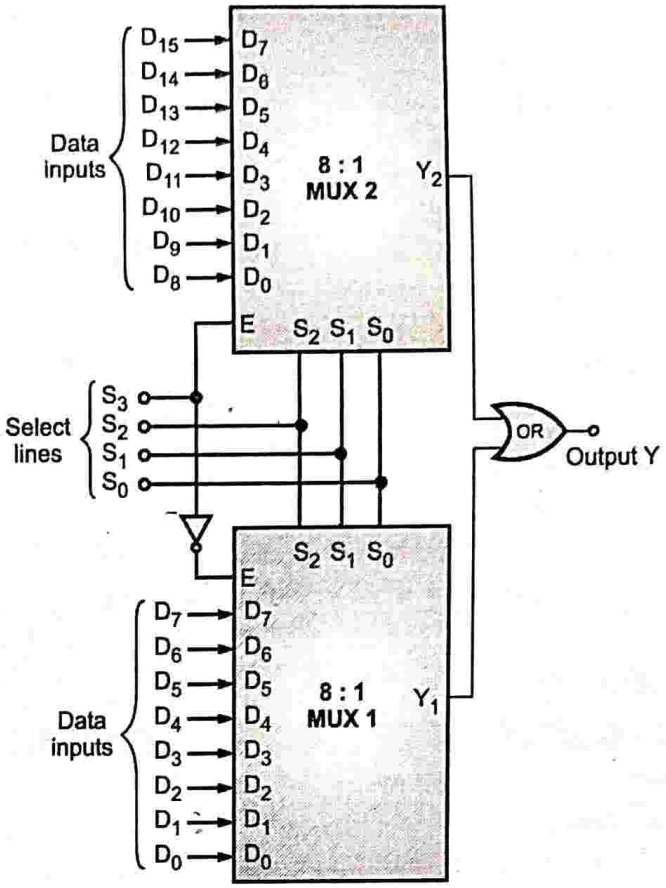
- A number of 2^n : 1 multiplexers arranged in a tree topology to obtain a m : 1 bigger multiplexer is called a **multiplexer tree** where $m > 2^n$.
- Small multiplexers can be cascaded to obtain bigger multiplexer.

Ex. 6.7.1

Design 16 : 1 multiplexer using 8 : 1 multiplexer.

Ans. :

- **Step I :** To obtain a 16 : 1 multiplexer using 8 : 1 multiplexer, we need two 8 : 1 multiplexers.
- **Step II :** The S_2, S_1 and S_0 select lines of two 8 : 1 multiplexers are connected in parallel.
- **Step III :** The most significant select line S_3, is used for enabling one multiplexer at once. When $S_3 = 0$, MUX-1 is enabled and when $S_3 = 1$, MUX-2 is enabled. S_3 is directly connected to the enable input of MUX-2 $\overline{S}_3$ is connected to the enable (E) input of MUX-1.
- **Step IV :** The output of both the multiplexers are logically ORed to obtain final output Y. Fig. Ex. 6.7.1 shows a 16 : 1 multiplexer using two 8 : 1 multiplexers.



(1D41)**Fig. Ex. 6.7.1 : 16 : 1 multiplexer using two 8 : 1 multiplexers**

Ex. 6.7.2

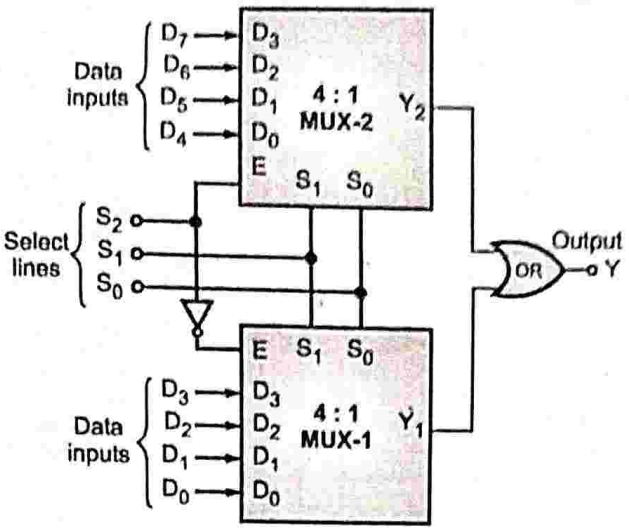
Design 8 : 1 multiplexer using 4 : 1 multiplexer.

Ans. :

- **Step I :** For obtaining 8 : 1 multiplexer, using 4 : 1 multiplexer, we need two 4 : 1 multiplexers.
- **Step II :** The select lines S_1, S_0 are connected in parallel.
- **Step III :** The most significant select line S_2 is used for enabling one multiplexer at once. When $S_2 = 0$ MUX-1 is enabled and when $S_2 = 1$, MUX-2 is enabled. S_2 is directly connected to the enable of MUX-2 and $\overline{S}_2$ is connected to the enable of MUX-1.
- **Step IV :** The output of both the multiplexers are logically ORed to obtain the final output.

Module 3

• NOTES •



(1D42)**Fig. Ex. 6.7.2 : 8 : 1 Multiplexer using two 4 : 1 multiplexers**

UEx. 6.7.3

MU - Q. 4(b), May 18, 5 Marks, Q. 3(a), Dec. 19, 10 Marks

Design 16 : 1 multiplexer using 4 : 1 multiplexers.

☑ **Ans. :**

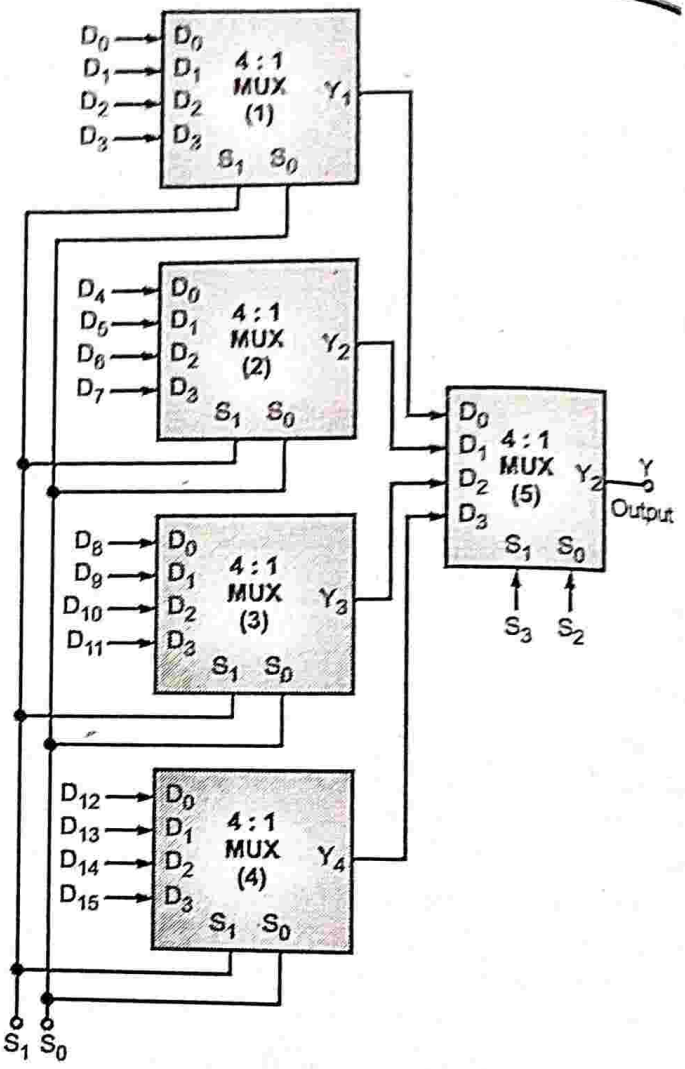
➪ **Step I :** For obtaining a 16 : 1 multiplexer, using 4 : 1 multiplexers, we need four 4 : 1 multiplexers. The four outputs from the multiplexers are again multiplexed to obtain the final output Y. Thus, for implementing a 16 : 1 multiplexer, we need five 4 : 1 multiplexers.

➪ **Step II :** Connect the S_1, S_0 select lines of the 4 multiplexers in parallel.

➪ **Step III :** The most significant select line S_3 and S_2 are connected to MUX–5.

➪ **Step IV :** The outputs from four multiplexers are applied as data inputs to the multiplexer-5.

• NOTES •



(1D43)**Fig. Ex. 6.7.3 : 16 : 1 Multiplexer using 4 : 1 multiplexers**

Ex. 6.7.4

Obtain 32 : 1 multiplexer using 16 : 1 multiplexers.

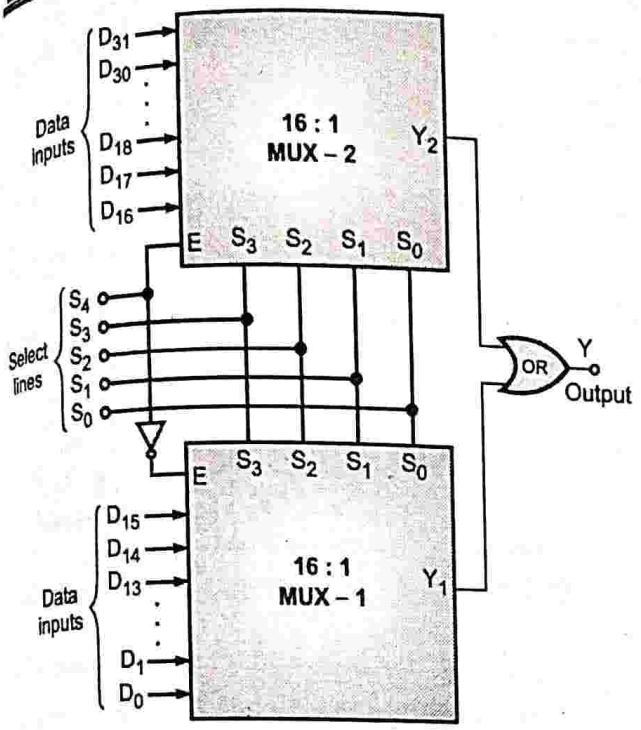
☑ **Ans. :**

➪ **Step I :** For obtaining a 32 : 1 multiplexer, we need two 16 : 1 multiplexers.

➪ **Step II :** The select lines S_3, S_2, S_1, S_0, of both the multiplexers are connected in parallel.

➪ **Step III :** The most significant line S_4 is used for enabling one multiplexer at a time. When $S_4 = 0$, MUX-1 is enabled and when $S_4 = 1$, MUX-2 is enabled. S_4 is directly connected to the enable input of MUX-2 and $\overline{S}_4$ is connected to the enable of MUX-1.

➪ **Step IV :** The outputs of both the multiplexers are logically ORed to obtain the final output.



(1D44)**Fig. Ex. 6.7.4 : Multiplexer using two 16 : 1 multiplexers**

Ex. 6.7.5

Design 4 : 1 multiplexer using 2 : 1 multiplexers.

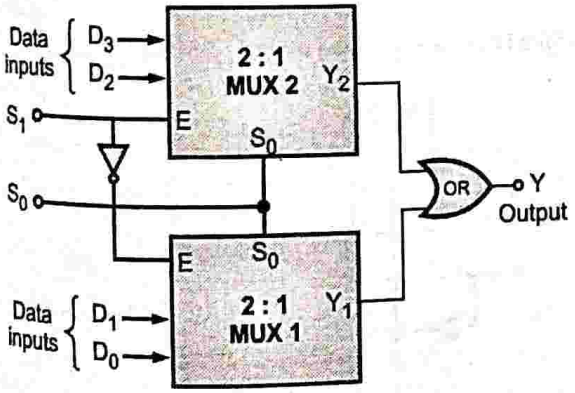
☑ **Ans. :**

➪ **Step I :** For obtaining 4 : 1 multiplexer, we need two 2 : 1 multiplexers.

➪ **Step II :** The S_0 lines of both the multiplexers are connected..

➪ **Step III :** The most significant select line S_1 is used for enabling one multiplexer at a time. When $S_1 = 0$, MUX - 1 is enabled and when $S_1 = 1$, MUX - 2 is enabled.

➪ **Step IV :** The output of both the multiplexers are logically ORed to obtain the final output.



(1D45)**Fig. Ex. 6.7.5 : 4 : 1 Multiplexer using two 2 : 1 multiplexers**

Truth table

Select inputs		MUX Output
S_1	S_0	Y
0	0	D_0 (MUX-1 selected)
0	1	D_1 (MUX-1 selected)
1	0	D_2 (MUX-2 selected)
1	1	D_3 (MUX-2 selected)

6.7.11 Implementing SOP and POS using MUX

- A multiplexer consists of a set of AND gates. The output of the AND gates are directly connected to the OR gate.
- This feature allows the implementation of Boolean logic functions in SOP form using a multiplexer.

Ex. 6.7.6

Implement the following function using 8 : 1 multiplexer :

$F(a, b, c) = \Sigma m\ (3, 5, 6)$

☑ **Ans. :**

➪ **Step I :** Select the multiplexer.

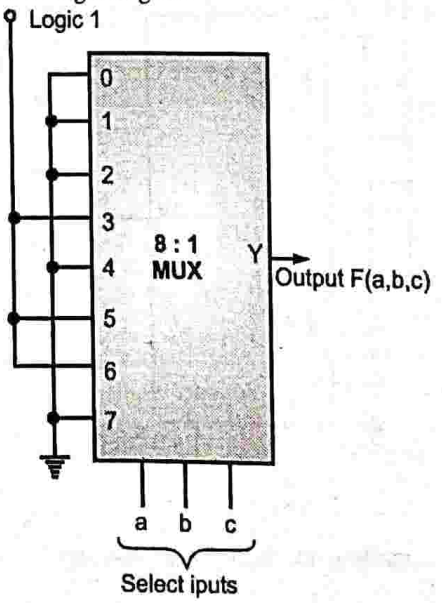
$F(a, b, c) = \Sigma m\ (3, 5, 6)$

The function has three variables. Hence, we need $2^3 = 8 : 1$ **multiplexer.**

➪ **Step II :** Connect the data inputs (3, 5, 6) to logic 1 and remaining data inputs to logic 0.

➪ **Step III :** Connect the input variables to the select lines of MUX.

➪ **Step IV :** Logic diagram.



(1D46)**Fig. Ex. 6.7.6 : Logic diagram**

Module 3

Ex. 6.7.7

Implement the Boolean function given by truth table using multiplexer.

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

☑ **Ans. :**

➪ **Step I :** Select the multiplexer.

From the truth table we can see that the function has three variables.

∴ we need $2^3 = 8 : 1$ **multiplexer** for implementation.

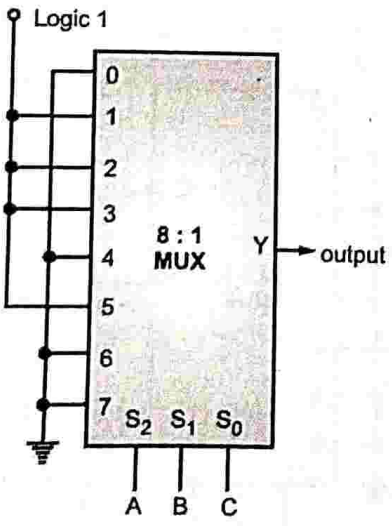
➪ **Step II :** Find the minterm expression

$Y = \sum m\ (1, 2, 3, 5)$ from the truth table.

➪ **Step III :** Connect inputs 1, 2, 3 and 5 to **logic 1**. Connect the remaining inputs to **logic 0**.

➪ **Step IV :** Connect the input variables to the select lines of MUX.

➪ **Step V :** Logic diagram.



(1D47)**Fig. Ex. 6.7.7 : Logic diagram**

Ex. 6.7.8

Implement the Boolean function using 4 : 1 multiplexer

$F(A, B, C) = \sum m\ (1, 3, 5, 6)$

☑ **Ans. :**

➪ *Step I : To prepare implementation table.*

Implementation Table

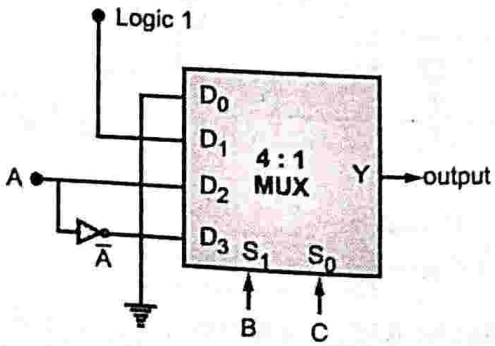
	D_0	D_1	D_2	D_3	
$\bar{A}$	0	(1)	2	(3)	Row 1
A	4	(5)	(6)	7	Row 2
	0	1	A	$\bar{A}$	

Most significant variable

(1D42)*Fig. Ex. 6.7.8*

- In the implementation table we have listed the minterms in two rows.
- In row 1, the minterms are listed where A is complemented and in row 2 the minterms are listed with A uncomplemented.
- The minterms in the given expression are circled and then each column is analyzed as follows :
  1. If two minterms, in a column are not circled then the minterms are connected to logic 0.
  2. If two minterms, in a column are circled then the minterms are connected to logic 1.
  3. If the minterms, in the first row is circled and minterms in the second row is not circled, connect the minterm to A.
  4. If the minterm, in the first row is not circled and minterms in the second row is circled, connect the minterm to A.

➪ **Step II :** Implementation using 4 : 1 MUX.



(1D49)**Fig. Ex. 6.7.8(a) : Logic diagram**

Ex. 6.7.9 **MU - Q. 3(a), May 19, 5 Marks**

Implement the expression using a 8 : 1 multiplexer

$f(A, B, C, D) = \Sigma m\ (0, 2, 3, 6, 8, 9, 12, 14)$

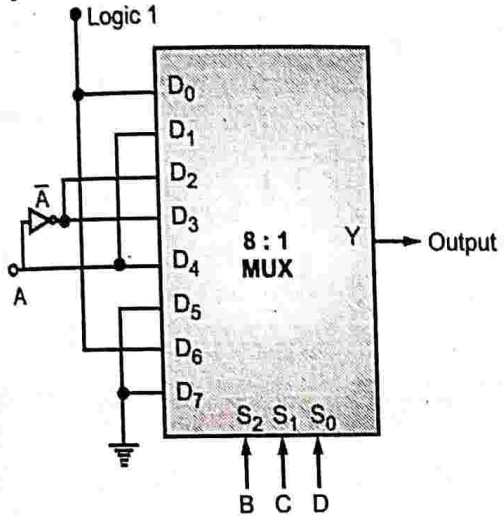
Ans. :

Step I : To prepare implementation table.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	(0)	1	(2)	(3)	4	5	(6)	7
A	(8)	(9)	10	11	(12)	13	(14)	15
	1	A	$\bar{A}$	$\bar{A}$	A	0	1	0

(1D50)**Fig. Ex. 6.7.9**

Step II : Implementation using logic diagram.



(1D51)**Fig. Ex. 6.7.9(a) : Logic diagram**

Ex. 6.7.10

Implement full adder using 8 : 1 multiplexer and draw the diagram.

Ans. :

Step I : Truth table of full adder.

Table Ex. 6.7.10 : Truth table of full adder

Inputs			Outputs	
A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = \Sigma m(1, 2, 4, 7)$

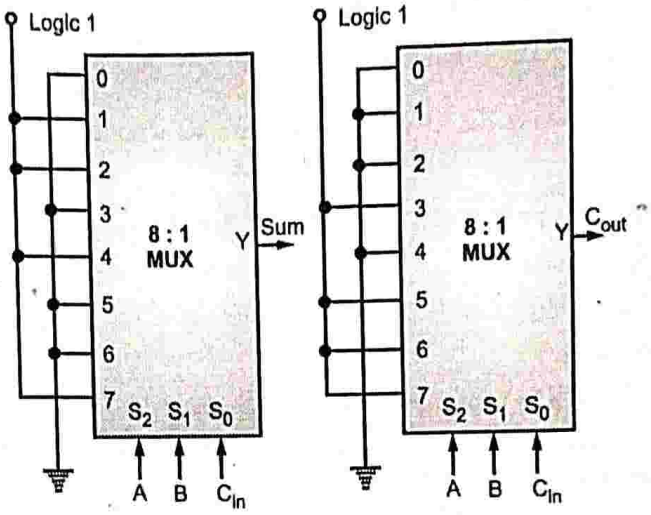
$C_{out} = \Sigma m(3, 5, 6, 7)$

Step II : For sum : Connect the inputs (1, 2, 4, 7) to logic 1 and remaining inputs to logic 0

For carry : Connect the inputs (3, 5, 6, 7) to logic 1 and remaining inputs to 0.

Step III : Connect the input variables to the select lines of MUX.

Step IV : Implementation



(a) Logic diagram for Carry **(b) Logic diagram for Sum**

(1D52)**Fig. Ex. 6.7.10 : Implementation of full adder using 8 : 1 MUX**

Ex. 6.7.11

Design full adder using 4 : 1 multiplexer.

Ans. :

Refer Table Ex. 6.7.10.

Step I : To prepare the implementation table for sum and carry.

Implementation table for Sum

	D_0	D_1	D_2	D_3
$\bar{A}$	0	(1)	(2)	3
A	(4)	5	6	(7)
	A	$\bar{A}$	$\bar{A}$	A

(1D53)(a)

Implementation table for Carry

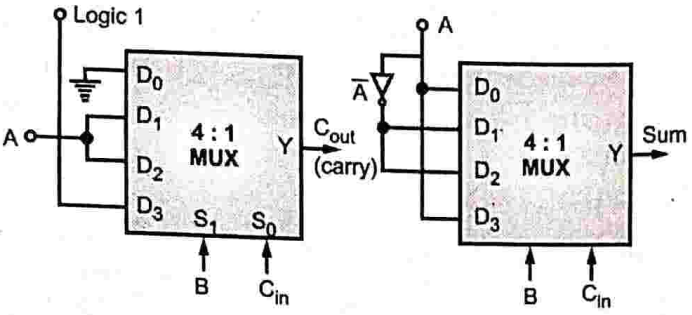
	D_0	D_1	D_2	D_3
$\bar{A}$	0	1	2	(3)
A	4	(5)	(6)	(7)
	0	A	A	1

(1D54)(b)

Fig. Ex. 6.7.11

Module 3

➪ **Step II :** Implementation.



(1D55)**Fig. Ex. 6.7.11(c) : Implementation of full adder using 4 : 1 MUX**

UEx. 6.7.12 **MU - Q. 4(a), May 14, 10 Marks**

Implement the following 8 : 1 MUX.

$F(A, B, C, D) = \pi m (1, 3, 5, 9, 11, 12, 13)$

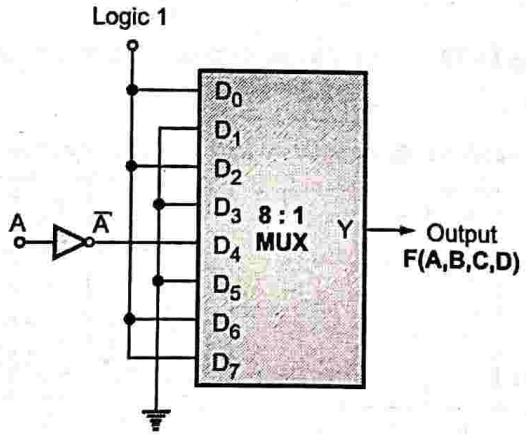
☑ **Ans. :**

➪ **Step I :** To prepare the implementation table

In the given function maxterms are specified. Hence, we will circle the terms that are not present in the function.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	⓪	1	②	3	④	5	⑥	⑦
A	⑧	9	⑩	11	12	13	⑭	⑮
	1	0	1	0	$\bar{A}$	0	1	1

➪ **Step II :** Implementation using 8 : 1 MUX



(2E2)**Fig. Ex. 6.7.12**

UEx. 6.7.13 **MU - Q. 4(a), Dec. 14, 5 Marks, Q. 3(b), May 17, 5 Marks**

Implement the following using 8 : 1 MUX.

$F(A, B, C, D) = \Sigma m (0, 1, 3, 5, 7, 10, 11, 13, 14, 15)$

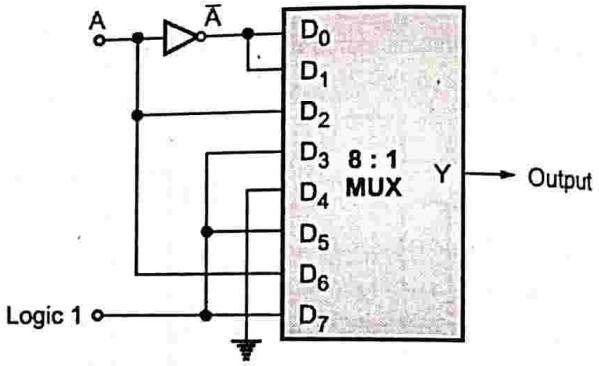
☑ **Ans. :**

➪ **Step I :** To prepare the implementation table

In the given function maxterms are specified. Hence, we will circle the terms that are not present in the function.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	⓪	①	2	③	4	⑤	6	⑦
A	8	9	⑩	⑪	12	⑬	⑭	⑮
	$\bar{A}$	$\bar{A}$	A	1	0	1	A	1

➪ **Step II :** Implementation using 8 : 1 MUX



(2E3)**Fig. Ex. 6.7.13**

UEx. 6.7.14 **MU - Q. 2(b), Dec. 16, 5 Marks**

Implement the following function using 4 : 1 multiplexer and few gates.

$F(A, B, C, D) = \Sigma m (0, 1, 2, 3, 6, 7, 9, 10, 13, 15)$

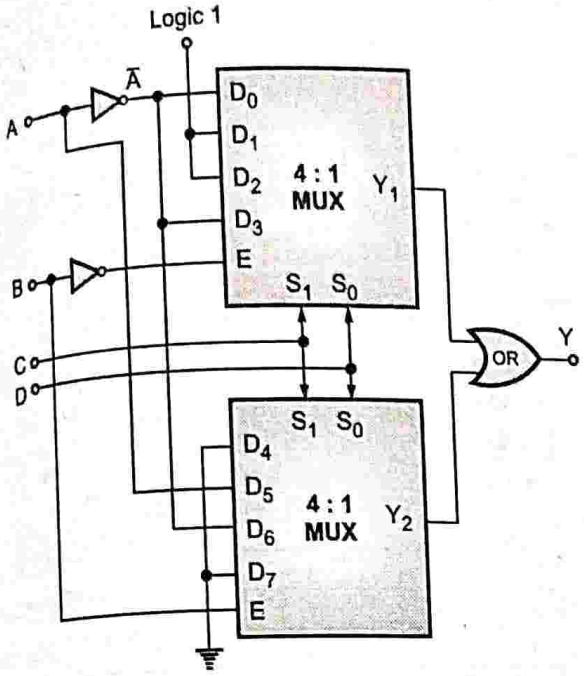
☑ **Ans. :**

➪ **Step I :** To prepare implementation table

The function has four variables. To implement this function we need 8 : 1 multiplexer i.e. two 4 : 1 multiplexers.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	⓪	①	②	③	4	5	⑥	⑦
A	8	⑨	⑩	11	12	⑬	14	⑮
	$\bar{A}$	1	1	$\bar{A}$	0	A	$\bar{A}$	0

Step II : Implementation using 4 : 1 MUX



(2E4)**Fig. Ex. 6.7.14 : Implementation using two 4 : 1 multiplexer**

UEx. 6.7.15 **MU - Q. 2(b), May 15, 5 Marks**

Implement the following expression using single 4 : 1 MUX.

$F(A, B, C, D) = \Sigma m\ (0, 1, 2, 4, 6, 9, 12, 14)$

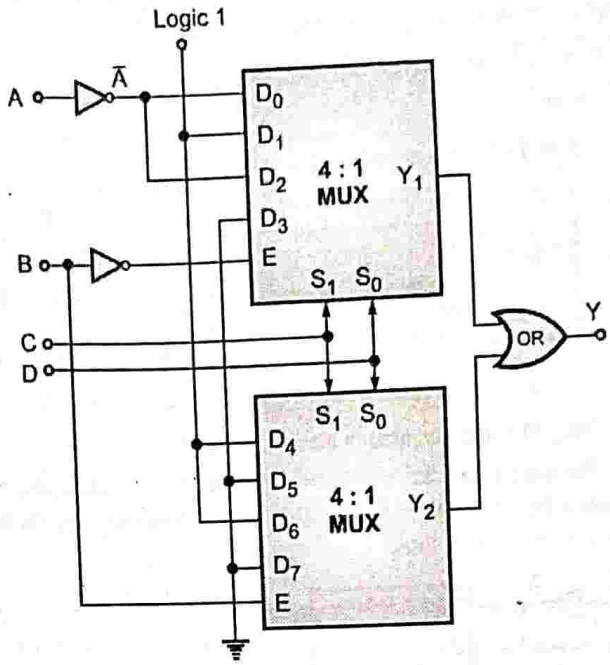
Ans. :

Step I : To prepare implementation table

The function has four variables. To implement this function we need one 8 : 1 multiplexer i.e. two 4 : 1 multiplexers.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	⓪	①	②	3	④	5	⑥	7
A	8	⑨	10	11	⑫	13	⑭	15
	$\bar{A}$	1	$\bar{A}$	0	1	0	1	0

Step II : Implementation using 4 : 1 multiplexers



(2E5)**Fig. Ex. 6.7.15 : Implementation using 4 : 1 multiplexers**

UEx. 6.7.16 **MU - Q. 3(b), May 16, 5 Marks**

Implement the following using only one 8 : 1 MUX and few gates.

$F(A, B, C, D) = \Sigma m\ (0, 3, 5, 7, 9, 13, 15)$

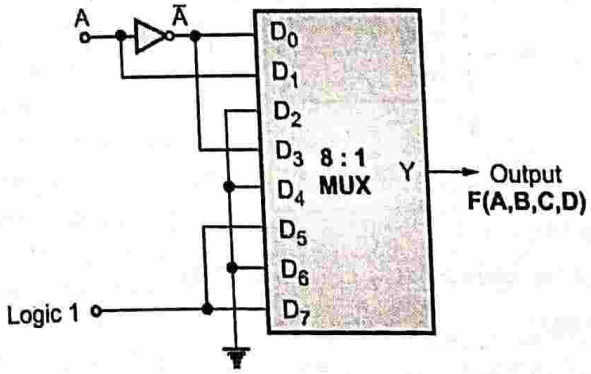
Ans. :

Step I : To prepare implementation table

The function has four variables. To implement this function we need 8 : 1 multiplexer i.e. two 8 : 1 multiplexers.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	⓪	1	2	③	4	⑤	6	⑦
A	8	⑨	10	11	12	⑬	14	⑮
	$\bar{A}$	A	0	$\bar{A}$	0	1	0	1

Step II : Implementation using 8 : 1 multiplexer



(2E6)**Fig. Ex. 6.7.16 : Implementation using 8 : 1 multiplexer**

Module 3

UEx. 6.7.17 **MU - Q. 3(b), Dec. 13, 10 Marks, Q. 4(a), Dec. 18, 10 Marks**

Implement the following logic function using all 4 : 1 multiplexers with select inputs as 'B', 'C', 'D', 'E' only.

$F(A, B, C, D, E) = \sum m\ (0, 1, 2, 3, 6, 8, 9, 10, 13, 15, 17, 20, 24, 30)$

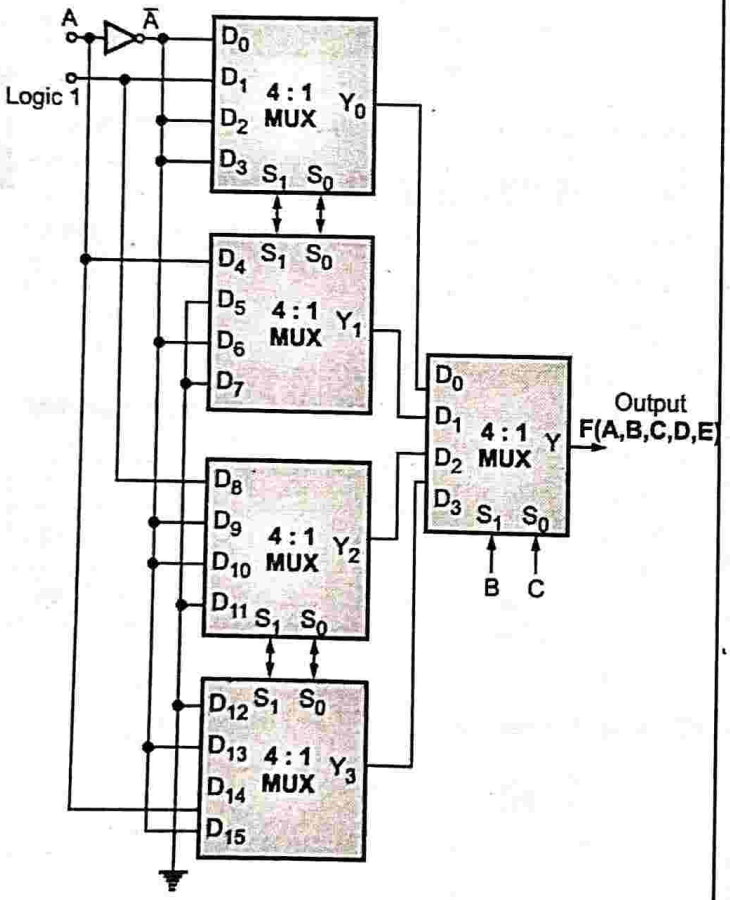
☑ **Ans. :**

➪ **Step I :** Implementation table

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}
$\bar{A}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	$\bar{A}$	1	$\bar{A}$	$\bar{A}$	A	0	$\bar{A}$	0	1	$\bar{A}$	$\bar{A}$	0	0	$\bar{A}$	A	A

➪ **Step II :** Implementation using 4 : 1 MUX

We have to use 4 : MUX. We will need 5, 4 : 1 multiplexers to form a 16 : 1 MUX. A 16 : 1 MUX has 4 select lines. Inputs B, C, D, E will be select lines.



(2E1)Fig. Ex. 6.7.17

Ex. 6.7.18

Design full subtractor using multiplexer IC74151.

☑ **Ans. :**

➪ **Step I :** Truth table of full subtractor

Table Ex. 6.7.18 : Truth table of full subtractor

Inputs			Outputs	
A	B	B_{in}	D (Difference)	B_{out} (Borrow)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

From the truth table.

$$D = \sum m\ (1, 2, 4, 7)$$

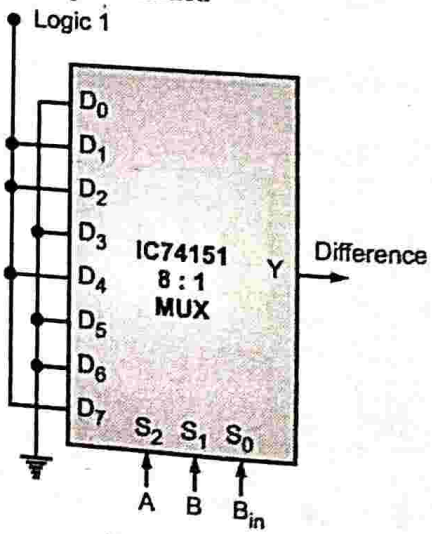
$$B_{out} = \sum m\ (1, 2, 3, 7)$$

➪ **Step II :** For difference cannot the inputs (1, 2, 4, 7) to logic 1 and remaining inputs to logic 0.

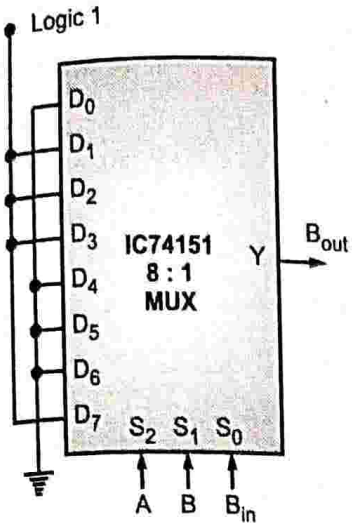
For borrow cannot the (1, 2, 3, 7) inputs to logic 1 and remaining inputs to logic 0.

➪ **Step III :** Connect the input variables to the select lines to MUX.

➪ **Step IV :** Implementation



(1D#1)Fig. Ex. 6.7.18



(1D#2)**Fig. Ex. 6.7.18(a)**

Ex. 6.7.19

Design the following logic expression using single 8 : 1 multiplexer. $F(A, B, C, D) = \sum m\ (0, 2, 3, 6, 8, 9, 14) + d\ (12, 13)$

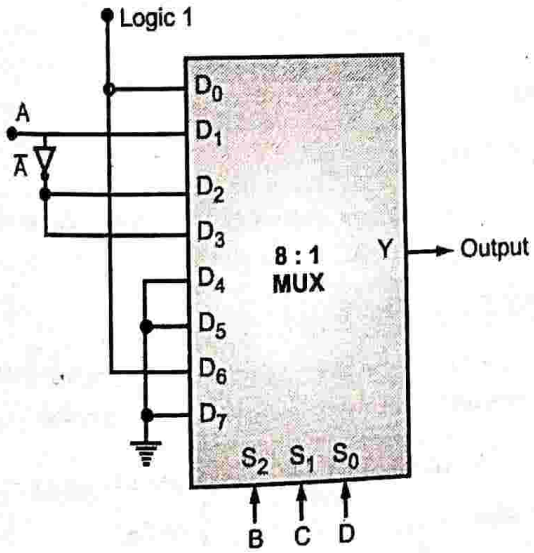
☑ **Ans. :**

➪ **Step I :** Implementation table.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	⓪	1	②	③	4	5	⑥	7
A	⑧	⑨	10	11	12	13	⑭	15
	1	A	$\bar{A}$	$\bar{A}$	0	0	1	0

We treat the don't care conditions 12, 13 to be 0's

➪ **Step II :** Implementation.



(1D59)**Fig. Ex. 6.7.19 : Implementation**

Ex. 6.7.20

Implement the following using 8 : 1 multiplexers.

$F(A, B, C, D) = \prod M\ (0, 3, 5, 7, 12, 15) + d\ (2, 9)$

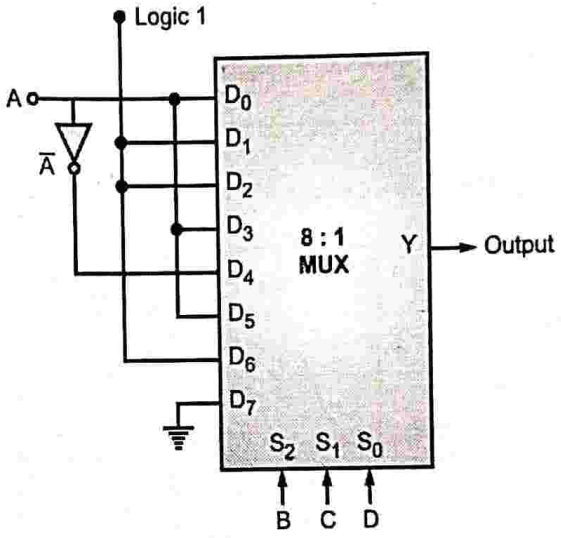
☑ **Ans. :**

➪ **Step I :** To prepare implementation table. $f(A, B, C, D) = \prod M\ (0, 3, 5, 7, 12, 15) + d\ (2, 9)$. In the given function maxterms are specified. Hence, we will circle the terms that are not present in the function.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	0	①	②	3	④	5	⑥	7
A	⑧	⑨	⑩	⑪	12	⑬	⑭	15
	A	1	1	A	$\bar{A}$	A	1	0

The don't care conditions are considered to be 1's.

➪ **Step II :** Implementation using 8 : 1 MUX



(1D61)**Fig. Ex. 6.7.20 : Logic diagram**

Module 3

⏭ 6.8 DEMULTIPLEXER

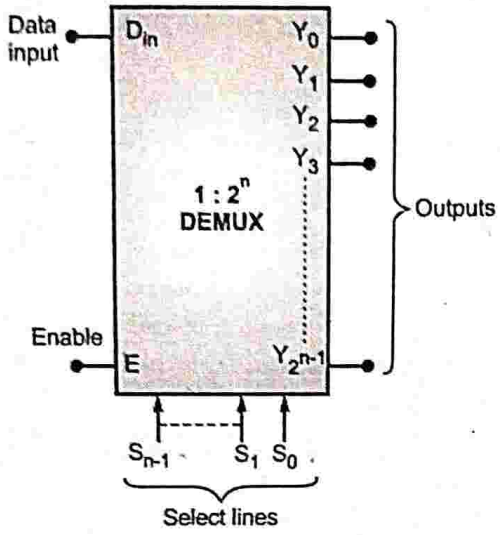
Q. 6.8.1 What is a demultiplexer ? Why is it called data distributor ?

Q. 6.8.2 Write a short note on demultiplexer.

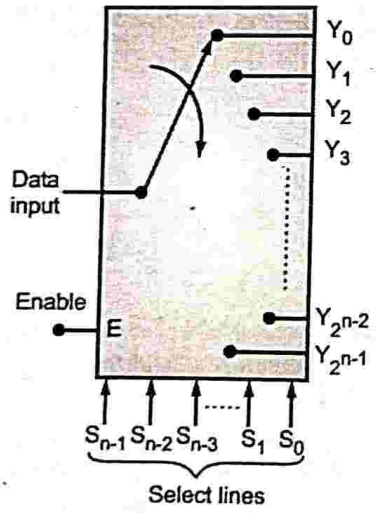
☑ **Ans. :**

- A demultiplexer is a combinational logic circuit that receives information or data on a single line and transmits it.
- This information / data is routed to one of the 2^n output lines. The output line to be selected depends on the combination selected by the select lines.
- A demultiplexer is a combinational logic circuit that depending on the status of the select inputs, routes the data input to one of the several data outputs.

- As, a demultiplexer has single input and it routes over multiple data output lines, it is called as a **data distributor.**
- Fig. 6.8.1(a) shows the functional block diagram of $1 : 2^n$ demultiplexer.



(1D108)**(a) Block diagram of a Demultiplexer**



(1D109)**(b) Equivalent circuit**

Fig. 6.8.1

- It has one input data line D_{in}, 2^n output lines, n select lines and one enable input. The select lines are also called as select inputs.

Q. 6.8.3 *Explain the advantages of demultiplexers.*

☑ **Ans. :**

The **advantages** of demultiplexer are as follows :

1. The logic design need not be simplified.
2. In multiple output circuits the IC package count is decreased.
3. The system reliability is improved.
4. It can convert the serial data into parallel.

Q. 6.8.4 *Explain the applications of demultiplexers.*

☑ **Ans. :**

Applications of demultiplexers are as follows :

1. Communication systems
2. Arithmetic logic unit.
3. Serial to parallel converter.
4. They are used in time demultiplexing systems.
5. They are used in computer memory.
6. They are used in data acquisition systems.

Q. 6.8.5 *List the types of demultiplexers. List demultiplexer ICs.*

☑ **Ans. :**

The types of demultiplexers are

(a) 1 : 2 demultiplexer (c) 1: 4 demultiplexer

(b) 1 : 8 demultiplexer (d) 1 : 16 demultiplexer

Demultiplexer ICs

IC	Function
IC74154	1 : 16 Demultiplexer
IC74155	Dual 1 : 4 Demultiplxer

6.8.1 1 : 2 Demultiplexer

Q. 6.8.6 *Explain the working of a 1 : 2 demultiplexer.*

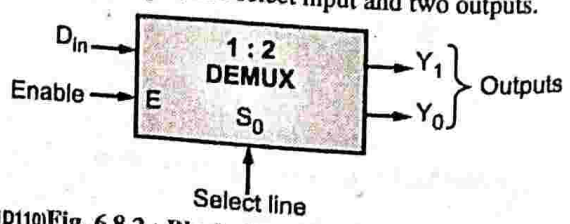
☑ **Ans. :**

(A) Function

A 1 : 2 demultiplexer routes the input to one of its two output depending on the status of the select line.

(B) Block Diagram

Fig. 6.8.2 shows the block diagram of a 1 : 2 demultiplexer. It has one data input, one select input and two outputs.



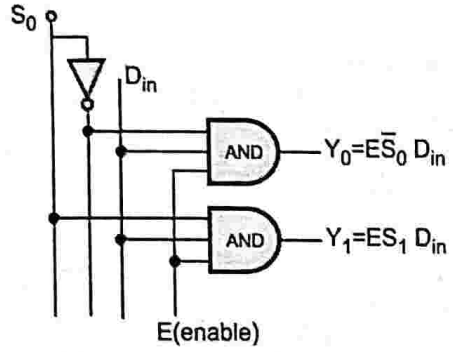
(1D110)**Fig. 6.8.2 : Block diagram of 1 : 2 Demultiplexer**

(C) Function Table

Table 6.8.1 : Function table of 1 : 2 DEMUX

Enable	Select input	Output		
	S_0	Y_0	Y_1	
0	X	0	0	$\leftarrow Y_0 = E\bar{S}_0 D_{in}$
1	0	D_{in}	0	$\leftarrow Y_1 = ES_1 D_{in}$
1	1	0	D_{in}	

(D) Logic Realization of 1 : 2 Demultiplexer



(1D111)Fig. 6.8.3

6.8.2 1 : 4 De-multiplexer

Q. 6.8.7 Draw the logical circuit diagram and describe the working of a 1 : 4 demultiplexer.

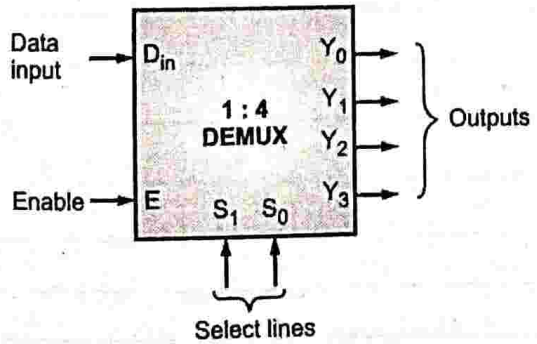
Ans. :

(A) Function

A 1 : 4 demultiplxer depending on the status of its select lines routes the input, to one of its four outputs.

(B) Block Diagram

Fig. 6.8.4 shows the block diagram of a 1 : 4 DEMUX. It has one data input, two select lines, four output lines and one enable signal.



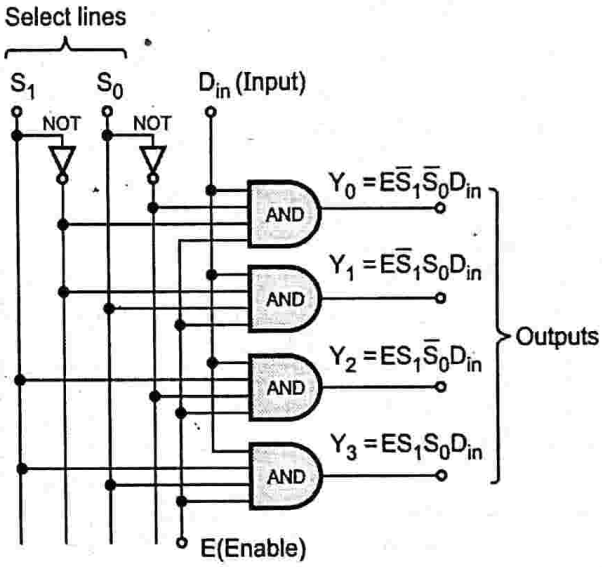
(1D112)Fig. 6.8.4 : 1 : 4 Demultiplexer

(C) Function table

Table 6.8.2 : Function table of 1 : 4 DEMUX

Enable	Select lines		Output				
	S_1	S_0	Y_0	Y_1	Y_2	Y_3	
0	X	X	0	0	0	0	
1	0	0	D_{in}	0	0	0	$\leftarrow Y_0 = E\bar{S}_0\bar{S}_1 D_{in}$
1	0	1	0	D_{in}	0	0	$\leftarrow Y_1 = ES_0\bar{S}_1 D_{in}$
1	1	0	0	0	D_{in}	0	$\leftarrow Y_2 = E\bar{S}_0 S_1 D_{in}$
1	1	1	0	0	0	D_{in}	$\leftarrow Y_3 = ES_1 S_0 D_{in}$

(D) Logic Realization of 1 : 4 DEMUX



(1D113)Fig. 6.8.5 : Logic Realization of 1 : 4 Demultiplexer

6.8.3 1 : 8 Demultiplexer

Q. 6.8.8 Draw the logical circuit diagram and explain working of a 1 : 8 demultiplexer.

Ans. :

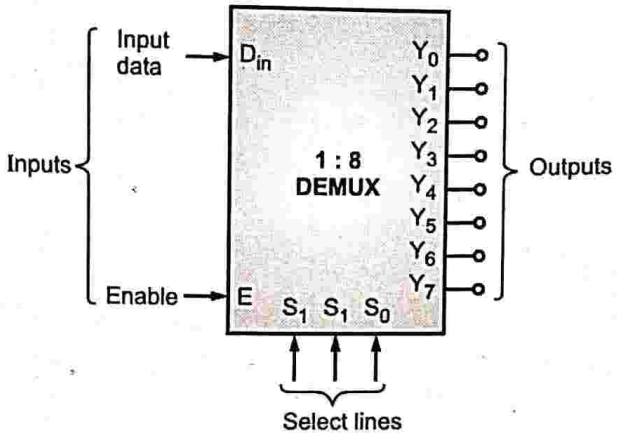
(A) Function

A, 1 : 8 demultiplexer depending the status of the select lines routes, it input to one of the eight outputs.

(B) Block Diagram

Fig. 6.8.6 shows the block diagram of a demultiplexer. It has one input, one enable input, three select lines and eight outputs.

Module 3



(1D114)**Fig. 6.8.6 : Block diagram of 1 : 8 Demux**

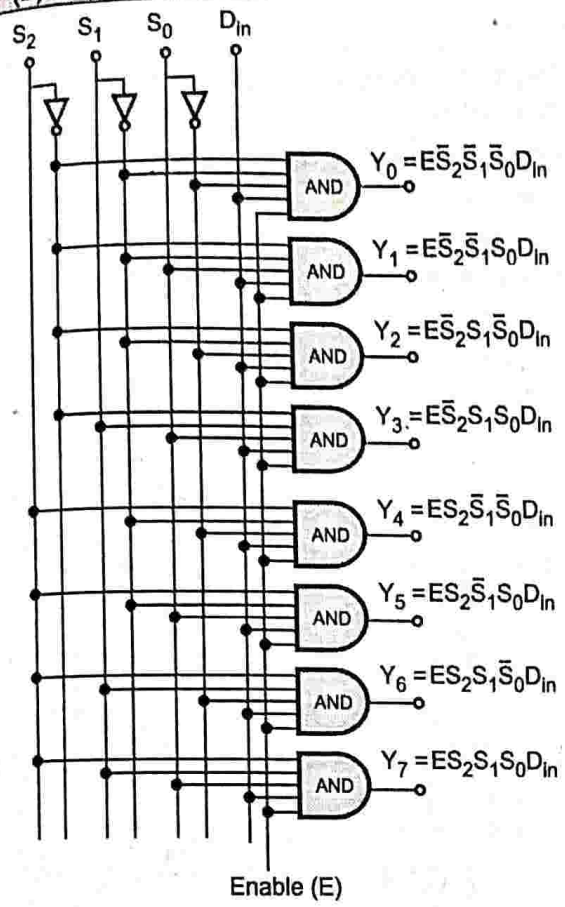
(C) Function Table

Table 6.8.3 : Function table of 1 : 8 DEMUX

Enable	Select lines			Output								
	S_2	S_1	S_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	
0	X	X	X	0	0	0	0	0	0	0	0	
1	0	0	0	D_{in}	0	0	0	0	0	0	0	$\leftarrow Y_0 = E\,\bar{S}_2\,\bar{S}_1\,\bar{S}_0\,D_{in}$
1	0	0	1	0	D_{in}	0	0	0	0	0	0	$\leftarrow Y_1 = E\,\bar{S}_2\,\bar{S}_1\,S_0\,D_{in}$
1	0	1	0	0	0	D_{in}	0	0	0	0	0	$\leftarrow Y_2 = E\,\bar{S}_2\,S_1\,\bar{S}_0\,D_{in}$
1	0	1	1	0	0	0	D_{in}	0	0	0	0	$\leftarrow Y_3 = E\,\bar{S}_2\,S_1\,S_0\,D_{in}$
1	1	0	0	0	0	0	0	D_{in}	0	0	0	$\leftarrow Y_4 = E\,S_2\,\bar{S}_1\,\bar{S}_0\,D_{in}$
1	1	0	1	0	0	0	0	0	D_{in}	0	0	$\leftarrow Y_5 = E\,S_2\,\bar{S}_1\,S_0\,D_{in}$
1	1	1	0	0	0	0	0	0	0	D_{in}	0	$\leftarrow Y_6 = E\,S_2\,S_1\,\bar{S}_0\,D_{in}$
1	1	1	1	0	0	0	0	0	0	0	D_{in}	$\leftarrow Y_7 = E\,S_2\,S_1\,S_0\,D_{in}$

• NOTES •

(D) Logic Diagram



(1D115)**Fig. 6.8.7 : Logic diagram of 1 : 8 DEMUX**

6.8.4 1 : 16 DeMultiplexer

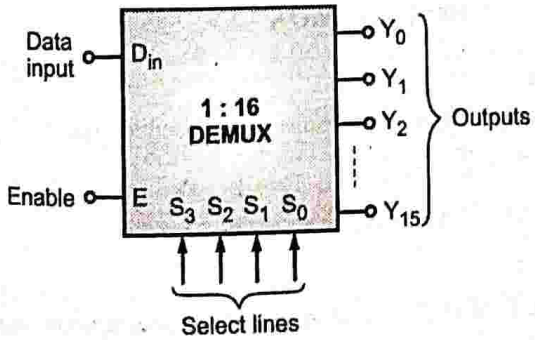
Q. 6.8.9 Write brief about 1 : 16 DeMultiplexer.

Ans. :

A) Function

A 1 : 16 demultiplexer routes the input to one of 16 outputs depending on the status of the select lines.

B) Logical symbol of 1 : 16 DEMUX



(1D116)**Fig. 6.8.8 : 1 : 16 demultiplexer**

6.8.5 Demultiplexer Tree

Q. 6.8.10 What do you understand by a demultiplexer tree ?

Ans. :

- 1 : 16 demultiplexers are the largest available demultiplexer ICs.
- Hence, in order to obtain a demultiplexer with more number of outputs, we need to cascade several demultiplexers.
- The demultiplexers with more number of outputs can be obtained by cascading demultiplexers with less number of outputs. Such a configuration is called as a **demultiplexer tree.**
- A number of 1 : 2^n demultiplexers are arranged in a tree topology to obtain a 1 : m bigger demultiplxer is called a **demultiplexer tree** where $m > 2^n$.

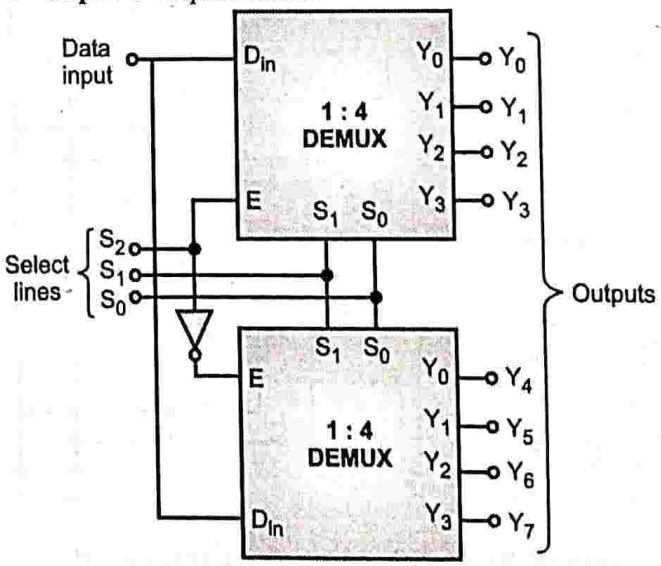
  Thus, small demultiplexers can be cascade to obtain a bigger demultiplexer.

Ex. 6.8.1

Design 1:8 de-multiplexer using 1:4 demultiplexer.

Ans. :

- **Step I :** To obtain a 1 : 8 demultiplexer using 1 : 4 demultiplexer, we need two 1 : 4 demultiplexers.
- **Step II :** Connect the D_{in} signal of both demultiplexers.
- **Step III :** Connect the most significant line S_2 such that when $S_2 = 0$, DeMUX-1 is enabled and when $S_2 = 1$, DeMUX 2 is enabled.
- **Step IV :** Implementation



(1D117)**Fig. Ex. 6.8.1 : 1 : 8 demultiplexer using 1 : 4 demultiplexer**

Module 3

UEx. 6.8.2 **MU - Q. 4(b), May 17, 5 Marks**

Design 1 : 16 Demultiplexer using 1 : 4 demultiplexer.

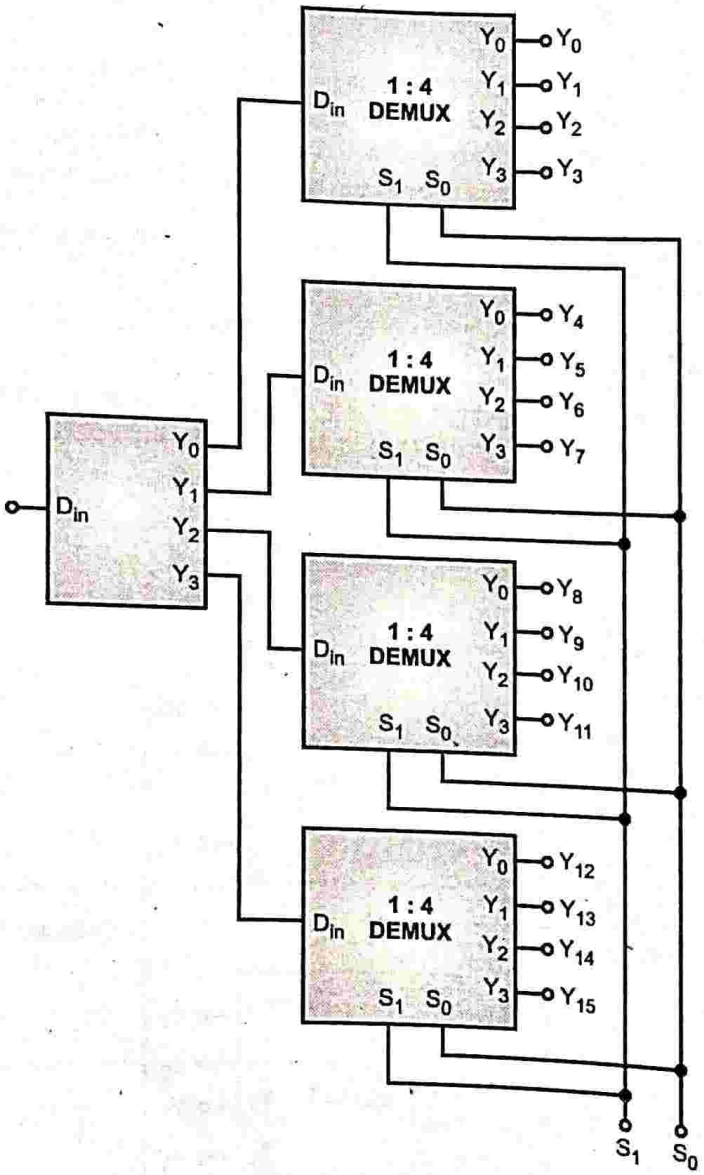
Ans. :

Step I : To obtain a 1 : 16 demultiplexer using 1 : 4 demultiplexers, we need 4, 1 : 4 demultiplexers.

Step II : Connect the S_1 S_0 lines of four demultiplexers.

Step III : Connect one more demultiplexer so that its four outputs are routed to the data inputs of the four demultiplexer. Connect the select lines S_3 S_2.

Step IV : Implementation.



(1D118)**Fig. Ex. 6.8.2 : 1 : 16 demultiplexer using 1 : 4 demultiplexers**

Ex. 6.8.3

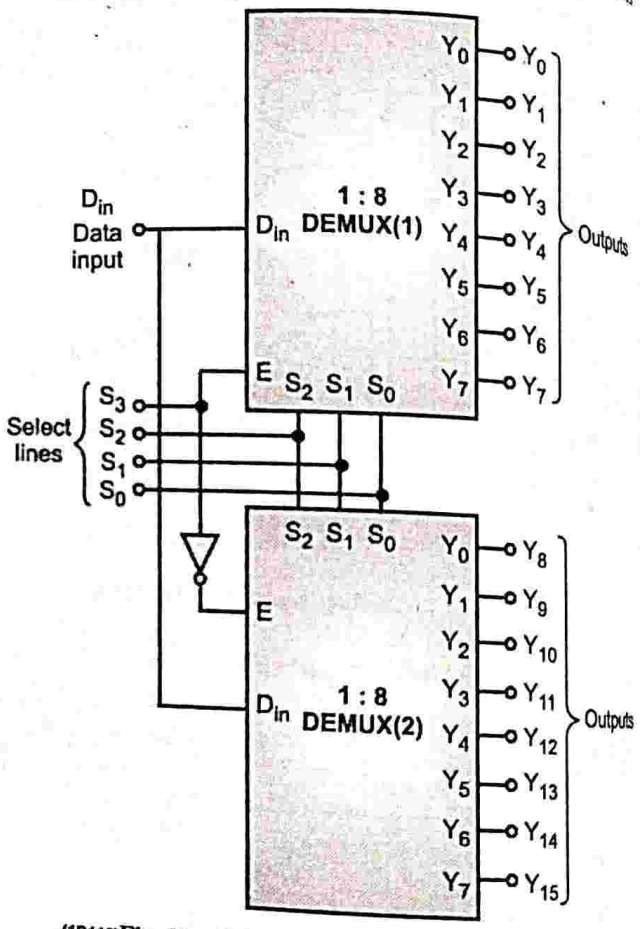
Implement 1 : 16 demultiplexer using 1 : 8 demultiplexer.

Ans. :

Step I : To obtain a 1:16 demultiplexer using 1 : 8 demultiplexer, we need two 1 : 8 demultiplexers.

Step II : Connect the D_{in} and S_2 S_1 S_0 lines of both the multiplexers.

Step III : Connect the most significant line S_3 such that when $S_3 = 0$, DEMUX-1 is enabled and when $S_3 = 1$, DEMUX-2 is enabled.



(1D119)**Fig. Ex. 6.8.3 : 1 : 16 demultiplexer using 1 : 8 demultiplexer**

6.8.6 Implementation of SOP and POS using DEMUX

Ex. 6.8.4

Implement the following functions using demultiplexer.

$Y_1 (A, B, C) = \sum m (0, 3, 7)$

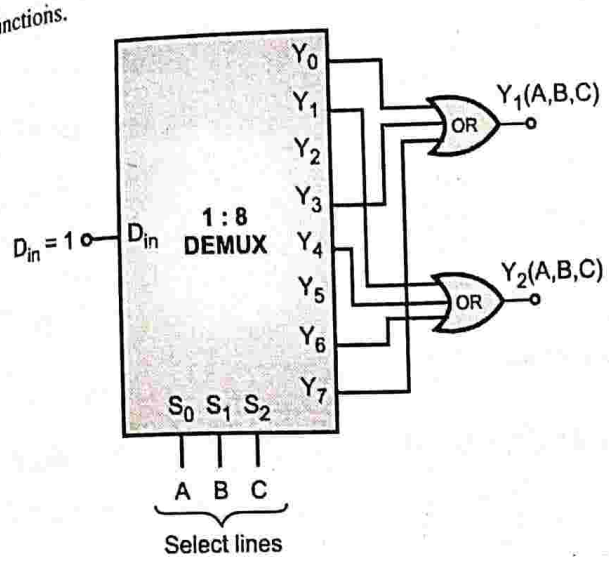
$Y_2 (A, B, C) = \sum m (1, 4, 6)$

Ans. :

Step I : With $D_{in} = 1$, at the output of demultiplexer we get minterms.

Step II : In order to implement the Boolean functions we need to logically OR these minterms.

Fig. Ex. 6.8.4 shows the implementation of Y_1 and Y_2 functions.



(1D120)**Fig. Ex. 6.8.4 : Implementation of Y_1 and Y_2 functions**

UEx. 6.8.5 **MU - Q. 2(b), May 18, 10 Marks**

Implement full adder using demultiplexers.

Ans. :

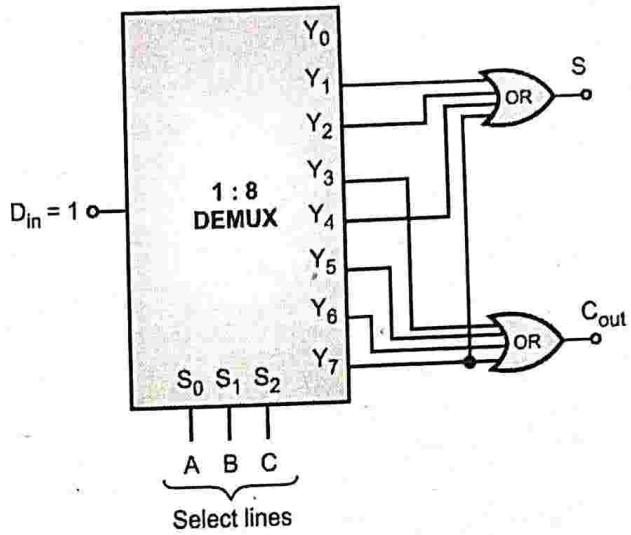
Step I : Truth table of full adder

Select input			Output	
A	B	C	Sum (S)	Carry (C_{out})
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = \sum m (1, 2, 4, 7)$$

$$C_{out} = \sum m (3, 5, 6, 7)$$

Step II : Implementation



(1D121)**Fig. Ex. 6.8.5 : Full adder using demultiplexer**

UEx. 6.8.6 **MU - Q. 4(b), May 16, 5 Marks**

Implement full subtractor using demultiplexer.

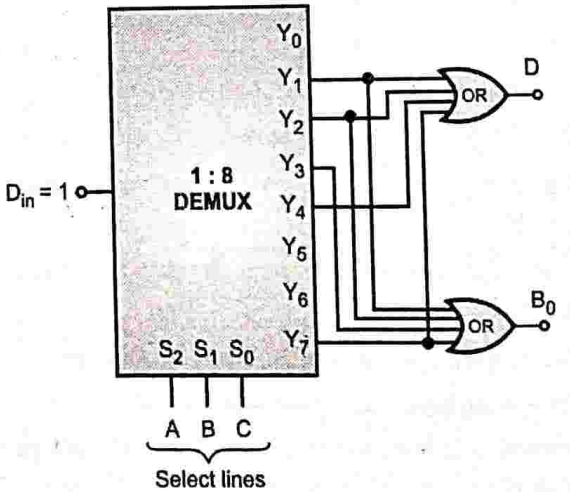
Ans. :

Step I : Truth table of full subtractor

Select input			Output	
A	B	C	D	B_0
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$D = \sum m (1, 2, 4, 7)$; $B_0 = \sum m (1, 2, 3, 7)$

Step II : Implementation



(1D122)**Fig. Ex. 6.8.6 : Implementation of full adder substractor using 1 : 8 Demultiplexer**

Module 3

6.9 ENCODERS

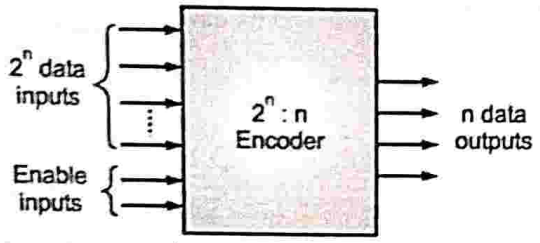
Q. 6.9.1 What do you mean by encoders ?

Ans. :

- An encoder is a combinational logic circuit that generates a specific code at its outputs.
- It encodes the information from 2^n inputs into an n bit code.

Block diagram

- Fig. 6.9.1 shows the block diagram of an encoder.



(1D105)**Fig. 6.9.1 : Block diagram of an encoder**

The types of encoders are :

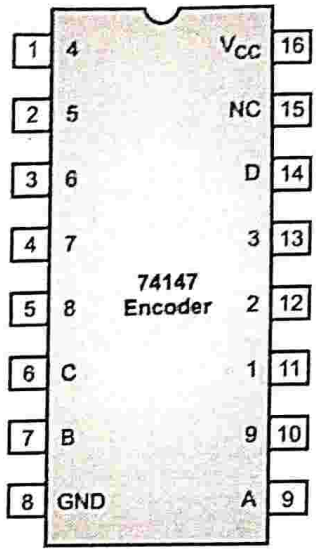
1. Priority encoders.
2. Decimal to BCD encoder.
3. Octal to binary encoder.
4. Hexadecimal to binary encoder.

6.9.1 Priority Encoder (IC74147)

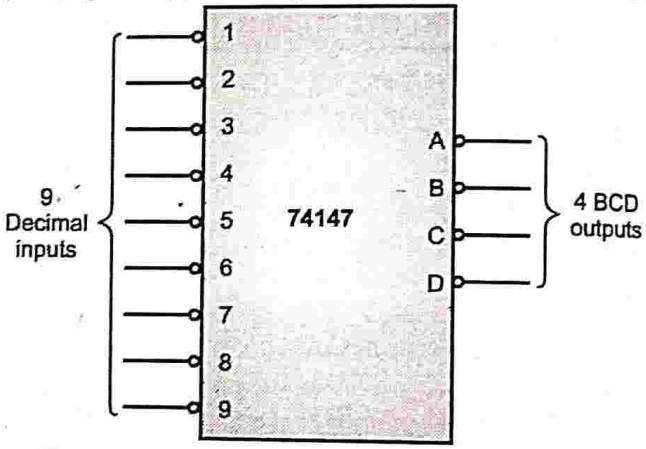
Q. 6.9.2 Explain in detail priority encoder.

Ans. :

- A **priority encoder** is a combinational logic circuit that responds to one input depending on the priorities assigned to the inputs.
- IC 74147 is a decimal to BCD encoder. It has 10 input lines that act an input for the encoder.
- The encoder has four output lines on which encoded BCD output is available.
- It is also called 10 to 4 line encoder.
- Fig. 6.9.2(a) shows the pin diagram and logic symbol for 74147.
- It is a TTL encoder that ensures that only the highest order data line is encoded.
- The 9 inputs as well as 4 BCD outputs are active low.
- There is no input line for the decimal 0. A zero is encoded when all data lines are at high logic level. All the inputs are buffered.



(1D106)**Fig. 6.9.2(a) : Pin diagram of 74147 10 : 4 line encoder**



(1D107)**Fig. 6.9.2(b) : Logic symbol for 74XX147 priority encoder**

- **Truth table :** Table 6.9.1 shows the truth table of 74147.

Table 6.9.1 : Truth table for 74XX147 decimal to BCD Encoded

Decimal value	Inputs									Outputs			
	1	2	3	4	5	6	7	8	9	D	C	B	A
0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	0
2	X	0	1	1	1	1	1	1	1	1	1	0	1
3	X	X	0	1	1	1	1	1	1	1	1	0	0
4	X	X	X	0	1	1	1	1	1	1	0	1	1
5	X	X	X	X	0	1	1	1	1	1	0	1	0
6	X	X	X	X	X	0	1	1	1	1	0	0	1
7	X	X	X	X	X	X	0	1	1	1	0	0	0
8	X	X	X	X	X	X	X	0	1	1	1	1	1
9	X	X	X	X	X	X	X	X	0	0	1	1	0

X indicates don't care condition.

Applications of Encoders

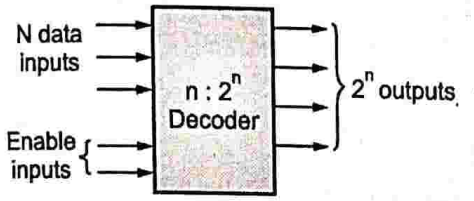
1. Email
2. Video encoders
3. Communication systems and networking

6.10 DECODER

Q. 6.10.1 Explain in brief the working of decoders.

Ans. :

- A **decoder** is a combinational logic circuit that converter an n-bit binary code to coded output.
- It is a multiple input multiple output combinational logic circuit.
- The input and output codes are different.



(1D132)**Fig. 6.10.1 : Block diagram of a decoder**

- The decoder has n inputs and 2^n outputs. The enable inputs are used to control the operation of the decoder.
- If any of the enable inputs is disabled, all outputs of the decoder are disabled.

6.11 74138 : 3 : 8 LINE DECODER

Q. 6.11.1 Write short note on : 3 to 8 line decoder.

MU - Q. 6(d), May 17, 5 M, Q. 6(d), Dec. 18, 5 M

Ans. :

- 74138 is most widely used 3:8 line decoder in microprocessor or microcomputer based system.
- 74138 is high speed 1 of 8 decoder / demultiplexer.

Pin names and description

Pin names	Description
A, B, C	Address inputs (Select lines)
$\overline{E_1}$ – $\overline{E_2}$ ($\overline{G_2A}$, $\overline{G_2B}$)	Enable inputs (Active LOW)
E_3 (G_1)	Enable input (Active HIGH)
$\overline{O_0}$ – $\overline{O_7}$	Outputs (Active LOW)

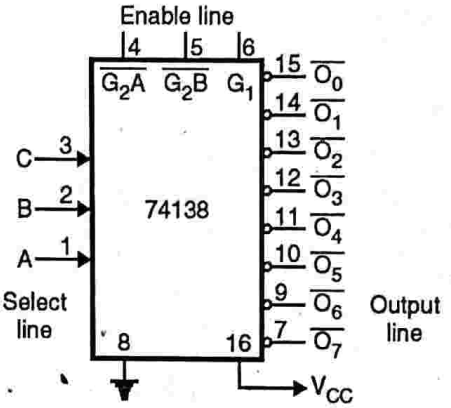


Fig. 6.11.1 : 74138

- 74138 chip is 16 pin chip with A, B and C select lines to select required output line to be active. and, two inputs are active LOW enable inputs is Active HIGH enable input. Finally three are, eight active LOW outputs present.
- To understand the working of functional diagram, let's see truth table of the chip.

Table 6.11.1 : Truth table of 3 : 8 decoder

Inputs						Outputs							
$\overline{G_2A}$	$\overline{G_2B}$	G_1	A	B	C	$\overline{O_0}$	$\overline{O_1}$	$\overline{O_2}$	$\overline{O_3}$	$\overline{O_4}$	$\overline{O_5}$	$\overline{O_6}$	$\overline{O_7}$
H	X	X	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	L	X	X	X	H	H	H	H	H	H	H	H
L	H	L	L	L	L	H	H	H	H	H	H	H	H
L	L	H	H	L	L	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	L	H	H	H	H	H
L	L	H	H	H	L	H	H	H	L	H	H	H	H
L	L	H	L	L	H	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	L	H	H
L	L	H	L	H	H	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	L

Module 3

A = LSB, C = MSB, H = HIGH Voltage Level,

L = LOW Voltage Level, X = Immaterial

Q. 6.11.2 Give the applications of Decoder.

☑ **Ans. :**

1. BCD to 7 segment decoder.
2. Address decoding.
3. Code converters
4. Implement Boolean function

Ex. 6.11.1

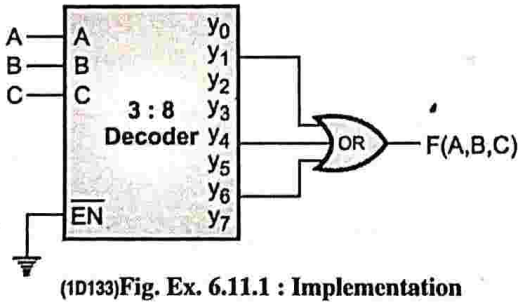
Implement Boolean function

$F_1 = \sum m\ (1, 4, 6)$ using 3 : 8 decoder.

☑ **Ans. :**

➪ **Step I :** Connect the variables of the given function as inputs of the decoder.

➪ **Step II :** Logically OR the outputs corresponding to the minterms the output.



(1D133)**Fig. Ex. 6.11.1 : Implementation**

Ex. 6.11.2

Design full adder using 3 : 8 decoder with active low outputs and NAND gates.

☑ **Ans. :**

➪ **Step I :** To obtain truth table of full adder

Table Ex. 6.11.2 : Truth table of full adder

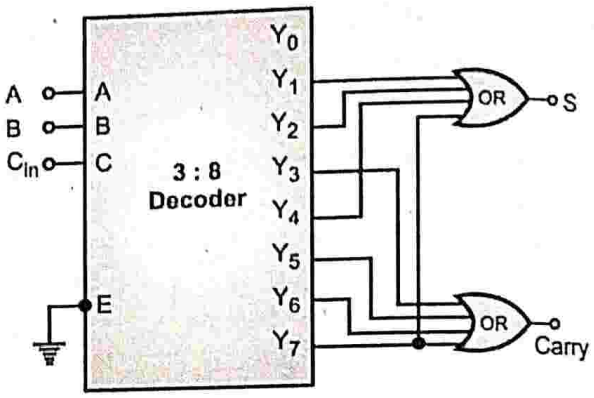
Inputs			Output	
A	B	C_{in}	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sum = $\sum m\ (1, 2, 4, 7)$

Carry = $\sum m\ (3, 5, 6, 7)$

➪ **Step II :** Connect the function variables (A, B, C_{in}) to the inputs of the decoder

➪ **Step III :** Logically OR the outputs corresponding to the present inputs to get the output



(1D134)**Fig. Ex. 6.11.2 : Implementation**

Ex. 6.11.3

Design a implement a full subtractor circuit using 3 : 8 using decoder.

☑ **Ans. :**

➪ **Step I :** To obtain truth table of full subtractor

Table Ex. 6.11.3 : Truth table of full subtractor

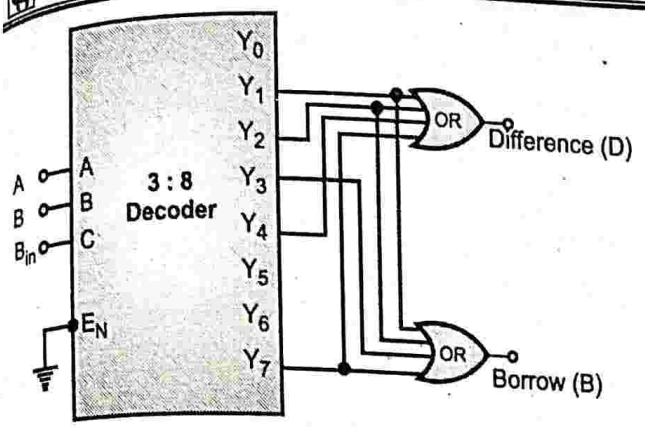
Inputs			Output	
A	B	B_{in}	Difference	B_0
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	0	1	1	0
1	1	0	0	1
0	1	0	1	0
1	1	0	0	0
1	1	1	1	1

Difference D = $\sum m\ (1, 2, 4, 7)$

Borrow B_0 = $\sum m\ (1, 2, 3, 7)$

➪ **Step II :** To connect function variables A, B, B_{in} to inputs of decoder

➪ **Step III :** Logically OR outputs corresponding to given minterms to get output.



(1D135)**Fig. Ex. 6.11.3 : Implementation**

UEx. 6.11.4 MU - Q. 5(a), Dec. 16, 10 Marks

Design a 3 bit binary to 3 bit gray code converter using 3 : 8 decoder.

Ans. :

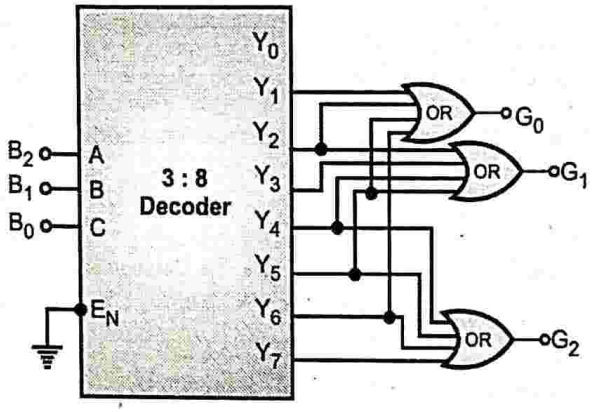
Step I : To obtain truth table of 3 : 8 decoder

Table Ex. 6.11.4 : 3-bit binary to gray code converter

Inputs			Output		
B_2	B_1	B_0	G_2	G_1	G_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

$G_2 = \sum m\,(4, 5, 6, 7)$ $G_1 = \sum m\,(2, 3, 4, 5)$

$G_0 = \sum m\,(1, 2, 5, 6)$



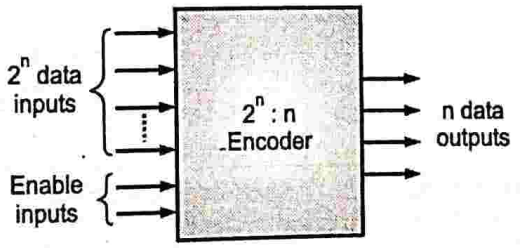
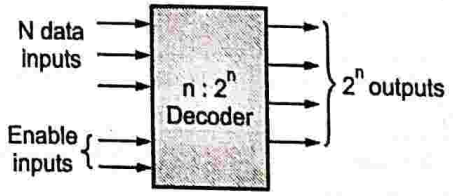
(1D136)**Fig. Ex. 6.11.4 : Implementation**

Module 3

6.12 COMPARISONS

Q. 6.12.1 Differentiate between encoder and decoder.

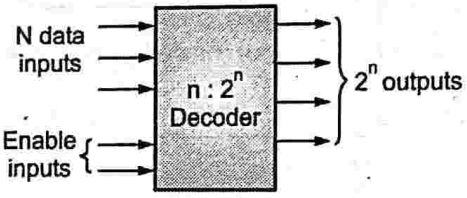
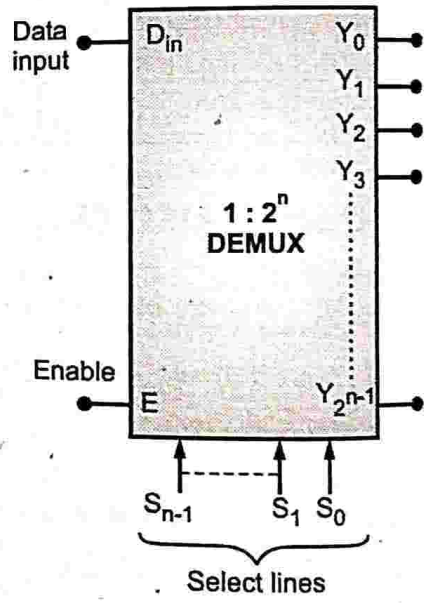
Ans. : Difference between encoder and decoder

Sr. No.	Parameter	Encoder	Decoder
1.	Definition	An encoder is a combinational logic circuit that generates a specific code at its outputs.	A decoder is a combinational logic circuit that converts an n-bit binary code to coded output.
2.	Block diagram	 (1D105)**Fig. 6.12.1 : Block diagram of an encoder**	 (1D132)**Fig. 6.12.2 : Block diagram of a decoder**
3.	Input applied	Active input signal	Coded binary input
4.	Output generated	Coded binary output	Active output signal

Sr. No.	Parameter	Encoder	Decoder
5.	Input lines	2^n	n
6.	Output lines	n	2^n
7.	Complexity	Simple	Complex
8.	Types	1. Priority encoder 2. Decimal to BCD encoder (10 to 4 linen output) 3. Octal to binary encoder. 4. Hexadecimal to binary encoder.	1. 2 to 4 line decoder 2. 3 : 8 line decoder 3. 4 : 16 line decoder
9.	Applications	1. Communication systems and networking 2. email 3. Video encoders	1. Wireless communication 2. Seven segment displays 3. Memory address decoding 4. Code converters

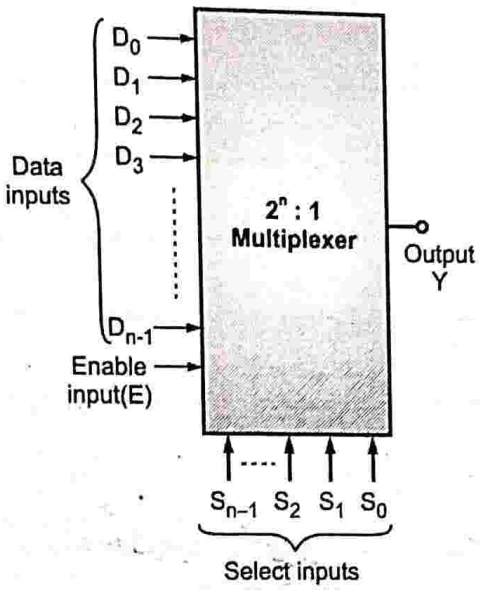
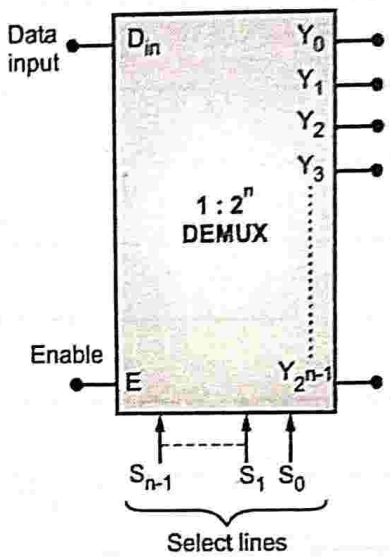
Q. 6.12.2 Differentiate between decoder and demultiplexer.

☑ **Ans. : Difference between decoder and demultiplexer**

Sr. No.	Parameter	Decoder	Demultiplexer
1.	Definition	**Decoder** is a combinational logic circuit that converts an n bit binary code into coded output.	**Demultiplexer is** a combinational logic circuit that depending on the status of the select inputs, routes the data input to one of the several data ouputs.
2.	Block diagram	 (1D132)**Fig. 6.12.3 : Block diagram of a decoder**	 (1D108)**Fig. 6.12.4 : Block diagram of a Demultiplexer**
3.	Characteristic	A decoder takes n input lines and produces 2^n output lines.	A demultiplexer transmits data from one input line D_{in} to 2^n possible output lines.
4.	Reverse function of	It performs the reverse function of an encoder.	It performs the reverse function of a multiplexer.
5.	Selection lines	There are no selection lines.	Output line is determined by the combination of select lines.
6.	Types	(a) 2 to 4 line decoders (b) 3 to 8 line decoders (c) 4 to 16 line decoders	(a) 1 : 4 demultiplexer (b) 1 : 8 demultiplexer (c) 1 : 16 demultiplexers

Q. 6.12.3 Differentiate between multiplexer and demultiplexer.

Ans. : Difference between multiplexer and demultiplexer

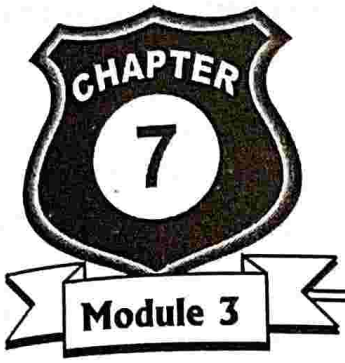
Sr. No.	Parameter	Multiplexer	Demultiplexer
1.	Definition	A multiplexer is a combinational logic circuit that selects one data input and routes it to the output.	A demultiplexer is a combinational logic circuit that routes the data input to the selected data output.
2.	Symbol	 (1D29)**Fig. 6.12.5 : Block diagram of 2^n : 1 multiplexer**	 (1D108)**Fig. 6.12.6 : Block diagram of a Demultiplexer**
3.	Number of data inputs	2^n	1
4.	Number of data outputs	1	2^n
5.	Device configuration	It is 2^n : 1 device it behaves as a data selector.	It is a 1 : 2^n device it behaves as a data distributor.
6.	Types of multiplexer.	(a) 4 : 1 multiplexer (b) 8 : 1 multiplexer (c) 16 : 1 multiplexer (d) 32 : 1 multiplexer	(a) 1 : 4 demultiplexer (b) 1 : 8 demultiplexer (c) 1 : 16 demultiplexer (d) 1 : 32 demultiplexer

Module 3

....Chapter ends

□□□

Note



Flip-Flops

University Prescribed Syllabus

Processor Organization and Architecture

3.2 Introduction to Flip Flop : SR, JK, D, T (Truth table).

Module 3

7.1 SEQUENTIAL CIRCUITS

Q. 7.1.1 Explain the need of sequential circuits.

Ans. :

- The combinational circuits are circuits whose output is dependent on the inputs present at that time.
- However, there are many applications where it is desired to generate digital output according to the sequence in which the input signals are received. Combinational circuits cannot produce such outputs.
- In such applications, the outputs generated depend on the present state inputs and also on the past state of the inputs. The past state is given as feedback from the output back to the input. Such circuits are called as **sequential circuits.**

Q. 7.1.2 Define sequential circuit.

Q. 7.1.3 What is sequential circuit? Write down its characteristics.

Ans. :

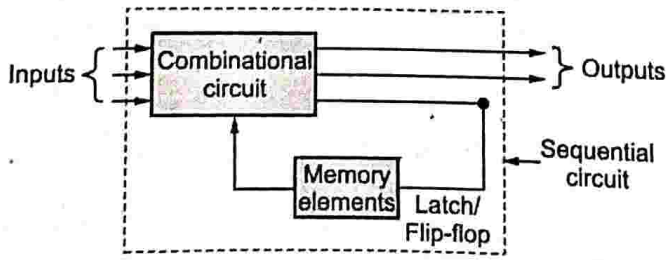
(A) Sequential Circuits

Definition : The circuits whose output at any instant of time is dependent on the present state of input as well as past state of inputs are called as sequential circuits.

- **Examples :** Counters, shift registers, serial adders, sequence generators, logic function generators, etc.

(B) Block Diagram of a Sequential Circuit

Please Refer Fig. 7.1.1.



(1E1)**Fig. 7.1.1 : Block diagram of a sequential circuit**

(C) Characteristics of a Sequential Circuit

1. The sequential circuit is a closed-loop system.
2. The sequential circuit is a recursive system; i.e., the output of a sequential circuit depends on its present state of input and also on its past output.
3. It is a combination of a combinational circuit and feedback/memory element. The feedback element is a memory storage device (**latch or flip-flop**).
4. The binary information/data stored in the memory element at any given time is referred to as the **present state of the sequential circuit.**
5. The combinational circuit performs operations on external inputs and on present state inputs to generate new outputs. They are stored in the memory element and are called as **next state of the sequential circuit**. Thus, the output of a sequential circuit is a function of the time sequence of the external inputs, internal states, (the present and next state) and outputs.

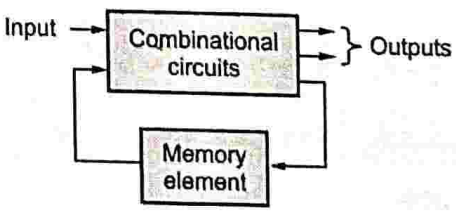
• NOTES •

7.2 COMPARISON OF COMBINATIONAL AND SEQUENTIAL CIRCUITS

UQ. 7.2.1 What is the difference between combinational and sequential circuits ?

(MU - Q. 1(e), Dec. 15, Dec. 18)

Ans. :

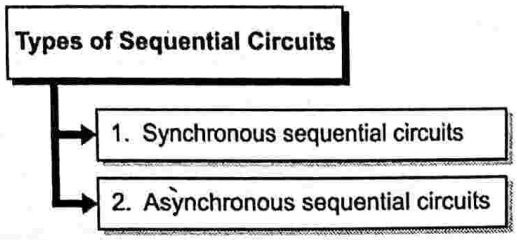
Sr. No.	Parameter	Combinational circuits	Sequential circuits
1.	**Output**	In combinational circuits, the output variables at any time instant are dependent on the present input variables.	In sequential circuits, the output variables at any instant of time are dependent on the present input variables and also on past state of the input variables.
2.	**Memory element**	Memory element is not required in combinational circuits.	Memory element is needed to store the past history of the input variables.
3.	**Speed**	The combinational circuits are faster in speed. This is because the delay between the input and output is only because of the propagation delay of gates.	The sequential circuits are slower in comparison to the combinational circuits.
4.	**Design**	They are easy to design	They are difficult to design.
5.	**Feedback**	There is no feedback between input and output	There is feedback between input and output.
6.	**Time**	Combinational circuits are time-independent	Sequential circuits are time-dependent.
7.	**Elementary building blocks**	Logic gates.	Flip-flops/Latches.
8.	**Use**	Combinational circuits are mainly used for arithmetic and Boolean operations	Sequential circuits are mainly used for storing data.
9.	**Cost**	More expensive circuit.	Cheap circuit.
10.	**Block diagram**	 (1E2)	 (1E3)
11.	**Examples**	Parallel adders, full adders, half adders, subtractors, encoders, decoders, multiplexers, demultiplexers, code converters, etc.	Counters, shift registers, serial adders, sequence generators, etc.

Module 3

7.3 TYPES OF SEQUENTIAL CIRCUITS

Q. 7.3.1 Define and classify sequential circuits.

Ans. :



(1E96)**Fig. 7.3.1 : Types of Sequential Circuits**

– There are two types of sequential circuits. They are :

1. Synchronous Sequential Circuits

Definition : The sequential circuits whose operation can be controlled by a clock are called as synchronous sequential circuits.

2. Asynchronous Sequential Circuits

Definition : The sequential circuits whose operations are not controlled by a clock are called as asynchronous sequential circuits. In asynchronous sequential circuits, the outputs can occur at any time the inputs are applied.

7.3.1 Comparison of Asynchronous and Synchronous Sequential Circuits

UQ. 7.3.2 Differentiate between asynchronous and synchronous sequential circuits. (MU - May 18)

Ans. :

Sr. No.	Parameter	Synchronous sequential circuits	Asynchronous sequential circuits
1.	**Definition**	The sequential circuits whose operation can be controlled by a clock are called synchronous sequential circuits.	The sequential circuits whose operations are not controlled by a clock are called as asynchronous sequential circuits.
2.	**Memory elements**	The memory elements are clocked flip-flops in synchronous sequential circuits.	The memory elements are time-delay elements or unclocked flip-flips in asynchronous sequential circuits.
3.	**Change in input signals**	In synchronous sequential circuits, if there is a change in the input signals, then it can affect the status of the memory element only when the clock signal is activated.	In asynchronous sequential circuits, if there is a change in the input signals, then it can affect the status of the memory element at any instant of time.
4.	**Speed**	The synchronous sequential circuits are a little slower because of time delays due to clock.	The asynchronous sequential circuits operate faster because of absence of clock.
5.	**Design**	Easy	Difficult

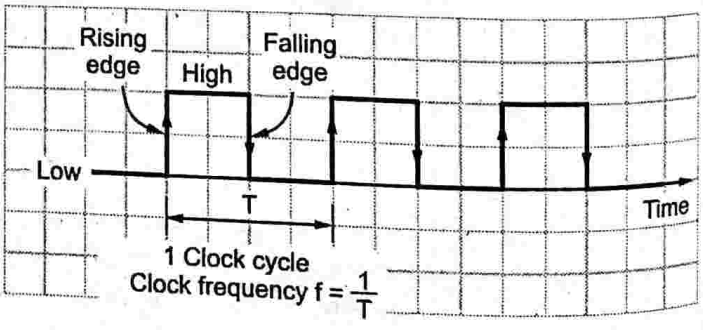
7.4 CLOCK SIGNAL

Q. 7.4.1 What is a clock signal ?

Ans. :

– The **clock signal** is a timing signal. In the sequential circuits, the required operations occur only on the activation of the clock signal.

– Fig. 7.4.1 shows a clock signal with 50% duty cycle. On the occurrence of clock pulse, data flow begins in sequential circuits.



(1E4)**Fig. 7.4.1 : Clock signal**

7.5 FLIP-FLOPS

Q. 7.5.1 What is a flip-flop ? or Define flip-flop. Write down its characteristics.

Q. 7.5.2 With the help of a diagram, explain 1-bit memory cell.

Q. 7.5.3 Explain how the flip-flop acts like a storage device.

Q. 7.5.4 With the help of diagram, explain bistable multivibrator.

Ans. :

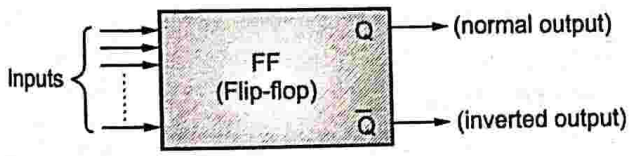
(A) Flip-flop

Definition : A flip-flop is an important memory element. It is constructed using an assembly of logic gates.

- A logic gate is not capable of storing data or information.
- However, in a flip-flip many logic gates are connected in such a manner that they allow data or information to be stored.

(B) Logical Symbol

- Fig. 7.5.1 shows logical symbol of a flip-flop.



(1E5)**Fig. 7.5.1 : Symbol of flip-flop**

(C) Characteristics

1. Flip-flops can be constructed by various methods using different gate arrangements. **Flip-flops are the basic building blocks of sequential circuits.**
2. A flip-flop is called as **a binary or 1-bit memory cell as it can store 1-bit data.** It is a **bistable multivibrator** with two stable states.
3. Two states are, state **0** and state **1**. A flip-flop may stay in either of state **0** or state **1** indefinitely.
4. To modify the state, an appropriate triggering signal needs to be applied to the flip-flop.
5. A flip-flop can be constructed using NAND or NOR gates. It has two outputs as shown in Fig. 7.5.1, Q and $\bar{Q}$. Q is the normal output from the flip-flop while $\bar{Q}$ is the inverted output.
6. If **Q = 1**, the flip-flop is said to be in 'logic 1' state or HIGH state or **SET state** and if **Q = 0** the flip-flop is said to be in 'logic 0' state or LOW state or **RESET or CLEAR state.**
7. A flip-flop can have one or more inputs.
8. An input signal that controls the flip-flop to modify its state is called as an **excitation.**
9. A trigger pulse needs to be applied to a flip-flop to cause a change in its output state. The flip-flop will retain the state even if the applied pulse is removed. Thus, **a flip-flop acts as a storage device.** When the output Q = 0, the flip-flop stores '0' and when the output Q = 1, the flip-flop stores '1'.
10. Flip-flops are commonly used in shift registers and counters.
11. The different flip-flops used for various applications are SR Flip-flop, JK Flip-flop, D flip-flop and T-Flip-flop.

Q. 7.5.5 What do you understand by a stable state ?

Ans. :

- The **stable state** is a state in which the circuit can permanently retain or hold itself.
- The circuit will change its state only on the application of an external signal.

Q. 7.5.6 What is flip-flop called ? How many states does it have ?

Ans. :

- A flip-flop is called as a **bistable multivibrator** or **1-bit memory cell.**
- It has two states : state **0** and state **1.**

Q. 7.5.7 What is the memory characteristic of flip-flops ?

Ans. :

- A trigger pulse needs to be applied to a flip-flop to cause a change in its output state.
- The flip-flop will retain the state even if the applied pulse is removed.
- Thus, **a flip-flop acts as a storage device.** When the output Q = 0, the flip-flop stores '0' and when the output Q = 1, the flip-flop stores '1'.
- This is the **memory characteristic of flip-flops.**

Module 3

Q. 7.5.8 How many flip-flops are required for storing n bits of information or data ?

☑ **Ans. :**

- One flip-flop can store 1-bit information at a time .
- Hence, for storing n-bits of information or data we will need 'n' flip-flops. E.g., for storing 2-bit data, we will need 2 flip-flops.

Q. 7.5.9 What do you mean by setting and resetting of flip-flop?

☑ **Ans. :**

- **Setting of Flip Flop :** The process of storing a '1' into a flip-flop is called **setting or presetting the flip-flop .**
- **Resetting of Flip Flop :** The process of storing a '0' into a flip-flop is called **resetting or clearing the flip-flop.**

UQ. 7.5.10 "Flip-flop is a sequential circuit". Explain. **(MU - Dec. 18)**

☑ **Ans. :** In a sequential circuit, the output depends on the present state of input as well as past output.

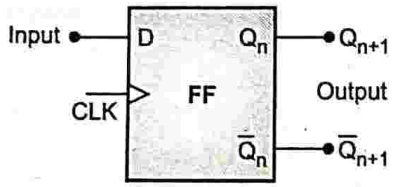


Fig. 7.5.2 : A symbol of a flip-flop

From Fig. 7.5.2, we can see that

Output, Q_{n+1} = f (input, Q_n) where, Q_n = Past output

In flip-flop, the output depends on the present state of input and past output. Hence, **flip-flop** is a **sequential circuit.**

⏭ 7.6 LATCHES

Q. 7.6.1 What is a latch ?

Q. 7.6.2 Define latch. Write down its characteristics. Describe its classification.

☑ **Ans. :**

☞ (A) Latch

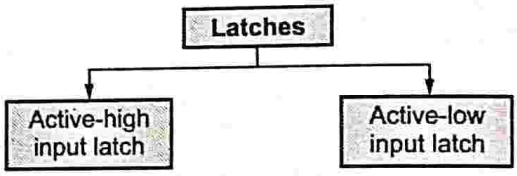
❒ **Definition :** A latch is a non-clocked flip-flop that checks all its inputs continuously and modifies its output at any time, independent of the clock signal.

☞ (B) Characteristics

1. As soon as a latch receives an input pulse, it **"latches on"** to a **1** if pulse is high or **SET** and latches to a **0** if a pulse is low or **RESET**. Thus, a latch is independent of the clock signal.
2. A latch continuously monitors the inputs and as soon as input pulse is received, it latches to that state.

☞ (C) Classification of Latches

The types of latches are :



(1E102)Fig. 7.6.1 : Classification of Latches

1. **Active-LOW input latch**

   Active-LOW indicates that the SET and RESET inputs are in the HIGH state and one of them will be pulsed LOW whenever it is required to change the output of the latch.

2. **Active HIGH input latch**

   Active-High input latch indicates that both the SET and RESET inputs are in the LOW state and one of them will be pulsed HIGH whenever it is required to change the output of the latch.

Q. 7.6.3 How can you build an SR latch using universal gates ?

☑ **Ans. :**

- An S-R latch can be constructed using two cross-coupled NOR gates or two cross-coupled NAND gates.
- An active-LOW SR latch can be constructed with two cross-coupled NAND gates and an active-HIGH SR latch can be constructed using two cross-coupled NOR gates.

7.7 COMPARISON BETWEEN FLIP-FLOPS AND LATCHES

UQ. 7.7.1 What is difference between latch and flip-flop ? MU - May 16

Ans. :

Sr. No.	Latches	Flip-flops
1.	Latches are the basic building blocks of sequential circuits. Latches are constructed from the logic gates.	Flip-flops are the basic building blocks of the sequential circuits. Flip-flops are constructed from the latches and include a clock signal.
2.	A latch checks all its inputs continuously and modifies its output at any time, independent of the clock signal.	A flip-flop checks all its input continuously and modifies its output only when the clock pulse is applied.
3.	Latches are level-triggered.	Flip-flops are edge-triggered.
4.	Latches are sensitive to the input switches. As long as latches are ON, they can send data.	Flip-flops are sensitive to the clock signal. They do not modify the output unless there is a change in the input clock signal.
5.	They cannot be used as registers.	Flip-flops can be used as registers. They include the input signals as well as a clock signal that enables cascaded flip-flops to be used as registers.
6.	Latches are asynchronous, as they are independent of the clock signal.	Flip-flops are synchronous, as they are dependent on the clock signal.
7.	They require less power.	They require more power.
8.	A latch operates depending on the enable signal.	A flip-flop operates depending on the clock signal.
9.	They operate faster in comparison to flip-flops due to absence of clock signal.	They operate slower than latches.
10.	Latches are simple to design.	Flip-flops are complex to design.
11.	Latches are more prone to glitches.	Flip-flops are highly immune to glitches.
12.	Latches are responsive towards the faults occurring on the enable pin.	Flip-flops are protected towards the faults.

Module 3

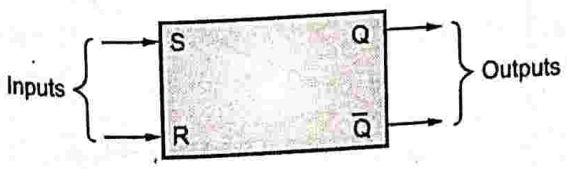
7.8 S-R LATCH

Q. 7.8.1 Describe Logic symbol Construction and Operation of an S-R latch.

Ans. :

(A) Logic Symbol

Logic symbol of SR latch is shown in Fig. 7.8.1



(1E6)Fig. 7.8.1 : Logic symbol of SR latch

(B) Construction and Operation

1. It is the simplest type of latch that can be constructed using two NAND or two NOR gates. In an S-R flip-flop, the letter S indicates SET and the letter R indicates RESET or CLEAR.
2. It is referred as **SR (set-reset) or RS (Reset-set) latch**. The S-R latch is also called as the **S-C (Set-Clear) latch**.
3. It has two inputs, S (set) and R (reset) and two outputs, Q and $\overline{Q}$. Fig. 7.8.1 shows its logic symbol.
4. An S-R latch can be constructed using two cross-coupled NOR gates or two cross-coupled NAND gates.
5. An active-LOW SR latch can be constructed with two cross-coupled NAND gates and an active-HIGH SR latch can be constructed using two cross-coupled NOR gates.

6. The corresponding state of the S-R latch determines the output Q and inverted output $\bar{Q}$.

   (a) If S = 1, R = 0, the output Q = 1 and $\bar{Q}$ = 0. The flip-flop is in **SET state.**

   (b) If S = 0, R = 1, the output Q = 0 and $\bar{Q}$ = 1. The flip-flop is in **RESET state.**

   (c) If both S = R = 0, the output of the latch remains in same state. It does not change. The flip-flop holds the previous state. Thus, the flip-flop is in **HOLD state** or **NO CHANGE state .**

   (d) If both S = R = 1, i.e. both the inputs are high, the output Q and $\bar{Q}$ are unpredictable, i.e., Q and $\bar{Q}$ may be high or low or any one of them can be high or low. This condition is not allowed. It is called as **invalid or indeterminate condition.** The flip-flop is in **indeterminate** or **invalid** or **prohibited state.**

7.8.1 S-R Latch using NOR Gates (Active-High SR Latch)

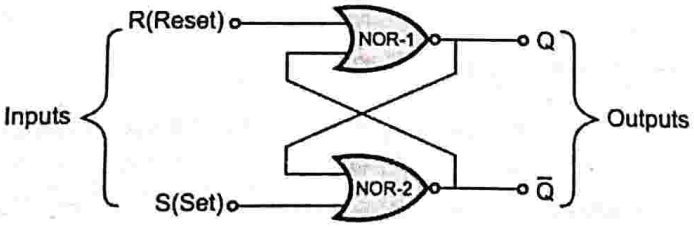
Q. 7.8.2 Explain the working of SR latch by using NOR gate.

Q. 7.8.3 Draw & explain in brief, a high assertion input SR latch.

Ans. :

(A) Circuit diagram

Circuit diagram of S-R latch using NOR gates is shown in Fig. 7.8.2(a).



(1E7)**Fig. 7.8.2(a) : Active-high SR latch using NOR gates**

(B) Construction

- Two NOR gates are cross-coupled to each other.
- The output of NOR-1 gate is connected to input of NOR-2 gate and output of NOR-2 gate is connected as input to NOR-1 gate, as shown in Fig. 7.8.2(a).
- The two outputs of the latch are, Q and $\bar{Q}$.

(C) Truth Table

Table 7.8.1 : Truth table of active-high SR latch

S	R	Q_n	Q_{n+1}		$\bar{Q}_{n+1}$	State
0	0	0	0	Q_n	1	No change (NC) or Hold
0	0	1	1		0	
0	1	0	0	0	1	Reset
0	1	1	0		1	
1	0	0	1	1	0	Set
1	0	1	1		0	
1	1	0	x	x	x	Indeterminate (invalid)
1	1	1	x		x	

(D) Operation

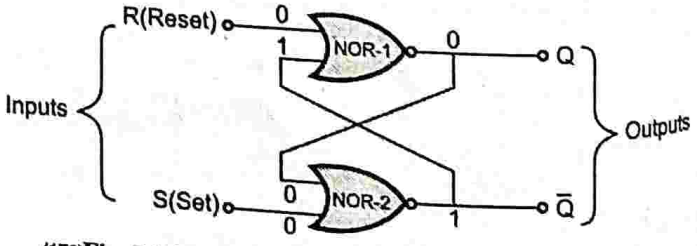
- The **output** of the NOR gate will be **0** if any of its **input** is at logic **1**.
- Initially the S(SET) and R(RESET) inputs are at low (logic 0) state.
- Whenever we want to latch the outputs, we need to pulse either S or R to logic **1**.
- We will analyze the active-high S-R latch for different input/output combinations, which are as follows :

Sr. No.	State	S(SET)	R(RESET)
(1)	Hold	0	0
(2)	RESET	0	1
(3)	SET	1	0
(4)	INDETERMINATE	1	1

1. Hold state

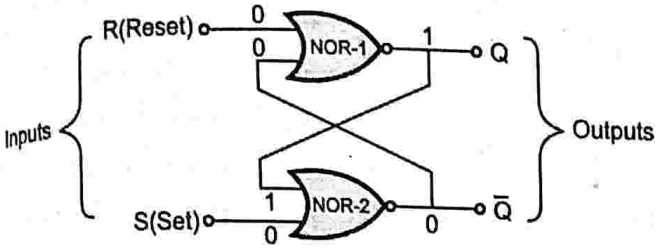
S (SET) = 0 and R (RESET) = 0

- This state is the normal resting state of the S-R latch.
- The latch will remain in the state it was before the occurrence of this input. Initially if Q = 0 and $\bar{Q}$ = 1, one input of the NOR-1 gate is logic **1**. Hence, its output is Q = 0.
- Both the inputs to NOR-2 gate are at logic **0**, making output $\bar{Q}$ = 1; i.e., there is no change in the output state. Fig. 7.8.2(b) shows this condition.



(1E8)**Fig. 7.8.2(b) : S = 0, R = 0 (Q = 0, $\bar{Q}$ = 1) (Hold state)**

- Initially if Q = 1 and $\bar{Q}$ = 0. If R = 0, both the inputs to the NOR-1 gate are at logic **0**. Thus, the output Q = 1.
- If Q = 1, S = 0, one input to NOR-2 is at logic **1**. Thus the output $\bar{Q}$ = 0.
- Fig. 7.8.2(c) shows this condition.

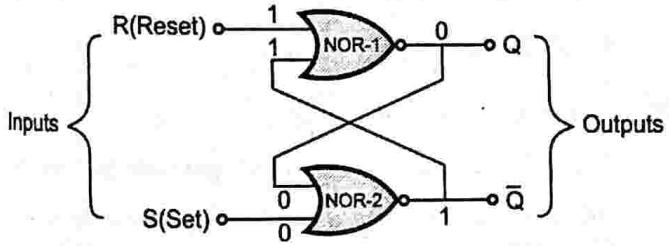


(1E9)**Fig. 7.8.2(c) : S = 0, R = 0 (Q = 1, $\bar{Q}$ = 0) (Hold State)**

- Thus, when S (SET) and R(RESET) both are at 'logic' 0' (low state), the state of output does not change.
- The output remains latched in its previous state.
- Hence, it is called as **inactive state** as there is no change in state.
- Thus, for **hold state** $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$

2. RESET state
S (SET) = 0, R (RESET) = 1.

- If R = 1, R input of the NOR-1 gate is at logic **1**.
- Thus, output Q = 0.
- If Q = 0, S = 0, both the inputs to NOR-2 gate are at logic 0, producing output $\bar{Q}$ = 1.



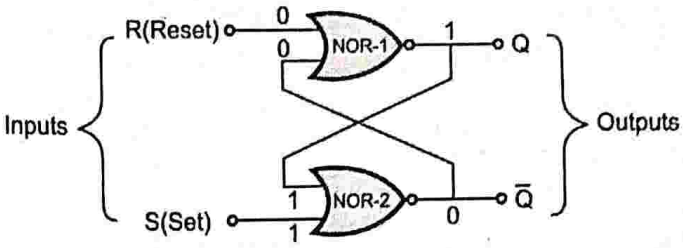
(1E10)**Fig. 7.8.2(d) : S = 0, R = 1 (RESET state)**

Thus, when S(SET) = 0 and R(RESET) = 1, the **output of latch will always RESET (Q = 0).**

For **RESET state :** $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$

3. SET state
S (SET) = 1, R (RESET) = 0

- If S = 1, S input to NOR-2 is at logic **1**, producing output $\bar{Q}$ = 0. If R = 0, $\bar{Q}$ = 0, both the inputs to the NOR-1 gate are at logic **0**, producing output Q = 1. Fig. 7.8.2(e) shows this condition.

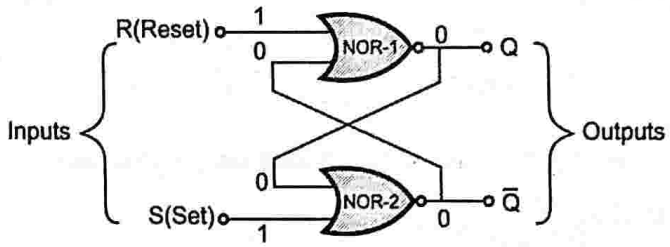


(1E11)**Fig. 7.8.2(e) : S = 1, R = 0 (SET state)**

Thus, when S(SET) = 1 and R(RESET) = 0, the output Q of latch will always be SET (Q = 1).

For SET state : $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$

4. INDETERMINATE state
S(SET) = 1 and R(RESET) = 1



(1E7A)**Fig 7.8.2(f) : S = 1, R = 1 (Indeterminate state)**

- When S = 1, S input of NOR-2 gate is at logic **1**, producing output $\bar{Q}$ = 0. If R = 1, $\bar{Q}$ = 0, the R input of NOR-1 gate is at logic **1** producing output Q = 0.
- The output $\bar{Q}$ is not complement of Q. Hence, this condition is not allowed. This condition is called as **invalid or indeterminate condition.**
- **Thus, when S(SET) = 1 and R(RESET) = 1, the output of the latch is unpredictable.** This condition must be avoided.
- **For indeterminate state :** $Q_{n+1} = x$, $\bar{Q}_{n+1} = x$

Summary of Operation

Sr. No.	State	Output	
		Q_{n+1}	$\bar{Q}_{n+1}$
(1)	Hold	Q_n	$\bar{Q}_n$
(2)	RESET	0	1
(3)	SET	1	0
(4)	INDETERMINATE	x	x

Module 3

7.8.2 SR Latch using NAND Gates/Active-Low SR Latch

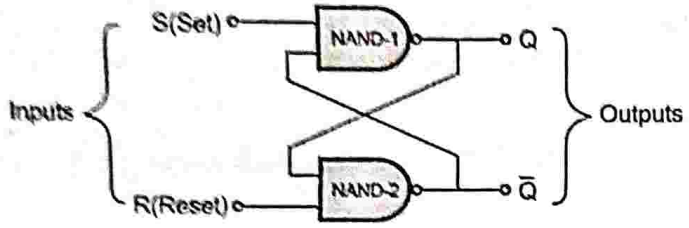
Q. 7.8.4 Explain the working of active-low SR latch by using NAND gates.

Q. 7.8.5 Draw low assertion input SR latches.

Ans. :

(A) Circuit Diagram

Circuit diagram of S-R latch using NAND gates is shown in Fig. 7.8.3.



(IET2)Fig. 7.8.3 : Active-low SR latch using NAND gates

(B) Construction

- Two NAND gates are cross-coupled to each other.
- The output of NAND-1 gate is connected to input of NAND-2 gate and output of NAND-2 gate is connected as input to NAND-1 gate, as shown in Fig. 7.8.3.
- The two outputs of the latch are, Q and $\bar{Q}$.

(C) Truth table

Table 7.8.2 : Truth table of Active-Low SR latch

S	R	Q_n	Q_{n+1}	$\bar{Q}_{n+1}$	State
0	0	0	X	X	Indeterminate (invalid)
0	0	1	X	X	
0	1	0	1	0	Set
0	1	1	1	0	
1	0	0	0	1	Reset
1	0	1	0	1	
1	1	0	0	1	No Change (NC) or HOLD
1	1	1	1	0	

(D) Operation of active-LOW NAND latch

- The operation of active-low SR latch is exactly reverse of the operation of active-high SR latch. **The output of the NAND gate will be 1, if any of its input is 0.**
- The S(SET) and R(RESET) inputs are at logic 1 (high) state initially. Whenever we want to latch the outputs we need to pulse either S or R to logic 0.
- We will analyze the active-high S-R latch for different input/output combinations, which are as follows :

Sr. No.	State	S(SET)	R(RESET)
(1)	INDETERMINATE	0	0
(2)	SET	0	1
(3)	RESET	1	0
(4)	Hold	1	1

1. INDETERMINATE state : S = 0, R = 0

- If S = 0, S input of NAND-1 gate will be 0, producing output Q = 1. R = 0, Q = 1, R input of NAND-2 gate will be logic 0, producing output $\bar{Q} = 1$.
- Thus, Q = 1, $\bar{Q} = 1$; hence this condition must be avoided. It is called as **invalid or indeterminate condition.**
- **Thus when S(SET) = 0 and R(RESET) = 0, the output of the latch is unpredictable.**

For indeterminate state : $Q_{n+1} = x$, $\bar{Q}_{n+1} = x$.

2. SET state : S = 0, R = 1

- If S = 0, S input of NAND-1 gate will be 0, producing output Q = 1. If Q = 1, R = 1, the output of NAND-2 gate will be $\bar{Q} = 0$.
- **Thus, when S(SET) = 0 and R(RESET) = 1, the output Q of latch will always be set (Q = 1).**
- **For SET state : $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$**

3. RESET state : S = 1, R = 0

- If R = 0, the R input of NAND-2 gate will be logic 0, producing output $\bar{Q} = 1$. If S = 1, $\bar{Q} = 1$, i.e., both the inputs of NAND-1 gate are at logic 1, producing output Q = 0.
- **Thus, when S(SET) = 1 and R(RESET) = 0, the output Q of latch will always RESET (Q = 0).**
- **For RESET state : $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$**

4. HOLD state : S = 1, R = 1

- If S = 1 and if Q = 0 and $\bar{Q} = 1$, then both inputs to the NAND-1 gate are high, producing Q = 0. If R = 1 and Q = 1, output of NAND-2 will be $\bar{Q} = 1$; i.e., there is no change in state.
- If S = 1 and if Q = 1 and $\bar{Q} = 0$, then $\bar{Q}$ input to NAND-1 gate is logic 0, Hence output Q = 1. R = 1 and Q = 1, hence output $\bar{Q}$ of NAND-2 gate is 0; i.e., there is no change in state.

- **Thus, when S = 1, R = 1 the output of the latch does not change. The output is latched in the previous state. It is called as inactive state.**
- **Thus, for hold state $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$**

Summary of operation

Sr. No.	State	Output	
		Q_{n+1}	$\bar{Q}_{n+1}$
(1)	INDETERMINATE	x	x
(2)	SET	1	0
(3)	RESET	0	1
(4)	Hold	Q_n	$\bar{Q}_n$

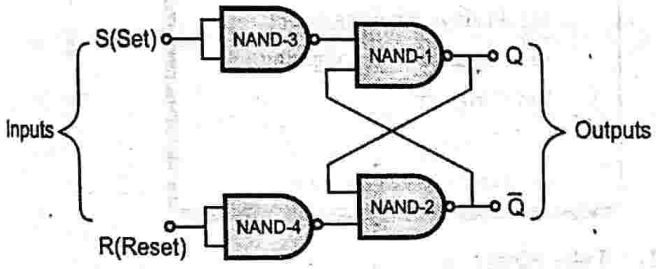
Q. 7.8.6 Explain the working of active-high SR latch by using NAND gates.

Q. 7.8.7 Draw high assertion input SR latches.

Ans. :

(A) Circuit diagram and construction

- Circuit diagram and construction of active-high S-R latch using NAND gates is shown in Fig. 7.8.4.
- An active-low SR latch is converted to active-high SR latch using NAND gates by adding inverters at the S and R inputs.



(1E13)Fig. 7.8.4 : Active-high SR latch using NAND gates

(B) Truth table of active-high S-R latch

Table 7.8.3 : Truth table of active-high SR latch

S	R	Q_n	Q_{n+1}	$\bar{Q}_{n+1}$	State
0	0	0	0	1	No change (NC) or Hold
0	0	1	1	0	
0	1	0	0	1	Reset
0	1	1	0	1	
1	0	0	1	0	Set
1	0	1	1	0	
1	1	0	x	x	Indeterminate (invalid)
1	1	1	x	x	

(C) Operation of active-high SR latch

- **The output of the NAND gate will be '1' if any of its input is at logic '0'.**
- Initially the S(SET) and R(RESET) inputs are at low (logic 0) state. Whenever we want to latch the outputs, we need to pulse either S or R to logic **1**.
- We will analyze the active-high S-R latch for different input/output combinations, which are as follows :

Sr. No.	State	S(SET)	R(RESET)
(1)	Hold	0	0
(2)	RESET	0	1
(3)	SET	1	0
(4)	INDETERMINATE	1	1

1. Hold state : S (SET) = 0 and R (RESET) = 0

- This state is normal resting state of the S-R latch.
- The latch will remain in the state it was before the occurrence of this input.
- When S (SET) and R(RESET) both are at logic **0** (low state), the state of output does not change. The output remains latched in its previous state.
- Hence, it is called as **inactive state,** as there is no change in state.
- **Thus, for hold state $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$**

2. RESET state : S (SET) = 0, R (RESET) = 1.

When S(SET) = 0 and R(RESET) = 1, the **output of latch** will always **RESET (Q = 0).**

For RESET state : $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$

3. SET state : S (SET) = 1, R (RESET) = 0

Thus, when S(SET) = 1 and R(RESET) = 0, the output Q of latch will always be SET (Q = 1).

For SET state : $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$

4. INDETERMINATE state
S (SET) = 1 and R (RESET) = 1

When S(SET) = 1 and R(RESET) = 1, the output of the latch is unpredictable. This condition must be avoided. It is called as **invalid or indeterminate condition.**

For indeterminate state : $Q_{n+1} = x$, $\bar{Q}_{n+1} = x$

Module 3

Summary of operation

Sr. No.	State	Output	
		Q_{n+1}	$\overline{Q}_{n+1}$
(1)	Hold	Q_n	$\overline{Q}_n$
(2)	RESET	0	1
(3)	SET	1	0
(4)	INDETERMINATE	x	x

7.9 CLOCKED FLIP-FLOPS

Q. 7.9.1 What is an asynchronous latch ?

Ans. :

- **Non-clocked** or **non-gated** latch is called as **asynchronous latch**.
- In asynchronous latch, the clock signal is not applied.
- Whenever input is applied, the latch will **latch its output to 0 or 1** asynchronously.

Q. 7.9.2 What is a synchronous latch ?

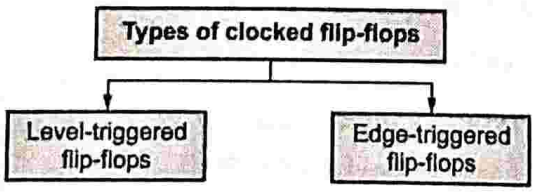
Ans. :

- The **clocked latch** is called as **gated** or **synchronous latch or flip-flops**.
- When the clock signal is applied, the output latches onto **0** or **1** only when the inputs are in synchronization with the clock.

Q. 7.9.3 What are clocked flip-flops? Describe its types.

Ans. :

- Flip-flops are constructed from latches and include a clock signal. Hence, they are called as clocked flip-flops.
- The output of the flip-flop will not change unless there is a change in the input clock signal.
- There are two types of clocked flip-flops :



(1E103)**Fig. 7.9.1 : Types of clocked Flip-Flops**

1. Level-triggered flip-flops

The flip-flops that respond to the changes in inputs only as long as their clock or enable is high are called as **level-triggered flip-flops.**

2. Edge-triggered flip-flops

The flip-flops that respond on the changes in input at the positive or negative-edge of the clock are called as **edge-triggered flip-flops.**

Q. 7.9.4 Define triggering. What are different methods for triggering flip-flops?

Ans. :

(A) Triggering

Definition : The clock signal is distributed to all parts of the circuit and most of the circuits outputs change state only when the clock makes a transition. This transition of clock signal from positive to negative or vice versa is known as triggering.

(B) Methods of Triggering

The different methods used for triggering are as follows :

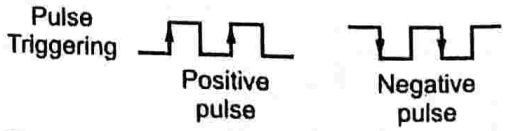
1. Pulse triggering
   (a) Positive pulse
   (b) Negative pulse
2. Edge triggering
   (a) Positive-edge triggering
   (b) Negative-edge triggering
3. Level triggering
   (a) Positive-level triggered
   (b) Negative-level triggered

1. Pulse triggering

There are two types of pulses :

(a) Positive pulse : A waveform in which the normal state is logic **0** and changes to logic **1** momentarily to produce a clock pulse.

(b) Negative pulse : A waveform in which the normal state is logic **1** and changes to logic **0** momentarily to produce a clock pulse.



(1E92)**Fig. 7.9.2 : Positive and Negative pulse triggering**

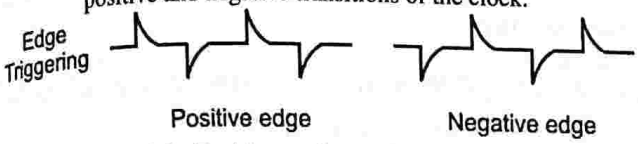
2. Edge triggering

Edge triggering means that the output state changes either at the positive-edge (rising edge) or at the negative-edge (falling edge) of the clock pulse.

(a) **Positive-edge triggering :** The output responds to the changes in the input only at the positive-edge of the clock pulse at the clock input.

(b) **Negative-edge triggering :** The output reacts to the changes in the input only at the negative-edge of the clock pulse at the clock input.

Fig. 7.9.3 shows the positive and negative-edges for positive and negative transitions of the clock.



(1E93)**Fig. 7.9.3 : Positive and negative-edge triggering**

3. **Level triggering :** In the level triggering, the output state is allowed to change according to input(s) when active level (either positive or negative) is maintained at the enable input.

There are two types of level triggered latches :

(a) **Positive-level triggered :** The output of flip-flop responds to the input changes only when its enable or clock input is **1** (HIGH).

(b) **Negative-level triggered :** The output of flip-flop responds to the input changes only when its enable or clock input is **0** (LOW).

(1E94)**Fig. 7.9.4 : Positive and negative-level triggering**

7.10 EDGE-TRIGGERED FLIP-FLOPS

Q. 7.10.1 What is difference between a gated-latch and an edge-triggered flip-flop?

Ans. :

- A **gated-latch** is a level triggered flip-flop that responds to changes in inputs as long as their clock or enable is high.
- An **edge-triggered flip-flop** is flip-flop that responds to the changes in input at positive and negative-edge of clock.

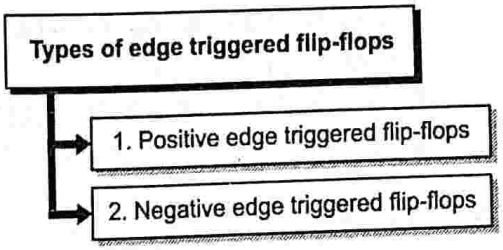
Q. 7.10.2 What do you understand by edge-triggered flip-flops ? Describe its classification.

Q. 7.10.3 What is dynamic triggering ?

Ans. :

Definition : The flip-flops that respond on the changes in input at the positive or negative-edge (rising or falling edge) of the clock are called as edge-triggered flip-flops.

There are two types of **edge-triggered flip-flops.**



(1E104)**Fig. 7.10.1 : Edge-triggered Flip-Flops**

1. **Positive (Rising) edge-triggered flip-flops** are those in which outputs can change state only at the positive (rising) edge (**0** to **1** or LOW to HIGH) of the clock pulse.

   Positive edge triggering is indicated by a triangle ▷ at the clock of a flip-flop.

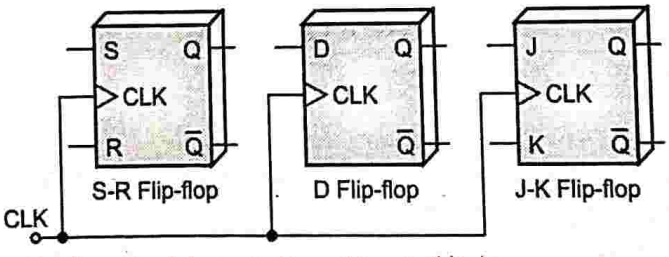
2. **Negative (Falling) edge-triggered flip-flops** are those in which outputs can change state only at the negative (falling) edge (**1** to **0** or HIGH to LOW) of the clock pulse.

   Negative edge-triggering is indicated by a triangle with bubble ○▷ at clock of the flip-flop.

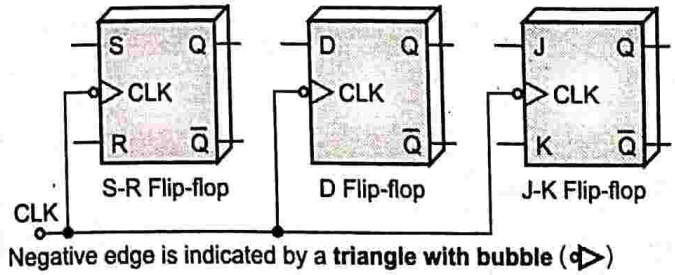
- The three edge-triggered flip-flops are :

  (1) S-R flip-flop (2) D flip-flop (3) J-K flip-flop

- Fig. 7.10.2 shows the edge-triggered flip-flops. D and JK flip-flop are used in many applications in-comparison to the SR flip-flop.
- The SR flip-flop is considered because it is easy to construct the D and JK flip-flops using the SR flip-flop.
- Edge-triggered flip-flops are also called as **dynamic triggered flip-flops.**



(1E14) **Fig. 7.10.2(a) : Positive edge-triggered flip-flops**



(1E15)**(b) Negative edge-triggered flip-flops**

Fig. 7.10.2 : Edge-triggered flip-flops

Module 3

7.11 TRUTH TABLE, CHARACTERISTIC TABLE, CHARACTERISTIC EQUATIONS & EXCITATION TABLE OF FLIP-FLOPS

Q. 7.11.1 Define : Truth table.

Ans. :

Truth Table

❒ *Definition : A truth table is a table that shows all the input-output combinations of a logic circuit.*

- In a truth table, all the input combinations of binary 0's and 1's are listed in numerical order.
- Then output corresponding to every input combination is also listed; i.e., a truth table shows how a logic circuit's output will react to the different input combinations.

Q. 7.11.2 Define characteristic equation and characteristic table.

Ans. :

Characteristics Equation

❒ *Definition : The characteristic equation of a flip-flop provides the relation between the next and present state of the flip-flop.*

Characteristic Table

❒ *Definition : Characteristic table lists the present state input excitations and next state of the flip-flop.*

- In order to obtain the characteristic equation of a flip-flop, we write the characteristic table, draw a K-map for next state of the flip-flop in terms of the present state and inputs and simplify it.

Q. 7.11.3 What do you understand by an excitation and excitation table ?

Ans. :

Excitation

❒ *Definition : An input signal that controls the flip-flop to modify its state is called as an excitation.*

Excitation Table

❒ *Definition : An excitation table shows the excitations that are needed to change the flip-flops present state to the next state.*

An excitation table is also called as **transition table** or **application table.** It is obtained from the truth table.

Q. 7.11.4 Differentiate between excitation table and truth table.

Ans. :

Sr. No.	Excitation table	Truth table
1.	An excitation table shows the excitations that are needed to change the flip-flop's present state to the next state.	A truth table is a table that shows all the input-output combinations of a logic circuit.
2.	An excitation table is also called as transition table or application table. It is obtained from the truth table.	A truth table shows how a logic circuit's output will react to the different input combinations.

7.12 CLOCKED SR FLIP-FLOP

Q. 7.12.1 Draw and explain SR flip flop using NAND gates.

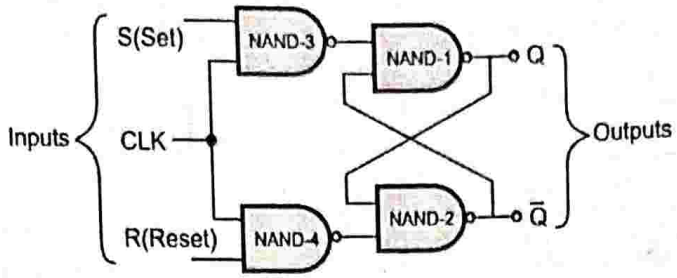
Ans. :

(A) Description

- It is also called as **gated SR latch.** It needs a Enable (EN) or clock (CLK) input.
- Only when the EN(CLK) is HIGH, the S and R inputs will change their state.
- It is also called as **synchronous SR latch.**
- If the EN(CLK) is low, there will be no state change. The flip-flop changes its state only when the clock is HIGH. Hence, they are called **level-triggered flip-flops.**

(B) Block diagram

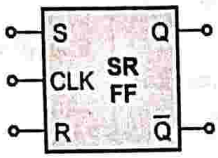
Fig. 7.12.1 shows the logic diagram of clocked S-R flip-flop.



Clocked S-R Flip-flop (Gated SR latch)

(1E16)**Fig. 7.12.1**

(C) Symbol



(1E17)**Fig. 7.12.1(a) : Logic symbol of clocked SR flip-flop**

(D) Truth table

Table 7.12.1 : Truth table of clocked SR flip-flop

EN (CLK)	S	R	Q_n	Q_{n+1}	$\bar{Q}_{n+1}$	State
1	0	0	0	0 } Q_n	1	No change NC)
1	0	0	1	1 }	0	
1	0	1	0	0 } 0	1	RESET
1	0	1	1	0 }	1	
1	1	0	0	1 } 1	0	Set
1	1	0	1	1 }	0	
1	1	1	0	x } x	x	Indeterminate (invalid)
1	1	1	1	x }	x	
0	x	x	0	0 } Q_n	1	No change (NC)
0	x	x	1	1 }	0	

≡

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	x

Module 3

(E) Operation

- We will analyze the clocked S-R flip-flop for different input/output combinations, which are as follows :

Sr. No.	State	S(SET)	R(RESET)
(1)	Hold	0	0
(2)	RESET	0	1
(3)	SET	1	0
(4)	INDETERMINATE	1	1

1. Hold state : S (SET) = 0 and R (RESET) = 0

- This state is normal resting state of the S-R latch.
- The latch will remain in the state it was before the occurrence of this input.
- When S (SET) and R(RESET) both are at 'logic 0' (low state), the state of output does not change.
- The output remains latched in its previous state. Hence, it is called as **inactive state as there is no change in state.**
- Thus, for **Hold state** $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$

2. RESET state : S (SET) = 0, R (RESET) = 1.

- When S(SET) = 0 and R(RESET) = 1, **the output of latch will always be RESET (Q = 0).**

For RESET state : $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$

3. SET state : S (SET) = 1, R (RESET) = 0

Thus, when S(SET) = 1 and R(RESET) = 0, the output Q of latch will always be SET (Q = 1).

For SET state : $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$

4. INDETERMINATE state
S (SET) = 1 and R (RESET) = 1

- When S(SET) = 1 and R(RESET) = 1, the output of the latch is unpredictable. This condition must be avoided. It is called as **invalid or indeterminate condition.**

For INDETERMINATE state : $Q_{n+1} = x$, $\bar{Q}_{n+1} = x$

Summary of operation

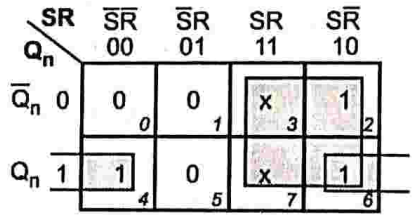
Sr. No.	State	Output	
		Q_{n+1}	$\bar{Q}_{n+1}$
(1)	Hold	Q_n	$\bar{Q}_n$
(2)	RESET	0	1
(3)	SET	1	0
(4)	INDETERMINATE	x	x

(F) Characteristic Equation

Table 7.12.2 : Characteristic Table (Truth table) for SR flip-flop

Present state	Inputs		Next state
Q_n	S	R	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	x
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	x

K-map Simplification for S-R flip-flop



$$Q_{n+1} = S + \bar{R}\,Q_n$$

(1E60)**Fig. 7.12.2 : K-map for S-R flip-flop**

Characteristics equation

The characteristic equation for SR flip-flop is,

$$Q_{n+1} = S + \bar{R}\,Q_n$$

(G) Excitation Table of SR Flip-Flop

Q. 7.12.2 Draw the excitation table of S-R flip-flop.

Ans. :

- Table 7.12.3 shows the excitation table of SR flip-flop.

Table 7.12.3 : Excitation table of SR flip-flop

Present state	Next state	Required inputs	
Q_n	Q_{n+1}	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

➪ **Step I** : Before time t_0, the output Q does not change because the EN or Clock (CLK) is LOW.

➪ **Step II** : At time instant $t = t_0$, S = 0, R = 1, CLK = 1 will reset the flip-flop output to Q = 0 and $\bar{Q} = 1$.

➪ **Step III** : At time instant $t = t_1$, S = 1, R = 0, CLK = 1 will set the flip-flop changing its output to Q=1 and $\bar{Q} = 0$. Before time t_2, though S = 0, R = 1, the output Q and $\bar{Q}$ does not change because Clock (CLK) is LOW.

➪ **Step IV** : At time instant $t = t_2$, S = 0, R = 1, CLK = 1 will reset the flip-flop output Q = 0 and $\bar{Q} = 1$.

➪ **Step V** : At time instant $t = t_3$, S = 1, R = 0, CLK = 1 will set the flip-flop output Q = 1 and $\bar{Q} = 0$.

Q. 7.12.3 Fill in values for S & R to cause the Q values of the SR FF given in Fig. 7.12.4.

	t_0	t_1	t_2	t_3
S	0			
R	0			
Q	1	0	0	1

Fig. 7.12.4

☑ **Ans. :**

➪ **Step I :** At $t = t_0$, S = 0, R = 0.

The SR flip-flop is in hold state with output Q = 1.

➪ **Step II :** At $t = t_1$,

The output Q = 0.

The transition of output from 1 → 0 is when S = 0, R = 1 (Reset condition).

$\therefore$ **S = 0, R = 1 at $t = t_1$**

➪ **Step III :** At $t = t_2$,

The output Q = 0.

The transition of output from 0 → 0 is when S = 0, R = 0 (no change) state or S = 0, R = 1 (Reset condition).

$\therefore$ **S = 0, R = x at $t = t_2$**

➪ **Step IV :** At $t = t_3$,

The output Q = 1.

The transition of output from 0 → 1 is when S = 1, R = 0 (Set condition).

$\therefore$ **S = 1, R = 0 at $t = t_3$**

	t_0	t_1	t_2	t_3
S	0	0	0	1
R	0	1	x	0
Q	1	0	0	1

Flip-flop Reset **Flip-flops Set**

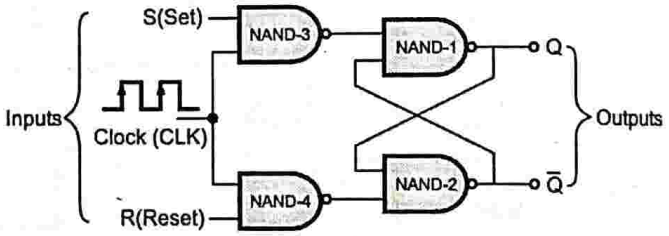
⏭ 7.13 POSITIVE EDGE-TRIGGERED SR FLIP-FLOP

Q. 7.13.1 Draw the logic diagram of a positive edge-triggered SR Flip-Flop and describe it's working.

☑ **Ans. :**

☞ **(A) Block Diagram**

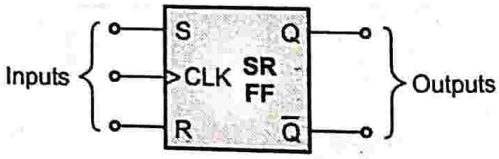
Block diagram of a positive edge-triggered S-R flip-flop is shown in Fig. 7.13.1.



(1E19)Fig. 7.13.1 : Clocked SR flip-flop (positive edge-triggered)

☞ **(B) Symbol**

Fig. 7.13.2 shows the symbol of positive edge-triggered SR flip-flop.



(1E20)Fig. 7.13.2 : Logic symbol of positive edge-triggered SR flip-flop

☞ **(C) Truth Table**

Table 7.13.1 shows the truth table of positive edge-triggered flip-flop.

Table 7.13.1 : Truth table of positive edge-triggered flip-flop

Inputs			Outputs		State
Clock	S	R	Q_n	Q_{n+1}	
↑	0	0	0	0	No Change (NC) or Hold
↑	0	0	1	1	
↑	0	1	0	0	RESET
↑	0	1	1	0	
↑	1	0	0	1	SET
↑	1	0	1	1	
↑	1	1	0	x	Indeterminate (invalid)
↑	1	1	1	x	
0	x	x	0	0	No Change (NC)
0	x	x	1	1	

(D) Operation

- The S(SET) and R(RESET) inputs are called as **synchronous control inputs**.
- Only when the SR flip-flops clock is positive edge-triggered, the output of the flip-flop will change.
- The S and R inputs cannot alter the flip-flops output in the absence of the clock pulse.
- We will analyze the positive edge-triggered S-R flip-flop for different input/output combinations, which are as follows :

Sr. No.	State	S(SET)	R(RESET)	Clock
(1)	Hold	0	0	↑
(2)	RESET	0	1	↑
(3)	SET	1	0	↑
(4)	INDETERMINATE	1	1	↑

1. Hold State : S = 0, R = 0, clock ↑

If S = 0, R = 0 and positive edge-triggered clock pulse is applied, the output of the flip-flop remains in its previous state (i.e. $Q_{n+1} = Q_n$). **There is no change in the flip-flop's output.**

- Thus , for hold state $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$

2. RESET State : S = 0, R = 1, clock ↑

If S = 0, R = 1, the output Q becomes LOW on positive edge of the clock pulse.

The SR flip-flop is RESET (cleared Q = 0).

For RESET state : $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$

3. SET State : S = 1, R = 0, clock ↑

If S = 1, R = 0, the output Q becomes HIGH on the positive edge of the clock pulse. The SR flip-flop is **SET (Q = 1).**

For SET state : $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$

4. INDETERMINATE state

S = 1, R = 1, clock ↑

- IF S = 1, R = 1, and positive edge-triggered clock pulse is applied, the output of the flip-flop is unpredictable.
- This condition is indeterminate (invalid) condition and must be avoided.

For INDETERMINATE state : $Q_{n+1} = x$, $\bar{Q}_{n+1} = x$

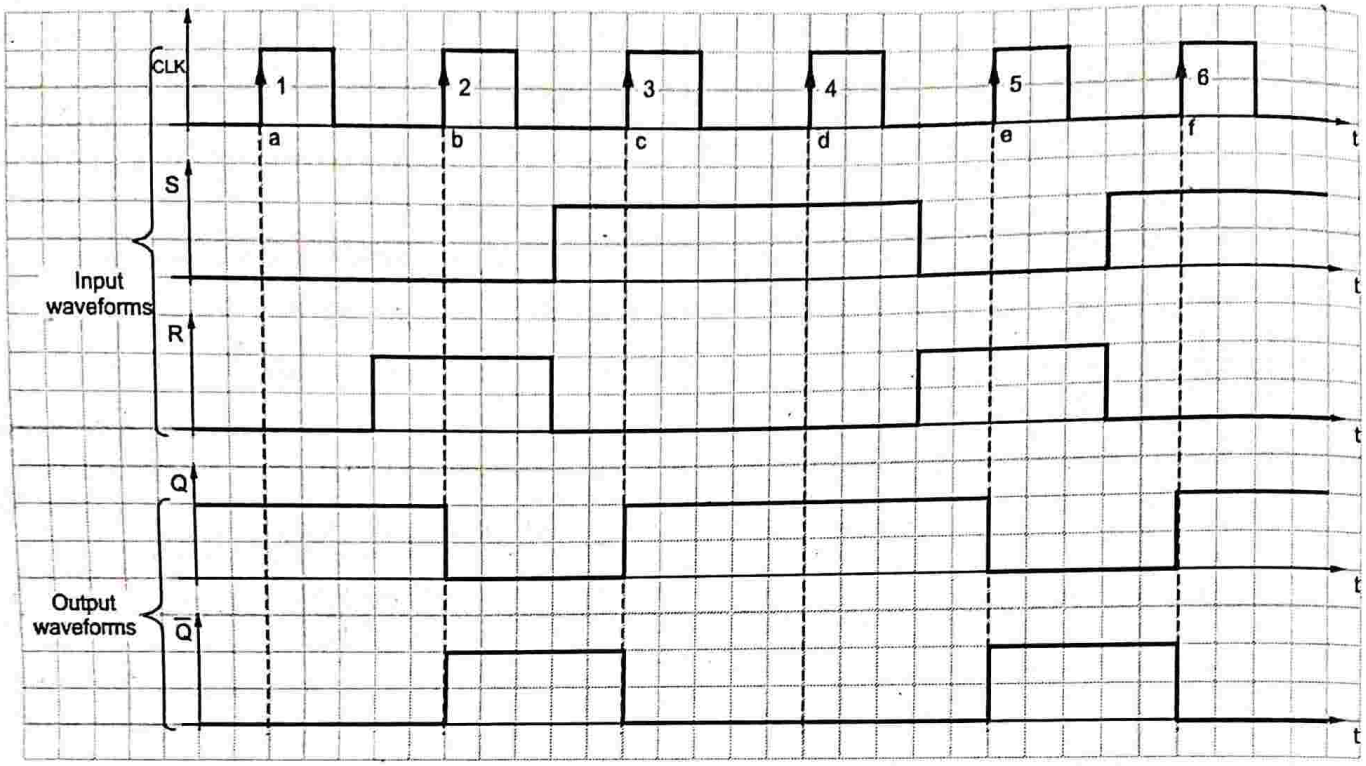
Module 3

Summary of operation

Sr. No.	State	Output	
		Q_{n+1}	$\bar{Q}_{n+1}$
(1)	Hold	Q_n	$\bar{Q}_n$
(2)	RESET	0	1
(3)	SET	1	0
(4)	INDETERMINATE	x	x

(E) Timing diagram

Fig. 7.13.3 shows the input and output waveforms of positive edge triggered SR flip-flop.



(1E21)**Fig. 7.13.3 : Input and output waveforms of positive edge-triggered SR flip-flop**

➪ **Step I :** Initially we assume S = 0, R = 0, Q = 1. At time instant t = a , when the first positive edge clock pulse is applied, S = 0, R = 0. Hence, the output of flip-flop will remain in its previous state with Q = 1, $\bar{Q}$ = 0 till the second clock pulse is applied.

↓

➪ **Step II :** At time instant t=b , when the second positive edge clock pulse is applied, S = 0, R = 1. Hence, the S-R flip-flop will RESET causing Q = **0** (low) and $\bar{Q}$ = **1** (high).

↓

➪ **Step III :** At time instant t=c , when the third positive edge-triggered clock pulse is applied, S = 1, R = 0 will set the flip-flop. Output Q will be **1**(high) and $\bar{Q}$ will be **0** (low).

↓

➪ **Step IV :** At time instant t=d , when the fourth positive edge clock pulse is applied, S = 1, R = 0 will set the flip-flop. Output Q will be **1**(high) and $\bar{Q}$ will be **0**·(low).

↓

➪ **Step V :** At time instant t = e, when the fifth positive edge clock pulse is applied, S = 0, R = 1. The flip-flop will be Reset. The output Q = **0**(low) and $\bar{Q}$ = **1** (high).

↓

➪ **Step VI :** At time instant t= f , when the sixth positive edge clock pulse is applied, S = 1, R = 0. The flip-flop will be set and the output Q = **1** (high) and $\bar{Q}$ = **0** (low), as shown in Fig. 7.13.3.

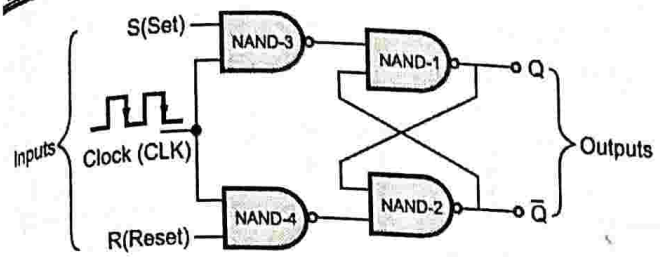
⏭ 7.14 NEGATIVE EDGE-TRIGGERED SR FLIP-FLOP

Q. 7.14.1 *Draw logic diagram of S-R flip-flop with negative edge-triggering and write its truth table.*

☑ **Ans. :**

☞ **(A) Block diagram**

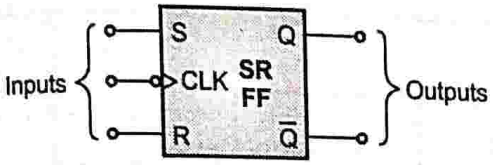
Block diagram of a negative edge-triggered S-R flip-flop is shown in Fig. 7.14.1.



(1E19)**Fig. 7.14.1 : Clocked SR flip-flop (negative edge-triggered)**

(B) Symbol

Symbol of a negative edge-triggered S-R flip-flop is shown in Fig. 7.14.2.



(1E22)**Fig. 7.14.2 : Logic symbol of negative edge-triggered SR flip-flop**

(C) Truth Table

Truth table of a negative edge-triggered S-R flip-flop is shown in Table. 7.14.1.

Table 7.14.1 : Truth table of negative edge-triggered SR flip-flop

Inputs			Outputs			State
Clock	S	R	Q_n	Q_{n+1}		
↓	0	0	0	0	Q_n	No Change (NC)
↓	0	0	1	1		
↓	0	1	0	0	0	RESET
↓	0	1	1	0		
↓	1	0	0	1	1	SET
↓	1	0	1	1		
↓	1	1	0	x	x	Indeterminate (invalid)
↓	1	1	1	x		
0	x	x	0	0		No Change (NC)
0	x	x	1	1		

(D) Operation

- Only when the SR flip-flops clock is negative edge-triggered, the output of flip-flop will change. The S and R inputs cannot alter the flip-flops output in absence of the clock pulse.
- Table 7.14.1 shows the truth table of negative edge-triggered flip-flop. The flip-flop will trigger on the falling edge of the clock pulse, i.e., when the clock pulse goes from 1 to 0.
- We will analyze the positive edge-triggered S-R flip-flop for different input/output combinations, which are as follows :

Sr. No.	State	S(SET)	R(RESET)	Clock
(1)	Hold	0	0	↓
(2)	RESET	0	1	↓
(3)	SET	1	0	↓
(4)	INDETERMINATE	1	1	↓

1) Hold State : S = 0, R = 0, clock ↓

- If S = 0, R = 0 and negative edge-triggered clock pulse is applied, the output of the flip-flop remains in its previous state (i.e., $Q_{n+1} = Q_n$).
- There is no change in the flip-flop's output.
- Thus , for **hold state** $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$

2) RESET State : S = 0, R = 1, clock ↓

If S = 0, R = 1, the output Q becomes LOW (flip-flop is reset) on the negative (falling) edge of the clock pulse. The SR flip-flop is **RESET (cleared Q = 0).** Thus, for **RESET state** $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$

3) SET State : S = 1, R = 0, clock ↓

- If S = 1, R = 0, the output Q becomes 1 (HIGH) on the negative edge of the clock pulse.
- The SR flip-flop is **SET (Q = 1).**
- Thus , for **SET state** $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$

4) Indeterminate State : S = 1, R = 1, clock ↓

- If S = 1, R = 1 and negative edge-triggered clock pulse is applied, the output of the flip-flop is unpredictable.
- This condition is indeterminate (invalid) condition and must be avoided.

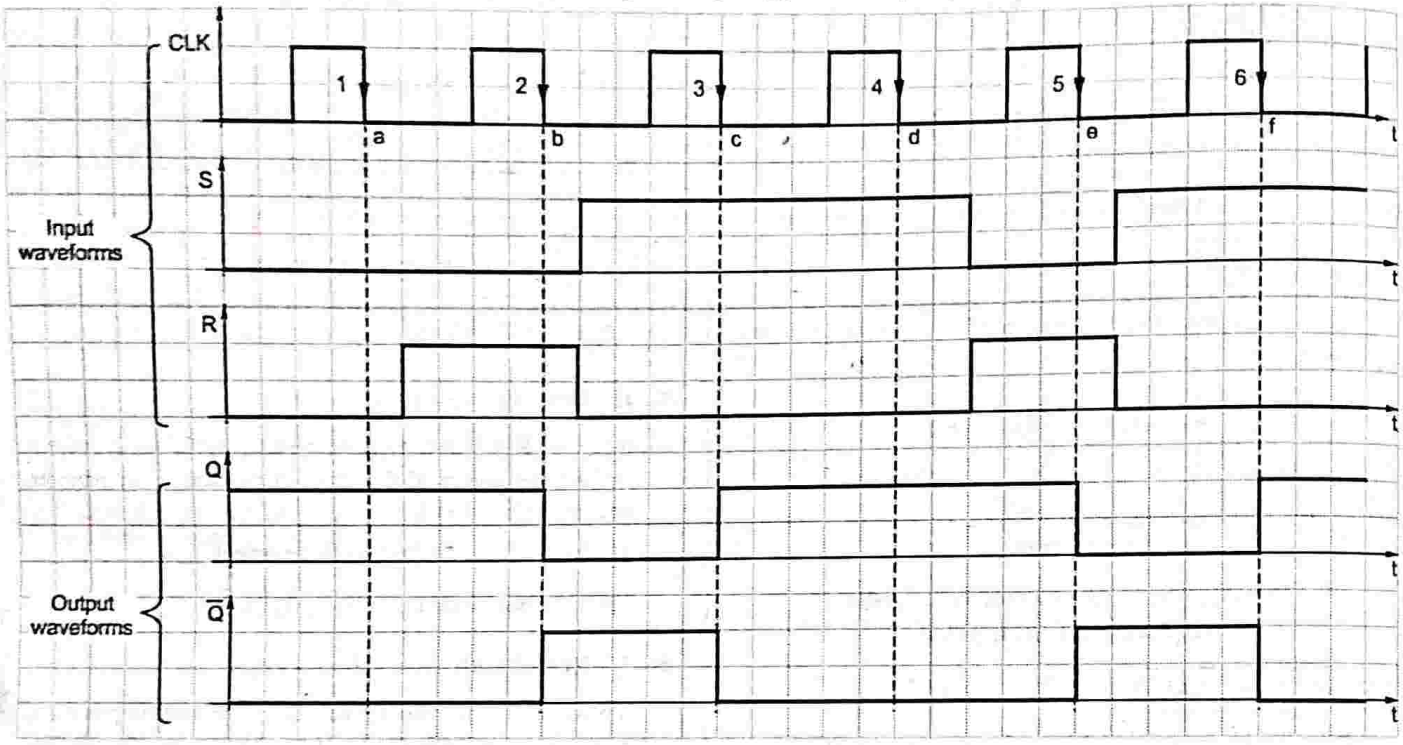
For INDETERMINATE state : $Q_{n+1} = x$, $\bar{Q}_{n+1} = x$

Sr. No.	State	Output	
		Q_{n+1}	$\bar{Q}_{n+1}$
(1)	Hold	Q_n	$\bar{Q}_n$
(2)	RESET	0	1
(3)	SET	1	0
(4)	INDETERMINATE	x	x

Module 3

(E) Timing Diagram

Fig. 7.14.3 shows the input and output waveforms of negative edge-triggered SR flip-flop.



(1E23)**Fig. 7.14.3 : Input and output waveforms of negative edge-triggered SR flip-flop**

- ***Step I :*** Initially we assume S = 0, R = 0, Q = 1. At time instant t = a, when the first clock pulse is applied at the falling edge, S = 0, R = 0. The output of flip-flop will remain in previous state Q = 1, $\overline{Q}$ = 0 till the falling edge of second clock pulse.
- ***Step II :*** At time instant t= b , on the falling edge of second clock pulse, S = 0, R = 1. The output of the flip-flop will be Reset. Q = 0 and $\overline{Q}$ = 1.
- ***Step III :*** At time instant t=c , on the falling edge of third clock pulse, S = 1, R = 0. The output of flip-flop will be set. Q = 1 and $\overline{Q}$ = 0.
- ***Step IV :*** At time instant t= d , on the falling edge of fourth clock pulse, S = 1, R = 0. The flip-flop output will be set. Q = 1, $\overline{Q}$ = 0.
- ***Step V :*** At time instant t=e , on the falling edge of fifth clock pulse, S = 0, R = 1 will reset the flip-flop output Q = 0 and $\overline{Q}$ = 1.
- ***Step VI :*** At time instant t= f , on the falling edge of sixth clock pulse, S = 1, R = 0, will set the flip-flop output Q = 1 and $\overline{Q}$ = 0, as shown in Fig. 7.14.3.

7.15 PRESET AND CLEAR

Q. 7.15.1 Explain the need of preset and clear pins.

Q. 7.15.2 What are PRESET and CLEAR inputs?

Ans. :

- When the S-R flip-flop is powered on, the state of the outputs Q and $\overline{Q}$ is uncertain. The outputs can be (Q = 0, $\overline{Q}$ = 1) or vice-versa.
- However, in some applications it is essential to set or reset the flip-flops; i.e., an initial state must be assigned to a flip-flop.
- It can be obtained with the help of **preset (PR) and clear (CR) inputs.** PR and CR are **asynchronous or direct inputs that can be applied to the SR flip-flop at any instant of time between the clock pulses irrespective of the synchronization with clock.**
- The preset and clear inputs override all other inputs.
- PR is also called as **DC SET** or **Direct Set (S_D).** CR is called as **DC RESET** or **DC CLEAR** or **Direct Reset (R_D).**
- The preset input forces output Q = 1 (high) and clear input forces Q = 0 (low).

Q. 7.15.3 *If $\overline{PR}$ = 1 and $\overline{CR}$ = 1, can a flip-flop respond to its control and clock inputs?*

Ans. :

- PR and CR are both **active-low inputs**. When $\overline{PR}$ = 1 and $\overline{CR}$ = 1, a flip-flop can respond to its control and clock inputs.
- With $\overline{PR}$ = 1 and $\overline{CR}$ = 1, the preset and clear inputs do not affect the flip-flop operation.

Q. 7.15.4 *Why are asynchronous inputs called overriding inputs ?*

Ans. :

- The **asynchronous inputs** PRESET (PR) and CLEAR (CR) are called as **overriding inputs,** as they override the control inputs.
- It indicates that in the presence of the asynchronous inputs the other inputs become ineffective.

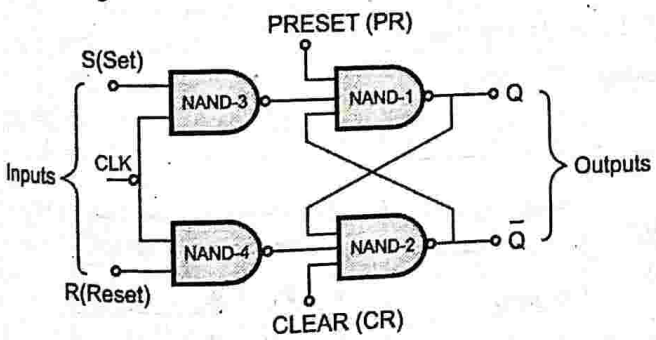
7.16 SR FLIP-FLOP WITH PRESET AND CLEAR

Q. 7.16.1 *Explain preset and clear pins in RS flip flop ?*

Ans. :

(A) Circuit Diagram

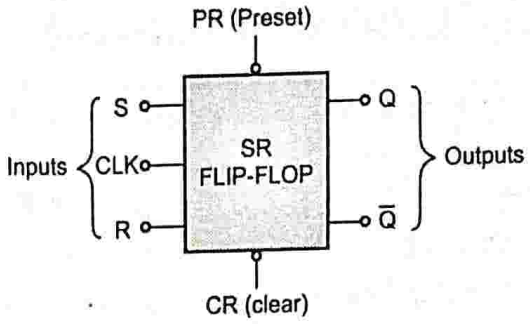
Circuit diagram of S-R flip-flop with preset and clear is shown in Fig. 7.16.1.



(1E24)**Fig. 7.16.1 : SR flip-flop with preset and clear**

(B) Logic Symbol

Logic symbol of SR flip-flop with preset and clear is shown in Fig. 7.16.2.



(1E25)**Fig. 7.16.2 : Logic symbol of SR flip-flop with preset and clear**

- Preset (PR) and Clear (CR) are active-low inputs, indicated by bubbles as shown in Fig. 7.16.2.

(C) Operation

1. Normal SR flip-flop : PR = 1, CR = 1

If PR = 1, CR = 1, the SR flip-flop operates like standard SR flip-flop.

2. SET state : PR = 0, CR = 1

- If PR = 0, CR = 1, the output of NAND1 will be 'logic 1' (Q = 1). All the inputs to NAND2 gate will be '1'.
- Hence, output of NAND2 will be 'logic 0' ($\overline{Q}$ = 0). Thus **PR = 0, CR = 1 will set the flip-flop.**

3. RESET state : PR = 1, CR = 0

- If CR = 0, the output of NAND2 gate will be 'logic 1' ($\overline{Q}$ = 1).
- All the inputs to NAND1 gate will be '1'.
- Hence, the output of NAND1 gate will be 'logic 0' (Q = 0). Thus, **PR = 1, CR = 0 will reset the flip-flop.**

4. Indeterminate state : PR = 0, CR = 0

PR = 0 and CR = 0 condition leads to uncertain or indeterminate state of outputs, Q and $\overline{Q}$. Hence, it should be avoided.

Table 7.16.1 : Operation of SR flip-flop

Inputs			Outputs	Operation
CLK	PR	CR	Q_{n+1}	
1	1	1	Q_{n+1}	Normal SR flip flop
x	0	1	1	Flip-flop is set
x	1	0	0	Flip-flop is reset

Module 3

7.17 CLOCKED JK FLIP-FLOP

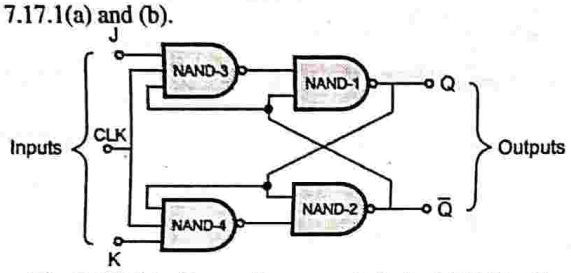
UQ. 7.17.1 Draw the logic diagram of a clocked JK flip-flop. Describe it's working.
MU - Q. 5(b), Dec. 19, 5 Marks

UQ. 7.17.2 Draw JK flip-flop using SR flip-flop and additional gates.
MU - Q. 1(e), Dec. 16, 3 Marks

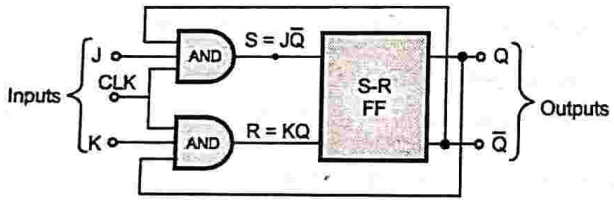
Ans. :

(A) Circuit Diagram

Circuit diagram of clocked JK flip-flop is shown in Figs. 7.17.1(a) and (b).



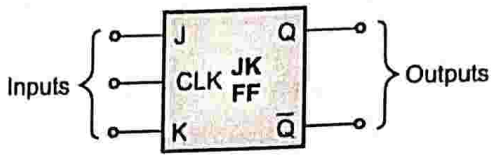
(1E26)**Fig. 7.17.1(a) : Logic diagram of clocked JK flip-flop**



(1E27)**Fig. 7.17.1(b) : Logic diagram of clocked JK flip-flop using SR flip-flop**

(B) Logic Symbol

Logic symbol of clocked JK flip-flop is shown in Fig. 7.17.1(c).



(1E28)**Fig. 7.17.1(c) : Logic symbol of clocked JK flip-flop**

(C) Description

- It is a modification of SR flip-flop. In JK flip-flop, the letter **J stands for SET** and **K stands for RESET**. The inputs J and K behave as S and R inputs to the SR flip-flop.
- The unpredictable state S = 1, R = 1 is defined in the JK flip-flop as toggle state (J = 1, K = 1).
- It is also called as JK latch or level-triggered JK flip-flop.
- J and K inputs are ANDed with outputs $\bar{Q}$ and Q to get the S and R inputs.

$$S = J \cdot \bar{Q}$$

$$R = K \cdot Q$$

(D) Truth table

Table 7.17.1 : Truth table of clocked JK flip-flop

Inputs			Outputs			Inputs to SR Flip-flop		State
EN (CLK)	J	K	Q_n	$\bar{Q}_n$	Q_{n+1}	S	R	
1	0	0	0	1	0 } Q_n	0	0	No Change (NC)
1	0	0	1	0	1 } Q_n	0	0	
1	0	1	0	1	0 } 0	0	0	RESET
1	0	1	1	0	0 } 0	0	1	
1	1	0	0	1	1 } 1	1	0	SET
1	1	0	1	0	1 } 1	0	0	
1	1	1	0	1	1 } $\bar{Q}_n$	1	0	Toggle
1	1	1	1	0	0 } $\bar{Q}_n$	0	1	
0	x	x	0	1	0	0	0	No Change (NC)
0	x	x	1	0	1	0	0	

≡

Inputs		Outputs
J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\bar{Q}_n$

(E) Operation

The operation of the J-K flip-flop is similar to the operation of SR flip-flop.

We will analyze the operation of clocked JK flip-flop for different input/output combinations, which are as follows :

Sr. No.	State	J(SET)	K(RESET)
(1)	Hold	0	0
(2)	RESET	0	1
(3)	SET	1	0
(4)	TOGGLE	1	1

1. Hold state : J = 0, K = 0, CLK = 1

Initially Q = 0, $\bar{Q} = 1$, J = 0, K = 0

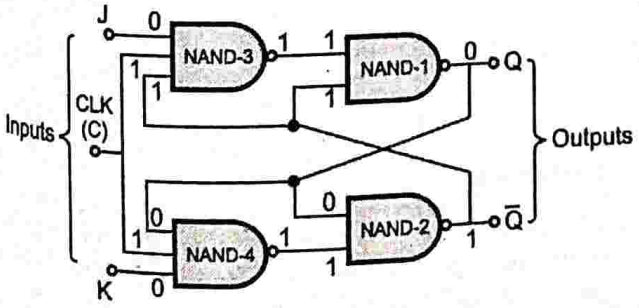
S = $J\bar{Q} = 0.1 = 0$

R = $KQ = 0.0 = 0$

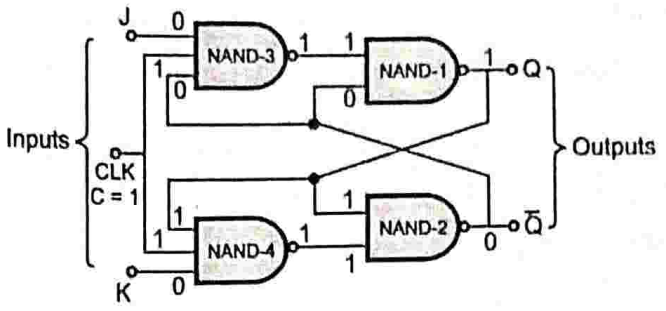
- When J = 0, $\bar{Q}$ = 1, CLK = 1 the output of NAND-3 gate will be 1.
- The two inputs to the NAND-1 gate are at logic 1. Hence, the output of NAND-1 is Q = 0.
- When K = 0, Q = 0, CLK = 1, the output of NAND-4 gate will be logic 1.
- The Q input to NAND-2 gate is at logic 0, resulting in output $\bar{Q}$ = 1; i.e., there is no change in the state of output when J = K = 0 (Fig. 7.17.2(a)).

Initially if Q = 1, $\bar{Q} = 0$.

- When J = 0, $\bar{Q}$ = 0, CLK = 1, the output of NAND-3 gate will be logic 1.
- The $\bar{Q}$ input to the NAND-1 gate is at logic 0, resulting in output Q = 1(high).
- When K = 0, Q = 1, CLK = 1, the output of NAND-4 gate will be logic 1. Both the inputs to NAND-2 gate are at logic 1.
- Hence, the output of the NAND-2 gate is $\bar{Q}$ = 0; i.e., there is no change in the output state. (Fig. 7.17.2(b)).



(1E29)**Fig. 7.17.2(a) : J = 0, K = 0, Q = 0, $\bar{Q}$ = 1 (Hold state)**



(1E30)**Fig. 7.17.2(b) : J = 0, K = 0, Q = 1, $\bar{Q}$ = 0 (Hold state)**

- Thus, when J = K = 0, CLK = 1, there is no change in the state of output. The output remains in its previous state.

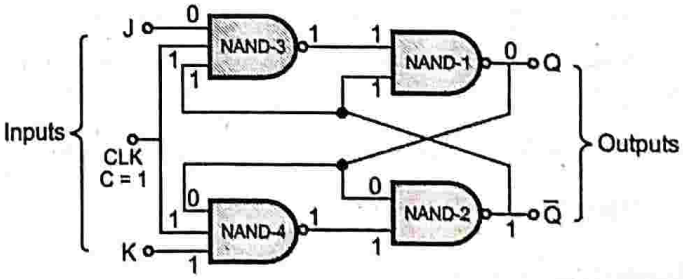
Thus, for HOLD state $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$

2. RESET state : J = 0, K = 1, CLK = 1

- When J = 0, CLK = 1, irrespective of the value of $\bar{Q}$, the output of NAND 3 gate will be logic 1. If both the inputs to NAND-1 gate are at logic 1, the output $\bar{Q}$ = 0.
- If K = 1, Q = 0, CLK = 1, the output of NAND-4 will be logic 1.
- As the input to NAND-1 gate is at logic 0, the output of NAND-2 gate will be logic 1 ($\bar{Q}$ = 1), as shown in Fig. 7.17.2(c).

Thus, when J = 0, K = 1, CLK = 1, the flip-flop resets (Q = 0).

Thus, for RESET state $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$.



(1E31)**Fig. 7.17.2(c) : J = 0, K = 1, CLK = 1 (RESET state)**

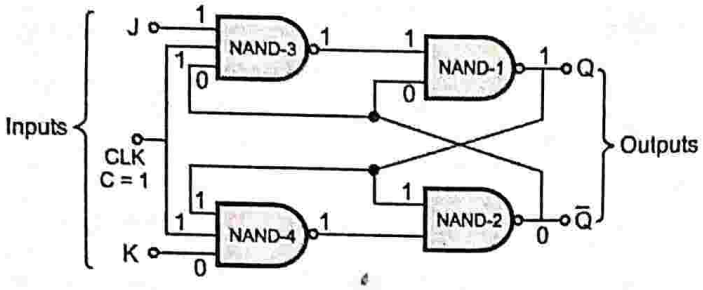
3. SET state : J = 1, K = 0, CLK = 1.

- When K = 0, CLK = 1, the output of NAND 4 will be logic 1. If both the inputs to NAND 2 gate are at logic 1, the output $\bar{Q}$ = 0.
- If J = 1, CLK = 1, $\bar{Q}$ = 0, the output of NAND 3 gate will be logic 1. The $\bar{Q}$ input to NAND 1 gate is at logic 0. Hence, output of NAND-1 gate is Q = 1, i.e., J K flip-flop is set.

Module 3

Thus, when J = 1, K = 0, CLK = 1, the flip flop is set (Q = 1).

Thus, for SET state $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$



(1E32)**Fig. 7.17.2(d) : J = 1, K = 0, C = 1 (SET)**

4. TOGGLE state : J = 1, K = 1, CLK = 1

- When clock pulse is applied, let Q = 0, $\bar{Q}$ = 1. As J = 1, CLK = 1, $\bar{Q}$ = 1, the output of NAND-3 gate will be logic **0**. As one input to NAND-1 gate is logic **0**, the output NAND-1 gate is logic **1 (Q = 1)**.
- If K = 1, Q = 0, CLK = 1, the output of NAND-4 gate will be logic **1**. As both the inputs to NAND-2 gate are logic **1**, the output of NAND-2 gate is $\bar{Q} = 0$.
- When clock pulse is applied, let Q = 1, $\bar{Q} = 0$.
- As K = 1, CLK = 1, Q = 1, the output of the NAND-4 gate will be logic **0**. As one input to the NAND-2 gate is logic **0**, its output is $\bar{Q} = 1$.
- As J = 1, CLK = 1, $\bar{Q} = 0$, the output of NAND 3 gate will be logic **1**. As both the inputs to the NAND 1 gate are at logic 1, the output of the NAND 1 gate is **Q = 0**.

Thus, when J = K = 1, CLK = 1, the JK flip-flop toggles.

Thus, for TOGGLE state $Q_{n+1} = \bar{Q}_n$, $\bar{Q}_{n+1} = Q_n$

Summary of operation

Sr. No.	State	Output	
		Q_{n+1}	$\bar{Q}_{n+1}$
(1)	Hold	Q_n	$\bar{Q}_n$
(2)	RESET	0	1
(3)	SET	1	0
(4)	TOGGLE	$\bar{Q}_n$	Q_n

(F) Characteristic Equation

Table 7.17.2 : Characteristic table for JK flip-flop

Present state	Inputs		Next state
Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

K-map simplification for clocked JK flip-flop

Q_n \ JK	$\bar{J}\bar{K}$ 00	$\bar{J}K$ 01	JK 11	$J\bar{K}$ 10
$\bar{Q}_n$ 0	0	0	1	1
Q_n 1	1	0	0	1

$$Q_{n+1} = J\bar{Q}_n + \bar{K}_n Q_n$$

(1E61)**Fig. 7.17.3 : K-map for clocked JK flip-flop**

Characteristic equation for clocked JK flip-flop :

$$Q_{n+1} = J\bar{Q}_n + \bar{K}_n Q_n$$

(G) Excitation Table

Q. 7.17.3 Draw excitation table of JK flip flop.

Ans. :

Table 7.17.3 : Excitation table of JK flip-flop

Present state	Next state	Required inputs	
Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

0 → 0 transition

- If the present state of the flip-flop is 0 and next state is to be **0**, then the inputs can be J = 0, K = 0 or J = 0, K = 1 (reset condition). J = 0 and K = 0 or 1.
- Hence, **JK = 0x for 0 → 0 transition.**

0 → 1 transition

- If the present state of the flip-flop is 0 and next state is to be 1, then the inputs can be J = 1, K = 0 (set condition) or J = K = 1 (toggle condition). J = 1 but K = 0 or 1.
- Hence, **JK = 1x for 0 → 1 transition.**

1 → 0 transition

- If the present state of the flip-flop is 1 and next state is to be 0, then the inputs can be J = 0, K = 1 (reset condition) or J = K = 1 (toggle condition). J = 0 or 1, K = 1.
- Hence **JK = x 1 for 1 → 0 transition.**

1 → 1 transition

If the present state of the flip-flop is 1 and the next state is to be 1, then the inputs can be either J = 0, K = 0 (no change condition) or J = 1, K = 0 (set condition). K = 0, J = 0 or 1.

∴ JK = x 0 for 1 → 1 transition

Q. 7.17.4 What do you understand by toggling ?

☑ **Ans. :**

On every clock pulse, changing the output to its opposite state or complementing the output is called as **toggling.**

7.17.1 Race Around Condition

UQ. 7.17.5 Explain the race around condition in JK flip-flop. State various methods to overcome it.

MU - Q. 1(h) Dec. 15, 2 M, Q. 1(e), Dec. 16, 5 M

UQ. 7.17.6 What is race around condition ? How to overcome it ?

MU - Q. 3(a) Dec. 17, 10 Marks

UQ. 7.17.7 Write short note on : Race around condition.

MU - Q. 6(c) May 18, 5 Marks

☑ **Ans. :**

- When J = K = 1 and clock is applied, the output will complement itself till the clock is present. Hence, at the end of clock pulse, the output is unpredictable. This condition is called as **race around condition.**
- Fig. 7.17.4 shows the timing diagram of the clocked JK flip-flop.
- Practically, the JK latch or clocked JK flip-flop cannot be used. This is because the output is fedback to the input. Hence, any change in the output will affect the input. Fig. 7.17.4 shows this.
- Before $t = t_0$, Q is HIGH. Although J = 0, K = 1, the output Q will not change because CLK is LOW.

➪ **Step 1 :** During interval $t_0 - t_1$, J = 0, K = 1, CLK = 1. The flip-flop is reset and Q = 0.

➪ **Step 2 :** During interval $t_1 - t_2$, J = 1, K = 1, CLK = 1. The flip-flop toggles.

➪ **Step 3 :** During interval $t_2 - t_3$, there will be no change in the flip-flop output even though J = K = 1 as CLK = 0.

➪ **Step 4 :** During interval $t_3 - t_4$, J = 1, K = 1, CLK = 1. The flip-flop toggles.

➪ **Step 5 :** During interval $t_4 - t_5$, J = 0, K = 0, CLK = 1. The flip-flop remains in the same state, even after CLK = 0.

➪ **Step 6 :** During interval $t_5 - t_6$, J = 1, K = 0, CLK = 1 will set the flip-flop and Q = 1.

➪ **Step 7 :** During interval $t_6 - t_7$, J = 1, K = 1, CLK = 1, the flip-flop toggles.

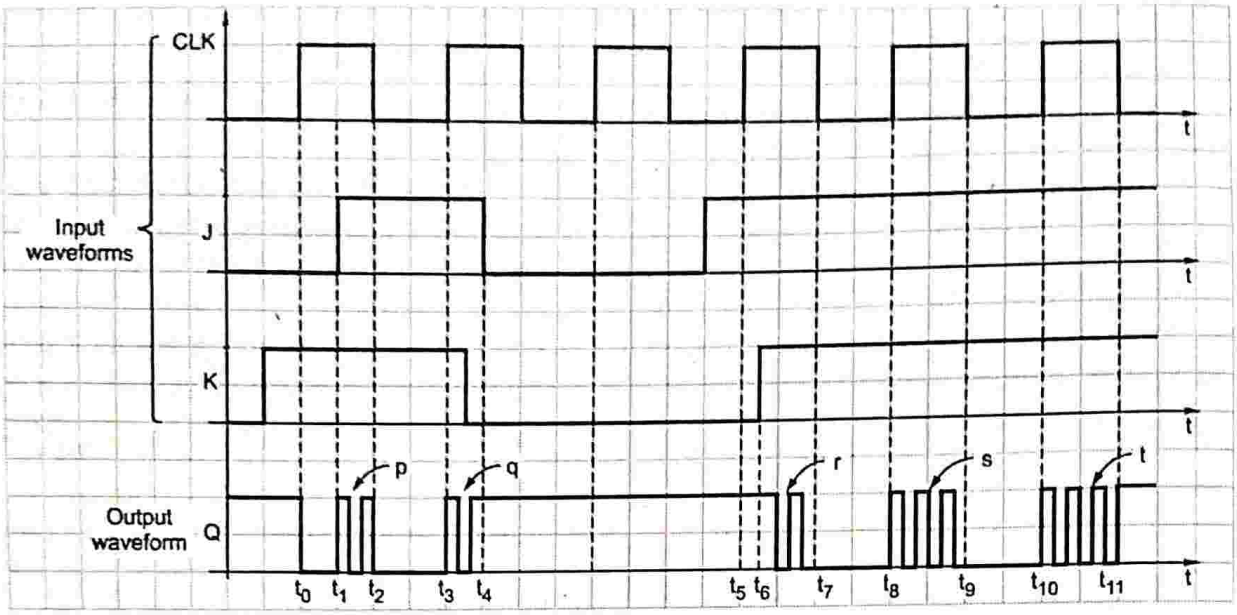
➪ **Step 8 :** During interval $t_7 - t_8$, J = 1, K = 1, CLK = 0. The flip-flop remains in the same state and Q = 0.

➪ **Step 9 :** During interval $t_8 - t_9$, J = 1, K = 1, CLK = 1. The flip-flop toggles.

➪ **Step 10 :** During interval $t_9 - t_{10}$, J = K = 1, CLK = 0. The flip-flop remains in same state Q = 0.

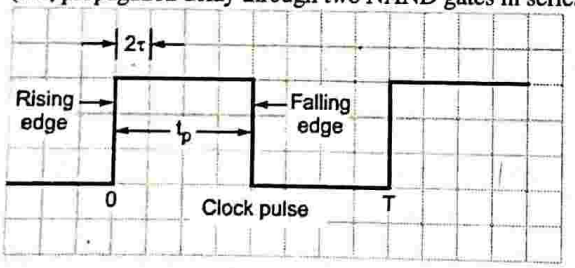
➪ **Step 11 :** During interval $t_{10} - t_{11}$, J = K = 1, CLK = 1, the flip-flop toggles.

Module 3



(1E33)**Fig. 7.17.4 : Input and output waveforms for JK flip-flop**

- During instants p, q, r, s and t as shown in Fig. 7.17.4 the output continuously toggles.
- If t_p is the clock pulse-width and t_p is long, then the output of the flip-flop continuously changes from 0 to 1, 1 to 0, 0 to 1 and so on; i.e., the output oscillates (toggles) between 0 and 1.
- At the end of the clock pulse, the output Q will be unpredictable. This condition is called as **race around condition.**
- **If τ is the propagation delay of each NAND gate and if $t_p >> \tau$, then the flip-flop output toggles itself for every 2τ** (i.e., propagation delay through two NAND gates in series).



(1E34)**Fig. 7.17.5 : Clock pulse**

☞ Methods to avoid race around condition

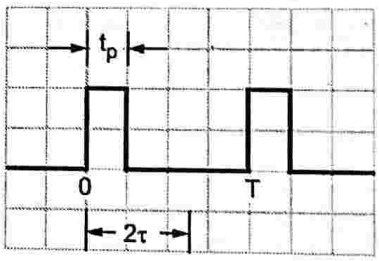
Q. 7.17.8 Explain different methods to avoid race around condition.

☑ **Ans. :** The race around condition can be avoided if :

(i) $t_p < 2\tau < T$
(ii) Using edge-triggered JK flip-flop
(iii) Using master-slave JK flip-flop

▶ (i) $t_p < 2\tau < T$ (Fig. 7.17.6)

- In order to avoid the race around condition, the pulse-width of clock pulse t_p is reduced such that $t_p < 2\tau < T$.
- The propagation delay 2τ can be increased with the help of lumped delay lines in series with the feedback. However, it is not feasible.



(1E35)**Fig. 7.17.6 : $t_p < 2\tau < T$**

▶ (ii) Using edge-triggered JK flip-flop

- In clocked JK flip-flop, the clock or enable signal is high for a long time period. Hence, the problem of racing (multiple toggling) takes place.
- In edge-triggered JK flip-flop, the rising or positive edge of the clock pulse is available for a short time. Hence, **multiple toggling will not occur.**

▶ (iii) Using master-slave JK flip-flop

- In master-slave JK flip-flop, two flip-flops are used. One flip-flop acts as a master and the second flip-flop acts like a slave.

The master flip-flop is positively clocked and the slave flip-flop is negatively clocked. It indicates that when the clock is low, master is inactive and slave is active.

When the clock is high, the master flip-flop will be active and slave flip-flop will be inactive, as it needs negative edge at its clock input. **Thus, the race around condition is avoided in master-slave JK flip-flop.**

Q. 7.17.9 Which flip-flop is most widely used flip-flop ?

☑ **Ans. :**

The **JK flip-flop** is most widely used flip-flop. It is the **most** versatile of all flip-flops.

Q. 7.17.10 How does JK flip-flop differ from SR flip-flop in its operation ? What is its advantage over SR flip-flop ?

☑ **Ans. :**

Difference

- In a **SR flip-flop,** the condition S = 1, R = 1 is invalid or indeterminate condition, as output becomes unpredictable.
- In a **JK flip-flop,** the condition J = 1, K = 1 is toggle mode; i.e., the state of output complements on every clock pulse. i.e., if Q = 1, flip-flop switches to Q = 0 and vice-versa.

Advantage over SR flip-flop

The toggle mode operation of JK flip-flop is an advantage over SR flip-flop, which makes them suitable to be used in ripple counters. Ripple counters need the flip-flops to be in toggle mode.

Q. 7.17.11 Which flip-flop is preferred for counting ?

☑ **Ans. :** The **JK flip-flop** is preferred for counting.

Q. 7.17.12 Which flip-flop is used for data transfer ?

☑ **Ans. :** The **D-flip-flop** is used for data transfer.

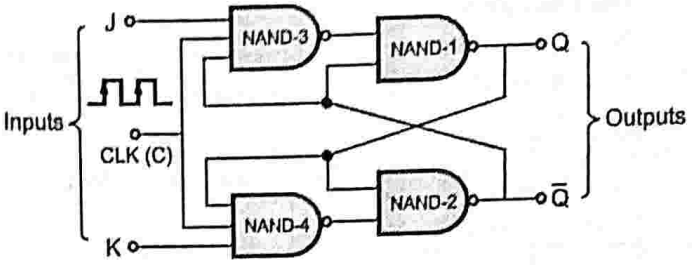
7.18 POSITIVE EDGE-TRIGGERED JK FLIP-FLOP

Q. 7.18.1 Draw the logic diagram of a positive edge-triggered JK Flip-Flop and describe it's working.

☑ **Ans. :**

(A) Logic diagram

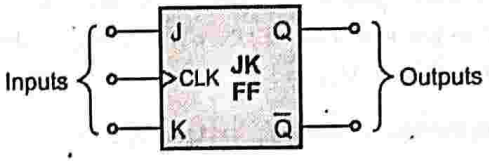
Logic diagram for positive edge-triggered JK flip-flop is shown in Fig. 7.18.1.



(1E36)**Fig. 7.18.1 : Positive edge-triggered JK flip-flop**

(B) Logic symbol

Logic symbol of positive edge-triggered JK flip-flop is shown in Fig. 7.18.2.



(1E37)**Fig. 7.18.2 : Logic symbol of positive edge-triggered JK flip-flop**

(C) Truth Table

Table 7.18.1 : Truth table of positive edge-triggered JK flip-flop

Inputs			Outputs		State
Clock C	J	K	Q_n	Q_{n+1}	
↑	0	0	0	0	Q_n No change (NC)
↑	0	0	1	1	
↑	0	1	0	0	0 RESET
↑	0	1	1	0	
↑	1	0	0	1	1 SET
↑	1	0	1	1	
↑	1	1	0	1	$\bar{Q}_n$ Toggle
↑	1	1	1	0	
0	x	x	0	0	No Change (NC)
0	x	x	1	1	

Module 3

(D) Operation

- The J and K inputs are **synchronous control inputs,** as data on them will modify the state of flip-flop on the positive edge of the clock pulse.
- On absence of the clock pulse, the J and K inputs will not affect the output of the flip-flop.
- We will analyze the operation of positive edge-triggered JK flip-flop for different input/output combinations, which are as follows :

Sr. No.	State	J(SET)	K(RESET)	Clock
(1)	Hold	0	0	↑
(2)	RESET	0	1	↑
(3)	SET	1	0	↑
(4)	TOGGLE	1	1	↑

1. HOLD state : J = 0, K = 0, CLK↑.

When both J and K are LOW, the output does not change from its previous state; i.e., there is no change in the flip-flop output (if Q = 1, Q_{n+1} = 1 and if Q = 0, Q_{n+1} = 0).

Thus, for HOLD state $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$

2. RESET state : J = 0, K = 1, CLK↑.

When J = 0, K = 1, on positive (rising) edge of the clock pulse, the flip-flop is clear or RESET with (Q = 0).

Thus, for RESET state $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$

3. SET state : J = 1, K = 0, CLK↑.

When J = 1, K = 0, on positive (rising) edge of the clock pulse, the output of NAND-1 gate becomes HIGH (Q = 1). JK flip-flop is SET (Q = 1).

Thus, for SET state $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$

4. TOGGLE state : J = 1, K = 1, CLK↑.

When J = 1, K = 1, on positive (rising) edge of the pulse, the flip-flop toggles; i.e., output changes to 1 if it was 0 or vice-versa.

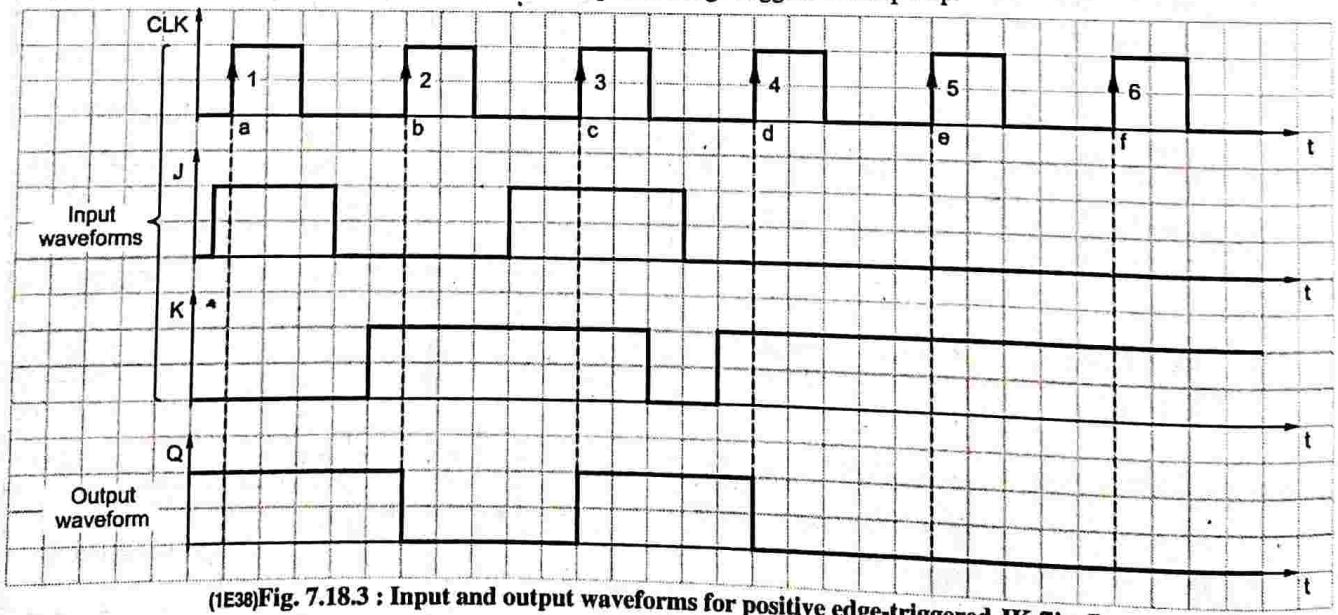
Thus, for TOGGLE state $Q_{n+1} = \bar{Q}_n$, $\bar{Q}_{n+1} = Q_n$

Summary of operation

Sr. No.	State	Output	
		Q_{n+1}	$\bar{Q}_{n+1}$
(1)	Hold	Q_n	$\bar{Q}_n$
(2)	RESET	0	1
(3)	SET	1	0
(4)	TOGGLE	$\bar{Q}_n$	Q_n

(E) Timing Diagram

Fig. 7.18.3 shows the input and output waveforms of the positive edge-triggered JK flip-flop.



(1E38)**Fig. 7.18.3 : Input and output waveforms for positive edge-triggered JK flip-flop**

Step 1 : Initially J = 0 K = 0, and CLK = 0. Let the initial state of flip-flop be Q = 1. At time instant t = a, at the positive edge of the first clock pulse, J = 1, K = 0. Hence, the flip-flop sets. The output Q = 1 till the positive edge of the next clock pulse.

Step 2 : At time instant t = b, at the positive edge of the second clock pulse, J = 0, K = 1. The flip-flop resets. Hence, Q goes LOW (Q = 0) and $\bar{Q}$ goes HIGH ($\bar{Q}$ = 1).

Step 3 : At time instant t = c, at the positive edge of the third clock pulse, J = 1, K = 1. The flip-flop toggles; i.e., Q changes from 0 to 1 and $\bar{Q}$ changes from 1 to 0.

Step 4 : At time instant t = d, at the positive edge of the fourth clock pulse, J = 0, K = 1. The flip-flop is RESET, the output Q = 0 and $\bar{Q}$ = 1.

Step 5 : At time instant t = e, at the positive edge of the fifth clock pulse, J = 0, K = 1. The flip-flop remains in RESET state with output Q = 0 and $\bar{Q}$ = 1.

Step 6 : At time instant t = f, at the positive edge of the sixth clock pulse, J = 0, K = 1. The flip-flop remains in RESET state with output Q = 0 and $\bar{Q}$ = 1.

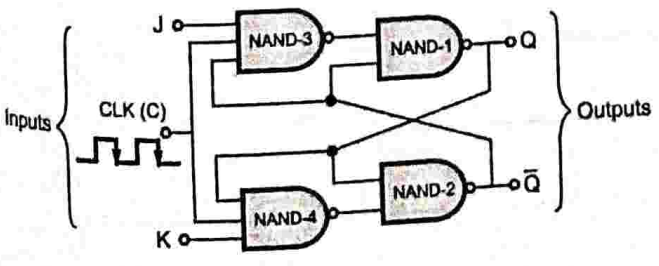
7.19 NEGATIVE EDGE-TRIGGERED JK FLIP-FLOP

Q. 7.19.1 *Draw the logic diagram of a negative edge-triggered JK Flip-Flop and describe it's working.*

Ans. :

(A) Logic diagram

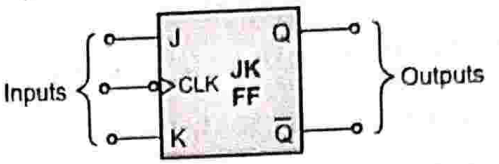
- Fig. 7.19.1 shows the logic diagram for negative edge-triggered JK flip-flop and its symbol.



(1E39)**Fig. 7.19.1 : Negative edge-triggered JK flip-flop**

(B) Logic symbol

Logic symbol of negative edge-triggered JK flip-flop is shown in Fig. 7.19.2.



(1E40)**Fig. 7.19.2 : Logic symbol of negative edge-triggered JK flip-flop**

(C) Truth table

Table 7.19.1 : Truth table of negative edge-triggered JK flip-flop.

Inputs			Outputs		State
C	J	K	Q_n	Q_{n+1}	
↓	0	0	0	0	Q_n — No change (NC)
↓	0	0	1	1	
↓	0	1	0	0	0 — RESET
↓	0	1	1	0	
↓	1	0	0	1	1 — SET
↓	1	0	1	1	
↓	1	1	0	1	$\bar{Q}_n$ — TOGGLE
↓	1	1	1	0	
0	x	x	0	0	No Change (NC)
0	x	x	1	1	

(D) Operation

- J and K are synchronous control inputs, as data on these pins will modify the change of the flip-flop on the negative edge of the clock pulse.
- On the absence of the clock pulse, the J and K inputs will not affect the output of the flip-flop.
- We will analyze the operation of negative edge-triggered JK flip-flop for different input/output combinations, which are as follows :

Sr. No.	State	J(SET)	K(RESET)	Clock
(1)	Hold	0	0	↓
(2)	RESET	0	1	↓
(3)	SET	1	0	↓
(4)	TOGGLE	1	1	↓

Module 3

1. HOLD state : J = 0, K = 0, CLK ↓.

When both J and K are LOW, the output does not change from its previous state; i.e., there is no change in the flip-flop output (if Q = 1, $Q_{n+1} = 1$ and if Q = 0, $Q_{n+1} = 0$) even after the negative edge of clock pulse is applied.

Thus, for HOLD state $Q_{n+1} = Q_n$, $\bar{Q}_{n+1} = \bar{Q}_n$

2. RESET state : J = 0, K = 1, CLK ↓.

When J = 0, K = 1, on the negative (falling) edge of the clock pulse, the output Q is LOW. The flip-flop is cleared or RESET (Q = 0).

Thus, for RESET state $Q_{n+1} = 0$, $\bar{Q}_{n+1} = 1$

3. SET state : J = 1, K = 0, CLK ↓.

When J = 1, K = 0, on the negative (falling) edge of the clock pulse, the output of the NAND 1 gate becomes HIGH (Q = 1). The flip-flop is SET (Q = 1).

Thus, for SET state $Q_{n+1} = 1$, $\bar{Q}_{n+1} = 0$

4. TOGGLE state : J = 1, K = 1, CLK ↓.

When J = 1, K = 1, on the negative (falling) edge of the clock pulse, the flip-flop output toggles; i.e., output changes to 1 if it was 0 and vice-versa.

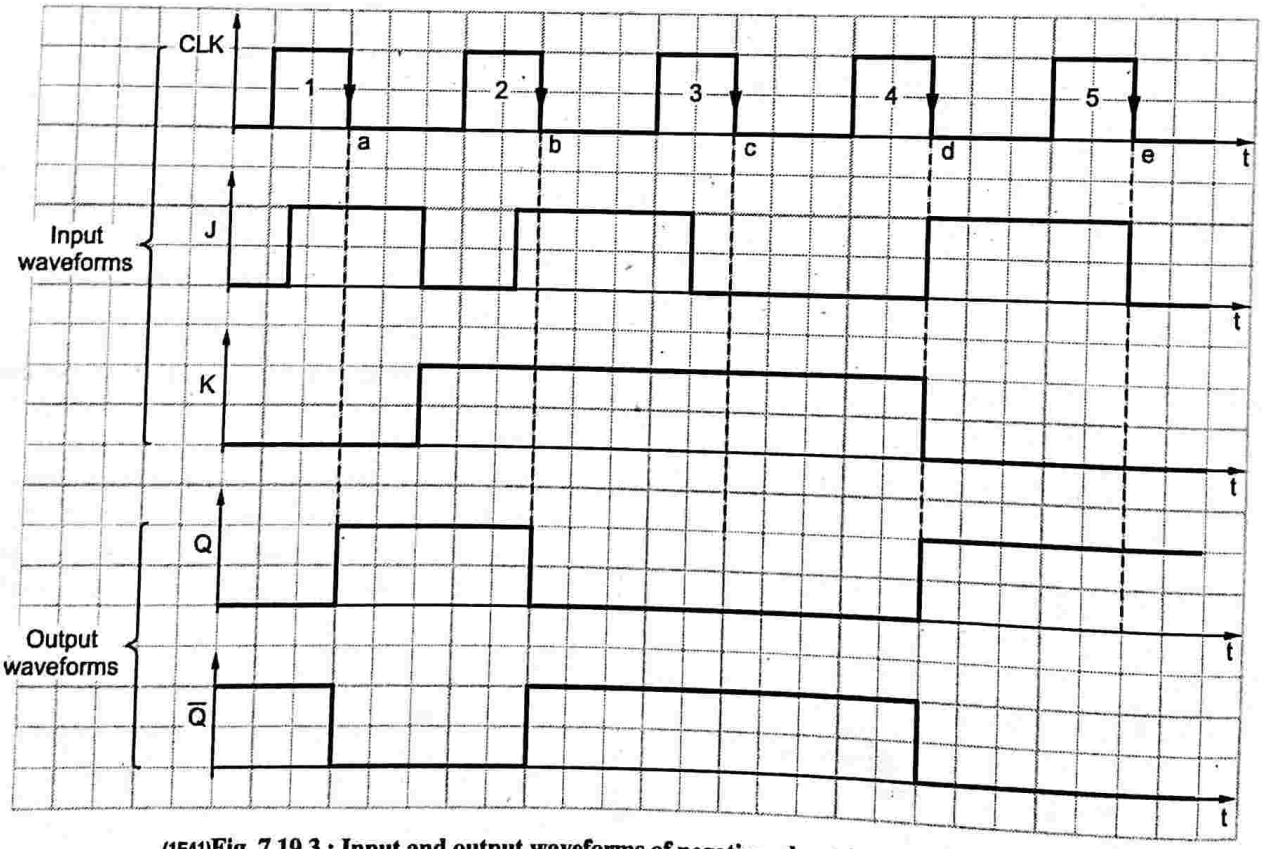
Thus, for TOGGLE state $Q_{n+1} = \bar{Q}_n$, $\bar{Q}_{n+1} = Q_n$

Summary of operation

Sr. No.	State	Output	
		Q_{n+1}	$\bar{Q}_{n+1}$
(1)	Hold	Q_n	$\bar{Q}_n$
(2)	RESET	0	1
(3)	SET	1	0
(4)	TOGGLE	$\bar{Q}_n$	Q_n

(E) Timing diagram

Fig. 7.19.3 shows the input and output waveforms of the negative edge-triggered JK flip-flop.



(1E41)**Fig. 7.19.3 : Input and output waveforms of negative edge-triggered JK flip-flop**

Step 1 : Initially J = 0, K = 0 and CLK = 0. Let the initial state of flip-flop be Q = 0, $\overline{Q}$ = 1. At time instant t = a, at the negative edge of the first clock pulse , J = 1, K = 0. The JK flip-flop sets and output Q = 1 (HIGH) and $\overline{Q}$ = 0 (LOW).

Step 2 : At time instant t = b, at the negative edge of the second clock pulse, J = 1, K = 1. The flip-flop toggles, i.e., Q changes from 1 to 0 and $\overline{Q}$ changes from 0 to 1.

Step 3 : At time instant t = c, at the trailing edge of the third clock pulse, J = 0, K = 1. The flip-flops RESET with Q = 0 and $\overline{Q}$ = 1.

Step 4 : At time instant t = d, at the falling edge of the fourth clock pulse, J = 1, K = 0. The flip-flop will SET with Q = 1 and $\overline{Q}$ = 0.

Step 5 : At time instant t = e, at the falling edge of the fifth clock pulse J = 0, K = 0 the flip-flop will remain in the same state with Q = 1 and $\overline{Q}$ = 0.

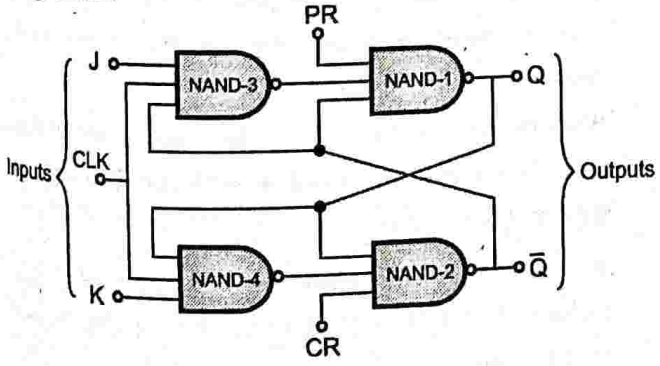
7.20 J-K FLIP-FLOP WITH PRESET AND CLEAR

Q. 7.20.1 Describe the function of preset and clear terminals in JK flip-flop. Write truth table of it.

Ans. :

(A) Circuit Diagram

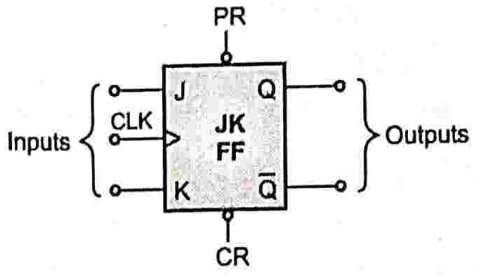
Circuit diagram of JK flip-flop with preset and clear is shown in Fig. 7.20.1.



(1E42)**Fig. 7.20.1 : JK flip-flop with preset and clear**

(B) Logic symbol of JK flip-flop (positive edge-triggered) with active-LOW PRESET (PR) and CLEAR (CR) inputs

Refer Fig. 7.20.2.



(1E43)**Fig. 7.20.2 : Logic symbol of JK flip-flop (positive edge-triggered) with active-LOW PRESET (PR) and CLEAR (CR) inputs**

– PR and CR are both active-low asynchronous inputs.

(C) Operation

1. Normal JK flip-flop : PR = CR = 1

If PR = 1, CR = 1, i.e., both are inactive, the flip-flop will operate as a normal JK flip-flop.

2. Preset state : PR = 0, CR = 1

If PR = 0, the output of NAND-1 gate will be logic **1**, i.e., (Q = 1). The flip-flop is SET. All the three inputs to the NAND-2 are logic **1**. Hence its output, $\overline{Q}$ = 0.

Thus, if PR = 0, CR = 1, the JK flip-flop is SET.

3. Clear state : PR = 1, CR = 0.

If PR = 1, CR = 0, the JK flip-flop is RESET, i.e., Q = 0.

4. Indeterminate state : PR = 0, CR = 0.

The condition PR = 0, CR = 0 should be avoided because it results in invalid or indeterminate state.

Table 7.20.1 shows the summary of operation of positive edge-triggered JK flip-flop.

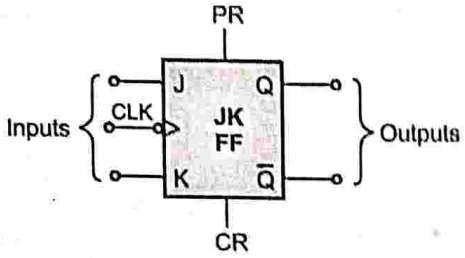
Table 7.20.1 : Summary of operation of positive edge-triggered JK flip-flop

Inputs			Output	Operation performed
CLK	PR	CR	Q	
1	1	1	Q_{n+1}	Normal JK flip-flop
0	0	1	1	PRESET
0	1	0	0	CLEAR

Module 3

(D) Logic symbol of negative edge-triggered JK flip-flop with active-high PR and CR

Refer Fig. 7.20.3.



(1E44)**Fig. 7.20.3 : Negative edge-triggered JK flip-flop with active-high PR and CR**

Table 7.20.2 shows summary of operation of negative edge-triggered JK flip-flop with active-high PR and CR.

Table 7.20.2 : Negative edge-triggered JK flip-flop with active-high PR and CR

Inputs			Output	Operation performed
CLK	PR	CR	Q	
1	0	0	Q_{n+1}	Normal JK flip-flop
0	0	1	0	CLEAR
0	1	0	1	PRESET
0	1	1	Invalid	Indeterminate state. Hence, not used.

7.21 CLOCKED D FLIP-FLOP

Q. 7.21.1 Draw the logic diagram of D flip-flop using NAND gates. Write its truth table.

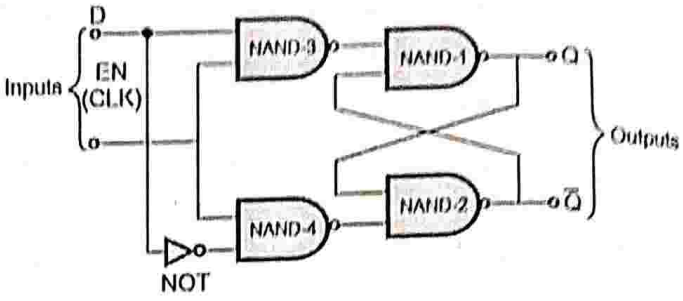
Ans. :

(A) Description

- Clocked D flip-flop is also called as **D (data) latch or transparent latch or Delay flip-flop.**
- Clocked D flip-flop has a single input called **D or data input.**

(B) Logic diagram

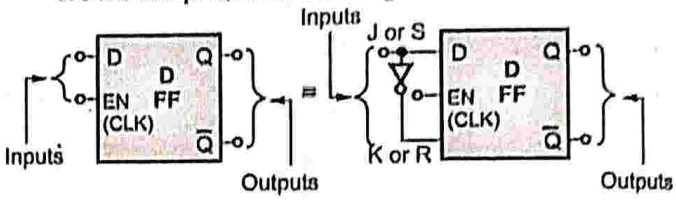
Logic diagram of clocked D flip-flop is shown in Fig. 7.21.1.



(1E46)**Fig. 7.21.1 : Logic diagram of gated D-latch**

(C) Logic symbol

- The D flip-flop is constructed from the SR flip-flop or JK flip-flop by inserting an inverter between the S and R inputs or J and K inputs, as shown in Fig. 7.21.2.



(1E45)**Fig. 7.21.2 : Logic symbol of a clocked D flip-flop**

(D) Truth Table

Table 7.21.1 : Truth table of gated D-latch

EN(C)	D	Q_n	Q_{n+1}	State
1	0	0	0	RESET
1	0	1	0	
1	1	0	1	SET
1	1	1	1	
0	x	0	0	No change (NC)
0	x	1	1	

≡

D	Q_{n+1}
0	0
1	1

(E) Operation

- When the EN(clock) is LOW, the clocked D flip-flop is inactive. Any change in the value of D will not modify the state of the output.
- When the EN(clock) is HIGH, a low value on the D input indicates S = 0, R = 1. It causes the clocked D flip-flop to RESET ($Q = 0, \overline{Q} = 1$).
- When the EN(clock) is HIGH, a high value on the D input indicates S = 1, R = 0. It causes the flip-flop to SET ($Q = 1, \overline{Q} = 0$).

Thus, when the EN(clock) is high, a high D input sets the flip-flop Q = 1 and low D input will RESET the flip-flop (Q = 0); i.e., the Q output follows the D input at the end of the clock pulse. Hence, this latch is called as a transparent latch.

Fig. 7.21.1 shows the logic diagram of gated D-latch.

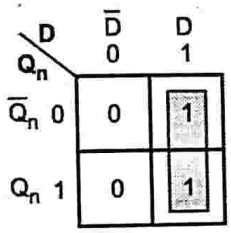
It is also called as Delay (D) flip-flop as there is some delay in transferring data from input to output.

(F) Characteristic Equation

Table 7.21.2 : Characteristic table for D flip-flop

Present state	Inputs	Next state
Q_n	D	Q_n
0	0	0
0	1	1
1	0	0
1	1	1

K-map simplification for clocked D flip-flop



$Q_{n+1} = D$

(1E62)**Fig. 7.21.3 : K-map for clocked D flip-flop**

The characteristic equation for D flip-flop is : $Q_{n+1} = D$

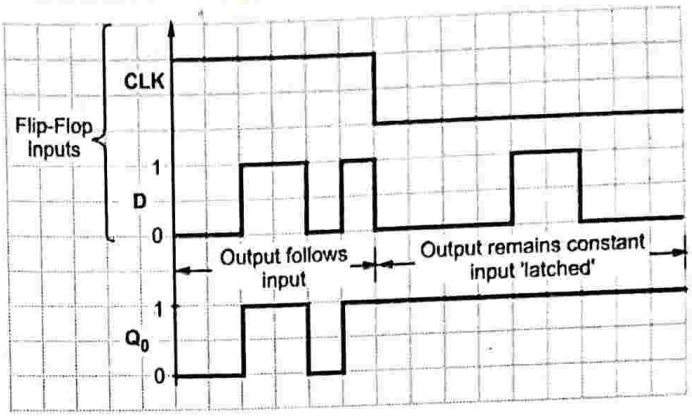
(G) Excitation Table

Table 7.21.3 : Excitation Table for D flip-flop

Present state	Next state	Required input
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

The D input will always be same as the next state, irrespective of the present state; i.e., if the next state required is **0**, then D input is to be **0** and if the next state required is **1**, then the D input is **1** irrespective of Q_n (present state value).

(H) Timing Diagram



(1E74)**Fig. 7.21.4 : Clocked D Flip-Flop**

(I) Application

It is used as a **latch** or a **delay element** for storing 1-bit binary data.

7.22 POSITIVE EDGE-TRIGGERED D FLIP-FLOP

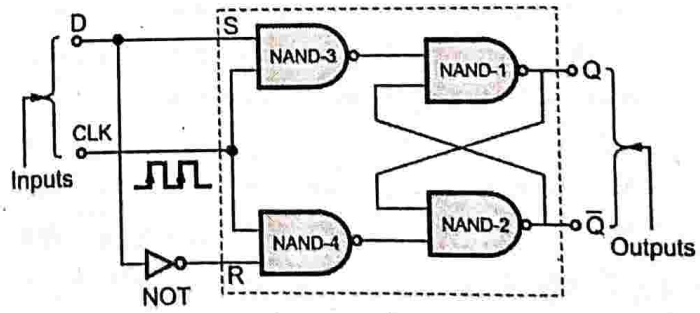
Module 3

Q. 7.22.1 Draw the logic diagram of a positive edge-triggered D Flip-Flop and describe it's working.

Ans. :

(A) Circuit Diagram

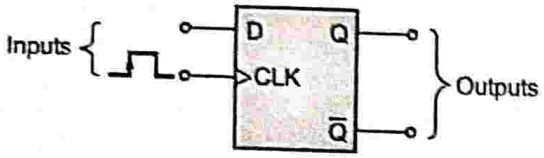
– Fig. 7.22.1 shows the logic diagram of the positive edge-triggered D flip-flop and Fig. 7.22.2 shows its logic symbol.



(1E47)**Fig. 7.22.1 : Positive edge-triggered D flip-flop**

(B) Logic Symbol

Logic symbol of positive edge-triggered D flip-flop is shown in Fig. 7.22.2.



(1E45)**Fig. 7.22.2 : Logic symbol of positive edge-triggered D flip-flop**

(C) Truth Table

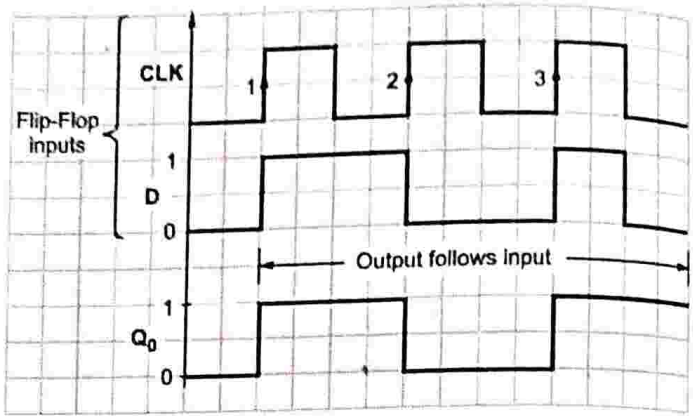
Table 7.22.1 : Truth table of positive edge-triggered D flip-flop

Inputs			Output	State
CLK(C)	D	Q	Q_{n+1}	
↑	0	0	0	RESET
↑	0	1	0	
↑	1	0	1	SET
↑	1	1	1	
0	x	0	0	No Change (NC)
0	x	1	1	

(D) Operation

- D(data) input is the only synchronous control input. In the absence of the clock pulse, the D input will not affect the flip-flop output.
- Due to the presence of inverter, the S and R inputs always be inverted; i.e., if S = 0, R = 1 and if S = 1, R = 0. Both the SR inputs will never be 00 or 11, avoiding the race around condition.
- At the positive (rising) transition of the clock pulse, the flip-flop output Q will be same as the value of D input; i.e., if D is low when rising edge of clock pulse is applied, then the flip-flop resets producing output Q = 0 and $\bar{Q}$ = 1.
- If D is high when the rising edge of clock pulse is applied, the flip-flop sets producing output Q = 1 and $\bar{Q}$ = 0.
- **Thus, on the rising (positive) edge of the clock pulse, the level present at D input is stored in the flip-flop.**

(E) Timing Diagram



(1E75)**Fig. 7.22.3 : Positive Edge-triggered D Flip-Flop**

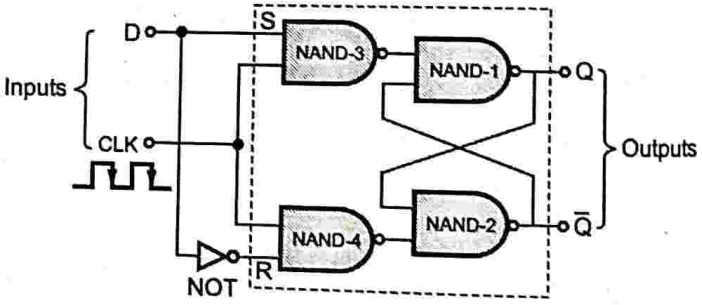
7.23 NEGATIVE EDGE-TRIGGERED D FLIP-FLOP

Q. 7.23.1 Draw the logic diagram of a negative edge-triggered D Flip-Flop and describe it's working.

Ans. :

(A) Logic Diagram

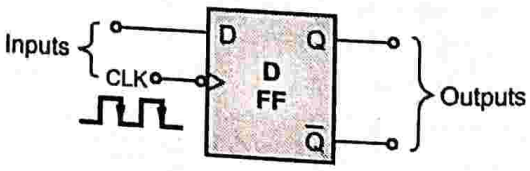
Logic Diagram of Negative edge-triggered D flip-flop is shown in Fig. 7.23.1.



(1E49)**Fig. 7.23.1 : Negative edge-triggered D flip-flop**

(B) Logic Symbol

Logic symbol of negative edge-triggered D flip-flop is shown in Fig. 7.23.2.



(1E50)**Fig. 7.23.2 : Logic symbol of negative edge-triggered D flip-flop**

(C) Truth Table

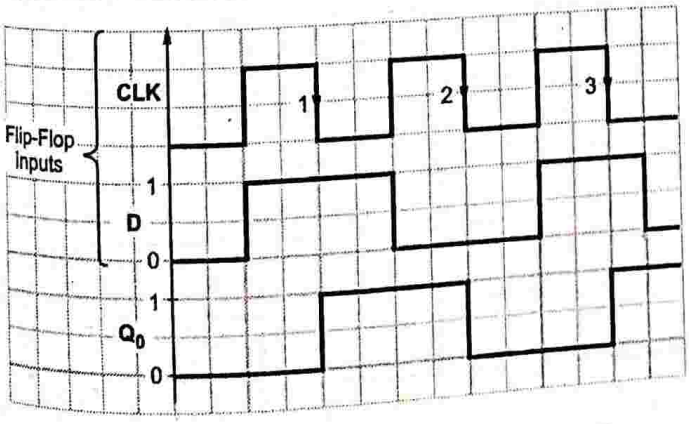
Table 7.23.1 : Truth table of negative edge-triggered D flip-flop

Inputs			Output	State
C	D	Q	Q_{n+1}	
↓	0	0	0	RESET
↓	0	1	0	
↓	1	0	1	SET
↓	1	1	1	
0	x	0	0	No Change
0	x	1	1	

(D) Operation

- D(data) input is the only synchronous control input apart from clock. Without the clock pulse, the D input will not affect the flip-flop output.
- At the negative (falling) edge of the clock pulse, the flip-flop output Q will be same as its D input; i.e., if D is low when the trailing edge of the clock pulse is applied, then the flip-flop RESETs producing output Q = 0 and $\bar{Q} = 1$.
- If D is HIGH when the trailing edge of the clock pulse is applied, then the flip-flop SETs producing output Q = 1 and $\bar{Q} = 0$.
- Thus, on the negative (trailing) edge of the clock pulse, the level present at the D input is stored in the flip-flop; i.e., Q output follows the D input.

(E) Timing Diagram



(1E76)**Fig. 7.23.3 : Negative Edge-triggered D Flip-Flop**

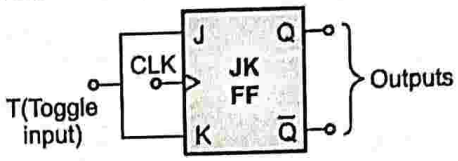
7.24 T FLIP-FLOP

Q. 7.24.1 Draw the truth table for T FF. Using the truth table, derive & explain the excitation table of T FF.

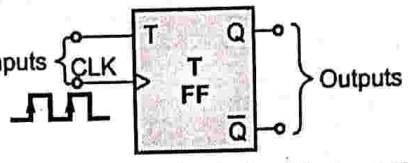
Ans. :

(A) Logic Symbol

- This flip-flop is also called as **Toggle flip-flop.**
- This flip-flop can be constructed using a JK flip-flop if the J and K both the inputs are tied together, as shown in Fig. 7.24.1.



(1E51)**(a) JK flip-flop converted to T flip-flop**



(1E52)**(b) Logic symbol of positive edge-triggered T flip-flop**

Fig. 7.24.1

(B) Truth Table

Table 7.24.1 : Truth table of positive edge-triggered T flip-flop

Inputs			Output	State
C	T	Q_n	Q_{n+1}	
↑	0	0	0	No Change (NC)
↑	0	1	1	
↑	1	0	1	Toggle
↑	1	1	0	
0	x	0	0	No Change (NC)
0	x	1	1	

Table 7.24.2 : Truth table of T flip-flop

T	Q_{n+1}
0	Q_n
1	$\bar{Q}_n$

(C) Operation

- In absence of clock, the T input will not affect the flip-flop output; i.e., if Q = 0 then $Q_{n+1} = 0$ and if Q = 1. $Q_{n+1} = 1$.
- On the positive (rising) edge of the clock pulse, if T = 0 (i.e., J = 0, K = 0), there will be no change in the flip-flop's output, Q and $\bar{Q}$.
- On the positive (rising) edge of the clock pulse, if T = 1 i.e., J = K = 1, the flip-flop output toggles; i.e., if Q = 0 then Q = 1 or vice-versa.
- If T = 1, then the flip-flop output will continuously toggle on every rising edge of the clock pulse.

Module 3

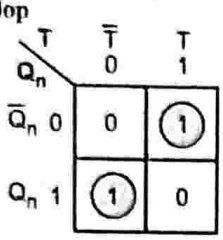
(D) Characteristic Equation

Table 7.24.3 : Characteristic table for T flip-flop

Present state	Input	Next state
Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

K-map simplification for T flip-flop

$$Q_{n+1} = \bar{Q}_n T + Q_n \bar{T}$$



(1E53)**Fig. 7.24.2 : K-map for T flip-flop**

The characteristic equation of T flip-flop is,

$$Q_{n+1} = \bar{Q}_n T + Q_n \bar{T}$$

(E) Excitation Table

Table 7.24.4 : Excitation table of T flip-flop

Present state	Next state	Required input
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

0 → 0 transition

If the present state of the flip-flop is 0 and next state of flip-flop is 0, when the clock is applied, the input must be T = 0.

0 → 1 transition

If the present state of the flip-flop is 0 and next state of the flip-flop is 1, then the T input should be T = 1 (toggle).

1 → 0 transition

If the present state of the flip-flop is 1 and the next state of flip-flop is 0, then the T input should be T = 0 (toggle).

1 → 1 transition

If the present state of the flip-flop is 1 and the next state is to 1, then the T input must be T = 0 (no change).

(F) Timing diagram

Fig. 7.24.3 shows the input and output waveforms or T flip-flop. Flip-flop output toggles at every positive edge of the clock pulse.

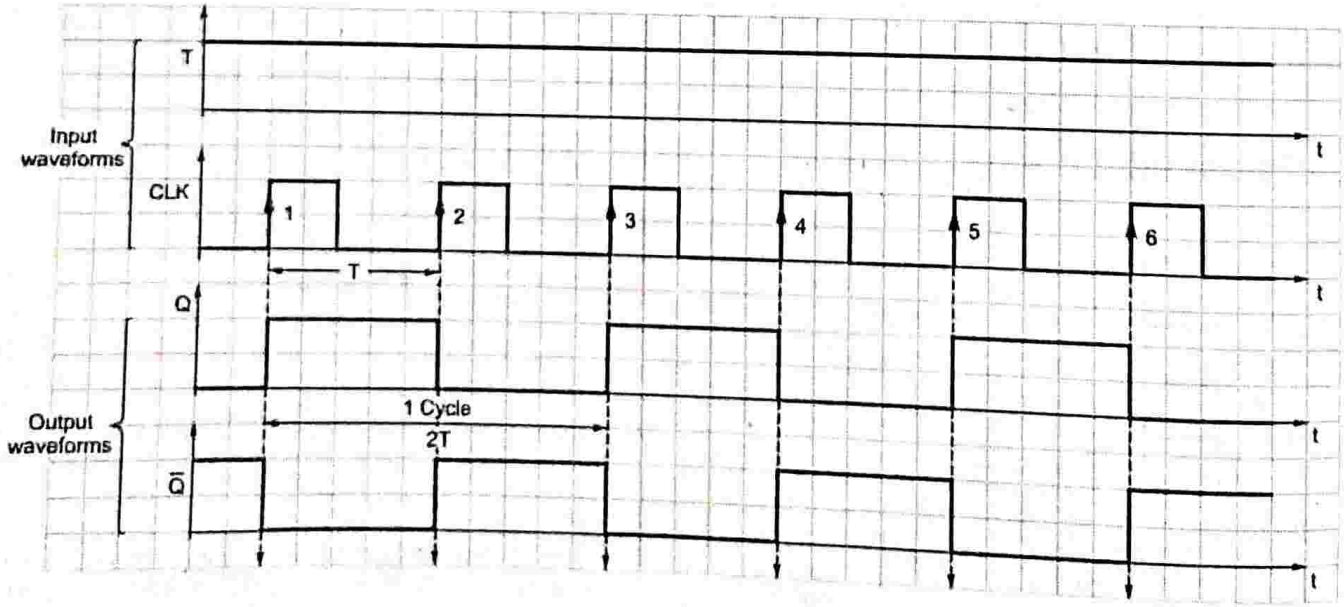
$$\text{Clock frequency} = f_{CLK} = \frac{1}{T}$$

$$\text{1 cycle for Q output} = 2T$$

$$\therefore \text{ Output frequency of flip-flop, } f_0 = \frac{1}{2T}$$

$$\therefore \quad f_0 = \frac{f_{CLK}}{2} \qquad \left(\because f_{CLK} = \frac{1}{T}\right)$$

i.e., the **T flip-flop divides the clock frequency by 2.** Hence, it can be used as a **frequency divider.**



(1E53)**Fig. 7.24.3 : Input and output waveforms for T flip-flop**

(G) Application

It can be used as a **frequency divider**. It is also used in ripple counters.

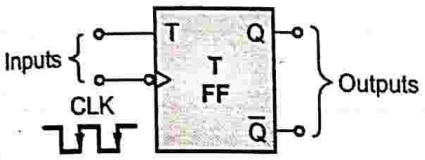
7.25 NEGATIVE EDGE-TRIGGERED T FLIP-FLOP

Q. 7.25.1 Draw the logic diagram of a negative edge-triggered T Flip-Flop and describe it's working.

Ans. :

(A) Logic Symbol

Logic symbol for negative edge-triggered T flip-flop is shown in Fig. 7.25.1.



(1E54)**Fig. 7.25.1 : Logic symbol for negative edge-triggered T flip-flop**

(B) Truth Table

Table 7.25.1 : Truth table for negative edge-triggered T flip-flop

Inputs			Outputs	State
C	T	Q_n	Q_{n+1}	
↓	0	0	0	No Change (NC)
↓	0	1	1	
↓	1	0	1	Toggle
↓	1	1	0	
0	x	0	0	No Change (NC)
0	x	1	1	

(C) Operation

- In absence of clock, the T input will not affect the flip-flop output; i.e., if Q = 0 then $Q_{n+1} = 0$ and if Q = 1 then $Q_{n+1} = 1$.
- On the negative falling edge of the clock pulse, if T = 0, i.e., J = K = 0, there will be no change in flip-flops output, Q and $\overline{Q}$.
- **On the trailing edge of the clock pulse, if T = 1, i.e., J = K = 1, the flip-flop output toggles; i.e., if Q = 0 then Q = 1 or vice-versa.**
- Similar to the positive edge-triggered T flip-flop, the negative edge-triggered T flip-flop acts like a frequency divider with output frequency $f_0 = \frac{f_{CLK}}{2}$.

7.26 APPLICATIONS OF FLIP-FLOPS

Module 3

Q. 7.26.1 List application of flip-flops.

Ans. :

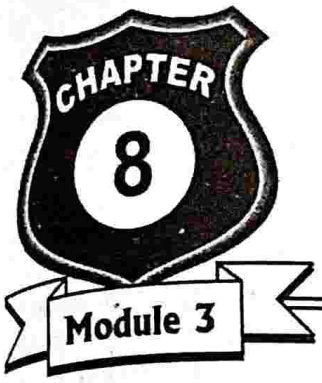
The applications of flip-flops are :

1. Bounce elimination switch
2. Shift registers
3. Counters
4. Memory
5. Frequency dividers

....Chapter ends

❑❑❑

Note



Processor Organization and Architecture

University Prescribed Syllabus

Processor Organization and Architecture

3.3 Register Organization, Instruction Formats, Addressing modes, Instruction Cycle, Interpretation and sequencing.

8.1 REGISTER ORGANIZATION

GQ. 8.1.1 What is register organization ? What are different types of registers ? Explain in detail.

GQ. 8.1.2 How registers are organized in the Processor or CPU architecture?

UQ. 8.1.3 Explain role of different registers like IR, PC, SP, AC, MAR and MDR used in Von Neumann model.

MU - Q. 1(a), Dec. 15, 5 Marks

UQ. 8.1.4 Describe the register organization within the CPU.

MU - Q. 4(b), May 16, 10 Marks

- In the processor register function as a level of memory above cache and main memory in the hierarchy. There are two main categories of registers.

(a) User Visible registers
(b) Control and Status registers

(a) User Visible registers

- These registers are used by the processor and programmer for minimizing the usage of memory references. These registers are further classified as:

(i) General Purpose registers
(ii) Data Registers and Address register
(iii) Conditional codes

(i) General Purpose Register

- These registers are used to hold operand for opcode, but there are certain restrictions like some dedicated registers are used for floating point and stack operations.
- These registers are also used for addressing functions.

(ii) Data and Address registers

- **Data registers** are used to hold explicitly the data used for processing, while address registers are used as general purpose as well as address pointers in some addressing modes.
- **Address registers** includes; segment registers, Index registers, and Stack pointers.
- **Segment registers :** These registers hold the segment base address in segmented addressing.
- **Index registers :** These are used to hold index address in indexing addressing mode.
- **Stack pointer :** This is dedicated register used as pointer to point to the top of the stack memory.

(iii) Control Codes

- These are partially visible to the programmer. These registers are used to hold the conditional bits which are set and reset by the processor hardware to indicate the status of the operation.
- Implicit machine instructions are used to read the status of the control code. Some machine instructions use control word to test the condition as a part of branch operation.

(b) Control Register and Status

- This registers are used to control the operation of the processor, most of these registers are invisible to the programmer and used by the processor. Some of the control registers are visible in machine control or operating system modes.
- Basic four registers are present in control and status group:
  - Program counter (PC)
  - Instruction register (IR)
  - Memory address register (MAR)
  - Memory buffer register (MBR)
- **Program register (PC) :** It is used to hold the address of the next instruction to be fetched from the memory.
- **Instruction register (IR) :** It holds the instruction most recently fetched from the memory.
- **Memory address registers (MAR) :** It holds the address of the memory location.
- **Memory buffer register (MBR) :** It is a buffer which holds the data to be written to memory or most recently read from memory.
- Most of the processor contains set of registers which are used to indicate the status information these are called as program status word (PSW). PSW contains conditional codes plus status information, commonly used fields are sign, zero, carry, equal, overflow, interrupt enable/disable, supervisor.

8.2 INSTRUCTION FORMAT

UQ. 8.2.1 Give different instruction format.

MU - Q. 1(b), Dec. 18, Q. 1(c), May 19, 5 Marks

- Every computing machine requires instructions, which defines the operations to be performed on data. The instructions are represented in the form of binary bits in which every bit has its own significance, the layout of each bit in terms of its fields position is called as instruction format.

Fig. 8.2.1 shows a simple 16 bit instruction format

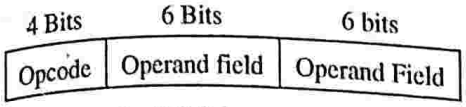


Fig. 8.2.1 : 16-bit instruction format

- It consists of an Opcode field which is a mandatory field and operand field which is flexible in size.
- The opcode field indicates the type of operation to be performed by the instruction where as operand field indicates the method in which the data is obtained and operated on.

Instruction format Design Issues

GQ. 8.2.2 Explain key design issues in designing an instruction format.

- To create an instruction format for a given processor is a complex task several design considerations need to be addressed to obtain an optimum instruction format.
- Following are some key design issues which are need to be addressed :

(a) Instruction length (b) Allocation of Bits
(c) Variable length instruction

(a) Instruction length

- For any processor more and complex instructions mean more computation capabilities. This factor will require more opcodes which demands more bits.
- As every processor relies on memory to fetch its data and instructions. It requires larger memory addressability and various addressing modes which requires more bits.
- The size of data on which processor can operate at a time determines the speed of the processor, larger the size of data better computation capabilities, to bring in large data more number of bits are required which further increases the length of instruction format.
- All of the above factors (opcdes, operands, addressing modes and address range) effects the length of the instruction format.
- It is also affected by several other factors such as memory size, memory organization, bus structure, processor complexity, processor speed.
- All the instructions are fetched from memory hence bus size plays a vital role in deciding the size of the instruction, the instruction length is usually kept equal to bus transfer length or multiple of the bus transfer length.
- A considerable tradeoff between the factors needs to be met to optimize the processor operation.

(b) Allocation of Bits

- As we know every instruction format is divided into fields and each field has several number of bits allocated. The second major factor is how to allocate bits for each field.
- For a fixed length format if the number of opcodes are increased then the number of bits for opcode also increases. This will reduce the bits of operand field which directly affects the addressing range and addressing modes.
- Therefore for fixed length instruction there is a tradeoff between opcodes and operand bits.
- Other factors which affects the operand addressing bits are :
  - Number of addressing modes
  - Number of operands
  - Register versus memory
  - Number of register sets
  - Address range
  - Address Granularity

(c) Variable length instruction

- In this strategy the length of the instruction format changes which gives a greater flexibility in designing large number of opcodes, addressing modes with various combination of registers and memory references.
- As the length of the instruction is unknown the fetch cycle is always equals to the longest length instruction this strategy is sometimes helpful as more than one small instruction can be fetched in one fetch cycle.
- As each instruction is of variable in length this requires more complex processor decoding circuitry.

Module 3

8.3 ADDRESSING MODE AND FORMATS

8.3.1 Addressing Modes

UQ. 8.3.1 Explain in detail different types of addressing modes.
MU - Q. 2(a), Dec. 14, 10 Marks

Describing the operand and its location is referred to as addressing modes. Following are the list of commonly used addressing modes.

(a) Immediate (b) Direct
(c) Indirect (d) Register
(e) Register indirect (f) Displacement
(g) Stack

(a) Immediate Addressing mode

- In this addressing mode, the operand value is present as a part of the instruction. This mode is used for accessing constants and set values of variables.

- The advantage of this addressing mode it requires no memory reference for accessing the operand. Generally the operand value is stored in twos complement form.

Fig. 8.3.1 : Immediate Addressing mode

▶ (b) Direct Addressing mode

- In this addressing mode the effective address of the operand is present as a part of the instruction. It is a simplest form of addressing mode as it requires only one memory reference.
- The major disadvantage of this addressing mode it can access limited range of address.

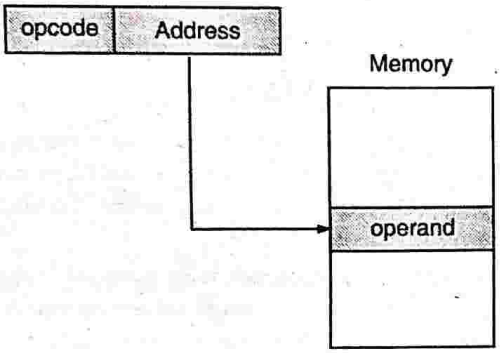


Fig. 8.3.2 : Direct Addressing mode

▶ (c) Indirect Addressing mode

- In this addressing mode the operand address is obtained from word of memory addressed by the instruction address field. The main advantage of this addressing mode is that it can access large address range.
- Let say if N number of bits can be stored at one location of memory then the total available space is 2^N. The disadvantage of this mode is that it requires two memory references to fetch the operand.

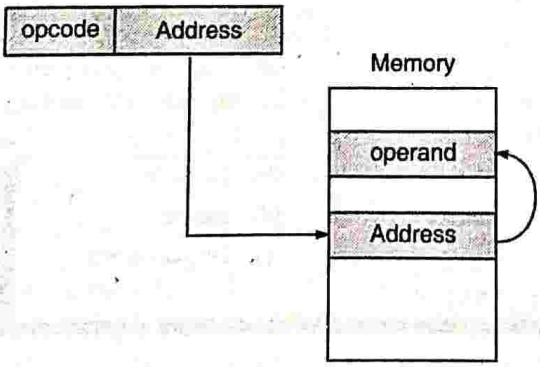


Fig. 8.3.3 : Indirect Addressing mode

▶ (d) Register Addressing mode

- In this addressing mode the operands are placed in the register and the address of register is indicated in the instruction. There are 3 to 5 bits for address field of register which in turn provides reference to 8 to 32 general purpose registers.
- The main advantage of this register is no memory references hence it's a fast transfer, small address field.
- The disadvantage of this mode is that it has limited number of registers which requires greater management efforts on programmer side.

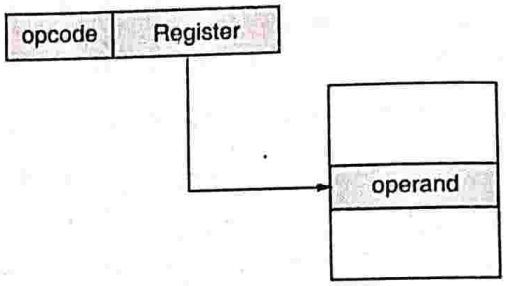


Fig. 8.3.4 : Register Addressing mode

▶ (e) Register Indirect Addressing mode

- The address of the operand is provided by the register instead of memory. The space available in this mode is proportional to the length of the register.
- The main advantage of this mode is it requires only one memory fetch cycle.

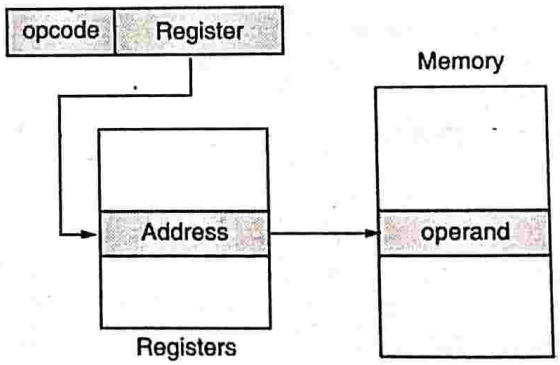


Fig. 8.3.5 : Register Indirect Addressing mode

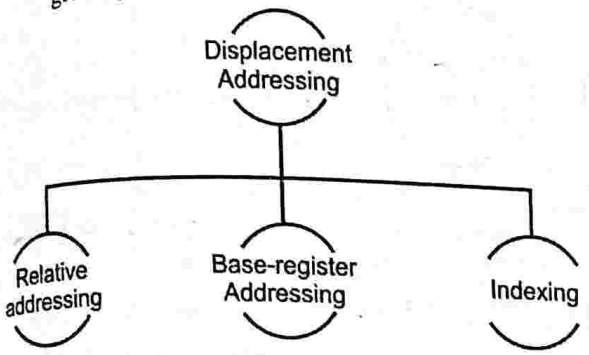
▶ (f) Displacement addressing mode

- This addressing mode requires two address fields in which one is explicit.
- The other implicit reference address refers to a register whose content value is added to the explicit address to form the effective address of the operand.

(i) **Relative Addressing :** In this addressing mode the implicit address is given by the program counter i.e. The address field is added with the next instruction address to form the effective address.

(ii) **Base-Register Addressing :** In this addressing mode the effective address is formed by adding address field displacement value to the referenced register value. The register reference may be explicit or implicit.

(iii) **Indexing :** In this mode the effective address is formed by adding address field to a positive displacement value given by referenced register.



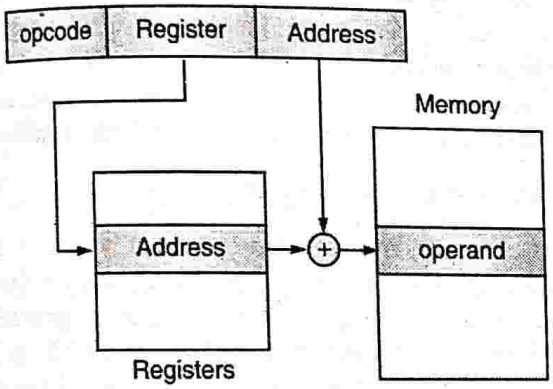


Fig. 8.3.6 : Displacement Addressing mode

(g) Stack Addressing

- This addressing mode is an implicit addressing mode in which no memory reference is mentioned. The address of the operand is always the top of the stack.

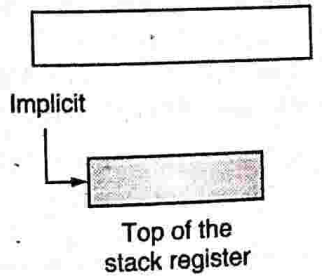


Fig. 8.3.7 : Stack Addressing mode

8.4 THE INSTRUCTION CYCLE

UQ. 8.4.1 Explain Instruction and Instruction cycle.

MU - Q. 1(a), Dec. 18, Q. 1(A), May 19, 5 Marks

- The Fig. 8.4.1 shows the state diagram of an instruction cycle it can be divided into two sections the upper section caters to exchange between memory-processor and I/O-processor while the second section caters to internal processor operations.
- Following is the description of each stage :

(a) Instruction address calculation (IAC)
(b) Instruction fetch (IF)
(c) Instruction operation decoding (ID)
(d) Operand address calculation (OAC)
(e) Operand fetch (OF)
(f) Data operation(DO or EXEC)
(g) Operand Store (OS)

(a) Instruction address calculation (IAC)

- In this stage the address of the next instruction is computed by adding a fixed number to the previous instruction address.
- In general, the required Instruction address calculations are carried out.

(b) Instruction fetch (IF)

- In this stage, the calculated instruction address is sent on the external Address bus and from the memory system; the instruction is fetched from the into the processor or CPU.
- Usually, it gets stored in the Instruction Register (IR) in the Processor or CPU.

(c) Instruction operation decoding (ID)

- In this stage the meaning of the instruction is extracted which determines the type of operation to be performed and operands to be used.
- This process is called as Instruction decoding. It is carried out in the part of control unit, conventionally called as Instruction Decoder.

(d) Operand address calculation (OAC)

- The address of the operand is computed that is to be fetched from memory or I/O space.
- This stage is the function of addressing mode being used by the instruction and the translations of address needed by the virtual memory system.

(e) Operand fetch (OF)

- In this stage, the operand is fetched from the memory or is read from an I/O device by sending the operand address computed in the previous stage, over the address bus and receiving operand over the data bus.
- The operand could be a register, a memory location or an I/O port.

Module 3

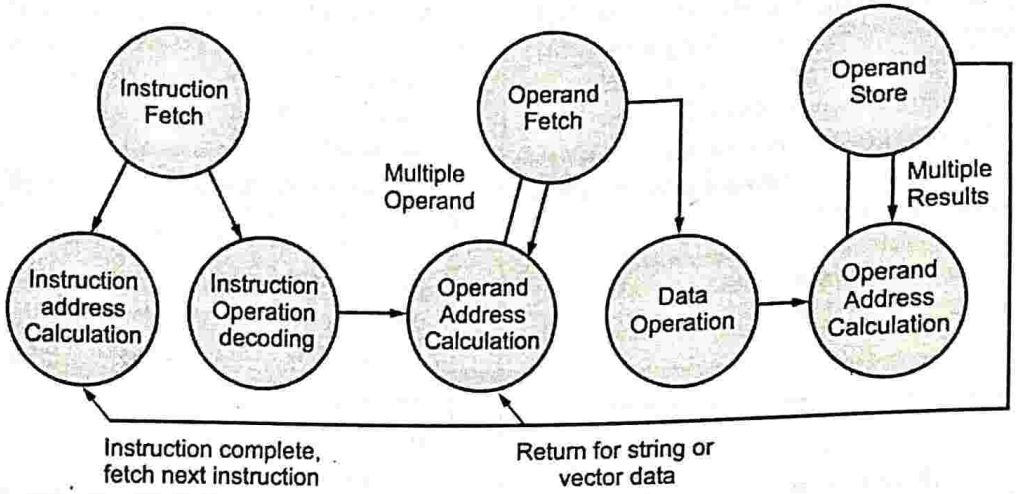


Fig. 8.4.1 : State diagram of instruction cycle

▶ (f) Data operation(DO or EXEC)

- This stage performs operation indicated in the instruction.
- Therefore, this is the stage where the instruction currently under processing; is actually executed.

▶ (g) Operand Store (OS)

- Write the result into memory or out to I/O device. In this stage, again the operand addresses are calculated and the processed operand in the processor or CPU is stored back to the destination.
- It could be a register, a memory location or an I/O port.

⏭ 8.5 INSTRUCTION INTERPRETATION AND SEQUENCING

Instructions are fetched or received from the program memory in to the CPU. They must be first understood or interpreted and then they should be sequenced for their execution. Therefore, Instruction interpretation and sequencing is an important aspect in the life cycle of Instructions over the CPU :

☞ Instruction Interpretation

- Instructions are fetched from the program memory and they are sent to the Instruction Decoder. In instruction decoder, the instructions are decoded and their meaning and functionality is understood. This is then used for generating the different control signals internally over the CPU as well as externally on the system buses. This operation of decoding and understanding the instructions is called as **Instruction Interpretation**. Based on the interpretation of the instruction, next steps are taken in order to execute that instruction.

☞ Instruction Sequencing

- Once the instructions are decoded and interpreted, they are sequenced for the execution. Firstly, the control signals required for the executing the instruction, are generated by an appropriate control unit (either hardwired or micro-programmed) Then the instruction is executed in sequence by first reading the data operands necessary for the instruction, the subsequently the instruction operates and executes on the operands; finally the resultant operands of the instruction, are stored back to their destination. This process is termed as **Instruction sequencing and execution**.
- For the purpose of interpretation of instructions, digital logic is needed that is provided in the Control unit of the CPU. For Sequencing and execution, again functional logic blocks are required that are provided through the Data Path of the CPU. In next section we discuss the different types of Data path with Controls and types of Control units.

...Chapter Ends

□□□

CHAPTER 9

Control Unit Design

Module 4

University Prescribed Syllabus

Control Unit Design

4.1 Hardwired Control Unit : State Table Method, Delay Element Methods.

4.2 Microprogrammed Control Unit : Micro Instruction-Format, Sequencing and execution, Micro operations, Examples of microprograms.

9.1 CONTROL UNIT

UQ. 9.1.1 State the functions of control unit.
MU - Q. 5(b), May 18, 5 Marks

1. Functional Requirement of Control Unit

- The Instruction Cycle stated above has a very important stage, namely Instruction Operation Decode (ID or IOD) which is responsible for decoding the instruction and understanding the meaning of the instruction.
- Once the exact functionality of the specified instruction is known, the next very important task is to control the CPU logic to get that specific instruction successfully executed and produce specified outputs.
- This forms the very basic necessity or need of the logic that would perform the control operations. This logic is called as the control logic of the CPU. The functional unit that consists of the control logic is called as the control unit. Therefore, the control unit becomes the most critical part in the processor of CPU.

2. Operations/ Functions of the Control Unit

- The control unit operates by receiving the input information which it converts into control signals. It is then sent to the CPU and attached hardware performing the desired operations in the desired sequence.
- The internal and external control signals generated by the control unit carry out control operations on various functional units of the CPU. This result in the successful execution of the specific instruction.
- The functions of the control unit are as follows :
  - Coordination of data movement between processors subunits and its sequencing.
  - Interpretation of the instructions fetched from the memory.
  - Controlling the data flow over the CPU.
  - Conversion of external instructions or commands into the sequence of control signals.
  - Controlling the subunits of the CPU such as ALU, registers data buffers etc.
  - Scheduling the operations of the CPU in an appropriate flow so as to complete the execution of instruction.

3. Block Schematic & Control Signals

Fig. 9.1.1 shows the block schematic of the control unit. The operations of the control units are based on the following inputs.

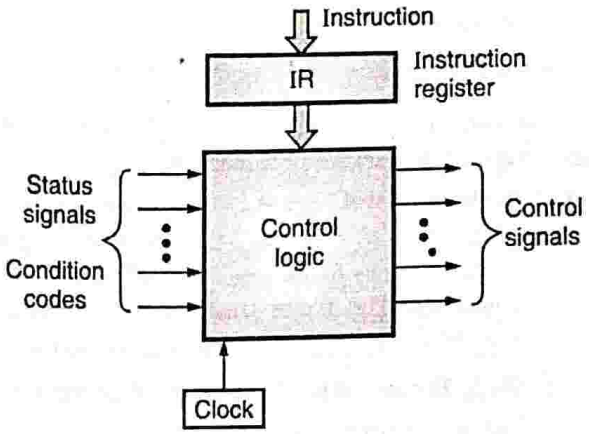


Fig. 9.1.1 : Block Schematic of Control Unit

(A) Opcode of the instruction

- The operational code of the instruction embeds the meaning of the instruction.
- It contains information about the instruction functionality, location and encoding of the operands used and the addressing mode of the instruction.

(B) The Condition codes and status flags

- The condition codes reflect the current status of certain digital signals generated in the processor on the basis of which control signals are generated.
- Similarly, status flags reflect the current state of operation performed in the ALU and the control signals generated are dependent on this state.

(C) The current state of the processor

The control signals and the next state of the processor is directly dependent on the current state of the processor.

(D) Clock signal

It is used for synchronization of the timings for state transition and activation of control signals.

The control logic that resides on the control unit, receives these inputs in order to generate the necessary control signals as output. Different types of control signals generated are as follows :

(a) External control signals

These control signals are issued to external components in the system such as memory and I/O devices.

1. **RD :** Read Control signal. It means that the data will be read from Memory or IO device to the CPU. RD indicates active high and #RD indicates active low control signal.
2. **WR :** Write Control signal. It means that the data will be written to the Memory or IO device from the CPU. WR indicates active high and #WR indicates active low control signal.

Module 4

3. A control signal differentiating the memory access and IO access.

(b) Internal control signals

These control signals are issued inside the CPU and its subunits to control the data flow and functionality.

1. **Register control Signals :** For managing the data flow in and out of a register, pair of control signals is required namely In and Out. For a given register R_X the control signals would be denoted as $R_{X\,in}$ and $R_{X\,out}$.

   $R_{X\,in}$: For controlling the data flow to the register

   $R_{X\,out}$:For controlling data flow out of the register

   Different registers used are:

   $R_0 - R_N$: General Purpose Registers

   MAR: Memory Address Register

   MDR: Memory Data Register

   PC: Program Counter

   SP: Stack Pointer

   IR: Instruction register

   TMP, Z: Temporary Registers

2. **ALU control Signals :**

ALU Data Flow Control signals :

ALU has In and Out control signals to control the data Flow namely ALU_{in} and ALU_{out}. These signals operate similar to Register In and Out signals.

☞ ALU Function Control signals

Set of control signal lines are associated that specify the function to be performed in the ALU. For each function in the ALU there is a unique combination of function control signals. The typical ALU functions are Arithmetic functions (such as ADD, SUB, MUL, DIV, INC, DEC, CMP, etc.) and Logical functions (such as AND, OR, XOR, NOT, etc.)

⏭ 9.2 TYPES OF CONTROL UNIT

UQ. 9.2.1 Explain different techniques for design of control unit of computer.

MU - Q. 3(A), May 19, 10 Marks

- Control Unit of the computer system is very important component of the system. Based on the decoded instruction, it is responsible for generating all internal as well as external control signals over the computer system so as to execute that instruction correctly. The implementations of the Control Unit can be primarily classified into two types –
  1. Hardwired Control Unit
  2. Microprogrammed Control Unit
- Hardwired control unit implementation is done with the help of hardware i.e. It consists of digital combinational and sequential logic components. They operate as a Finite State Machine (FSM). In the process, the digital FSM generates desired sequence and combination of control signals.
- Microprogrammed control unit is a software implementation. It stores a specific sequence of micro-instructions for every instruction and these sets of micro-instructions are stored in a control memory which is accessed by Micro-PC decoded based on the instruction to be executed on the system. A specific sequence of Micro-instructions is operated to generate the desired combination and sequence of control signal and therefore, executing the specific instruction.

⏭ 9.3 HARDWIRED CONTROL UNIT

UQ. 9.3.1 Describe hardwired control unit and specify its advantages.

MU - Q. 2(b), May 17, Q. 2(b), Dec.17, Q. 4(a), Dec. 16, Q. 3(a), Dec. 15, 10 Marks

UQ. 9.3.2 Explain with diagram functioning of Hardwired Control unit.

MU - Q. 2(a), May 15, 8 Marks

- Hardwired control unit implementation is done with the help of hardware i.e. It consists of digital combinational and sequential logic components.
- Based in Instruction to be executed and the status of condition codes, they operate a Finite State Machine (FSM). In the process, the digital FSM generates they require combination and sequence of control signals needed to successfully execute the instruction.
- Design of Hardwired control unit is guided by following factors :
  (i) Complexity and amount of Hardware used
  (ii) Desired speed of generation of control signals
  (iii) Costing (Economy) of the design
- Hardwired Control Unit implementations are possible to be done using different hardware designing techniques. Based on the technique used for its implementation, methods of Hardwired control units are classified as :
  1. **State-table Method** : It is a method classical in nature and creates Moore or Mealy state tables to generate control signal outputs.
  2. **Delay-element Method** : It is the method of implementation that used stages of building blocks of clocked Delay-element (i.e. Series of D-Flipflops) in order to generate the desired control signals.
  3. **Other Methods - Sequence-counter Method and PLA Based Methods** : This method used Counters. It tries to reduce the amount of Sequential Hardware (Flipflops) and generate the desired control signals. PLA Method uses Programmable Logic Arrays to implement the Hardwired control unit.

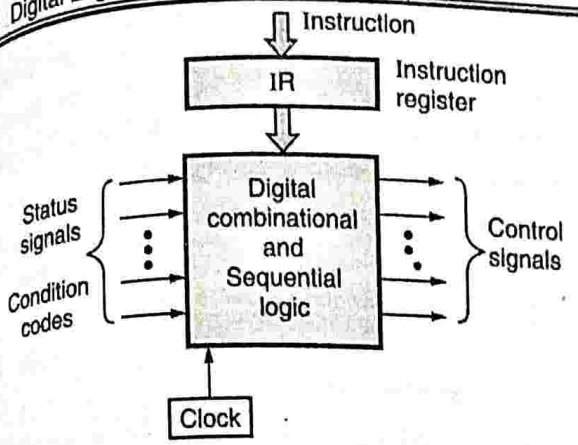


Fig. 9.3.1 : Basic Hardwired Control Unit

9.3.1(a) State-table Method or Classical Hardwired Control Unit

- State transition table are created in the State-table method implementation of Hardwired control unit. In every state, the control unit generates the desired sent of control signals. As the states are transited, sequence of appropriate control signal sets get generated, in turn executing the specific instruction.
- The transition of the state machine to the next state is done on the basis of Current state and the current Inputs to the control circuit.
- The book implements Hardwired control unit design taking the example of an unsigned multiplication operation.

9.3.1(b) A Multiplier Implementation for Hardwired Control Unit (State Table Method)

- Fig. 9.3.2 shows the flowchart explaining the unsigned Multiplication logic. Let us have Multiplicand in M register and Multiplier in Q register respectively. C contains Carry generated during any additions.
- A register is an accumulator and used for computational purpose in the multiplication. Let n be the count (i.e. Number of bits to be multiplied).
- The states of multiplication logic are as follows :

S_0: Start State

S_1 : Initialization State – In this state, registers are initialized. C and A are cleared. M and Q are loaded with Multiplicand and Multiplier respectively. Q_0 bit of register Q is checked

IF $Q_0 = 1$ THEN GO TO state S_2 ELSE GO TO state S_3

S_2 : Addition State – In this state, addition operation is done, $A \leftarrow A + M$; if carry $C \leftarrow$ carry

S_3 : Shift State – In this state shifting is done, Right-Shift C, A, Q; count $\leftarrow$ count – 1

IF count > 0 THEN GOTO state S_2 ELSE Done.

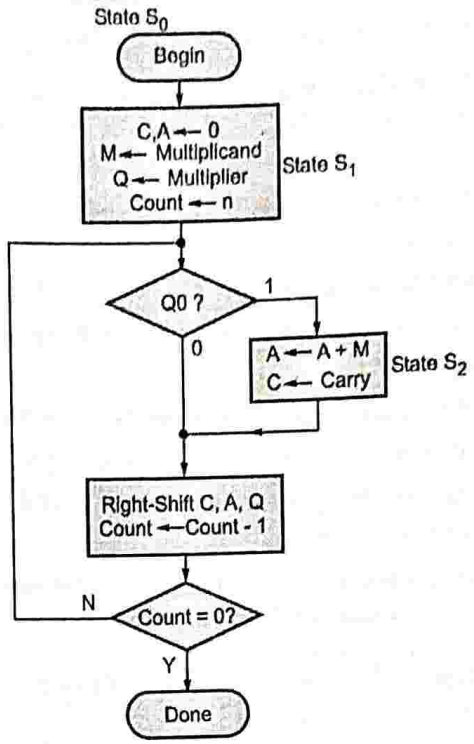


Fig. 9.3.2 : Flow chart for multiplier implementation

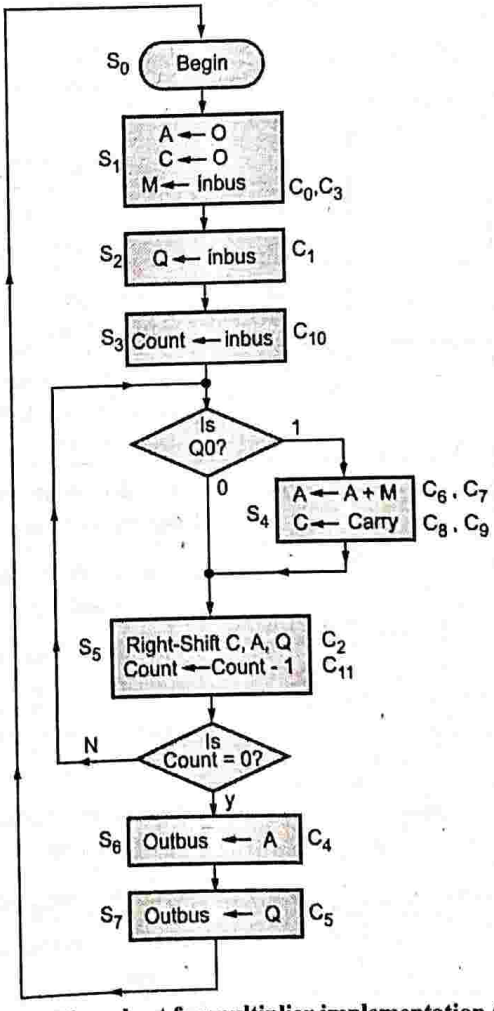


Fig. 9.3.3 : Flow chart for multiplier implementation (added with control signals)

Module 4

Control Signal Generation for Multiplier

- Control signals are needed to be determined for the multiplier unit. Let us consider schematic of the multiplier implementation with control signals.
- In order to handle single transfer of contents on the buses at any given instant, we are needed to modify the state machine flowchart adding 2 more states. The flowchart of Multiplier hardwired unit implementation with control signals is as given in Fig. 9.3.3.
- Table 9.3.1 State table describes the States of Multiplier implementation of Hardwired controls, input signals triggering the state transitions, output control signals generated and their details:

Table 9.3.1 : State Diagram for Multiplier Implementation using State Table Method

State	Input Signal evaluated	Output Control Signal generated	Details of Control Signal
S_0	Begin	--	Beginning of the Multiplication operation
S_1		C_0	Transfer (input) Multiplicand M from Inbus
		C_3	Clear Accumulator Register A and Carry C.
S_2		C_1	Transfer (input) Multiplier Q from Inbus
S_3		C_{10}	Transfer (input) count value from Inbus into count register.
S_4	Q_0	C_6	Transfer M to the adder unit for addition
		C_7	Transfer A to the adder unit for addition
		C_8	Transfer addition results to A upon addition
		C_9	Transfer carry out to C after addition
S_5	count	C_2	Right-shift contents of C,A,Q by 1 bit
		C_{11}	Decrement the Count by 1
S_6		C_4	Output A contents to the Outbus
S_7		C_5	Output Q contents to the Outbus

- Using the States and State transitions described in above-mentioned table, we are able to determine the functional blocks needed to implement the multiplier unit and its control signals needed for the hardware implementation.
- It is clear that the implementation needs registers, adders, comparators, multiplication hardware and shifters. The implementation is as given in Fig. 9.3.4.

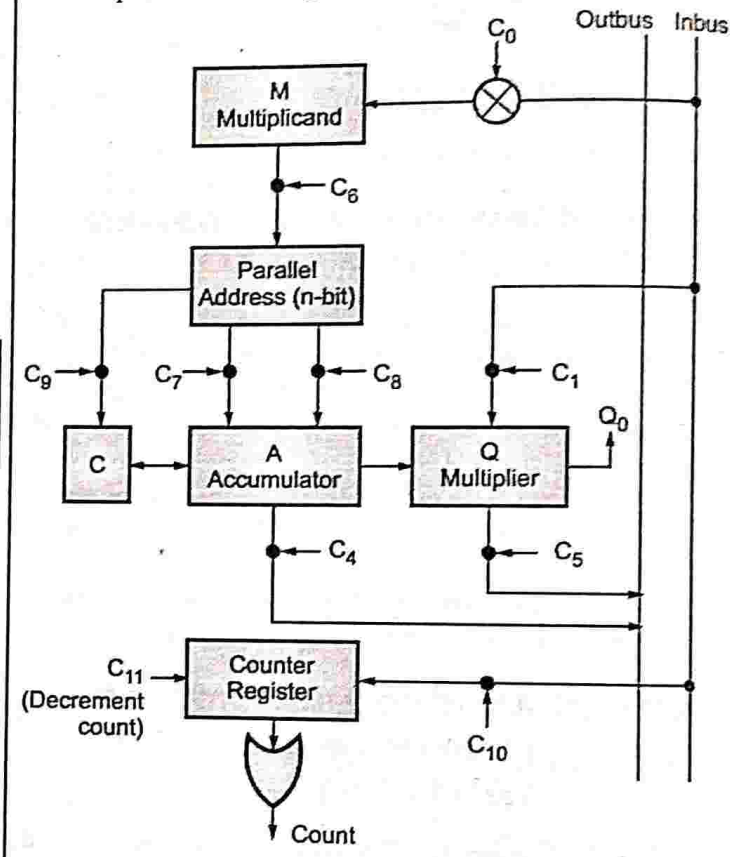


Fig. 9.3.4 : Hardware Block Diagram of Multiplier Unit Implementation

9.3.2(a) Delay - Element Method

- The control signals from the Hardwired control unit are needed to be generated in an appropriate sequence. Since the states are drawn to identify the control signals to be generated simultaneously in a state. As the state transition occurs, next set of control signals is generated in the next state and so on.
- There is a specific time delay between the generation of two consecutive sets of control signals. Therefore, a sequence of delay elements can be used to generate control signals one after the other. To ensure proper operation synchronized to the clock signal, the delay elements are implemented by D flip-flops. A common clock signal is used to synchronize.

9.3.2(b) Multiplier Implementation for Hardwired Control Unit (Delay Element Method)

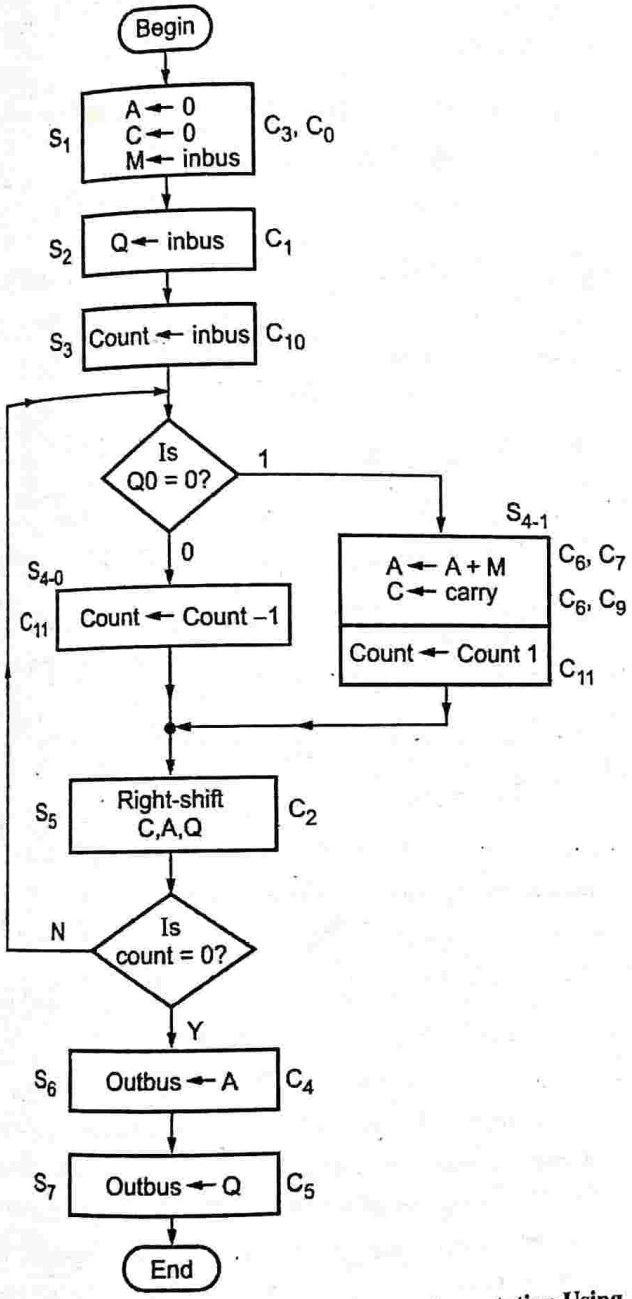


Fig. 9.3.5 : Flow chart for multiplier implementation Using Delay-element Method

- Fig. 9.3.5 shows the flowchart used for the Delay-element method implementation of the Hardwired control unit, for the Multiplier logic.
- Set of Guidelines for using Delay-element Method :
- The implementation of Hardwired control unit using delay elements can be conceived and derived from the flowchart that specifies required control signal sequences. We use the same Multiplier unit implementation example for this purpose.
- Every state requires a delay element. The Delay element is implemented through Clocked D-Flip-flops.

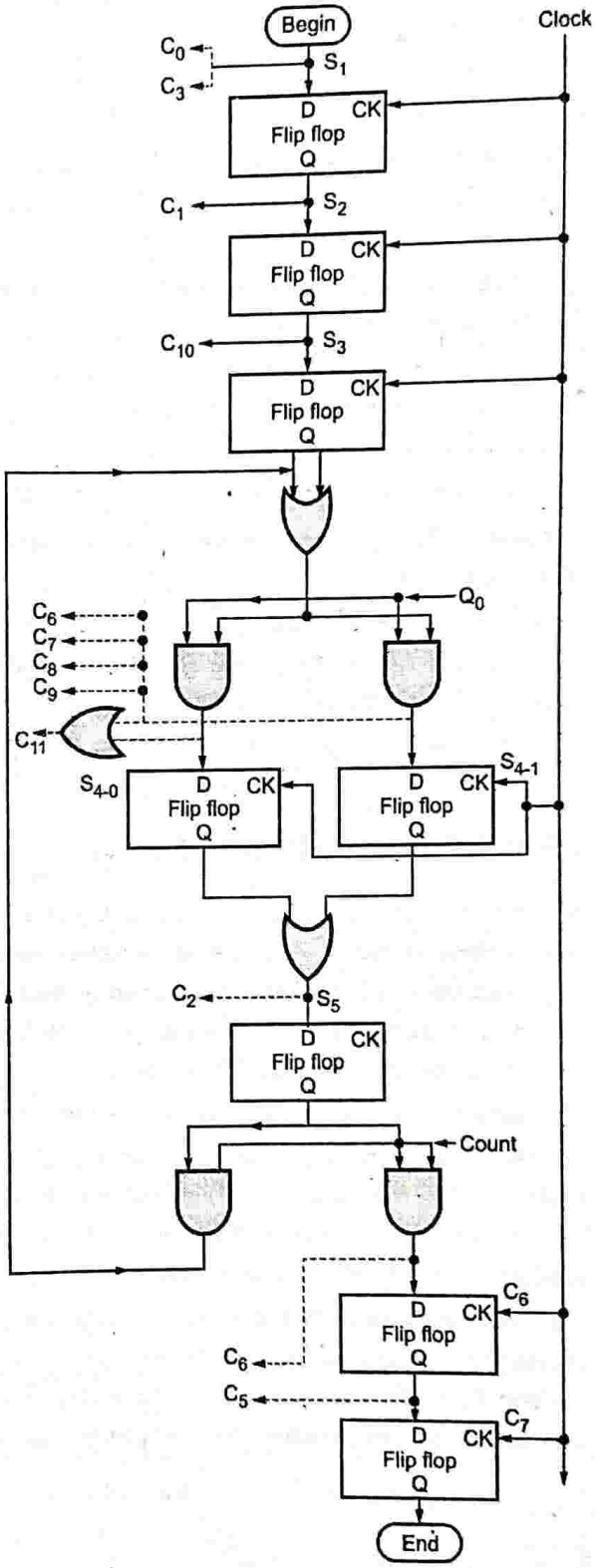


Fig. 9.3.6 : Flow chart for multiplier implementation using Delay-element Method

Module 4

- The signals in different states that activate same control signals are logically 'OR' ed to get one common output control signal.
- Multiple control signals generated in only one state are simply wired OR.
- When multiple lines in the flowchart merge to a common point then these lines are connected to an equal input OR Gate.
- A decision box is implemented by two 2-input AND gates. The first input of each AND gate is driven by the desired input and complement of the desired input respectively, while the second input of both gates is common, and it is the output of the D Flip-flop representing the Delay-element.
- States are identified at each of the Delay-element Flip-flop.
- Common synchronizing clock signal is connected to all Delay element Flip-flops.
- Using these guidelines for the hardware implementation of Multiplier unit for constructing the Delay-element method based Hardwired Control Unit. The resultant hardware implementation of the Multiplier unit is as shown in Fig. 9.3.6.

9.3.3(a) Sequence Counter Method

- Sequence Counter method is the classical digital design method where we use the Moore and Mealy tables generated in the state table method to find the functional synthesis and derivation for the next state functions and output functions in terms of current state variables and input variables.
- This method uses either AND-OR-Inverter or NAND-NAND or NOR-NOR logic implementations for realizing the combinational circuits and uses JK flip-flops or D flip-flops for the encoding of states and thereby the Finite State Machine of the sequential logic to be implemented.
- The method achieves the hardware circuit implementation of combinational as well sequential digital logic with minimum number of state flip-flops. However, as the number of state and input variables increase, the complexity increases exponentially for synthesizing the functions for implementing the next states and output (control signals)
- Following sections demonstrate the Hardwired control unit component implementations for multiplier and divisor using Sequence counter method.

9.3.3(b) Multiplier Implementation for Hardwired control unit (Sequence counter method)

- Referencing the State table generated for the multiplier, we can derive the functions for Next state variables in terms of current state variables and input variables as :

$$D_0^+ = D_0 \cdot \overline{BEGIN} + D_7$$

$$D_1^+ = D_0 \cdot BEGIN$$

$$D_2^+ = D_1$$

$$D_3^+ = D_2 \cdot Q[0] + D_4 \cdot Q[0] \cdot \overline{COUNT7}$$

$$D_4^+ = D_2 \cdot \overline{Q[0]} + D_3 + D_4 \cdot \overline{Q[0]} \cdot \overline{COUNT7}$$

$$D_5^+ = D_4 \cdot Q[0] \cdot COUNT7$$

$$D_6^+ = D_5 + D_4 \cdot \overline{Q[0]} \cdot \overline{COUNT7}$$

$$D_7^+ = D_6$$

- Similarly, we can derive the functions for output variables in terms of current state variables and input variable as :

$$c_0 = c_1 = c_{11} = D_4 \qquad c_2 = c_3 = c_4 = D_3 + D_5$$

$$c_5 = D_5 \qquad c_6 = D_6$$

$$c_7 = D_7 \qquad c_8 = D_2$$

$$c_9 = D_{10} = D_1 \qquad END = D_0$$

- On the basis of the derived Next states and Output variables (Control signals) in terms of current states and Input variables, using the all-NAND-NAND logic implementation for combinational circuits and using D flip-flops for the sequential circuit, the hardwired control unit (using sequence counter method) implementation for Multiplier unit is constructed. The implementation is as shown in Fig. 9.3.7.

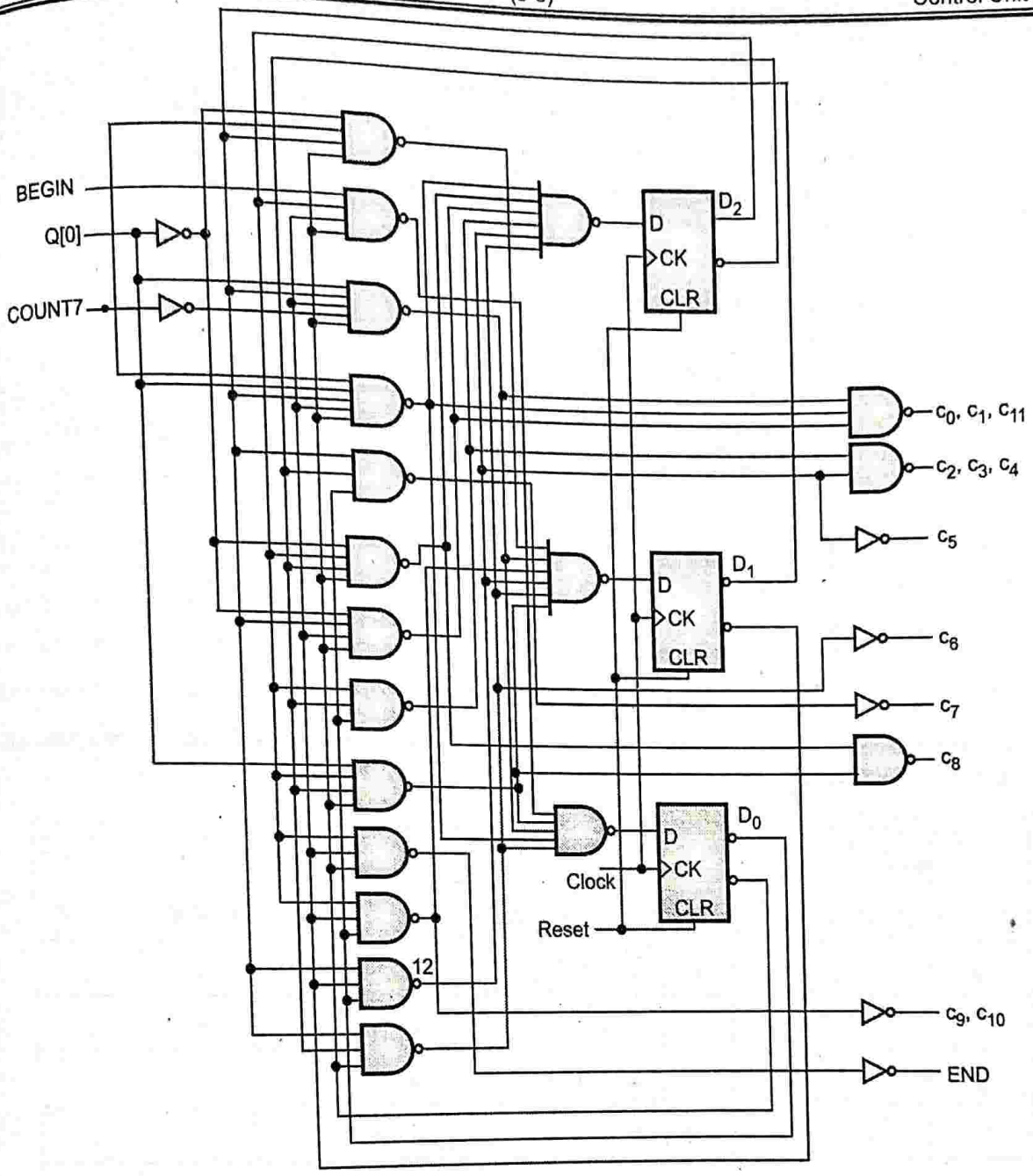


Fig. 9.3.7 : Multiplier implementation using Sequence Counter method

9.3.4 Division Implementation for Hardwired Control Unit

- Division is repeated subtraction till the remainder is 0 or positive. Using this property of Division as the repetitive subtraction, it is possible to implement the division logic. Let us consider the problem statement of Greatest Common Divisor (GCD) which can be demonstrated as the application that incorporates the division logic as stated above. Let us consider 2 registers XR and YR as general purpose registers. Also, the implementation would use a multiplexer, a subtractor and a few comparators.
- Fig. 9.3.8 shows the schematic of hardware implementing GCD.

Module 4

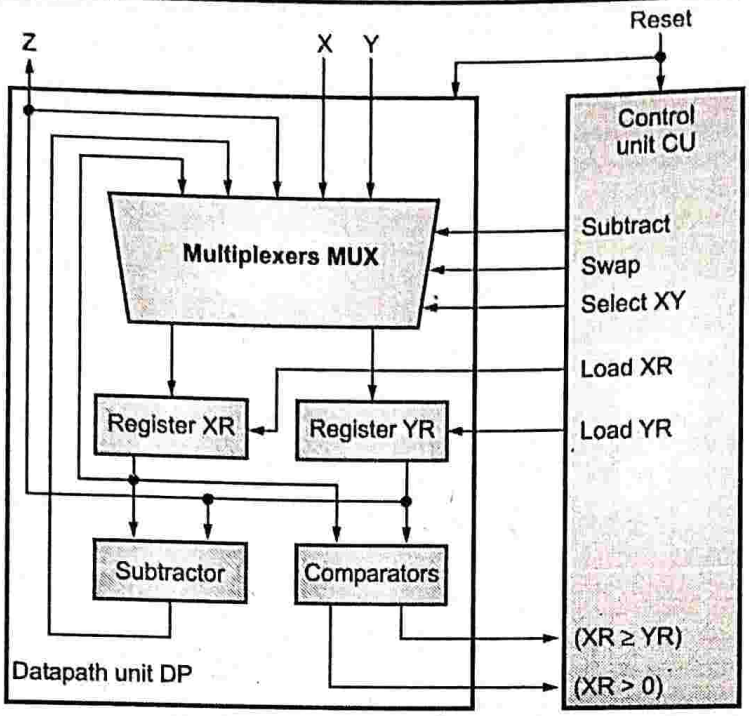


Fig. 9.3.8 : GCD implementation

- Using the GCD implementation logic, we can arrive at the excitation state table for the GCD. It states the next states and output control signals in terms of current states and input variables of the system.
- The state table is given as :

Inputs		Present state		Next state		Outputs				
(XR > 0)	(XR ≥ YR)	D_1	D_0	D_1^+	D_0^+	Subtract	Swap	Select XY	Load XR	Load YR
0	d	0	0	1	1	0	0	1	1	1
0	d	0	1	1	0	0	1	0	1	1
0	d	1	0	1	1	1	0	0	1	0
0	d	1	1	1	1	0	0	0	0	0
1	0	0	0	0	1	0	0	1	1	1
1	0	0	1	1	0	0	1	0	1	1
1	0	1	0	0	1	1	0	0	1	0
1	0	1	1	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	1	1	1
1	1	0	1	1	0	0	1	0	1	1
1	1	1	0	1	0	1	0	0	1	0
1	1	1	1	1	1	0	0	0	0	0

Using this state table, for implementing the required combinational and sequential logic, we synthesize and derive the digital functions for Next state and output control signal variables. The signals are derived as :

$$D_0^+ = 0$$

$$D_1^+ = D_0 \cdot (XR > 0) \cdot \left(\overline{XR \geq XR}\right) + D_2 \cdot (XR > 0) \cdot \left(\overline{XR \geq XR}\right)$$

$$D_2^+ = D_0 \cdot (XR > 0) \cdot (XR \geq XR) + D_1 + D_2 \cdot (XR > 0) \cdot (XR \geq XR)$$

$$D_3^+ = D_0 \cdot \left(\overline{XR > 0}\right) + D_2 \cdot \left(\overline{XR > 0}\right) + D_3$$

$$\text{Subtract} = D_2$$

$$\text{Swap} = D_1$$

$$\text{Select XY} = D_0$$

$$\text{Load XR} = D_0 + D_1 + D_2$$

$$\text{Load YR} = D_0 + D_1$$

On the basis of the derived Next states and Output variables (Control signals) in terms of current states and Input variables, using the all-NAND-NAND logic implementation for combinational circuits and using D flip-flops for the sequential circuit, the hardwired control unit (using sequence counter method) implementation for GCD unit is constructed.

The implementation is as shown in Fig. 9.3.9.

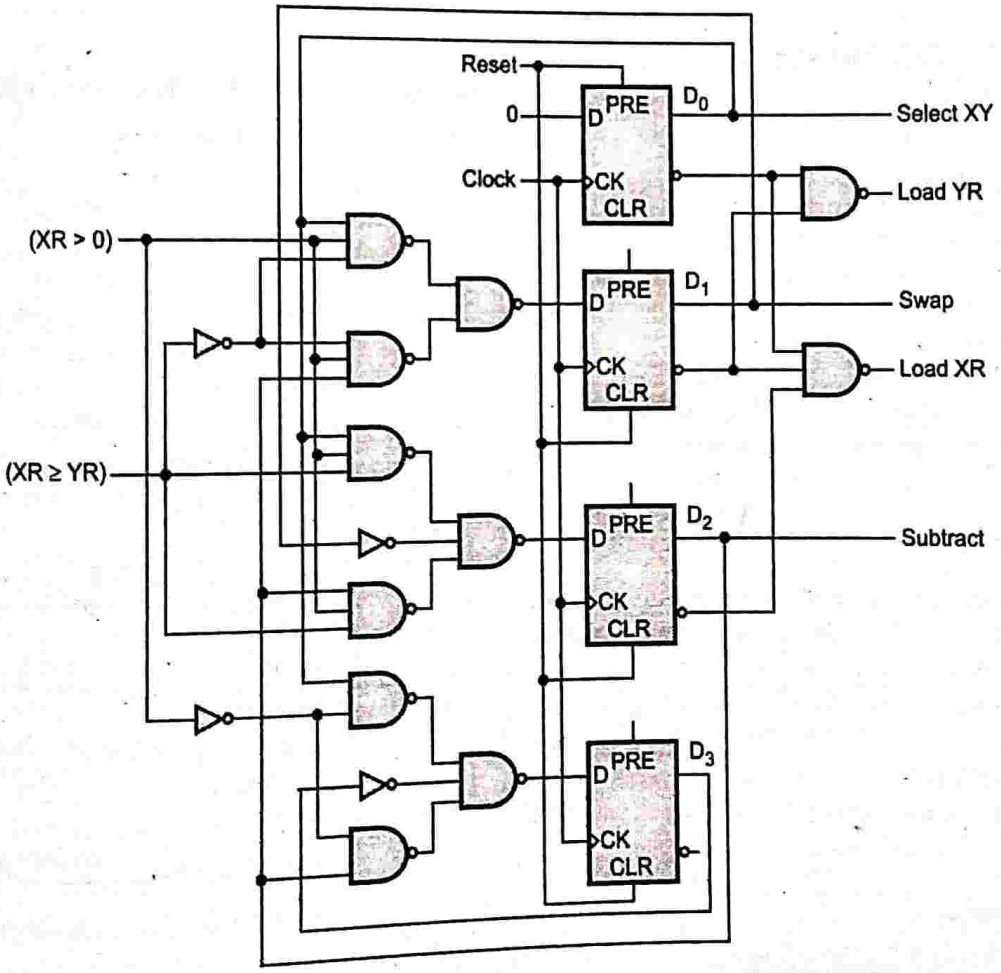


Fig. 9.3.9

Module 4

9.3.5 Advantages of Hardwired Control Unit

1. The execution speed of Hardwire control unit is fast as compared to micro programmed control unit.
2. It can accommodate less number of instructions, it is better suited for RISC architecture implementation.

9.3.6 Disadvantages of Hardwired Control Unit

1. The probability of error is more in Hardwire control unit.
2. It is difficult to handle complex instructions.
3. The design process for hardwire control unit is more complicated.
4. The chip area of Hardwire control unit is more and the cost of manufacturing is high.
5. Instruction sequencing and decoding is difficult to implement.
6. Entire control unit design needs to be changed in order to add new instructions for the system.

9.4 MICRO PROGRAMMING TERMINOLOGIES

1. Micro – operation

- Micro – operations are the smallest low-level, detailed instructions used to develop complex machine instructions. Usually, for every micro – operation, the control unit only generates a set of control signals.
- Thus for each micro – operation, the control lines from the control unit are either ON or OFF that is having binary representation. So every micro – operation can be represented by a pattern of 1s and 0s in the control word.

2. Micro – Instruction

- Micro – instructions are a set of micro – operations that are supposed to be executed at one time. Each assembly language instruction defines a symbolic micro – instruction.
- A symbolic micro – instruction is the one that can be easily translated by assembler into its binary equivalent. Each symbolic micro – instruction can be divided into five fields namely: label, micro – operation, condition, branch and address.

3. Micro-program

UQ. 9.4.1 What is Microprogram ?

MU - Q. 3(b), Dec. 18, Q. 3(B), May 19, 10 Marks

- A sequence of micro instructions is known as a micro program or firmware. Micro program is a midway between the hardware and the software.
- It is easier to design a firmware than the hardware, hence micro programmed control units are preferred over hard wired control units. Micro programs are stored in the control memory.
- Once the control unit is in operation, alterations should not be made to the micro program. Hence the control memory is usually selected as the read only memory, ROM.
- The micro program is used to place all the control variables forming a word in ROM that can be used by the control unit through successive reads. Each word in the ROM at a specific address consists of a micro instruction.

4. Micro-code

- Commonly used subroutines in the micro program are called as micro codes. E.g. The sequence of micro operations that are needed to generate the effective address of an operand in an instruction is common for all memory access instructions.
- This sequence of micro operations can be put together to form a micro code.

9.5 MICRO INSTRUCTIONS AND ITS FORMAT

- Micro instructions consist of the micro operation, next instruction's address field, conditional parameters, conditional execution's address and label. The format for the micro instruction or control word is as follows:
- There is one bit for each internal control unit's control line and one bit each for the system bus control line. There is a conditional field, consisting of conditional flags, that indicate the condition for branching. Also, one address field that has address of the micro instruction to be executed if branching takes place.

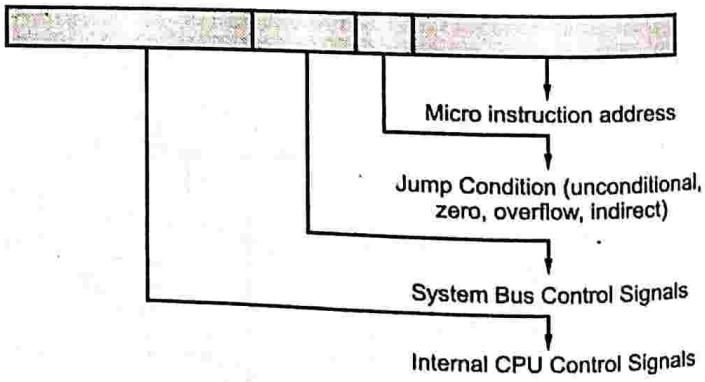


Fig. 9.5.1 : Horizontal Micro instruction

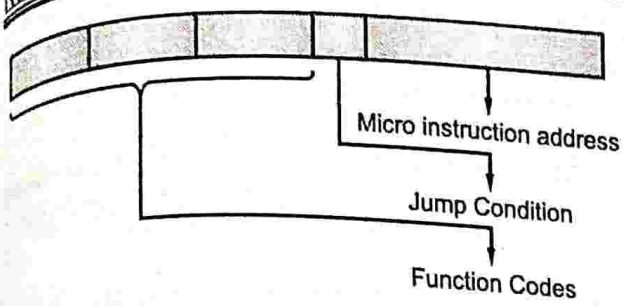


Fig. 9.5.2 : Vertical Micro instruction

Such a micro instruction is interpreted as follows :

1. To execute a micro instruction, all the control lines should be turned ON, indicated by bit 1, whereas turn OFF the control lines indicated by bit 0. The resulting control signals will result into one or more micro operations to be performed.
2. If the condition indicated by the conditional bits is false, execute the next micro instruction in sequence.
3. If the condition indicated by the conditional bits is true, execute the next micro instruction whose address is specified in the address field.

- As shown in the Fig. 9.5.1, the horizontal microinstructions have individual bits for the control signals, therefore they are directly connected to the control lines of the control unit.
- In Vertical microinstructions, individual control signal is not generated rather a function code is generated depending upon these control signals. Therefore, when using vertical micro instructions, the control lines are not directly connected to the register (Command Buffer Register). The function code is first translated into individual control signals using a decoder which are then connected to individual control signals.

Observation	Horizontal Micro instruction	Vertical Micro instruction
Control Signals generation	Individual control signals are generated.	Control signals in the form of function code is generated
Decoder required	No	Yes. To translate the function code into individual control signal.
Connection to control lines	Each control signal is directly connected to control line.	The function code is decoded to give individual control signals which are connected to control lines.
Access of individual control bit or signal.	Possible	Not Possible.

9.6 CONCEPT OF CONTROL MEMORY

- Control memory is used to store all the micro instructions for the processor or control unit. There are several micro instructions in any routine to be performed. All micro instructions in a routine are to be executed sequentially.
- Every routine ends with a branch or jump instruction that gives information about the next instruction or routine to be executed.
- The control memory thus defines the sequence of micro operations to be performed during each of the fetch or indirect or execute or interrupt cycle. It also specifies the sequencing of these cycles.
- Thus, control memory forms a way to implement a control unit.

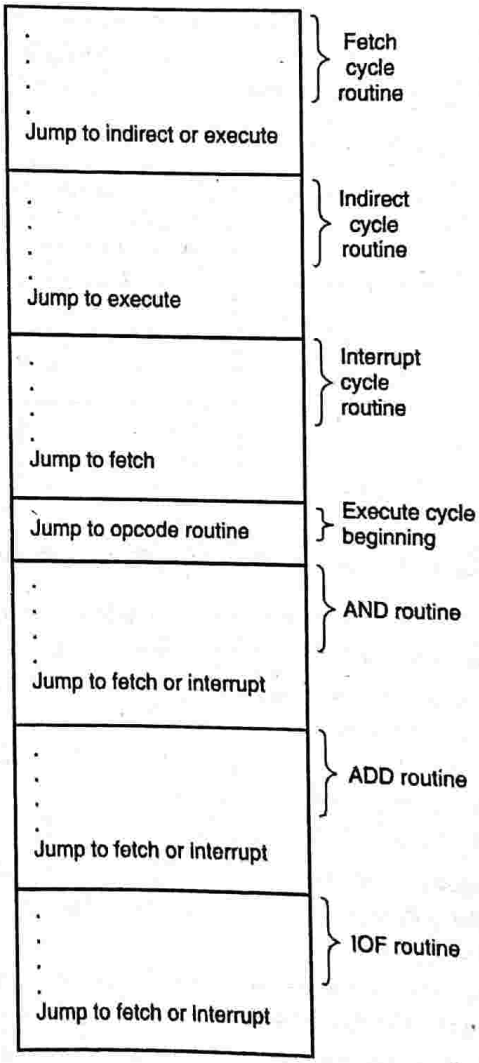


Fig. 9.6.1 : Organization of Control Memory

Module 4

9.7 MICRO PROGRAMMED CONTROL UNIT

UQ. 9.7.1 Explain Micro-programmed control unit. MU - Q. 5(b), May18, 5 Marks

9.7.1 Micro Programmed Control Unit : Concept

- The control unit is a simple device. To design it as interconnection of basic logical units is a very difficult task because it needs to have logic for sequencing the micro-operations, logic for executing those micro-operations, logic for interpreting opcodes, and logic for making decisions based on ALU flags.
- It will be very difficult to test and implement such complex hardware, Also, if at all any new instruction is to be added, making changes will be even more difficult. Instead a **Microprogrammed Control Unit** is implemented.

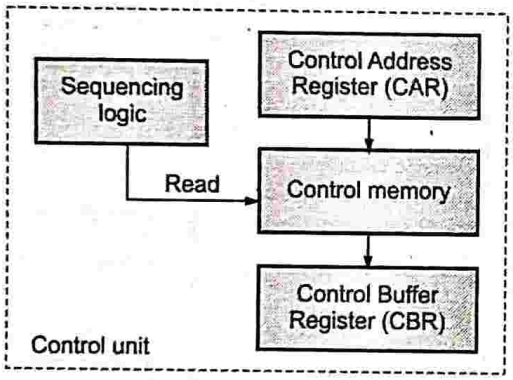


Fig. 9.7.1 : Micro Architecture of a control unit

- Thus, a micro programmed control unit is a control unit whose binary control variable are stored in memory.
- The control memory contains a program that describes the behavior of the control unit. In micro programmed control unit, implementing the control unit would be equivalent to executing this program in the memory.
- Fig. 9.7.1 shows the Microarchitecture of control unit.
- It basically consists of Control Memory, Control Buffer Register (CBR), Control Address Register (CAR), Sequencing Logic.
- Control memory stores the micro instructions. The CAR stores the address of the next micro instruction to be read.
- The CBR stores micro instruction after it has been fetched from the control memory.
- The sequencing unit loads the CAR with the address of the next instruction depending upon branching and issues a read command.
- The micro instructions have fields that describe the System bus control and the Internal CPU control signals. These fields are connected to the command or control lines emanating from the control unit.
- Thus, reading a micro instruction is equivalent to changing the control signals of the control line, which in turn is equivalent to executing that micro instruction.

9.7.2 Functioning of Microprogrammed Control Unit

UQ. 9.7.2 Explain with diagram functioning of Micro programmed Control Unit. MU - Q. 2(b), May 14, 8 Marks

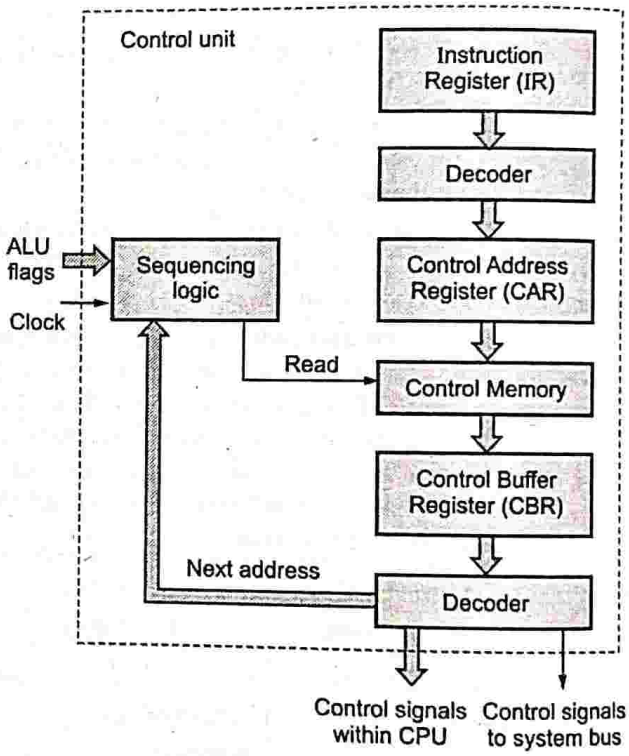


Fig. 9.7.2 : Functioning of the Micro programmed Control Unit

The control unit has three inputs namely : ALU flags, clock and Instruction Register (IR). It gives two groups of control signals : one to external system bus and other internal within CPU. To perform any operation the control unit performs the following tasks in one clock pulse :

1. The sequencing logic issues a READ command for the control memory.
2. The instruction from the address specified in the CAR is read into CBR.
3. The contents of the CBR, i.e. the micro instruction, generates the control signals for internal as well as external bus and the information about the next address for the sequencing unit.

4. Finally, the sequencing logic decides the address of the next instruction to be loaded into CAR depending upon information given by CBR and ALU flags. One of the three decisions is made to generate the next instruction address.
   (a) No branching, so Get the next Instruction by adding 1 to the CAR.
   (b) Jump to a new routine based on micro instruction jump : If the ALU flags satisfy the condition field of the micro instruction, load the CAR with the address field of Control Buffer Register i.e. the address field of the micro instruction.
   (c) Jump to a machine instruction routine : Use the opcode from the IR to load the CAR.

- There are two decoders present in the Micro programmed Control Unit apart from the CAR, CBR, control memory and Sequencing Logic.
- The upper decoder is used to translate the opcode of the IR into the address of the control memory.
- This element is present in both Horizontal and Vertical micro instructions.
- For vertical micro instructions, since the control signals are generated in the form of a code rather than direct signals, this code needs to be converted into individual control signals.
- The lower decoder does the job of converting this functional code to individual signals to be connected to control lines.

9.7.3 Advantages of Microprogrammed Control Unit

1. The main advantage of micro programmed control unit is that it simplifies the control unit design.
2. It is cheaper and is less error prone to implement.
3. The decoder and the sequencing logic is simple.
4. Easy to implement hence used for CISC and mainframes.

9.7.4 Disadvantage of Microprogrammed Control Unit

1. The main disadvantage of micro programmed control unit is that it is slower as compared to hardwired control unit.

9.7.5 Wilke's Control Unit

UQ. 9.7.3 Explain Wilke's Engine (Hardwired Control Unit) in detail.
MU - Q. 3(a), Dec. 14, 10 Marks

Wilkes first proposed the use of micro programmed control unit. The design is basically based upon a matrix that is partially filled with diodes as shown in Fig. 9.7.3 :

- During a machine cycle, one row of the matrix is activated by a pulse. This pulse generates signals at those points where diodes are present (represented as dots in Fig. 9.7.3). Each row is a microinstruction. It consists of two sections. One section generates control signals for the internal working of the control unit.
- The second section generates the address of the row to be activated in the next machine cycle. The entire matrix layout therefore resembles the storage area for all micro instructions. It is basically the control memory.
- Two registers are involved in Wilke's control unit. Register – I is used to store the address of the row to be pulsed at the start of the cycle. The address from Register – I is given to the address decoder.

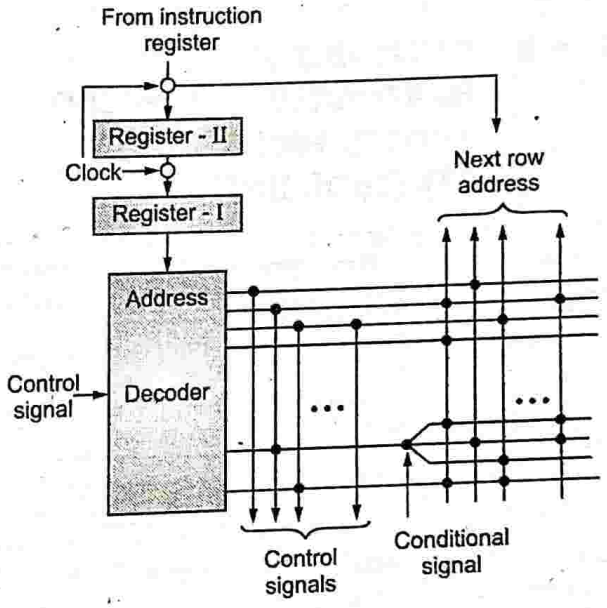


Fig. 9.7.3 : Wilke's Microprogrammed Control Unit

- When a clock pulse is applied to the decoder it activates the matrix row given by Register – I. Depending upon the control signals the address of the next row is either taken from the opcode in the Instruction Register or the from the second section of the pulsed row.
- This address is given the Register – II during the cycle. Next clock is then used to pass this row address from Register – II to Register – I. Therefore, two clock pulses are required to perform one operation. One clock pulse to activate the matrix row and second pulse to transfer address from register – II to register – I.
- Wilke's control unit behaves like horizontal microprogramming, but the next address is always present in the microinstruction.
- Therefore, each micro instruction consists of two addresses. One branched address and the second auto incremented address. The selection is decided by the control signals.

Module 4

- The Wilke's control unit has a total of 38 micro instructions that define the system completely. Also, it has two 1-bit flags and total seven registers as shown below.

A	Multiplicand
B	Accumulator (LSH)
C	Accumulator (MSH)
D	Shift Register
E	Memory Address Register (MAR) as well as temporary storage
F	Program Counter
G	Temporary Register for counting

9.8 COMPARISON OF HARDWIRED AND MICRO PROGRAMMED CONTROL UNIT

Observation	Hardwired Control Unit	Microprogrammed Control Unit
Execution Speed	Comparatively Fast	Comparatively Slow
Error Probability	More	Less
Control Function implementation	Hardware	Software (Micro operations)
Complex Instructions	Difficult to handle	Easy to Handle
Chip Area	More	Less
Cost	More	Less
Design Process	Complicated	Simple, Orderly and Systematic
Sequencing and Decoding	Complex	Easy
Capability to incorporate changes (new instruction)	Complete redesign is required.	Possible to alter since control unit is in software
Number of Instructions	Preferably Less	Preferably More
Application	RISC processors	CISC processors, Main frames

9.9 MICRO INSTRUCTION SEQUENCING

UQ. 9.9.1 Explain micro instruction sequencing.

MU - Q. 5(c), May 15, 7 Marks.
Q. 2(b), Dec. 16, 05 Marks

There are two basic tasks performed by microprogrammed control unit i.e. microinstruction sequencing and microinstruction execution.

- **Microinstruction Sequencing :** Once a microinstruction is executed, get the next instruction from control memory
- **Microinstruction execution :** This generates the required control signals for the microinstructions to get executed.
- In the following section we will discuss in detail some concepts of microinstruction sequencing.

Design Considerations

- Deciding the size of the microinstruction and the address generation time are the two major apprehensions for designing microinstruction sequencing techniques. Minimizing the size of microinstruction also reduces the size of components, hence reducing the cost. Lesser the address generation time faster will be the time to execute microinstruction.
- The address generated by microinstruction can be of one of the following categories:
  - Next address is determined by the instruction register
  - Next address can be the next sequential address
  - Branch
- The first category occurs when a new instruction is fetched, i.e. once per instruction cycle.
- The second category occurs normally in almost all the designs. But branching is the integral part of microprogramming and for optimum design only sequential design should not be included.
- The branching gives flexibility for the programmer in implementing different logical cases. Both conditional and unconditional branching are very important part of microprogramming.
- In microprogramming, branching is more often used. Hence it is very important to design compact, time-efficient techniques for branching.

9.9.1 Sequential Techniques - Micro Instruction with Next Address Field

- Whenever a microprogram is getting executed, the address of next instruction to be executed is generated.
- This address is generated by considering the contents of different register like instruction register, condition flags, etc.
- The technique used to generate address of next instruction is

called as sequential techniques.

These techniques can be classified based on the format of address information in the microinstruction. These categories can be given as follows :

(i) Two address fields
(ii) Single address field
(iii) Variable format

(i) Two address fields

- This is the simplest technique used for generating address of the next microinstruction.
- This technique provides two address fields in each microinstruction.
- Fig. 9.9.1 shows how this technique is implemented.
- The control buffer register contains two separate fields for address. A multiplexer is used to select any one of the two addresses.
- The address selection logic is implemented using Branch logic block which takes input from different flag bits and control bits. The selected address is then given to the Control Address Register.
- The address decoder then decodes the address present in the Control Address Register to get the next microinstruction to be executed.

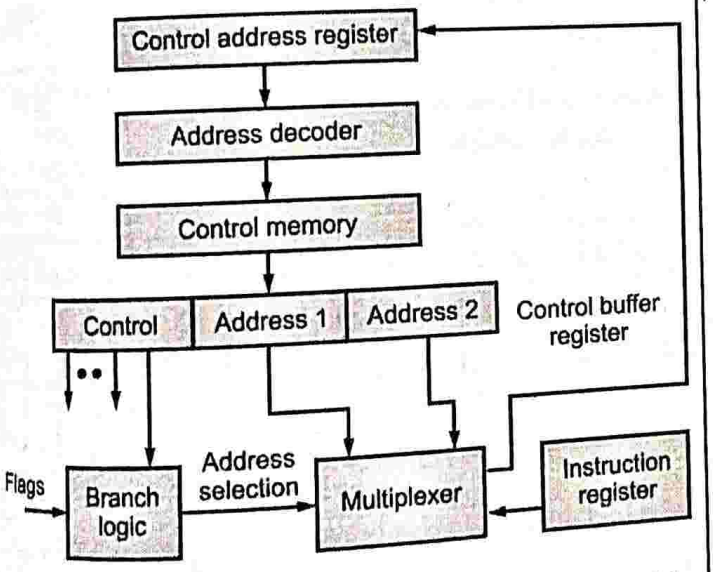


Fig. 9.9.1 : Microprogram Sequencing using two Address fields

(ii) Single address field

- The two address approach is easy to implement, but it requires a greater number of bits in microinstruction to implement.
- In single instruction format, some bits of microinstructions are saved by implementing some additional logic. The Fig. 9.9.2 shows the implementation of single address field.
- The options for next address in single address field are Address field, Instruction register code. Next sequential code. The Branch logic block generates address selection logic signal. This signal is given to multiplexer which selects one of the above-mentioned logic.

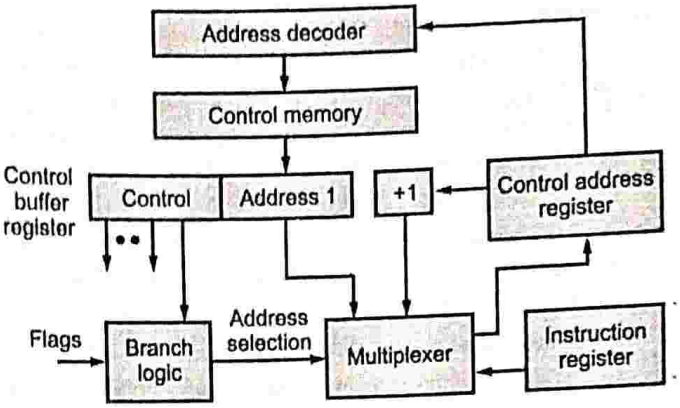


Fig. 9.9.2 : Microprogram sequencing using single Address field

(iii) Variable Format

- In this approach, the address field contains two different formats. One format contains address and remaining bits are used for control signals.
- The other format contains some bits contains branch logic module and the remaining bits contains address. The implementing of this approach is shown in the Fig. 9.9.3.
- In order to select one of the two formats, one dedicated bit is present in control buffer register. In the first format, the multiplexer selects address form Instruction register or the next address is the next sequential address.

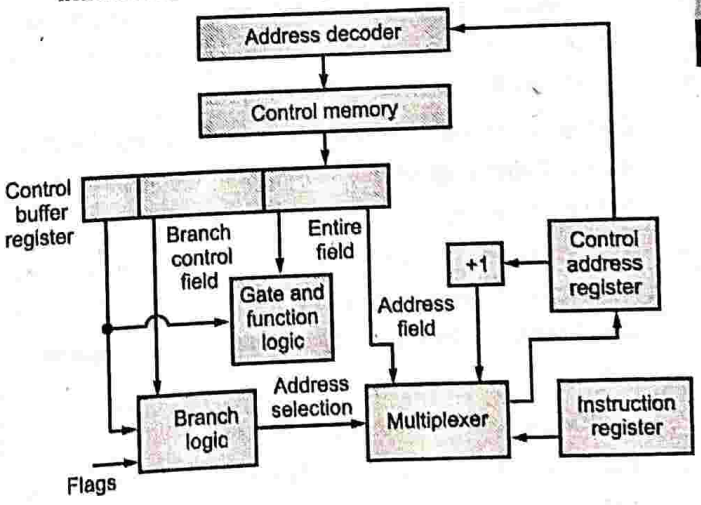


Fig. 9.9.3 : Microprogram sequencing using Variable Format

Address Generation

- The sequential techniques essentially deal with generation of address using different formats of control buffer register.

Module 4

- Another method finding different ways to compute next address.
- The various techniques of computing next address can be broadly classified into two categories namely implicit mapping and explicit mapping.
- Explicit mapping contains address mentioned into microinstruction itself. These techniques are already discussed in the previous sections. Hence, explicit mapping contains techniques such as Two-field, Unconditional branching, Conditional branching.
- Implicit mapping techniques require additional logic for address generation. These techniques are Mapping, Addition, and Residual Control.
- **Mapping Technique :** This is the commonly used implicit technique. In this technique the opcode portion of the machine instruction is mapped into a microinstruction address. This is done once per instruction cycle.
- **Addition Technique :** In this technique, two portions of the address are added and a complete address is formed. This technique was implemented in IBM3033 model. The control address of IBM3033 is shown in Fig. 9.9.4.

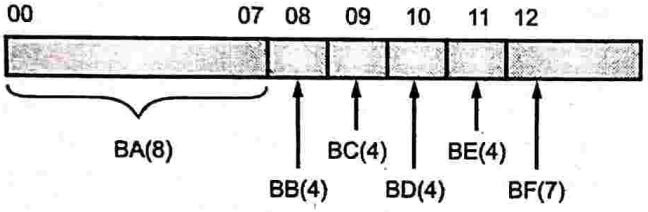


Fig. 9.9.4 : Microprogram Address Generation

- As shown in Fig. 9.9.4, the control address is 13 bit long and can be divided into two parts. The higher order bits remain same for all the microinstructions.
- During the execution of the microinstruction, the 8 bits of control address register are moved from the higher order 8 bits of control address.
- The remaining 5 bits of control address register specifies whether the microinstruction is going to be executed next or not. If these bits are set, the microinstruction will be executed next.
- If not set, the microinstruction will not be executed.
- **Residual Control :** In this technique, the microinstruction addresses which are previously used and stored in temporary storage within control unit are used again.

9.9.2 Wide Branch Addressing

- The instruction decoder decodes the instructions and determines which sequence of micro-instruction is needed to be generated for the execution of this specific instruction.
- As a result, it generates the starting address of the micro-routine (logical sequence of micro-instructions needed to execute any particular instruction) that implements the instruction that has just been loaded into the IR.
- If we take an example of Addition instruction ADD, register IR contains the ADD instruction coding, for which the instruction decoder generates the microinstruction address 1001.
- However, this address cannot be loaded as it is, into the microprogram counter. It is so that the source operand of the ADD instruction can be specified in several different addressing modes.
- Consider that there are five possible branches that the add instruction may follow based on the five different addressing modes in which our example ADD instruction can be operated, namely – indexed, auto decrement, auto increment, direct and register indirect addressing modes.
- Wide branch addressing technique allows the starting address of micro-routine be calculated differently for each addressing mode based ADD instruction.
- This is achieved by bitwise-'OR'ing technique to modify the starting address generated by the instruction decoder to reach the appropriate path.
- For the address shown in the Fig. 9.9.5, bitwise-'OR'ing changes the address field value of 1001 to one of the five possible address values 1011, 1021, 1031, 1041, or 1051 depending on the addressing mode used in the instruction.

Address field Instruction Op-Code specific								Addressing mode specific				Start Branch Address coding			
1				0				0				1			
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
								OR	OR	OR	OR				
Say for indexed addressing mode →								0	0	0	1				
0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1
1				0				1				1			

Fig. 9.9.5

Therefore, starting micro-instruction address for Indexed mode ADD instruction would be generated as 1011.

Addressing Mode	Common Start Address of Micro-instruction	OR	Addressing Mode Specific Bits in the Branch address	=	Final Start address for start of Micro-routine
IndexedAddressing	1001	V	0010	=	1011
Auto-Increment Addressing	1001	V	0020	=	1021
Auto-Decrement Addressing	1001	V	0030	=	1031
Direct Addressing	1001	V	0040	=	1041
Register Indirect Addressing	1001	V	0050	=	1051

- Similarly, for Auto-Increment mode 1021, for Auto-Decrement mode 1031, for Direct addressing mode 1041 and for Register Indirect addressing mode 1051 would be generated as the starting micro-instruction address in the micro-routine for execution of ADD instruction in those specific addressing modes respectively.

9.10 MICROOPERATIONS

- Primary functionality of any Computer or Processor based system is to execute the Instruction. The Instruction execution is operationally divided into smaller logical tasks. These tasks are called as Microoperations. Therefore, Instruction can be seen as the sequence of Microoperations.
- A micro-operation is a **fundamental action** performed by a machine on the data stored in the registers as a step towards or part of Instruction execution. Therefore, sequence of Micro-operations makes an Instruction to execute on the CPU.
- Based on type of Instructions to be executed, the Microoperations are classified as :
  1. Register transfer micro-operations
  2. Arithmetic micro-operations
  3. Logical micro-operations
  4. Memory Read Transfers (Fetching a Word from Memory)
  5. Memory Write Transfers (Storing a Word to Memory)
  6. Branch Instruction micro-operations
- A machine instruction is implemented by executing a set of micro-operations in a pre-defined sequence.

9.10.1 Register Transfer Microoperation

- This micro-operation transfers information from one register to another register.
- A register transfer micro-operation may be written in RTL (Register Transfer Language) as :

$$R_{destn} \leftarrow R_{src}$$

For example,

$R_2 \leftarrow R_1$, where R_1 is the source register and R_2 is the destination register.

$R_1 \leftarrow R_2$, where R_2 is the source register and R_1 is the destination register.

Operations involved in Register Transfer

- In order to transfer the contents of register R_1 to register R_2 :
- Contents of register R_1 are needed to be read and put on the internal CPU Bus. This can be done using the control signal R/#W of register R_1.
- When R/#W control signal is 1, read operation will be performed on R_1. Fig. 9.10.1 shows the schematic for such transfers :

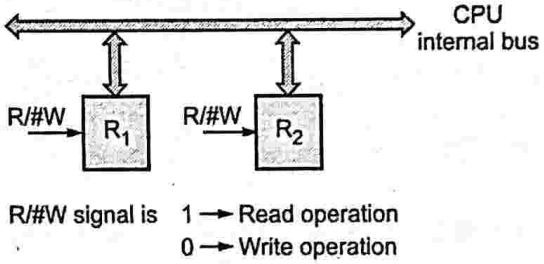


Fig. 9.10.1 : Register Transfer Micro-operation

- Data moving on internal CPU bus can be stored in register R_1 by making the control signal R/#W of register R_2 as 0.

R/#W of R_1	R/#W of R_2	Operation
0	0	No Operation
0	1	$R_1 \leftarrow R_2$
1	0	$R_2 \leftarrow R_1$
1	1	Invalid

Module 4

Register transfer operations with control signals

- The Fig. 9.10.1 shows such control signal functionality for facilitating Register transfer Microoperation.
- Usually, in and out signal conventions are used for Control signals for transferring data into the Register and transferring data out from the Register respectively.

Control signal $R_{1\,in}$ is equivalent to R/#W = 0

Control signal $R_{1\,out}$ is equivalent to R/#W = 1

Control signals asserted	Data Transfer Operation
$R_{1\ in}$	Internal bus to Register R_1
$R_{1\ out}$	Register R_1 to internal bus
$R_{2\ in}$	Internal bus to register R_2
$R_{2\ out}$	Register R_2 to internal bus

- Therefore, the operation represented by RTL :

  $R_2 \leftarrow R_1$ can be performed using the set of control signals → $R_{1\ out}$, $R_{2\ in}$.

- Similarly, the operation represented by RTL :

  $R_1 \leftarrow R_2$ can be performed using the set of control signals → $R_{2\ out}$, $R_{1\ in}$.

- The Fig. 9.10.2 shows the Register transfer operations using separate control signals.

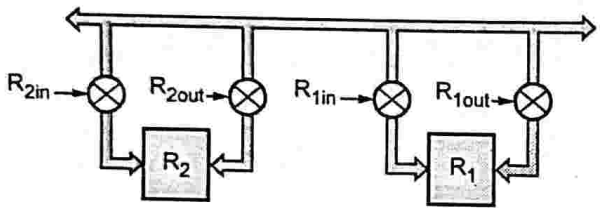


Fig. 9.10.2 : Register Transfer Micro-operation (with separate control signals)

- When the controller sends the control signals $R_{1\ in}$ and $R_{2\ out}$, two switches are operated
- This forms a path from register R_2 to R_1, via internal CPU bus, effectively transferring data from R_2 to R_1.

9.10.2 Arithmetic Microoperation

- Arithmetic microoperations are performed by the ALU inside the CPU. These microoperations perform some fundamental arithmetic operation on the numeric data.
- These basic operations are that of addition, subtraction, arithmetic comparison, increment and decrement operation etc.

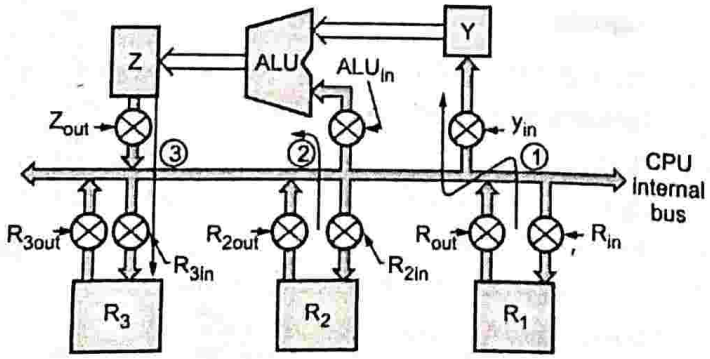


Fig. 9.10.3 : Arithmetic Micro-operations (Ex. Addition)

- For example, the addition microoperation can be specified as :

  Schematic for Arithmetic microoperation $R_3 \leftarrow R_1 + R_2$

- The sequence of operations to add contents of register R_1 and register R_2 and to store the result in register R_3 may be as follows :

Microoperation 1 :

Contents of R_1 are stored in register Y control signals : R_{1out}, Y_{in}.

Microoperation 2 :

Contents of R_2 and y are sent to ALU and control signal ADD is applied to ALU.

Control signals: $R_{2\ out}$, ALU_{in}, ALU control signal indicates ADD operation

Now the result gets stored in Z register.

Microoperation 3 :

Contents of register Z are transferred to the CPU register R_3.

Control signals: Z_{out}, $R_{3\ in}$.

RTL Statement for the Arithmetic addition operation $R_3 \leftarrow R_1 + R_2$		
Microoperation	**Operation**	**Control signals**
1	$Y \leftarrow R_1$	$R_{1\ out}$, Y_{in}
2	$Z \leftarrow Y + R_2$	$R_{2\ out}$, ALU_{in}, ALU control ADD
3	$R_3 \leftarrow Z$	Z_{out}, $R_{3\ in}$

9.10.3 Logical Microoperation

- Logical microoperations are performed by the ALU inside the CPU. These microoperations perform some fundamental logical operation on the numeric data.
- These basic operations are that of bitwise AND, bitwise OR, bitwise Exclusive-OR (XOR) and complement operation etc.

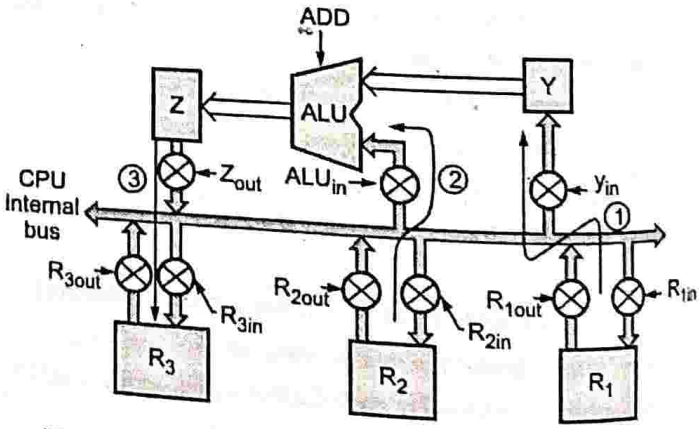


Fig. 9.10.4 : Logical Micro-operations (Ex. Bitwise AND)

Microoperation 1 : R_{1out}, Y_{in}

Microoperation 2 : $R_{2out.}$, ALU_{in}, ALU control ADD

Microoperation 3 : Z_{out}, R_{3in}

- For example, the bitwise AND microoperation can be specified as :

Schematic for Arithmetic microoperation $R_3 \leftarrow R_1 + R_2$

- The sequence of operations to add contents of register R_1 and register R_2 and to store the result in register R_3 may be as follows :

Microoperation 1 :

Contents of R_1 are stored in register Y control signals: R_{1out}, Y_{in}.

Microoperation 2 :

Contents of R_2 and y are sent to ALU and control signal ADD is applied to ALU.

Control signals : $R_{2\ out}$, ALU_{in}, ALU control signal indicates - AND operation

Now the result gets stored in Z register.

Microoperation 3 :

Contents of register Z are transferred to the CPU register R_3.

Control signals : Z_{out}, $R_{3\ in}$.

RTL Statement for the Arithmetic addition operation $R_3 \leftarrow R_1 + R_2$

Microoperation	Operation	Control signals
1	$Y \leftarrow R_1$	$R_{1\ out}$, Y_{in}
2	$Z \leftarrow Y + R_2$	$R_{2\ out}$, ALU_{in}, ALU control ADD
3	$R_3 \leftarrow Z$	Z_{out}, $R_{3\ in}$

☞ Memory Transfer Microoperation

- Memory transfer is achieved via the system bus of the computer system. Memories are accessed by issuing the address of the memory location to be accessed.
- The address is supplied by the CPU through the address bus (which is a part of the system bus) through the MAR (Memory Address Register).
- Based on the direction of the data-flow, there are two types of memory transfer operations :

Memory Read Transfers (Fetching a Word from Memory)

Memory Write Transfers (Storing a Word to Memory)

9.10.4 Memory Read Transfers (Fetching a Word from Memory)

The Memory Read Transfer Micro-operation can be achieved in following step sequence :

- Place the memory address in Memory Address Register (MAR). This is completed by generating the control signal MAR_{in}.
- Read the data of memory. This operation is achieved by putting the MAR data on address bus (Part of the System Bus) along with a memory read control signal on the control bus (Part of the system bus).
- The result of memory read operation is put on System Data bus (again part of system bus). Which in turn stores the data in Memory Data Register (MDR). This operation can be written as MDR ← M (MAR)

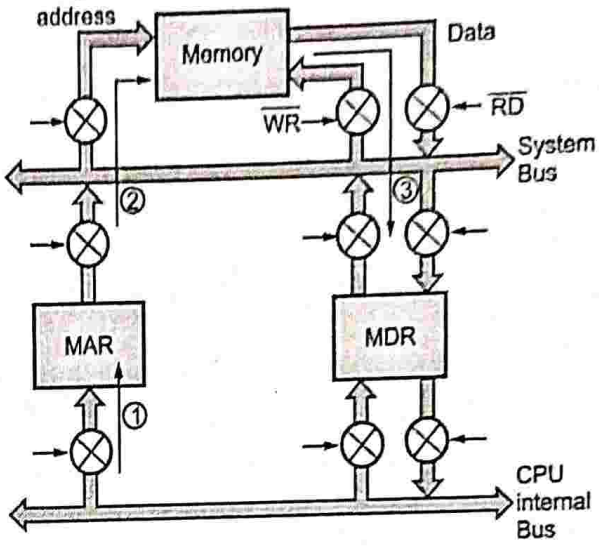


Fig. 9.10.5 : Memory Read Transfers (Fetching a word from the Memory)

☞ Sequence of control signals for read operation

1st Microoperation : $MAR_{in\ CPU}$(on Internal Bus)

2nd Microoperation : $MAR_{out\ system}$, RAM Address (on System Bus), RAM #RD Control Signal

3rd Microoperation : RAM_{out}, $MDR_{in,\ system}$(on System Bus)

Module 4

9.10.5 Memory Write Transfers (Storing a Word to Memory)

The Memory Write Transfer Microoperation can be achieved in following step sequence -

- Put the memory address in Memory Address Register (MAR). This can be done by generating the control signal MAR_{in}. From MAR, the address goes over the System Address Bus (Part of the System Bus).
- Write the data in MDR, that is to be written to memory. This operation is achieved by putting the data on local bus and then on to MDR i.e. MDR_{in}. From here, it goes over the System Data Bus (Part of the System Bus).

- Now the address bus along with a memory read control signal on the control bus (control bus is part of the system bus). The result of memory write operation is put on data bus (from MDR).

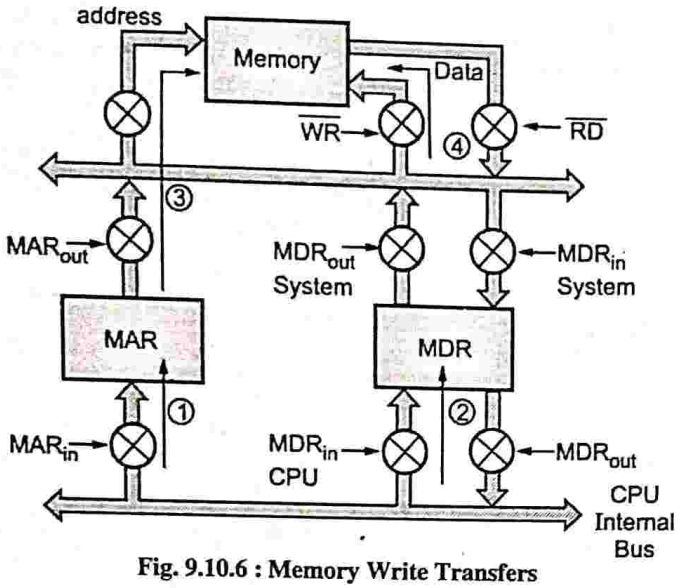


Fig. 9.10.6 : Memory Write Transfers (Storing a word to the Memory)

- Which in turn stores the data in Memory. This operation can be written as M (MAR) ← MDR.

☞ Sequence of control signals for write operation

1st Microoperation : $MAR_{in CPU}$ (on Internal Bus)

2nd Microoperation : $MDR_{in\ CPU}$ (on Internal Bus)

3rd Microoperation : $MAR_{out\ system}$, RAM Address, RAM #WR Control Signal

4th Microoperation : $MDR_{out,\ system}$, RAM_{out} (on System Bus)

9.10.6 Branch Instruction Micro-operations

Branch Instruction add decision making and therefore logic to the programming with systems.

Branch instructions can be broadly classified into Unconditional Branch Instructions and conditional Branch instructions.

9.10.6(A) Unconditional Branch Instructions

- In unconditional Branch Instructions, the program always takes the branch and therefore the next instruction to be executed is from the target address (address where the branch is taken).
- For such Instruction to work out, it is necessary that the PC be loaded with the target address. Since PC always points to the next instruction to be fetched, once the PC is loaded with Target address, the next instruction is fetched from that address and therefore, obviously the program branches to that (target) address, starting execution from there.
- Let us take example of Unconditional branch instruction :

  BR addr (Branch to addr)
- Where BR is the mnemonic for Unconditional branch instruction and addr is the Target address where the branch would be taken by the program, upon execution of this branch instruction. The RTL for the instruction is :

  PC ← addr
- The microoperations needed to be performed for this instruction are :

  Initially, $MAR_{in\ CPU}$ from the internal bus loads the address of program location where 'addr' is stored.

  Then, $MAR_{out\ system}$ on the System Address bus supplied to Memory along-with Memory read control signal #RD.

  The memory delivers 'addr' on the System Data bus, which is needed to be read in MDR. Therefore, operation needed is $MDR_{in\ system}$

  Now the 'addr' (which is the Target address for the branch) is contained in MDR is required to be loaded in PC. Therefore, operation needed is $MDR_{out\ CPU}$, PC_{in}

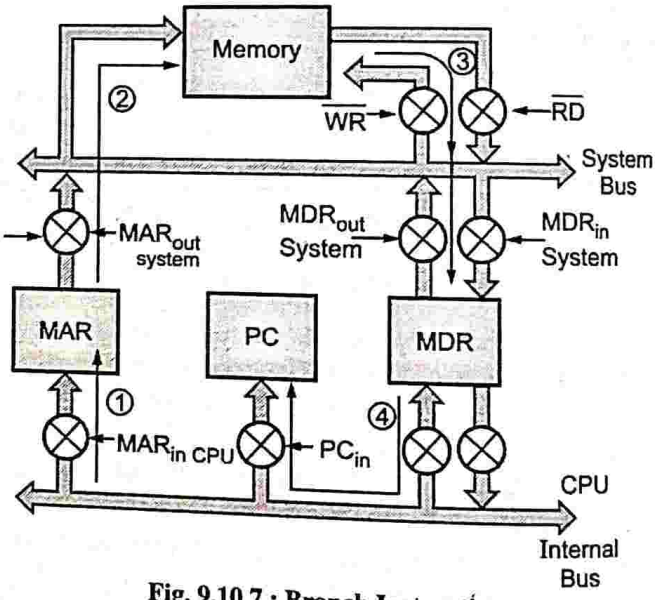


Fig. 9.10.7 : Branch Instructions

- The sequence of microoperations for unconditional branch instruction are :

1st Microoperation : $MAR_{in\ CPU}$

2nd Microoperation : $MAR_{out\ system}$, RAM Address (on System Bus), RAM #RD Control Signal

3rd Microoperation : RAM_{out}, $MDR_{in,\ system}$ (on System Bus)

4th Microoperation : $MDR_{out\ CPU}$, PC_{in}

- The Fig. 9.10.7 shows the schematic Branch Instruction micro-operations.

9.10.6(B) Conditional Branch Instructions

- In conditional Branch Instructions, the program takes the branch only if the condition is satisfied and therefore the next instruction to be executed is from the target address (address where the branch is taken).
- If the condition is not satisfied, program continues sequentially in the original program. The control unit is not needed to take any action, if the condition is not satisfied.
- For such Instruction, when the condition is satisfied, it is necessary that the PC be loaded with the target address. Since PC always points to the next instruction to be fetched, once the PC is loaded with Target address, the next instruction is fetched from that address and therefore, obviously the program branches to that (target) address, starting execution from there.
- Let us take example of Conditional branch instruction :

BEQ address (Branch if Equal to addr)

- Where BEQ is the mnemonic for conditional branch instruction and if the condition is satisfied (which is EQ – Equal) addr is the Target address where the branch would be taken by the program, upon execution of this branch instruction. The RTL for the instruction is :

IF EQ = True **THEN** **PC** ← addr

ELSE Continue

- To check whether EQ = True, control unit evaluates the status of ZF (Zero Flag). If ZF flag is Set, indicating EQ condition to be true. If the condition is satisfied.

The micro-operations needed to be performed for this instruction are :

- Initially, $MAR_{in\ CPU}$ from the internal bus loads the address of program location where 'addr' is stored.
- Then, $MAR_{out\ system}$ on the System Address bus supplied to Memory along-with Memory read control signal #RD.
- The memory delivers 'addr' on the System Data bus, which is needed to be read in MDR.
- Therefore, operation needed is $MDR_{In\ system}$
- Now the 'addr' (which is the Target address for the branch) is contained in MDR is required to be loaded in PC. Therefore, operation needed is $MDR_{out\ CPU}$, PC_{in}
- The sequence of micro-operations for conditional branch instruction, if the condition is satisfied, are –

1st Microoperation : $MAR_{in\ CPU}$

2nd Microoperation : $MAR_{out\ system}$, RAM Address (on System Bus), RAM # RD Control Signal

3rd Microoperation : RAM_{out}, $MDR_{in,\ system}$(on System Bus)

4th Microoperation : $MDR_{out\ CPU}$, PC_{In}

9.11 MICRO INSTRUCTION EXECUTION - EXECUTION OF A COMPLETE INSTRUCTION

UQ. 9.11.1 Explain micro instruction execution.

MU - Q. 5(c), May 15, 7 Marks.
Q. 2(b), Dec. 16, 08 Marks

GQ. 9.11.2 Write control sequence for the instruction ADD (R3), R1.

In the Instruction ADD (R3), R1 – We know that the standard generic form followed is –

- ADD destination, source.
- Therefore, first operand would be destination operand and second operand would be source operand. The instruction adds the data contents of memory location pointed by register R3 with the register contents of R1 and the result of the addition is stored back in the same memory location (pointed by register R3)
- The RTL statement for the instruction is :

M (R3) ← M(R3) + R1

- The sequence of Microoperations needed to successfully execute this instruction –

1st Microoperation :

$R3_{out}$, $MAR_{in\ CPU}$

In this microoperation, the contents of R3 register are loaded in the Memory Address Register (MAR).

2nd Microoperation :

$MAR_{out\ system}$, RAM Address (on System Bus), RAM #RD Control Signal

This microoperation sends the contents of MAR register on the System Address bus. Simultaneously, Read Control signal #RD is issued for the memory.

3rd Microoperation :

RAM_{out}, $MDR_{in,\ system}$(on System Bus)

The memory outputs data, it is received in the Memory Data Register (MDR) and therefore, it goes over the System Data bus.

4th Microoperation :

$MDR_{out\ CPU}$, ALU_{in}

Contents of MDR are loaded in the ALU as input. (Which are effectively, the contents of the memory location pointed by R3

Module 4

5th Microoperation :

$R1_{out}$, Y_{in}, ALU Function ADD (+)

- Contents of Register R1 are loaded in to Y register and therefore, loaded in to the second input of the ALU. Also, the ALU is commanded by an appropriate control signal, to carry out Addition operation.

6th Microoperation :

Z_{out}, $MDR_{in, CPU}$

The output of ALU operation gets loaded in Z register, from where it the microoperation loads the contents in to MDR register.

7th Microoperation :

MAR_{out}, RAM Address (on System Bus),
RAM #WR Control Signal

This microoperation sends the contents of MAR register on the System Address bus. Simultaneously, Write Control signal #WR is issued for the memory.

8th Microoperation :

$MDR_{out\ CPU}$, RAM_{in}

Finally, the result gets stored in the memory as MDR contents are written in to the memory. This is the memory location pointed by R3 register as that address from MAR is sent out for memory write operation.

GQ. 9.11.3 Write control sequence for the instruction SUB (R3), R1

In the Instruction SUB (R3), R1 – We know that the standard generic form followed is :

- SUB destination, source.
- Therefore, first operand would be destination operand and second operand would be source operand. The instruction adds the data contents of memory location pointed by register R3 with the register contents of R1 and the result of the addition is stored back in the same memory location (pointed by register R3)
- The RTL statement for the instruction is :

$$M(R3) \leftarrow M(R3) - R1$$

- The sequence of Microoperations needed to successfully execute this instruction –

1st Microoperation :

$R3_{out}$, $MAR_{in\ CPU}$

In this microoperation, the contents of R3 register are loaded in the Memory Address Register (MAR)

2nd Microoperation :

$MAR_{out\ system}$, RAM Address (on System Bus), RAM #RDControl Signal

This microoperation sends the contents of MAR register on the System Address bus. Simultaneously, Read Control signal #RD is issued for the memory.

3rd Microoperation :

RAM_{out}, $MDR_{in,\ system}$(on System Bus)

The memory outputs data, it is received in the Memory Data Register (MDR) and therefore, it goes over the System Data bus.

4th Microoperation :

$MDR_{out\ CPU}$, ALU_{in}

Contents of MDR are loaded in the ALU as input. (Which are effectively, the contents of the memory location pointed by R3.

5th Microoperation :

$R1_{out}$, Y_{in}, ALU Function SUB (-)

Contents of Register R1 are loaded in to Y register and therefore, loaded in to the second input of the ALU. Also, the ALU is commanded by an appropriate control signal, to carry out Subtraction operation.

6th Microoperation :

Z_{out}, $MDR_{in, CPU}$

The output of ALU operation gets loaded in Z register, from where it the microoperation loads the contents in to MDR register.

7th Microoperation :

MAR_{out}, RAM Address (on System Bus),
RAM #WR Control Signal

This microoperation sends the contents of MAR register on the System Address bus. Simultaneously, Write Control signal #WR is issued for the memory.

8th Microoperation :

$MDR_{out\ CPU}$, RAM_{in}

Finally, the result gets stored in the memory as MDR contents are written in to the memory. This is the memory location pointed by R3 register as that address from MAR is sent out for memory write operation.

GQ. 9.11.4 Write control sequence for the instruction ADD (Rsrc)+, Rdst

In the Instruction ADD (Rsrc)+, Rdst– We know that the standard generic form followed is :

- ADD source, destination.
- Therefore, first operand would be source operand and the + sign after the operand indicates that the operand would be post-auto-incremented. The second operand would be destination operand.
- The instruction adds the data contents of memory location pointed by register Rsrc; (automatically increment the contents of Rsrc) with the register contents of Rdst and the result of the addition is stored back in the register Rdst
- The RTL statement for the instruction is :

$$Rdst \leftarrow Rdst + M(Rsrc)$$

$Rsrc \leftarrow Rsrc + 1$

The sequence of Microoperations needed to successfully execute this instruction –

1st Microoperation :

$Rsrc_{out}$, $MAR_{in\ CPU}$

In this microoperation, the contents of Rsrc register are loaded in the Memory Address Register (MAR)

2nd Microoperation :

$MAR_{out\ system}$, RAM Address (on System Bus),
RAM #RD Control Signal
$Rsrc_{out}$, ALU_{in}, (On CPU internal bus)
ALU Function INC (+1)

This microoperation sends the contents of MAR register on the System Address bus. Simultaneously, Read Control signal #RD is issued for the memory.

At the same time, on the CPU internal bus, Rsrc contents are output and received in the ALU. ALU function control signal for Incrementing is given. Thus, the contents of Rsrc register are incremented by 1 and loaded in ALU output (Z Register)

3rd Microoperation :

RAM_{out}, $MDR_{in,\ system}$(on System Bus)
Z_{out},$Rsrc_{in}$(On CPU internal bys)

The memory outputs data, it is received in the Memory Data Register (MDR) and therefore, it goes over the System Data bus.

At the same time on the CPU internal bus Z contents are loaded back in Rsrc register. Effectively, Rsrc contents get post-incremented by 1.

4th Microoperation :

$MDR_{out\ CPU}$, ALU_{in}

Contents of MDR are loaded in the ALU as input. (Which are effectively, the contents of the memory location pointed by Rsrc

5thMicrooperation :

$Rdst_{out}$, Y_{in}, ALU Function ADD (+)

Contents of Register Rdst are loaded in to Y register and therefore, loaded in to the second input of the ALU. Also, the ALU is commanded by an appropriate control signal, to carry out Addition operation. The result goes to ALU output Z register.

6th Microoperation :

Z_{out}, $Rdst_{in}$

Finally, the result which is there in Z register now gets stored in the Rdst register.

UQ. 9.11.5 Write Microprogram for the instruction

i) ADD R1, M ii) MUL R1, R2

MU - Dec. 18, May 19

I) ADD R1, M

In the Instruction ADD R1, M – We know that the standard generic form followed is –

- ADD destination, source
- Therefore, first operand would be destination operand and second operand would be source operand. The instruction adds the data contents of memory location M with the register contents of R1 and the result of the addition is stored back in R1
- The RTL statement for the instruction is :

  $R1 \leftarrow R1 + M$
- The sequence of Microoperations needed to successfully execute this instruction –

1st Microoperation :

M, $MAR_{in\ CPU}$

In this microoperation, the memory address value M loaded in the Memory Address Register (MAR).

2nd Microoperation :

$MAR_{out\ system}$, RAM Address (on System Bus),
RAM #RD Control Signal

This microoperation sends the contents of MAR register on the System Address bus. Simultaneously, Read Control signal #RD is issued for the memory.

3rd Microoperation :

RAM_{out}, $MDR_{in,\ system}$(on System Bus)

The memory outputs data, it is received in the Memory Data Register (MDR) and therefore, it goes over the System Data bus.

4th Microoperation :

$MDR_{out\ CPU}$, ALU_{in}

Contents of MDR are loaded in the ALU as input. (Which are effectively, the contents of the memory location M

5thMicrooperation :

$R1_{out}$, Y_{in}, ALU Function ADD (+)

Contents of Register R1 are loaded in to Y register and therefore, loaded in to the second input of the ALU. Also, the ALU is commanded by an appropriate control signal, to carry out Addition operation.

6thMicrooperation :

Z_{out}, $R1_{in}$

The output of ALU operation gets loaded in Z register, from where it the microoperation loads the contents in the

Module 4

destination register R1, effectively completing the execution the instruction

ii) MUL R1, R2

In the Instruction MUL R1, R2 – We know that the standard generic form followed is :

- MUL destination, source.
- Therefore, first operand would be destination operand and second operand would be source operand. The instruction multiplies the data contents of R1 with the contents of R2
- The RTL statement for the instruction is :

  R1←R1 * R2
- The sequence of Microoperations needed to successfully execute this instruction –

1st Microoperation :

$R1_{out}$, ALU_{in}

Contents of R1 are loaded in the ALU as one of the inputs.

2nd Microoperation :

$R2_{out}$, Y_{in}, ALU Function MUL (*)

Contents of Register R2 are loaded in to Y register and therefore, loaded in to the second input of the ALU. Also, the ALU is commanded by an appropriate control signal, to carry out Multiplication operation.

3rd Microoperation :

Z_{out}, $R1_{in}$

The output of ALU operation gets loaded in Z register, from where it the micro-operation loads the contents in to R1 register. Hence the multiplication result is stored in R1, effectively completing the execution of the instruction.

UQ. 9.11.6 Write control sequence for the instruction MOV (R1), R2

MU - Q. 6(c), May 16, 6 Marks

- In the Instruction MOV (R1), R2 – We know that the standard generic form followed is –
- MOV destination, source.
- Therefore, first operand would be destination operand and second operand would be source operand. The instruction moves the data contents of the register R2 to the memory location pointed by register R1.
- The RTL statement for the instruction is :

  M(R1) ←R2
- The sequence of Microoperations needed to successfully execute this instruction –

1st Microoperation :

$R1_{out}$, $MAR_{in\ CPU}$

In this microoperation, the contents of R1 register are loaded in the Memory Address Register (MAR).

2nd Microoperation :

$R2_{out}$, $MDR_{in,\ CPU}$

The contents of register R2 are loadedas the contents in to MDR register.

3rd Microoperation :

MAR_{out}, RAM Address (on System Bus),

RAM #WR Control Signal

This microoperation sends the contents of MAR register on the System Address bus. Simultaneously, Write Control signal #WR is issued for the memory. Effectively sending contents of R1 register as the memory address.

4th Microoperation :

$MDR_{outsystem}$, RAM_{in}

Finally, the result gets stored in the memory as MDR contents are written in to the memory. This transfers the contents of register R2 as the contents of memory location pointed by R1. This effectively completes the execution of the instruction.

9.12 APPLICATIONS OF MICROPROGRAMMING

UQ. 9.12.1 What are applications of microprogramming ?

MU - Q. 1(c), May 14, Q. 1(a), May 15, 3 Marks

- Microprogrammed Control unit is simple to implement, it is capable to implement large variety of complex instructions without engaging in to hardware complexity and intricate synthesis that a Hardwired control unit needs.
- Further, Microprogrammed control unit offers flexibility of adding instructions and/or modifying existing instructions with ease.
- Addition an instruction in the instruction set of the Processor or CPU needs just addition of few Micro-instruction (i.e. adding a few locations in the Control memory). This makes Microprogrammed control unit, a very good choice for flexibly changing the definitions of the instructions.
- Therefore, different applications can be derived based on the concept of Microprogramming :

1. **Implementation of control unit :** Microprogrammed control units make use of Microprogramming. It is used most widely now for implementing the control unit of computer systems.
2. **Implementation of Operating system Primitives :** Microprograms are used to implement the primitives of Operating system. It provides the user with specific designed and privileged Instructions. It simplifies operation system implementation and also adds to the performance of the operating system.
3. **High-Level Language support :** High-Level language support is provided in processors at the machine level

with the help of Microprogramming. Different sub functions and data types are implemented using microprogramming. It makes compilation into an efficient machine language form possible.

4. **Micro-diagnostics** : Microprogramming is used for detection (identification), isolation of system faults and errors and further used for their monitoring and repairs. This technique is called Micro-diagnostics and it significantly enhances the fault tolerance and serviceability of the system.

5. **Re-configurable Processors** : Using RAM for implementing control memory (CM) allows the Micro-instruction to be loaded run-time into the Control Memory which can be done from an external memory source or on-system Flash-ROM. By changing the microcode of Micro-instruction in the external loading device or by burning it into the Flash-ROM, it is possible to tailor the machine to have different Instruction set for the different applications. Changing the Micro-Instruction code, therefore, allows the Processors to be re-configurable.

6. **In Emulation** : Emulation refers to the use of a microprogram on one machine to execute programs originally written for another machine. This is used widely as an aid for users in migrating from one computer to another.

9.13 NANO PROGRAMMING

UQ. 9.13.1 Write short note on Nano Programming.

MU - Q. 6(b), Dec. 14, Q. 6(e), Dec. 16, 10 Marks

UQ. 9.13.2 Explain concept of Nano programming.

MU - Q. 3(c), May 15,6 Marks

9.13.1 Concept of Nano Programming

1. In microprogrammed processors, an instruction fetched from memory is interpreted by a micro program stored in a single control memory CM; whereas in other microprogrammed processors, the micro instructions are not directly used by the decoder to generate control signals. This is achieved by the use of a second control memory called a Nano control memory (nCM).
2. So now there are two levels of control memories, a higher level control memory is known as micro control memory (μCM) and a lower level control memory is known as Nano control memory (nCM). This is shown in Fig. 9.13.1.
3. Thus, a microinstruction is in primary control-store memory, it then has the control signals generated for each microinstruction using a secondary control store memory The output word from the secondary memory is called Nano instruction.
4. The μCM stores micro instructions whereas nCM stores nano instructions. The decoder uses Nano instructions from nCM to generate control signals. Thus, Nano programming gives an alternative strategy to generate control signals. The process of generation of control signals using nano instructions is shown in Fig. 9.13.1.
5. Nano instruction addresses are generated by a nano program counter and nano instructions are placed in a register nIR. The next address of nIR is directly obtained. The next address is generated by either incrementing the nano program counter or loading it from external source (branch field or address from micro instruction opcode).

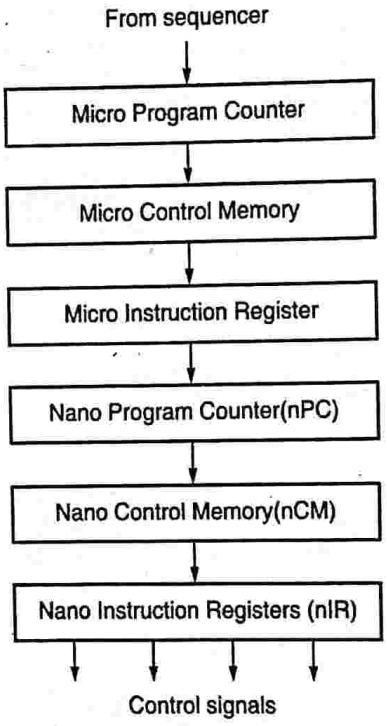


Fig. 9.13.1 : Nano Programming

9.13.2 Advantages of Nano programming

1. Reduces total size of required control memory

– In two level control design technique, the total control memory size S2can be calculated as

$$S2 = Hmx\ Wm + HnxWn$$

Where H – mn, represents the number of words in the high level memory

Wm, represents the size of word in the high level memory

Hn, represents the number of words in the low level memory

Module 4

Wn, represents the size of word in the low level memory

- Usually, the micro programs are vertically organized so *Hm*is large and *Wm*is small. In Nano programming, we have a highly parallel horizontal organization, which makes Wn large and Hnis small. This gives the compatible size for single level control unit as

  S1 = Hmx

  Wn which is larger than *S*2. The reduced size of control memory reduces the total chip area.

2. Greater design flexibility

Because of two level memories organization more design flexibility exists between instructions and hardware.

9.13.3 Disadvantage of Nano Programming

1. Increased memory access time

The main disadvantage of the two-level memory approaches is the loss of speed due to the extra memory access required for Nano control memory.

...Chapter Ends

❑❑❑

CHAPTER 10

Memory Organization

Module 5

University Prescribed Syllabus

5.1 Introduction and characteristics of memory, Types of RAM and ROM, Memory Hierarchy, 2-level Memory Characteristic,

5.2 Cache Memory : Concept, locality of reference, Design problems based on mapping techniques, Cache coherence and write policies. Interleaved and Associative Memory.

10.1 INTRODUCTION TO MEMORY

Memory is an essential component of the computer system. Its basic function is to store all the information in terms of data and programs required by the system. The organization, storage capacity, and speed of operation of the memory system heavily impacts the system's performance. Also, since the system memory cost is a significant factor in the overall cost of the system, the system designer must pay attention to memory designing.

The computer memory can be classified either depending upon the organization structure or based on the memory technology.

Classification based on **hierarchy** or **memory organization** can be shown as follows :

1. **Primary Memory :** Primary memory organizationally consists of the CPU register files, cache and the main memory of the computer system. It is the memory of the computer system where all the active programs and in-use data are stored. Cache are usually on the CPU chip while main memory is external to the CPU. Main memory can be directly accessed using the load store instructions of the processor. As a result, it has a fairly good speed.
2. **Secondary Memory :** Secondary memory is an external memory after the main memory used to store system programs and data that is not used continuously. It has huge capacity but is slower than primary memory. Hard disks, CD-ROMS, etc. are examples of secondary memory.

Based on the **semiconductor technologies** used, memories can be classified as follows :

1. **Random Access Memory (RAM) :** The memory that is used for read operation as well as write operation are called as RAM. The cache and part of the main memory require RAM technology implementation. RAM is further classified into Static RAM and Dynamic RAM.
2. **Read Only Memory (ROM) :** The memory used for storing the boot processes or is used for read operation only is called as ROM. ROMs are basically the non-volatile memories on the computing system. Part of the main memory requires ROM implementation. ROM is basically classified depending upon the erasing and re-writing methodologies used for ROM.

10.2 CLASSIFICATION OF PRIMARY AND SECONDARY MEMORIES

- Memories form the most essential element in a computing system. The main and internal memory of the computer system along with the magnetic disks form the important RAM, ROM and external storage memories required by the processor to perform any tasks. The memories can also be classified as primary and secondary memories depending upon their utility and accessibility for the processor.

10.2.1 Primary Memory

- The memory that is directly addressable or accessible on the memory map of the processor is called primary memory. It is accessed as the physical memory address space of the system.
- All types of RAMs and ROMs form the primary memory of a computing system. These memories are the ones that interact first with the processor. They form the internal memories of the computing system.

Features of Primary Memory

1. Being internal memory, the primary memory has a smaller access time i.e. it responds faster than the secondary memory.
2. Primary memory is accessed directly by the processor.
3. Every memory reference by the processor for data and code is performed on the primary memory.
4. They have smaller capacities compared to external i.e. secondary memory.
5. Few elements of the primary memory are volatile in nature i.e. their contents are lost once power is switched off.
6. Primary memory is expensive as compared to secondary

Primary memory consists of the sub parts as shown in Fig. 10.2.1.

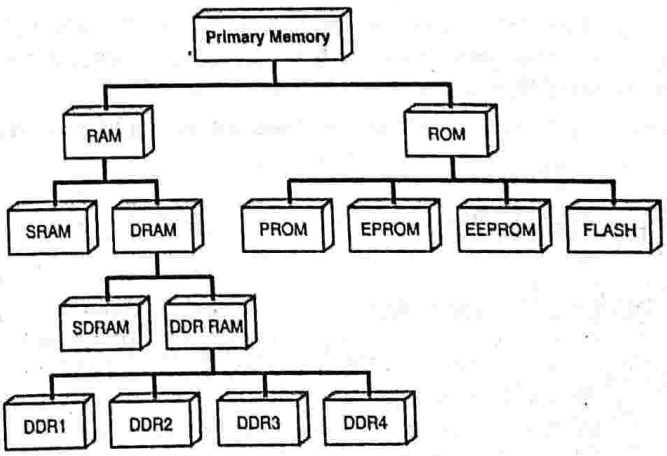


Fig. 10.2.1 : Classification of Primary Memory

Module 5

10.2.2 Secondary Memory

- The memory that is not directly addressable or accessible on the memory map of the processor is called secondary memory. It is accessed as the external memory device of the system.
- All external devices form the secondary memory. The memory could be incorporated as a fixed device or a removable device.

Features of secondary memory

1. Being external memory, the secondary memory requires more access time i.e. it responds slower than the primary memory.
2. Secondary memory cannot be accessed directly by the processor.
3. Secondary memory is first read into primary memory and then accessed.
4. They have bigger capacities compared to internal i.e. secondary memory.
5. Secondary memories are non-volatile in nature.
6. Secondary memory is cheaper as compared to primary.
7. Depending on whether secondary memory device is part of CPU or not, there are two types of secondary memory – fixed and removable.

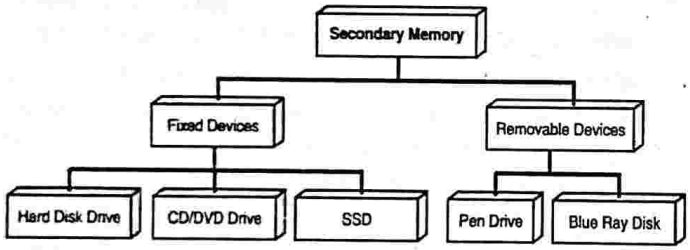


Fig. 10.2.2 : Classification of Secondary Memory

SSD

- A solid-state drive (SSD) is a solid-state external storage device or a type of secondary memory. It is used to store data persistently. SSDs can be implemented either using non-volatile NAND flash memory or volatile memory such as DRAM. It is also referred to as a solid-state device.
- SSDs are more resistant to physical shock and have quicker access time thereby reducing latency.

10.3 TYPES OF RAM AND ROM

UQ. 10.3.1 Write short note on Types of ROM.

MU - Q. 6(a), Dec. 16, 10 Marks

- The memories are always accessed randomly irrespective of the fact that whether they are RAM or ROM. Randomly accessed means that the words are directly accessed from the memory through wired-in addressing logic.
- In case of RAM, reading as well as writing is performed with the help of electrical signals. Also RAMs are volatile, meaning that the data is lost if power is cut to the memories.
- Read Only memory consists of a permanent pattern of data etched into the memory that cannot be modified but can be sensed or read. Therefore, ROMs are non- volatile i.e. once written, they store the data even without power. ROMs are Therefore, used for microprogramming the processor, or to write system programs, function tables or library sub routines etc.
- The classification of RAM and ROM is shown in Fig. 10.2.1
- Features of different memories are given in Table 10.3.1.

Table 10.3.1 : Memory Technologies Comparison

Memory type	Volatility	Storage Element/ Memory Cell	Read-Write	Erasure	Write Mechanism
Dynamic Random Access Memory (DRAM)	Volatile	Capacitor	Read Write	Electrical, Byte Level	Electrical
Static Random Access Memory (SRAM)	Volatile	Flip Flop Gate	Read Write	Electrical, Byte Level	Electrical
Read Only Memory (ROM)	Non Volatile	Wired during chip fabrication process	Read Only	Not Possible	Wired during chip fabrication process using Masks
Programmable ROM (PROM)	Non Volatile	Wired during chip fabrication process	Read Only	Not Possible	Electrical
Erasable ROM (EPROM)	Non Volatile	Transistor	Read Frequently	UV light, Chip Level	Electrical
Electrically EPROM (EEPROM)	Non Volatile	Transistor or Flip Flops	Read Frequently	Electrical, Byte Level	Electrical
Flash Memory	Non Volatile	Transistor	Read Frequently	Electrical, Block Level	Electrical

(I) Dynamic Random Access memory (DRAM)

- The memory cells in DRAM are made up of capacitors. Presence or absence of charge is interpreted as binary 1 or 0. Since capacitors tend to discharge, these RAMs require periodic refreshing to maintain the data.

- DRAM is further classified on the basis of data transfer rate as SDRAM and DDR RAM.

(a) Synchronous DRAM (SDRAM)

- It is a type of DRAM that runs in synchronization with the memory bus and delivers information in high speed bursts using clocked interface.

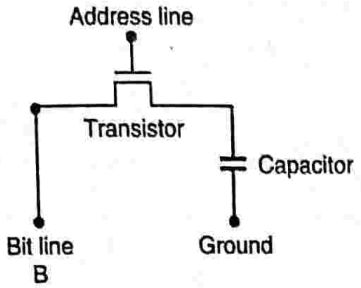


Fig. 10.3.1 : DRAM cell

- Due to the synchronization of signals with motherboard clock, the latency time of SDRAM is very small and one data unit is transferred synchronous to every clock cycle.

(b) Double Data Rate RAM (DDR RAM)

- This RAM has twice the data transfer rate as compared to SDRAM.
- This is achieved by transferring the data two times per transfer cycle, once at the rising age and once at the falling age of the cycle.
- Due to this even though the same clock and timing signals are used the effective transfer rate gets doubled.
- DDR RAMs are used as computer main memory with data transferred at a rate of 64 bits at a time. Over the period following DDR versions have evolved.
  - DDR1 : Maximum transfer rate of 1600 MB/s.
  - DDR2 : Maximum transfer rate of 4266 MB/s.
  - DDR3 : Maximum transfer rate of 6400 MB/s.
  - DDR4 : Maximum transfer rate of 10.6 GB/s.

(ii) Static Random Access Memory (SRAM)

- In SRAM, binary values are stored with the help of flip flops logic gates. As a result, they store the data as long as power is connected. Unlike DRAMs they do not require periodic refreshing to save the data.

(iii) Read Only Memory (ROM)

- The data is stored during the fabrication process by wiring it into the chip. This process is called as creating a mask for the data. The mask creation is a costly process, whether a single or multiple copy of ROM is to be made.
- Also, care must be taken that there is no error in the data mask. One error can lead to discarding of the ROMs fabricated.

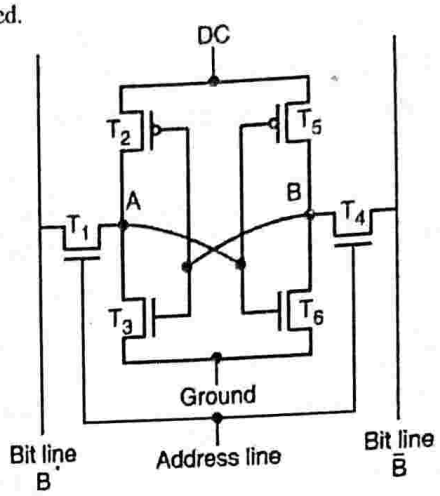


Fig. 10.3.2 : SRAM cell

(iv) Programmable Read Only Memory (PROM)

PROMs are non-volatile memories that are also written only once but are written electrically. The writing can be performed after the chip fabrication by any chip developers. Because of its flexibility and convenience in writing, PROMs are more desired for high volume chip developments.

(v) Erasable Programmable Read Only Memory (EPROM)

EPROMs are read as well as write memories but are read more frequently than written to. Hence are also called as read frequently or read mostly memories. They are optically erasable and hence can be rewritten. Before writing into the EPROM, the memory cells are required to be exposed to ultraviolet light equally. As a result, the erasure happens for the entire chip.

(vi) Electrically Erasable Programmable Read Only Memory (EEPROM)

EEPROMs are also read frequently type or memories. These memories can be erased electrically. Rather than erasing, these memories write new contents to dedicated address without affecting the other memory addresses. Therefore, byte level updates can take place in EEPROMs. EEPROM is Therefore, more costly than EPROM. Also, they are less dense (less number of memory cells in a given area) as compared to EPROMs.

Module 5

(vii) Flash Memory

Flash memories are the fastest in terms of rewriting the data onto them. They perform electrical erasing of data like the erasing process of EEPROM. But the entire flash memory is

erased like EPROM. It is Therefore, an intermediate between EPROM and EEPROM. Though individual bytes cannot be erased a section of memory cells can be erased at a time without affecting the contents of the entire memory.

☞ Comparison of DRAM and SRAM

UQ. 10.3.2 Differentiate between SRAM and DRAM. MU - Q. 5(b)(ii), Dec. 17, 5 Marks

Table 10.3.2 : Comparison of DRAM and SRAM

Observation	DRAM	SRAM
Memory cell construction	Simple and small	Complex and large
Cell Density (No of cells per unit given area)	More	Less
Cost	Less expensive	Costlier as compared
Refresh circuitry	Required	Not Required
Speed	Slight slower as compared	Faster as compared
Usage	Main memory	Cache

⏭ 10.4 MEMORY HIERARCHY

UQ. 10.4.1 Describe the memory hierarchy in the computer system. MU - Q. 1(c), May 18, Q. 1(D), Dec. 18, 5 Marks

UQ. 10.4.2 Explain in details Memory Hierarchy with examples. MU - Q. 4(a), May 14, 6 Marks

- The performance of the general purpose computers used mostly in day-to-day life can be improved by providing additional storage beyond the main memory. Single memory is not enough to store all the programs and continuously generated data. Also, not all the data and programs are required simultaneously.
- Single memory implementation technology cannot support the system requirements of high storage and faster execution. Hence, a combination of memory technologies can be used to store active programs and data together in faster smaller memories, whereas, save the rest of the data in low cost slightly slower huge memory devices. This technique is termed as **memory hierarchy**.
- The faster smaller memories consist of registers, cache and main memories. Magnetic disks, CDs, DVDs, tapes, etc. form the auxiliary or low cost slower memories. A typical memory hierarchy pyramid is shown in Fig. 10.4.1.

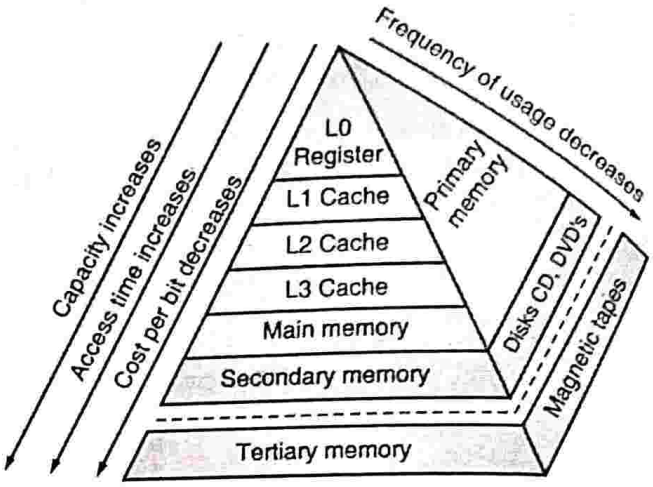


Fig. 10.4.1 : Memory Hierarchy or Organization

- As one goes down the hierarchy, following observations occur :
  1. Increase in the capacity.
  2. Decrease in the cost per bit.
  3. Increase in the access time.
  4. Decrease in the frequency of usage of the memory by the processor.
- Thus, smaller, costlier, faster memories are supplemented by bigger, cheaper, slower memories. This is termed as memory hierarchy. The key to success for the hierarchy is that the frequency of access also reduces as one goes down the pyramid.

⏭ 10.5 MEMORY CHARACTERISTICS

UQ. 10.5.1 List different memory organization characteristics. MU - Q. 3(c), May 14, 8 Marks. Q. 1(c), Dec. 15, Q. 1(a), Dec. 17, 5 Marks

UQ. 10.5.2 Describe the characteristics of Memory. MU - Q. 4(b), Dec. 16, 10 Marks

UQ. 10.5.3 Write notes on the characterizes of memory. MU - Q. 6(b), May 16, 8 Marks

The Memories or Memory devices are measured for their features and their performance based on different characteristics.

These characteristics are :

a. Location in Memory Hierarchy
b. Capacity and Addressability
c. Unit of Data Transfer
d. Access Method
e. Physical type
f. Volatility
g. Writable and Write Cycle Sustenance
h. Performance Parameters
i. Cost

(a) Location in the Memory Hierarchy

It deals with the location of the specified memory in the memory hierarchy of the computer system. There are different possible locations for the memories :

- **CPU :** CPU consists of many CPU registers which is the Processor's own private memory storage space for storing most required or hand elements of data references, instruction references and addresses. Many times, it is referred to as L0 cache of the system.
- **Cache Memory :** The high-speed memory that is introduced in the system organization with processor proximity and having the intention of reducing the average wait-states in the CPU machine cycles, is called as Cache Memory. Generally high-speed memories qualify to become cache memories. Currently up to 3 levels – Level-1 (L1), Level-2 (L2) and Level-3 (L3) – of cache memories are used in the computer systems.
- **Main Memory :** It is the bulk memory of the system. All the live data and program (instructions) elements are stored in the main memory. Main memory and Cache memory are directly addressable by the processor or CPU and therefore form the Primary Memory of the system.
- **External or secondary :** It comprises of secondary storage devices like hard disks, magnetic tapes. The CPU doesn't access these devices directly. It uses device controllers to access secondary storage devices.

(b) Capacity and Addressability

The capacity of the memory or memory device is represented in terms of: i) Word size and ii) Number of words or Addressability.

- **Word size :** Words are expressed in terms of no. of bits or bytes (8 bits). A word can however mean any number of bytes. Commonly used word sizes are 1 byte (8 bits), 2 bytes (16 bits) and 4 bytes (32 bits) or 8 bytes (64 bits). It is also called as the data width of the memory.
- **Addressability or Number of words :** This specifies the number of words available in the particular memory or how many words that are addressable in the memory.

Total capacity of the memory is Word Size X No. of Words. For example, if a memory device is specified as 8K x 16. This means that the device has a word size of 2 bytes (16 bits) and a total of 8192 (8K) words in memory.

(c) Unit of Transfer

It is the maximum number of bits that can be accessed (read or written into) by the memory at a time. In case of main memory or memories used as primary memory, it is generally same as its word size. In the case of external memory, unit of transfer may not be same as that of its word size. It is usually larger than the word size and it is referred to as blocks.

(d) Access Methods

It is the fundamental characteristic of the memory or memory device. It is the method or order in which memory elements can be accessed. There are three types of access methods:

1. **Random Access :** If storage locations in a particular memory device can be accessed in any order i.e. The access time of the memory is independent of the memory address or location being accessed. Any location in such memory requires exactly same to access. This type of memory is said to have a **Random Access** mechanism. Usually, most of the semiconductor memories and primary memories are Random Access in nature.
2. **Sequential or Serial Access :** If memory locations in the memory device, can be accessed only in a specific pre-determined sequence or order, then the access method is called as **Sequential or serial access.** Many secondary storage devices like magnetic tapes exhibit Sequential access to its contents.
3. **Semi random or Pseudo Random Access :** Memory devices such as Magnetic Hard disks and Optical disks use this type of access method. Here each disc surface has an independent read/write head thus each surface can be accessed randomly but access within the surface is sequential in nature as tracks and sector arranged can be accessed in serial order. Therefore the access is not fully or truly random in nature. This type of access is called as **Semi random or Pseudo Random Access.**

(e) Physical type

This type is essentially based on the technology by which the memory is constructed. The memories implemented with different IC technologies are Semiconductor memories. Semiconductor memories are further classified based on their Write characteristics as RAM – Random Access Memory (usually of Read/Write nature) and ROM – Read Only Memory.

Module 5

There are other physical types of memories such as Magnetic Access/ Recording type (ex. Hard disks) or Optical Access/Recording type (ex. CD, DVD, Blu-Ray Disks)

(f) Volatility

It is the characteristics that identifies the capacity of the memory to retain or hold its data contents when the power to memory is turned off. Based on this parameters we find the memories to be :

- **Volatile :** The memory that loses its data contents of its locations (or not able to retain) is called as Volatile Memory.
- **Non-Volatile :** The memory that holds on to the data contents of its locations (or is able to retain) is called as Non-Volatile Memory.

In case of Non-volatile memories, data can be retained of a specific amount of time. This property of Non-volatile memories is called as **Data Retainance**. For example EEPROM is said to have Data Retainance of 10 years.

(g) Writable and Write Cycle Sustenance

Not all memory technologies allow writing Data to the memory or writing data multiple times to the memory. Therefore, Non-volatile memories, usually ROMs are classified based on whether they are writable or not. They are referred to as Read Mostly memory if they are Writable. Such memories show different number of Write cycles that they can sustain. This property is called as Write Cycle Sustenance. For Example, OTPROMs can be written only once where as Flash ROM can be written 100,000 times.

(h) Performance Parameters

These characteristics are the parameters on which the performance of the memory is determined and measured. These parameters are discussed in detail, in the next section 10.6.

(i) Cost

It's a commercial characteristic of the memory and determines its implementation feasibility. This aspect is discussed in detail, in the next section 10.6.

10.6 MEMORY - COST AND PERFORMANCE MEASUREMENT

The memory selection is carried out on the basis of memory characteristics. The dominant determining factors in determining the feasibility of the memory implementations is governed by its Cost (commercial factor) and its Performance (technological factor).

A. Cost

- Cost is the commercial factor and governed by market forces. It comes into picture while determining the memory implementation due to economic feasibility. The Cost is measured as the cost incurred for some suitable unit of memory size i.e. Cost/bit or Cost/MB (Mega-Byte) or Cost/GB (Giga-Byte).
- Costlier is the memory per unit size it is more difficult to implement it economically. Right memory is needed to be chosen for the product in which memory is installed, to be affordable and cost-effective. Currently, DRAM varieties are found to be cost-wise most suitable for implementing main memories in computer systems while Flash memories are used for external drives along-with the Hard disks as secondary storage devices as these memories have lowest cost/bit in their respective domains.

B. Performance

The performance of the Memory determines its usage in the system. The performance measurement and comparison of different memories for their performance, is carried out on the basis of following Performance parameters.

C. Performance Parameters

The performance of the memory system is determined using three parameters :

1. **Memory Access Time :** For the random access or semiconductor type memories, it is the time taken by memory to complete the specified read/write operation from the instant, an address is supplied to the memory. Therefore, lower is the memory access time, faster is the memory response and is a desirable feature. It is measured in terms of appropriate unit of time such as Micro-Seconds (μS) or Nano-Seconds (nS).
2. **Memory cycle time :** It is defined only for Random Access Memories. It is defined as the time required for the specified read/write cycle of the processor to be completed with the memory under consideration. It is usually the sum of the access time of the memory and the additional time required (if any) before the second access can commence. It is measured in terms of appropriate unit of time such as Micro-Seconds (μs) or Nano-Seconds (ns).
3. **Peak Data Transfer rate :** It is defined as the maximum rate at which data can be transferred into or out of a memory unit. It is dependent on both the Memory Cycle time and the Word size of the memory. Peak data transfer rate can be computer as :

$$\text{Peak Data Transfer Rate} = \text{Memory Word size} \times (1/\text{Memory Cycle time})$$

It is also called as memory bus bandwidth. It is measured in terms of Kilo-Byte/Seconds (KB/s) or Mega-Bytes/Second (MB/s) or Giga-Bytes/Second (GB/s).

☞ Performance Measurement example

Ex. 10.6.1

A particular type of DRAM memory device is organized as 256K × 32 Bit. It has access time of 100 ns and refresh/recharge time of 25 ns. Determine the performance of the this memory device on the basis of Performance metrics or parameters.

☑ Soln. :

The Performance metrics or parameters for this memory are computed as follows :

Memory Access Time : 100 ns (as per the given data)

Memory Cycle Time = Memory Access time + any additional time needed between 2 accesses

= 100 ns + 25 ns = 125 ns

(This means the Memory Cycle can fastest be completed in 125 nS).

Peak Data Transfer Rate = Memory Word size × (1/Memory Cycle time)

= 4 Bytes (32 Bits) × (1/Memory Cycle Time)

= 4 Bytes × (1/125ns)

= 4 Bytes × 8 M

= 32 MB/s

⏭ 10.7 VIRTUAL MEMORY

UQ. 10.7.1 What is virtual memory?

MU - Q. 1(d), May 14, 4 Marks, Q. 1(d), Dec. 15, Q. 1(d), Dec. 17, 5 Marks

UQ. 10.7.2 Explain Virtual Memory.

MU - Q. 1(a), May 17, 5 Marks

UQ. 10.7.3 Explain virtual memory with reference to memory hierarchy, segments and pages. MU - Q. 3(b), Dec, 14, 10 Marks

10.7.1 Virtual Memory : Concept

- In the multi-tasking multiuser system environment, it is desirable that many programs run at the same time. They reside in the system main memory and it is necessary that the physical memory allocation where these programs are stored is abstracted from the programmer's writing those programs.
- Further, the total required memory space for the program code and data needed for all the application programs currently active on the system are far greater than the available physical memory space.
- Due to these factors, it becomes necessary that the user programs are not given access to physical memory addresses. These requirements give rise to a concept where-programmers write programs with all address references having logical addresses and Therefore, user programs do not have any access or control over the physical address space where their program would be loaded. This allows many programs to be created without worrying about whether the program would be fitted in the physical memory space or what is the exact location in physical memory space that program would access.
- While loading the application programs mentioned above, the OS kernel determines using its memory management algorithm, the physical memory space that would be allocated to the specified program. This is termed as **Memory Allocation** and is an important part of memory management.
- The OS kernel then loads the program in the allocated physical memory space. The point at which the program would be loaded in physical address space is called as the **Base Address**.
- When the program and data is accessed, for every access the logical address is translated into the physical address with respect to the base address, where the program has been loaded.
- Now, many programs having their memory requirements much bigger than the available memory space can be loaded in memory space. Also, one program is not allowed to access memory space allocated to another program.
- This technique where the user programs and applications visualize that their program code and data is loaded into a very bits memory space virtually and completed abstracted from the physical memory space; is called as the **Virtual Memory**.

☞ Virtual address

- The logical address which user programs and data use for addressing the memory is called as Virtual Address. The virtual address would typically have two components :
  1. One is an **index or ID** that refers to the base address or the starting address where the program or data has been loaded in the physical memory space and
  2. The other is the **offset** which is the relative address within the allocated memory space with respect to the base address.

☞ Virtual Memory management

The allocation and management techniques for managing the physical memory space by converting logical or virtual addresses

Module 5

into the physical addresses (for all memory accesses of the application programs and for the data) is called as **Virtual Memory Management**.

10.7.2 Virtual to Physical Address Translation : Address Translation Mechanism

UQ. 10.7.4 Explain in detail virtual memory segmentation and paging.
MU - Q. 4(b), May 15, 7 Marks

- Address translations is the most critical component in virtual memory management. It is necessary for the OS kernel to translate the virtual address into the physical address for each and every access of program code and data.
- Different techniques of virtual memory management use different mechanisms of address translation. As mentioned above, the **virtual address** consists of two components namely: the index and the offset.
- Index points to the base address of the allocated memory space. Usually it is done through a table maintained by the OS keeping track of all base addresses of the loaded memory structure (A segment or a page as described in later sections). Once the base address is obtained, the offset component is added to it to generate the **physical address**. This process is called as **address translation**. Fig. 10.7.1 shows the Virtual address to physical address translation.

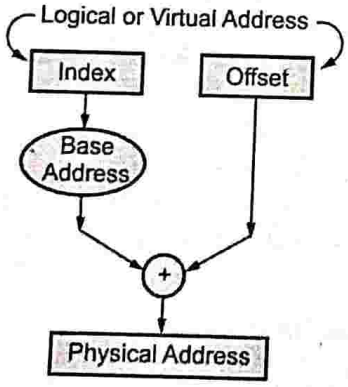


Fig. 10.7.1 : Virtual to Physical Address Translation

- There are two virtual memory management techniques used most widely. The allocation can be carried out by logical blocks such as the program code or data of a specific application. Such allocation is called as a **segment**, and the memory management technique is called as **Virtual Memory Segmentation**.
- The allocation can be carried out by physical size where, the virtual as well as the physical memory space is divided into fixed size memory blocks that can be allocated. They are called as pages and the Virtual memory management technique is called as **paging**.

10.7.3 Segmentation

UQ. 10.7.5 Describe memory segmentation in detail. Explain how address translation is performed in virtual memory.
MU - Q. 4(b), Dec. 18, 10 Marks

UQ. 10.7.6 What is Segmentation?
MU - Q. 1(d), May 17, 4 Marks

- Segmentation is a virtual memory management technique in which the memory allocation blocks are determined logically. The logical allocation is performed on the basis on control structure (the program code) or the data structure (data, stack or array).
- Such allocated memory blocks are called as segments. Segments are of variable size depending on the program code or data that they are needed to contain.

Segment address translation

- The user program specifies the virtual address in the form of two components namely : the **segment index** or ID and an **offset** within the segment. The segment index or ID refers to a segment table.
- The segment table contains the segment bases addresses of the loaded segments in the physical memory space. Therefore, the index provided in the virtual address, selects one of the segments in the segment table and the segment base address is obtained.
- Then to this segment base address, the offset provided in the virtual address is added to generate the **physical address**. This address points to the actual program code or data within the segment that is being accessed. Fig. 10.7.2 shows the mechanism of segment address translation.

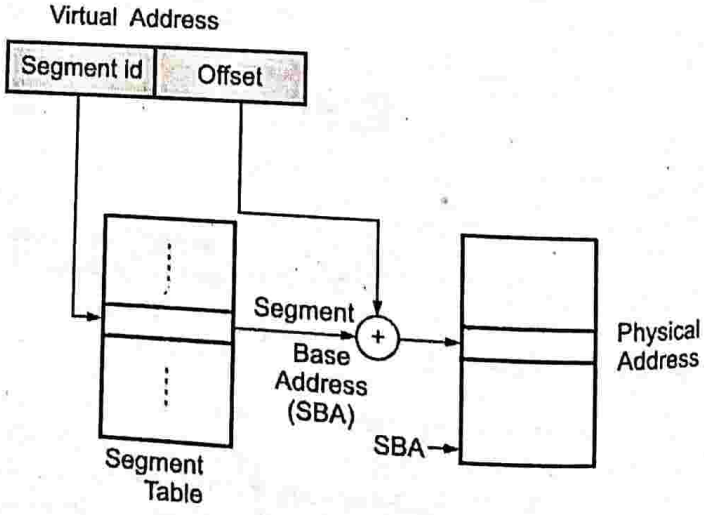


Fig. 10.7.2 : Segment Address Translation

Advantages of Segmentation

Segmentation makes the memory model to be more structured as the program code, data and stack are stored in different physical address space.

Segmentation allows the user program to be abstracted from the physical addresses of the memory, this makes the programs to be relocatable and to be loaded at any desired point in the physical memory space.

As segmentation is a virtual management technique, it allows bigger virtual space to be managed in a relatively smaller available physical memory space.

The points above make multitasking and multiuser environment with appropriate memory management feasible.

Segmentation provides OS with full control of memory allocation and management. It also allows OS to control access of each application program on the memory thereby not allowing any application program to access the memory space allocated to another program, either advertently or inadvertently.

10.7.4 Paging, Page Placement and Location

The main memory available in the computer system is usually partitioned to contain the OS programs and the user programs. Multiple processes are run simultaneously to improve the system performance. In such cases, the memory allotted for user programs needs to be divided and sub divided to ensure that simultaneous parallel tasks can be carried out. This task of division and subdivision is performed by the OS and is called as **Memory Management**.

- Creating these divisions with a fixed size can lead to waste of memory for smaller tasks whereas creating unequal divisions will make it difficult to access locations since their starting addresses won't be deterministic.
- In order to avoid these issues, not only the memory but the programs or processes are also divided i.e. **Paging** is performed. Paging involves the division of process/program or data into much smaller pieces of fixed size called as **Pages**. The memory is also divided into equal sized chunks called as **Page Frames** or **Frames**. The pages are then assigned to page frames. This ensures minimal wastage of memory, since the only frame that will have memory wasted will be the frame carrying last page of the process and that last page does not require entire frame memory locations.
- Fig. 10.7.3 shows an example of paging and allotting pages to frames. Consider a process X, that is divided into 4 pages, page 0, 1, 2 and 3. The memory is also divided into fixed equal subdivisions called as frames. Pages and frames have same size. Let the initial memory have 5 free frames namely frame 1, 2, 4, 5, 7 whereas frame 0, 3, 6, 8 be already occupied. When process X arrives, the OS shall assign any four free frames out of the five to process X. This frame assignment to pages may or may not be contiguous. As a result, the page to frame mapping needs to be stored. This storage is done in a table called as **Page Table**.

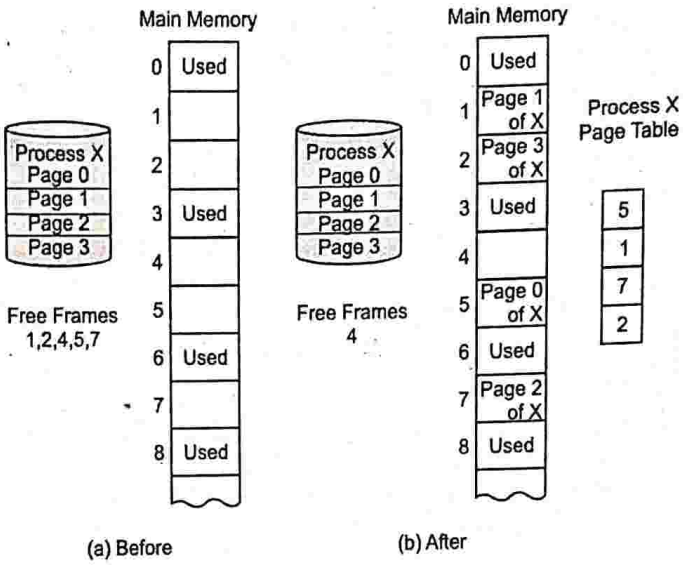


Fig. 10.7.3 : Paging and Page Frame Allocation

Page Address Translation

UQ. 10.7.7 Explain the page address translation in case of virtual memory.

MU - Q. 3(b), May 18, 5 Marks

Paging is a virtual memory management technique and it involves translation from virtual address to physical address in the paging systems. This translation is called as **page address translation**. The paging can be carried out by two different mechanisms namely : Single Level Paging and Two-Level Paging.

1. Single Level Paging

- In single level paging system, the Page address translation is carried out as shown in the Fig. 10.7.4.

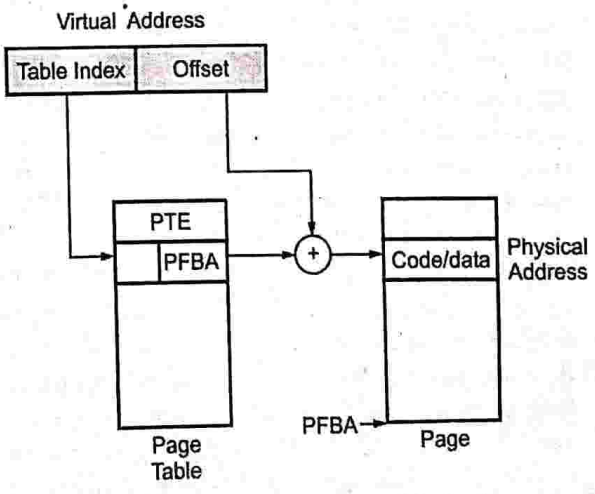


Fig. 10.7.4 : Single Level Paging Mechanism

Module 5

- The virtual address is having two components, an index to the Page Table and an offset within the page frame. The table index is used to select an entry one out of the entries stored in the page table. Page table stores number of entries, each pointing to the page frame in the physical memory space.
- It is called as Page Table Entry (PTE). It contains the Page Frame Base Address (PFBA).
- The page entry selected by the index and it provides the Page Frame Base Address. Then the offset is added to the Page Frame Base Address to generate the Physical address where the actual program code or data is accessed.

2. Two-Level Paging

- In two-level paging system, the Page address translation is carried out as shown in the Fig. 10.7.5.
- The virtual address is having three components, an index to the Page Directory, an index to the Page Table and an offset within the page frame. The Directory index is used to select an entry one out of the entries stored in the page directory.

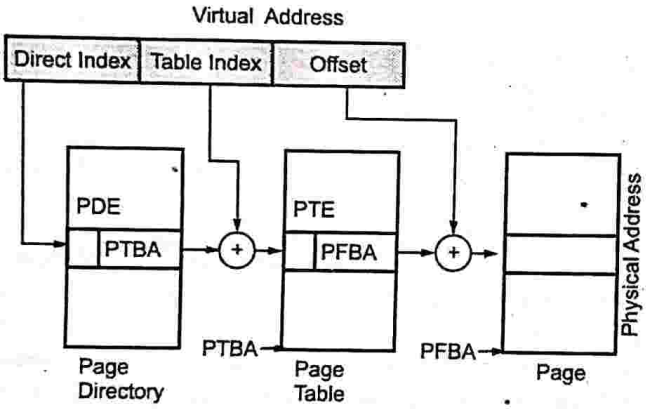


Fig. 10.7.5 : Two Level Paging Mechanism

- Page directory stores number of entries, each pointing to the page table in the physical memory space. It is called as **Page Directory Entry (PDE)**. It contains the Page Table Base Address (PTBA). The page table entry selected by the index and it provides the Page Table Base Address, which is used for accessing the selected page table.
- The table index is used to select an entry one out of the entries stored in the page table. Page table stores number of entries, each pointing to the page frame in the physical memory space. It is called as Page Table Entry (PTE). It contains the Page Frame Base Address (PFBA).
- The page entry selected by the index and it provides the Page Frame Base Address. Then the offset is added to the Page Frame Base Address to generate the Physical address where the actual program code or data is accessed. Page Directory Entry points to a Page Table and Page Table Entry points to a Page, the Page contains at the offset, the required piece of program data or code at the computed Physical address.

10.7.5 Page Faults, TLB in Address Translation

- The virtual memory space is divided in to fixed size Pages and the physical memory space is divided in to same fixed sized Page Frames.
- However, the number of pages in virtual memory space are much bigger in number as compared to the number of page frames in the physical memory space and therefore, only a few pages can be accommodated in the available page frames in the physical memory space.
- Therefore, two issues are confronted in loading and unloading the pages in page frames. They are
  1. Which pages would be loaded in the page frames in the physical memory space, i.e. Page Allocation.
  2. Which pages would be removed or unloaded when a new page is needed to be loaded and there is no page frame empty in the physical memory space. i.e. Page Replacement.

Page Faults

- When the Processor or CPU demands an element of program code or data and it is currently NOT available in the pages already loaded in the page frames in the physical memory space, processor or CPU generates an exception during the process of page address translation mechanism. This exception is called as **Page Fault.**
- The Page Fault indicates that the data or code element needed by Processor or CPU is currently not available in the physical memory space. In other words, the page containing that specific code or data element is not there in any of the page frames in the physical memory space. The page is lying out in the virtual memory space (which is on the secondary memory such as Hard disk).
- In Single level paging, the page fault is generated while accessing the Page Table Entries (PTE) indicating that the concerned page is not there in the physical memory space.
- In Two-level paging, the page fault is generated while accessing either the Page Directory Entry (PDE) or while accessing Page Table Entry (PTE). It indicates that either the concerned Page table or the concerned Page is not available in the Physical memory space.
- Page Fault is an exception and is handled in a way similar to an Interrupt. The page fault handling is discussed in the next section.
- This mechanism of loading the pages in the page frames in the physical memory space when the page is demanded by Processor or CPU program code/data reference and it is not found in the physical memory space; is called as **Demand Paging**.

Therefore, as a result of Page Fault mechanism, processor is able to handle the situation where it loads the desired pages into page frames in the physical memory space and Therefore, allows the programs having larger memory requirements to share the physical memory space efficiently and mutually exclusively.

10.8 CACHE MEMORY – CONCEPTS AND NEED

UQ. 10.8.1 What is the necessity of cache memory?

MU - Q. 3(a), May 18, 3 Marks

- The processor or CPU generates address for access to the memory, if the specified address is not available already in one of the cache memory lines or blocks then while it is being accessed from the main memory. It is also copied into the cache memory in one of its cache lines or blocks.
- Therefore, some memory address contents are available in the cache memory lines or blocks out of the total large amount of contents available in the main memory; main memory being much larger in size as compared to cache memory. This is the basic operating principle of Cache memory system. The cache principles can be detailed as follows.

10.9 CACHE MEMORY PRINCIPLES / PRINCIPLES OF LOCALITY

- In order to increase computational speed, the memory access time of Main Memory (MM) should be very low. But high-speed memory increases the cost of the system.
- Hence, a more economical solution is to connect a small high-speed memory in between CPU and Main Memory and load portion of Main Memory into this small high-speed memory. This small-high speed memory is called as Cache memory. The Fig. 10.9.1 illustrates the same.

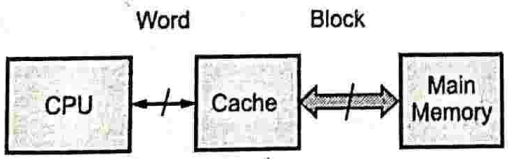


Fig. 10.9.1 : Cache Memory

- A block of Main Memory is transferred into cache memory. This memory block transfer is quite slow as memory access time of main memory is quite high. Whereas, a word value is transferred from cache to CPU and this transfer is at a very higher rate. The cache contains copy of portion of main memory. When processor tries to read a word from memory, a check is made to determine whether the word is present in cache. If it is so, the word is transferred from cache to CPU. Else, new block code is taken from main memory. When a new block of data is fetched into the cache for a single memory reference, the probability of having further memory reference from same memory block is more.
- The block diagram of Read Operation is shown in Fig. 10.9.2.

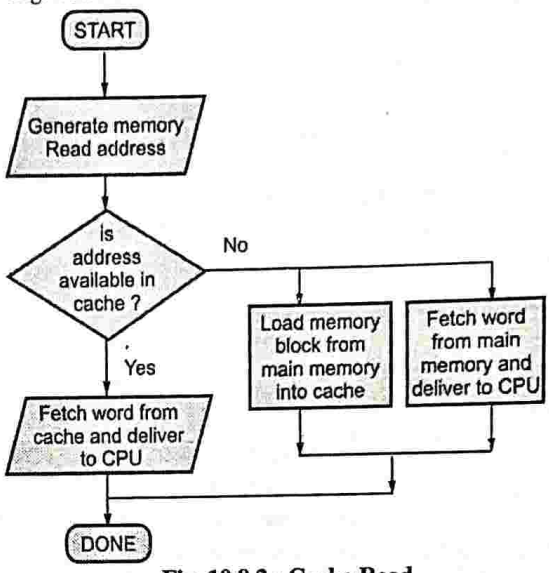
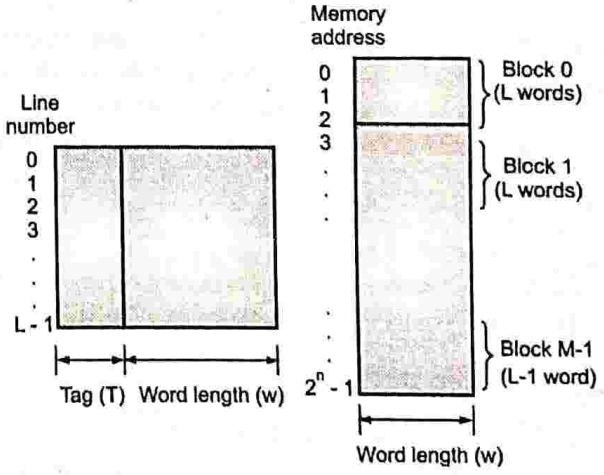


Fig. 10.9.2 : Cache Read

- CPU initiates memory read operation and generates a memory address from where data has to be accessed. This address is searched in Cache Memory. If this address is present in Cache, then the word data contained at that memory location is returned to the CPU. If the address generated by CPU is not contained in the Cache, then two simultaneous actions are performed. The Main memory is accessed and a block of memory containing the memory location is address is loaded into Cache Memory, and the word data is delivered to the CPU from Main Memory.
- Fig. 10.9.3 shows the Cache and Main Memory Structure.



(a) Cache memory cache length = word length (w) + Tag (T) **(b) Main memory**

Fig. 10.9.3

Module 5

- For Main Memory, assuming 'n' number of addressing lines, there will be 2^n different memory locations available. Each memory location can store a word of length 'W'. For Cache Memory, a block of 'L' word size is transferred. The data word from main memory is placed in the Cache memory along with a Tag. This tag contains a reference to memory location of main memory. Hence, the length of each line of cache will be Length of Tag (T) + Word Length (W).

UQ. 10.9.1 Write short note on Principle of locality of reference.

MU - Q. 6(a), May 18, 10 Marks

Principles of locality : Cache memories are successful due to following two principles of locality of reference.

(i) **Principle of Spatial Locality :** This principle state that if the processor or CPU references a particular memory address then there is a very high probability that the processor or CPU would reference to a nearby address in space, in the near future.

(ii) **Principle of Temporal Locality :** This principle state that if the processor or CPU is referencing a particular address in the memory then there is a very high probability that the processor or CPU would reference the same address location repeatedly in time (in near future).

10.10 CACHE MEMORY TERMINOLOGY / PERFORMANCE METRICS AND IMPROVEMENTS

- With respect to cache memories, following terminology is used :
  - **Cache hit :** The processor or CPU demands an address and the address is found in one of the cache lines or blocks. This event is called as the cache hit.
  - Cache miss : The processor or CPU demands an address and the address is not found in one of the cache lines or blocks. This event is called as the cache miss.
  - Hit Ratio (h) : It is the ratio of accesses made by processor or CPU to the cache memory to the total number of accesses made to the main memory and cache memory together.
- Let, N_1 be the number of accesses to cache memory and N_2 be the accesses to main memory the hit ratio is defined as,

$$h = \frac{N_1}{N_1 + N_2}$$

  - **Miss Ratio (m) :** It is the ratio of accesses made by processor or CPU to the main memory to the total number of accesses made to the main memory and cache memory together.

$$m = \frac{N_2}{N_1 + N_2} \qquad \text{Therefore, } h = 1 - m$$

  - **Look up penalty :** It is defined as the time required for the cache controller logic to ascertain the cache memory is a hit or a miss.

10.11 ELEMENTS OF CACHE DESIGN

UQ. 10.11.1 What are the features of cache memory design?

MU - Q. 3(a), May 15, 8 Marks, Q. 5(a), May 16, 10 Marks

UQ. 10.11.2 What are elements of cache design? Explain in detail.

MU - Q. 4(b), May 14, 8 Marks

- The processor or CPU generates address for access to the memory, if the specified address is not available already in one of the cache memory lines or blocks then while it is being accessed from the main memory, it is also copied into the cache memory in one of its cache lines or blocks. Various factors come into picture in the process of Organizing the Cache memory. The Cache memory organization factors are important in the design of Cache memory systems.
- Few basic design elements which are considered to classify and differentiate different Cache architecture are as follows:

1. Cache Addresses
2. Cache Size
3. Mapping Techniques
4. Replacement Algorithm
5. Write Policy
6. Cache coherency
7. Line Size
8. Number of Caches

1. Cache Addresses

- The addresses of the cache memory. The caching (i.e. Copying in to the faster memory) is done block by block or line by line. It is called as **cache block** or **cache line**.
- The addresses in the main memory that are to be cached or that are cached, termed as **cache addresses**. These addresses keep track of what part of main memory is actually lying in the Cache memory.

2. Cache Size

- It is the size of the Cache memory. Generally, cache memories, which are faster memories occupy higher space (i.e. their packing density on the chip is low) and therefore, large implementations of Cache memory are not practical.

- Small amount of memory, about 1% of the total memory implementation in the system; is usually populated as cache memory.

3. Mapping Techniques

- The rule that determines, exactly which line or block in the main memory would be populated or copied in the Cache memory and at what block position in the Cache memory; is called as the Mapping Function of the Cache memory.
- Most of the Cache memories are mapped using either Fully Associative or k-Way Set Associative (where k can be any power of 2) or they are Direct mapped.

4. Replacement Algorithm

- Replacement Algorithm specifying the cache line or block that would qualify to go out of cache memory when the current access determines (by virtue of an appropriate mapping function) to load a new block or line in to the Cache memory and replacement is necessary as currently the specified cache is full. Then, the replacement in such case is carried out as per the replacement algorithm.
- Most commonly used replacement algorithms are – LRU (Least Recently Used) or FIFO (First In First Out).

5. Write Policy

- Write policy determines the behaviour of Cache Memory and actions taken upon by the Cache system in case of the Processor or CPU Write cycles. When the processor or CPU performs a Write cycle, there is always the case of data integrity as The Cache and Main memory would then contain different data.
- To resolve this issue, write policies are formed and followed. In most of the cases, WT (Write Through) or WB (Write Back) are the two write policies that are followed by the Cache systems.

6. Cache coherency

- When a specific element is cached in the cache memory, there exist two copies of the information, one in the cache location and second one in its parent main memory location. The cached location gets modified if the processor or CPU performs a write operation on it.
- On the other hand, main memory location contents can get modified due to system bus access by the DMA controller or any other bus master.
- When such events occur, the contains of cache location and its parent main memory location would be different or incoherent. Such situation is undesirable and the contents must always be coherent and maintain the integrity. The set of techniques used to maintain the data integrity and coherency on the contents in the cache memory – main memory system, are called as **cache coherency** techniques.

7. Line Size

- As stated earlier, cache is never populated byte by byte but it is usually by line by line. Therefore, at any instant the loading (or filling) as well as unloading (writing back) is always done, one line at a time.
- The size of Cache line specifies that one Cache line is comprising of how many data units; i.e. Bytes or Words. The proper line size helps the Cache system in exploiting the spatial locality of reference.

8. Number of Caches

- Based on the system configuration, complexity and performance requirements, more that 1 cache memories are implemented in the system.
- They are called Levels of cache memory. The systems implement One, Two or Three Levels of Cache memories in their configurations.

10.11.1 Cache Address

Addressing/Positioning of the Cache Memory

- **Virtual Memory** is the term used to address memory from logical point of view without bothering about Physical memory actually available. This concept of Virtual Memory is used in almost all non-embedded applications and in some embedded applications. In order to implement virtual memory, the address generated by CPU are virtual addresses. In order to convert these addresses into actual physical addresses, a Memory Management Unit (MMU) is required.
- There are two ways in which Cache Addressing can take place in order to implement the concept of Virtual Memory using MMU.

1. By connecting a Cache Memory in between CPU and MMU as shown in Fig. 10.11.1.

   In this case, the CPU addresses the Cache Memory directly without going through MMU by generating Virtual Address. The data in the Cache Memory is stored using this virtual address, hence cache memory is also called as **logical cache** or **virtual cache.**

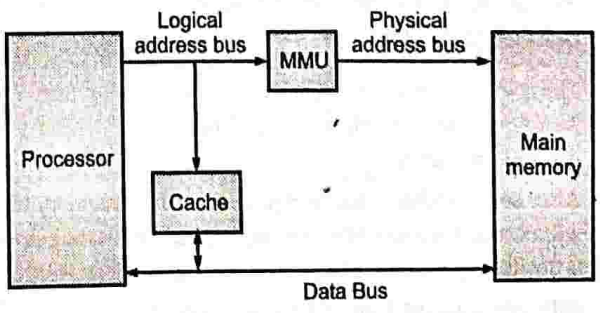


Fig. 10.11.1 : Virtual cache

Module 5

2. By connecting a Cache Memory in between MMU and Physical Memory as shown in Fig. 10.11.2.

In this case, cache memory is connected in between MMU and Main memory. Hence, the memory access time in this case is more as the MMU has to generate a physical address first from the logical address generated by CPU. Using the physical address, the cache memory can be accessed.

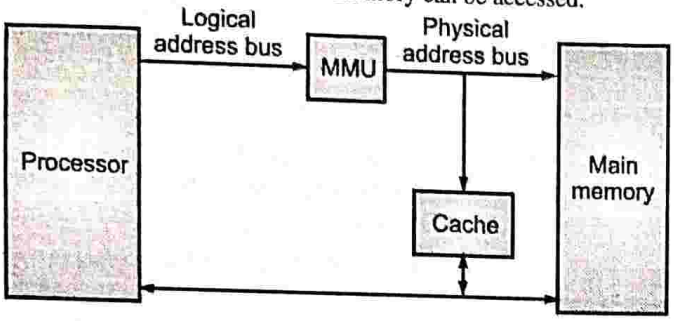


Fig. 10.11.2 : Physical cache

10.11.2 Cache Size

- As discussed in section 10.6, if the size of cache memory increases, the cost of system increases. Also, if the size of the cache is too small, the memory access time increases, reducing the speed of the entire system.
- There is a trade-off between speed and size of the cache.
- There are other factors while considering the size of cache. If the cache size increases, the number of transistors required addressing the cache also increases, increasing the complexity of the system.
- This also decreases the speed of memory access hence making the cache memory slightly slower. The size of cache is also limited by the size of available circuit board.

10.11.3 Cache Mapping Techniques

UQ. 10.11.3 Explain associative cache mapping techniques.

MU - Q. 6(a), Dec. 15, 5 Marks

Mapping function or mapping techniques of update policies

- The cache memory contains less memory lines than main memory, hence the main memory blocks are needed to be mapped into cache.
- Also, an algorithm is required to decide which memory block will occupy lines in cache. The organization of cache memory is decided by the type of mapping function implemented.
- Different mapping functions are as follows :

A. Direct mapping
B. Associative mapping (fully associative mapping)
C. Set associate mapping

A. Direct Mapping

- This is the simplest type of mapping function. As shown in Fig. 10.11.3, this technique maps each block of main memory into only one possible cache line.

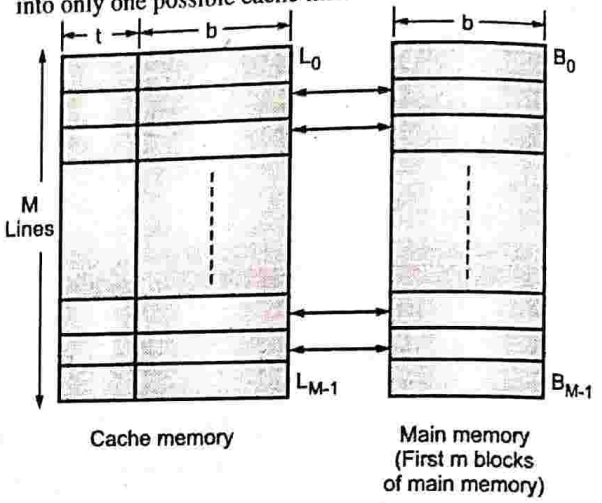


Fig. 10.11.3 : Direct Mapped Cache Memory

- The mapping function for direct mapping can be given as

$i = j \text{ modulo } m$

Where, i = cache line number
j = main memory block number
m = number of lines in the cache

- Fig. 10.11.3 shows how first m blocks of main memory are mapped into cache.
- The cache memory contains m different lines, and each line contains b + t bits. The main memory can be separated into different blocks of length m.
- There is direct one-to-one mapping between each memory location of main memory with cache as shown in the Fig. 10.11.3, i.e. B_0 memory location gets mapped with L_0 location of cache.
- For accessing cache fields, each main memory location can be divided into 3 different fields. In most of the modern machines, the least significant bytes represent addresses. Hence, the least significant w bits identify a unique word or byte within block of main memory.
- The remaining s bytes specify 2^s blocks of main memory. The cache logic identifies these bits as tags of s – r bits and line of field of r bits.

Address length = (s + w) bits

Number of addressable units = $2^{(s+w)}$ words or bytes

Block Size = line size = 2^w words or bytes

Number of blocks in main memory = $\frac{2^{(s+w)}}{2^w} = 2^s$

Number of lines in cache = $m = 2^r$

Size of cache = $2^{(r+w)}$ words or bytes

Size of tag = (s – r) bits

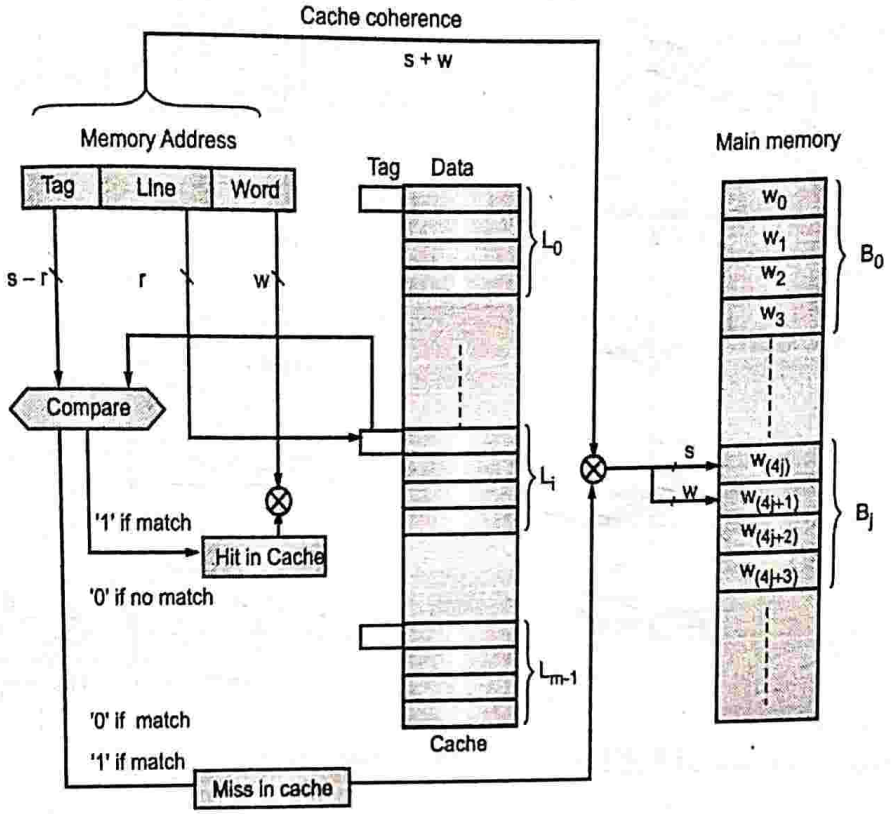


Fig. 10.11.4 : Direct mapping cache organization

Advantages of direct mapping

Simple to implement

It requires less amount of tag memory to be implemented per cache line.

It requires only one comparison to determine whether the cache is a hit or a miss. Therefore, look up penalty is low.

Disadvantages of direct mapping

Thrashing : For any given block, the cache location is fixed. If program refers to code from different blocks repeatedly, the current existing block from cache has to be thrashed every time.

Cache memory may not be full.

3. Due to thrashing and low memory occupancy, the hit ratio decreases.

Example of direct mapped cache

GQ. 10.11.4 Draw organization of direct mapped cache with specifications - 16 bytes/cache line or Block, 32 KB cache memory, 20 Bit Addresses generated by the processor.

– In direct mapped cache, the cache block is populated/depopulated by same numbered block of any tag zone in the main memory.

Module 5

• NOTES •

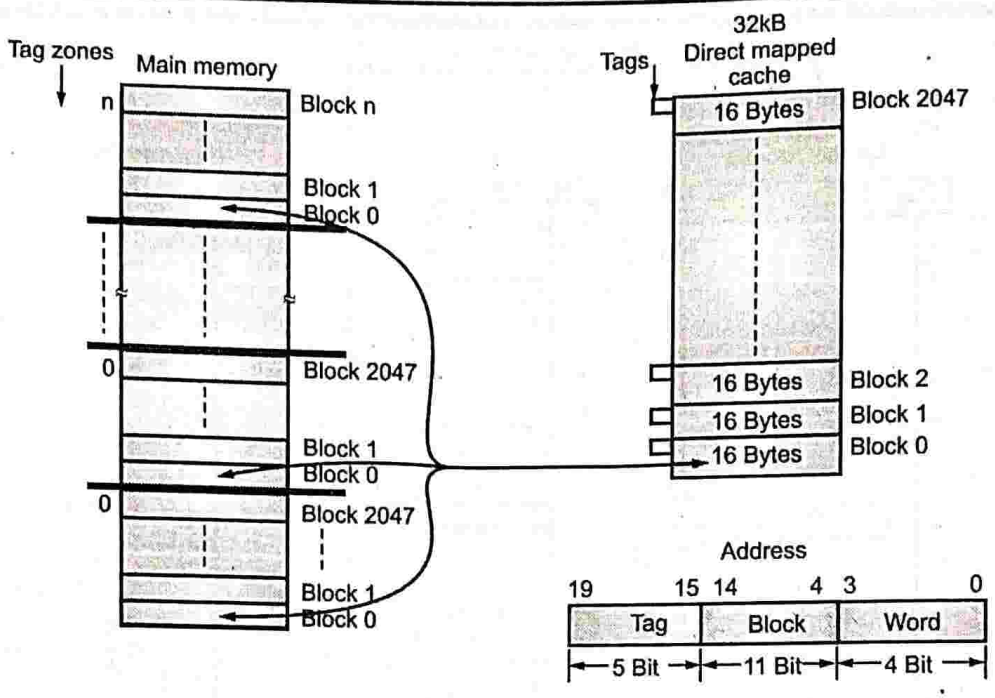


Fig. 10.11.5 : Direct Mapped Cache – Example

B. Associative Mapping (or Fully Associative Mapping)

- The disadvantage of direct mapping is overcome by Associative Mapping. Associative Mapping allows each memory block to be moved into any line of cache memory. The cache control block takes memory address as combination of Tag and Word field.
- Tag is used to exclusively identify a block of main memory. Cache control logic goes through tags of each line to find out whether a block is present in cache memory or not.
- As shown in the Fig. 10.11.6, the line number is not determined by any field in the address.

Address Length = (s + w) bits

Number of addressable units = $2^{(s+w)}$ words or bytes

Block size = line size = 2^{w} words or bytes

Number of blocks in main memory = $\frac{2^{(s+w)}}{2^{w}}$

Number of lines in cache = undetermined

Size of tag = s bits

☞ Advantages of fully associative mapping

1. As any cache block can be occupied from any memory block, thrashing never occurs.
2. Cache memory is usually fully occupied as the first replacement occurs only when all cache blocks are full.
3. Due to very high occupancy and no thrashing, this mapping technique has highest hit ratio.

☞ Disadvantages of fully associative mapping

1. Fully associative mapping technique requires more amount of tag memory to be implemented.
2. It requires n comparisons where n is number of cache blocks in the cache memory. Therefore, look up penalty is high.

☞ Example of fully associative cache

GQ. 10.11.5 Draw organization of fully Associative cache with specifications 16 Bytes/cache line or block, 32 KB cache memory, 20 Bit Address is generated by the processor.

- In fully associative cache, any block in cache can be populated/de-populated by any block in the main memory.

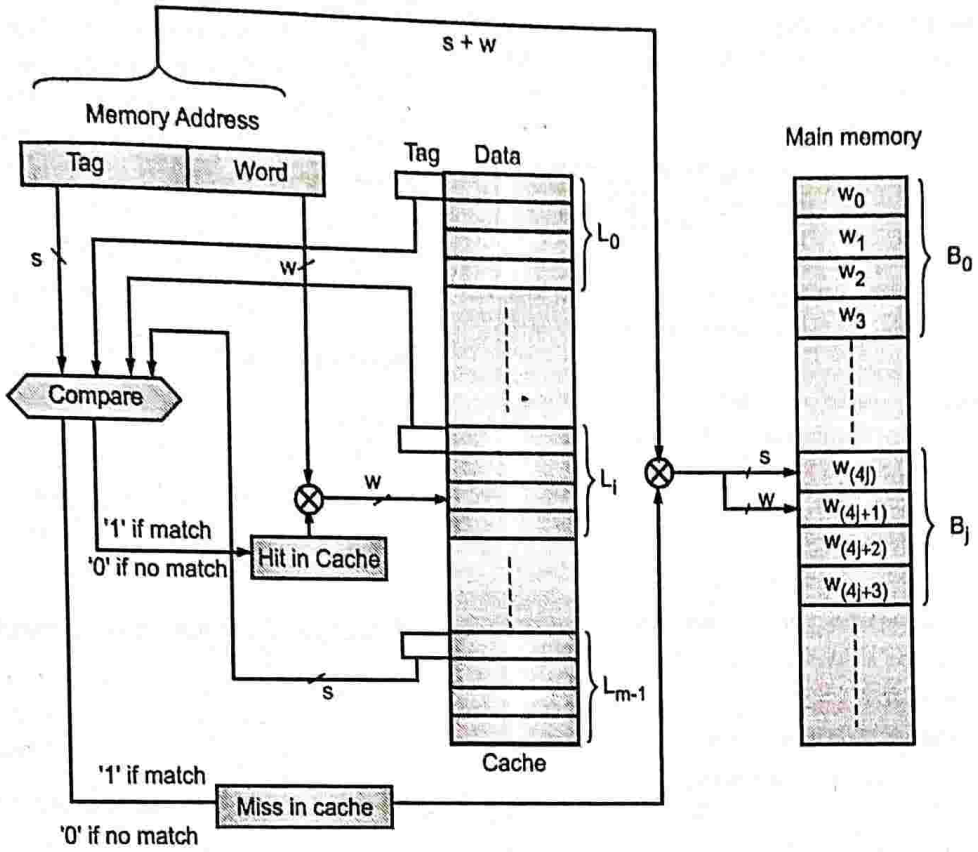


Fig. 10.11.6 : Fully associative cache organization

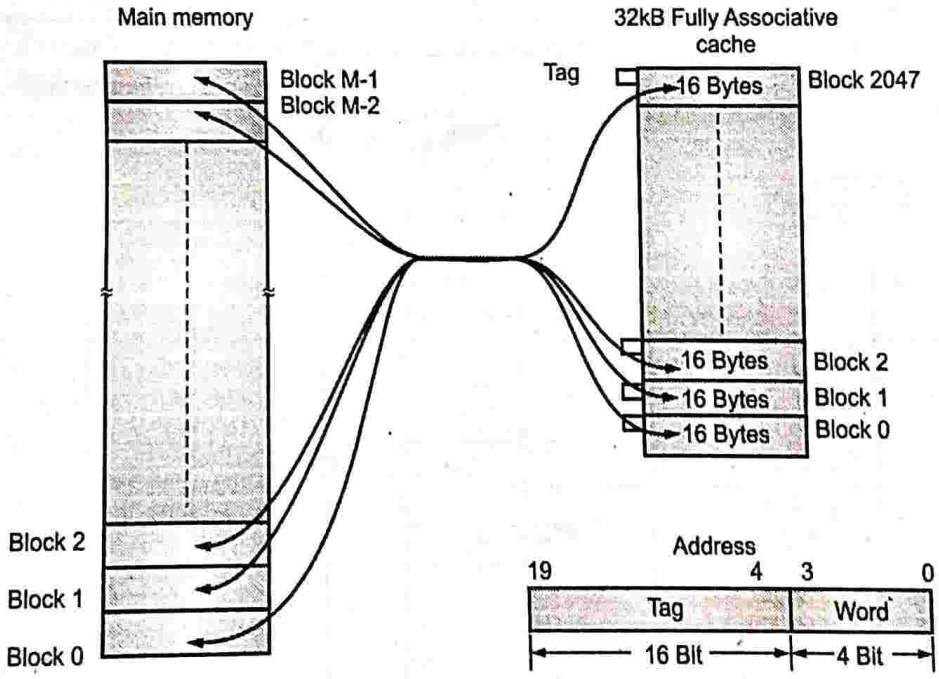


Fig. 10.11.7 : Fully Associative Cache Mapping – Example

Module 5

C. Set-Associative Mapping

UQ. 10.11.6 Explain set associative cache mapping.

MU - Q. 4(a), Dec. 17, 5 Marks.
Q. 3(a), May 18, 7 Marks

- Set-associative mapping is the combination of both direct mapping and associative mapping. By doing so, the disadvantages of both direct and associative mapping are eliminated.
- For this technique, the cache is divided into number of sets and each set consists of number of lines. This can be mathematically represented as follows :

$$m = v + k$$

$$i = j \text{ modulo } v$$

Where, i = cache set number; j = main memory block
m = number of lines in cache
v = number of sets ;
k = number of lines in each set

- The Fig. 10.11.8 is k-way set-associative mapping. In this type, block B_j can be mapped into any of the lines of set j.
- Each direct mapped cache is referred to as a *way* consisting of v lines. The first v lines of main memory are copied into first v lines of cache memory. The next v lines are further mapped into consecutive *way*.
- The cache control logic separates the memory address into three fields, Tag, Set and Word. Let there be total d set bits. Hence any one of the set v will be specified by $v = 2^d$ sets. The s bit of the Tag and Set field is used to select any one of the 2^s blocks of main memory. For k-way set-associative mapping, the tag in a memory address is smaller.

Address lines = (s + w) bits
Number of addressable units = $2^{(s+w)}$
Block size = line size = 2^w words or bytes
Number of blocks in main memory = $\frac{2^{(s+w)}}{2^s} = 2^s$
Number of lines in set = k
Number of sets = $v = 2^d$
Number of lines in cache = $m = kv = k \times 2^d$
Size of cache = $k \times 2^{(d+w)}$ words or bytes
Size of tag = (s – d) bits

☞ Advantage of set associative mapping

1. Set-associate mapping is the combination of direct mapping as well as associative mapping. Therefore, it eliminates the disadvantages of both the mapping techniques.

☞ Disadvantage of set associative mapping

1. The only disadvantage of this technique is that it is relatively complex to implement as compared to earlier two methods.

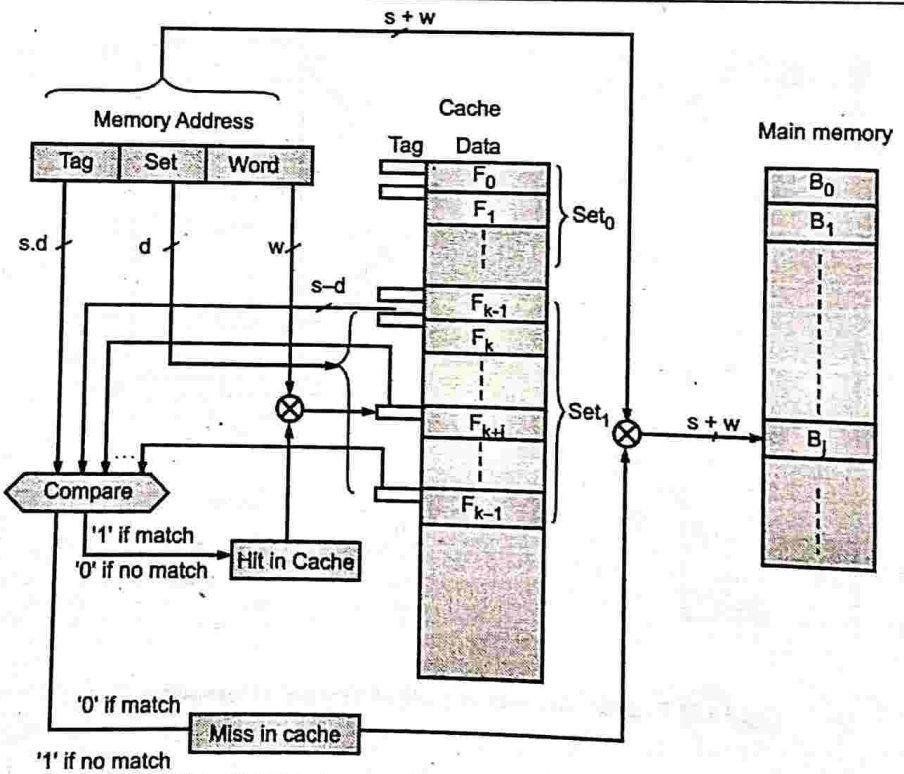


Fig. 10.11.8 : k-way set associative cache organization

Example of 2-way set associative cache

GQ. 10.11.7 Draw organization of 2-way set associative cache with specifications - 16 Bytes/cache line or Block, 32 KB cache (16 KB/way), 20 Bit Address is generated by the processor.

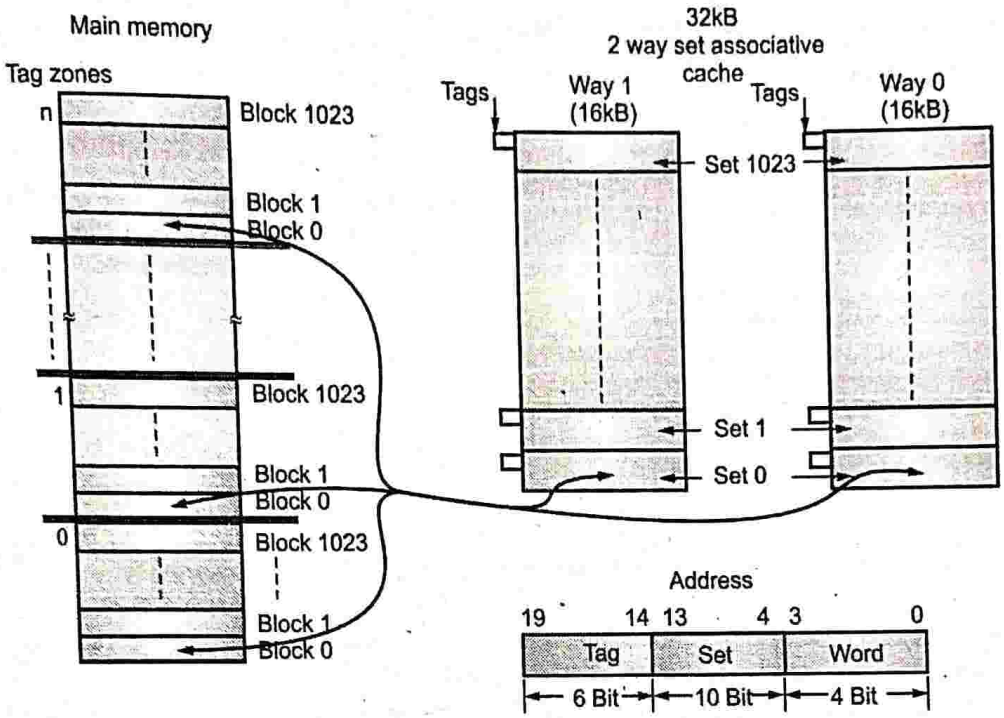


Fig. 10.11.9 : Two-Way Set Associative Cache Mapping - Example

In 2-way set associative cache, each set has 2 Blocks (i.e. set 0 has Block 0 in way 0 and Block 0 in way 1). Any (block of) given set number can be populated/de-populated by same numbered block in the main memory.

10.11.3(a) Comparison of Direct Mapped, Set Associative and Fully Associative Cache Memory Mapping

Sr. No.	Direct mapped	Fully associative	Set associative
1.	In this technique the look up penalty is less as only one comparison is needed to determine whether cache is a hit or miss.	In this technique, look up penalty is high as N comparisons are required to determine cache is hit or miss.	This technique requires two comparisons (2 way set associative) or four comparisons (4 way set associative) and therefore, look penalty is low.
2.	In this technique, tag field is small in length and therefore, tag memory required is less.	In this technique, tag field is much bigger and therefore, it requires high amount of tag memory.	In this technique, the tag field is smaller than fully associative but bigger than direct associative and therefore, tag memory requirement is intermediate.
3.	Chances of some tag lines remaining empty are very high.	Chances of some tag lines remaining empty are very low.	Chances of some tag lines remaining empty are moderate i.e. greater than Fully associative and less than Direct associative.
4.	This technique provides relatively lower hit ratio as cache memory is not always full.	This technique provides highest hit ratio as cache memory is almost always full.	This technique provides hit ration better than direct map but lower than fully associative.
5.	Chances of thrashing are relatively high.	Chances of thrashing very low.	Chances of thrashing are intermediate.

Module 5

10.11.3(b) Problems based on Cache Mapping Techniques

UQ. 10.11.8 *Consider a cache memory of 16 words. Each block consists of 4 words. Size of the main memory is 256 bytes. Draw associative mapping and calculate TAG and WORD size.*

MU - Q. 5(B), May 19, 10 Marks, Q. 5(b), Dec. 18, 10 Marks

Note : Both these examples are identical with same mapping technique and same set of values.

The Data from the example is

Cache Memory is of 16 Words – Each Block is consisting of 4 Words.

Therefore, the Cache memory is organized as 4 Blocks of 4 Words each.

Therefore, Word field (W) of the system is of $2^2 = 4 \rightarrow$ 2 Bits in width.

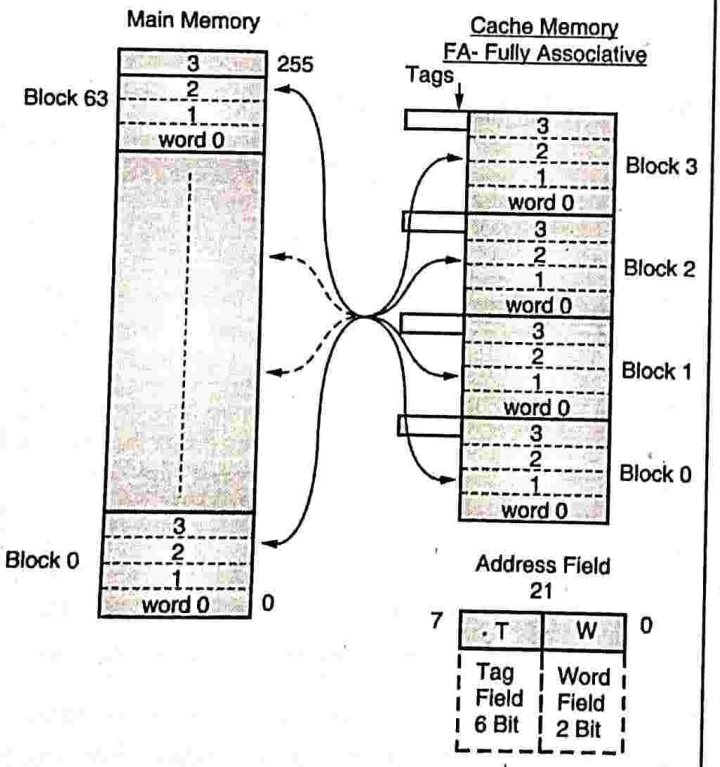


Fig. 10.11.10

The Main memory is of 256 Bytes – Total Address field (A) is of 8 Bits.

However, Address field (A) = Tag Field (T) + Word field (W) for Associative i.e. Fully Associative Cache Mapping.

Therefore, Tag Field is = A – W = 8 – 2 = 6 Bits.

The Main memory would be divided into 64 Blocks of 4 Words each and cache would be organized as 4 Blocks of 4 Words each. Any Cache block can be populated from/to any block in the Main memory.

The Fig. 10.11.10 shows organization of such cache memory.

UQ. 10.11.9 *A block set associative cache consists of 64 blocks divided in 4 block sets. The main memory contains 4096 blocks, each 128 words of 16 bit length.*

1. *How many bits are there in main memory address?*
2. *how many, bits are therein c cache memory address (tag, set, and word fields)?*

MU - Q. 3(b), May 17, 10 Marks

The Data from the example is :

Cache Memory is consisting of 64 Blocks divided into 4 block sets.

Therefore, this Cache Memory is organized as 4-Way Set Associative cache memory with each set having 4 blocks. These are, Therefore 64 ÷ 4 = 16 Sets.

No. of Sets = 16.

Therefore, Set Field (S) is of $2^4 = 16 \rightarrow$ 4 Bits in width ...(1)

No. of Ways = 4 (No. of Blocks in each set = 4)

No. of Words / Block = 128 and each word is of 16 Bits i.e. 2 Bytes

Therefore, Each Block is consisting of 128 × 2 = 256 Bytes

Therefore, Word field (W) of the system is of $2^8 = 256 \rightarrow$ 8 Bits in width. ...(2)

Total Main memory addressability is

$4096 \times 128 \times 2$ Bytes = 1048576 Bytes = 2^{20}

Therefore, Total Main Memory Address Field is = 20 Bits ...(3)

However, Address field (A) = Tag Field (T) + Set Field (S) + Word field (W) for Associative i.e. Fully Associative Cache Mapping.

Therefore, Tag Field (T) is = A – (S+W) = 20 – (4+8) = 8 Bits ...(4)

Hence, the answer –

1. Main Memory Address is of 20 Bits- As per calculations in (3) above
2. Cache memory Address Contains 8 Bits Tag Field, 4 Bits Set Field and 8 Bits Word Field – As per the calculations in (1), (2) and (4) above.

The Fig. 10.11.11 shows organization of such cache memory.

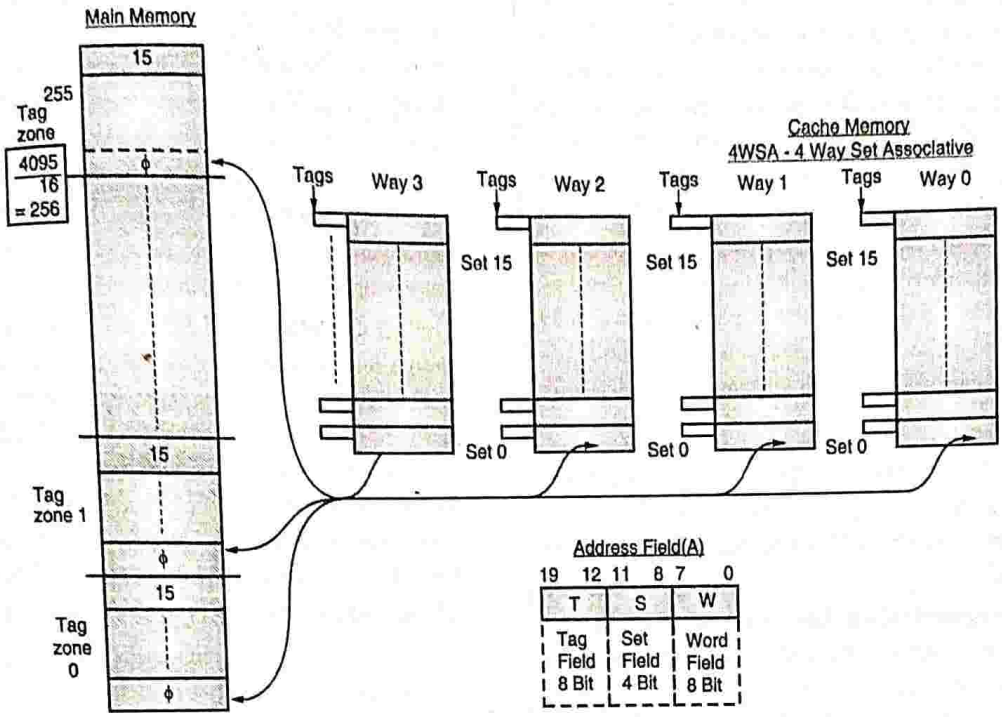


Fig. 10.11.11

10.11.3(c) Handling Cache Misses and Writes

- CPU generates an address when it tries to write some data into memory. If memory block containing that address is present in cache, it modifies cache or else that memory block is first brought into cache and then it is modified into cache memory itself. This modified data needs to be updated back into main memory as any other I/O device might need to use the data. If at least one memory write operation is performed on a word line of cache, the main memory should be updated with the updated data before bringing in new block into cache.
- There are two main situations needed to be taken care of. First, one i/o device along with CPU has read-write access to memory. If a word is altered only in cache, then the corresponding memory word is invalid. Or, if i/o device has altered main memory, the word in cache is invalid. A more complex situation arises when one memory is shared with multiple processors attached to a same bus and each processor has its own local cache.
- There are two techniques to address the above-mentioned problem.

1. **Write Through :** This is a very simple technique. In this technique a write operation is made both to cache and main memory thus ensuring the cache and main memory are both updated and synchronised. There is a disadvantage of this technique. This technique creates huge amount of memory traffic resulting into a bottleneck.
2. **Write Back :** In this method, the update is made only in cache memory. There is an additional flag bit called as DIRTY bit or USE bit. If an update is made in cache memory, the flag bit associated with that particular line is set to 1. When a block is getting replaced, the changes will be written into main memory only if the USE bit if the line is set. Also, if I/O tries to access main memory, first the USE bit of the corresponding word line in cache memory will be checked.

- If the bit is set, the updated value from the cache memory will be used else the previous memory value will be taken. The main drawback of Write Back method is entire memory access happens through cache memory. This increases the circuit complexity and results into bottleneck.

10.11.4 Replacement Algorithm

- In order to bring new block into cache and there has to be space available in cache memory. If the cache memory is full, some block of cache memory has to be replaced and new block of memory needs to be added.
- For direct mapping, there is only one line possible for any particular block. But for associative and set associative techniques, multiple lines are possible for any block. Hence an algorithm is needed for efficient allocation. This algorithm must be implemented in hardware in order to get high speed.
- Four of most commonly used algorithms are as follows :

1. **Least Recently Used (LRU) :** This algorithm is more effective when the program contained in main memory contains many repetitive instructions or loops. As name

Module 5

suggest, this algorithm replaces the block which is not used recently with new block which is supposed to be required by CPU. In order implement this algorithm, each line in cache memory is added with a flag bit. This bit is called as USE bit. If any of the line from a block is used by CPU, the USE bit of all the lines from that block is set to 1. By looking at value of USE bit, we can find out which block is recently used and which is not. The block whose value of USE bit is 0, gets replaced by new block. LRU is easy to implement for a fully associative cache. LRU is the most popular algorithm as it is the simplest algorithm to implement.

Other Algorithms are :

2. **First In First Out (FIFO) :** This algorithm replaces the block which is there in cache memory for the longest time. This algorithm is implemented as Round Robin technique.
3. **Least Frequently Used (LFU) :** This algorithm is used when the program contained in the main memory contains non repetitive, sequential set of instructions. LFU replaces the block from cache which has experienced fewer references.
4. **Random Method :** This method, as name suggests, randomly replaces a line from cache memory and allocates new line from main memory. There are some simulated results showing random method being slightly inferior to the other above-mentioned methods.

10.11.5 Number of Caches

- Initially, system designing was done using a single cache. Recently, due to advancement in designing and increase in logic density, number of caches in system design has increased. There are two main aspects in designing the system i.e. Multilevel caches and use of Unified vs Split caches.
- **Multilevel Caches :** The increase in logic density has made it possible to design on-chip cache memory. This on-chip cache memory has many advantages like faster data transfer rate, less utilization of system's external bus. Whenever a data or instruction request is made by CPU, it is found in the on-chip cache, hence the bus access is eliminated. Hence the bus is available for other transfer operations. This increases overall system's performance.
- Along with on-chip cache, most contemporary design includes off-chip external cache. This off-chip external cache is also called as Level 2 (L2) cache, Level 1 (L1) being on-chip cache. There are some advantages of including L2 cache.
- If processor makes a request to memory location which is not present in cache, then DRAM or ROM has to be accessed. DRAM and ROM being considerably slow as compared to cache, the performance of system decreases. Instead if secondary SRAM cache is used as L2 cache, the probability of missing information from L1 cache found in L2 cache increases. If the SRAM is fast enough such that it can be accessed within zero-wait state, the performance of the system will not degrade much.
- Many system designs do not use system bus for data transfer between CPU and L2 caches. Instead they use a separate bus for data transfer. This reduces burden on system bus. Also, with more advancement in logic density and shrinkage of processor components, the L2 cache are included on-chip, improving performance.
- **Unified v/s Split Caches :** Unified caches means traditional design consisting of both data and instruction stored in single cache memory. Nowadays, there are separate cache memory designed for data and instructions. This dedicated data and dedicated instruction cache memory both act as level 1 cache. When processor tries to fetch instruction from memory, L1 instruction cache is referred first. And when processor tries to fetch data from memory, L1 data cache is referred first.
- The **advantages of Unified Cache system** are as follows :
  1. As data and instruction cache are same, the hit ratio increases. This is because, if execution pattern involves a greater number of data fetch, the cache will get updated with more data from main memory and if execution pattern involves a greater number of instructions fetch, the cache will get updated with more instructions from main memory.
  2. Only a single cache needs to be design.

The main **advantages of Split Cache system** are as follows :

a. The contention between instruction fetch/decode unit and execution unit is eliminated.

b. This helps in the design where pipelining of instruction is used.

c. The combination of unified and split cache is implemented in current systems. The split caches are implemented as level 1 cache and unified cache are implemented as level 2 cache.

A. One Level and Two Level Cache

- Cache memories are the faster memories used for temporary storage by the processor.
- All the active data, information, code etc. is stored in temporary storage. Mostly the processors requiring smaller cache due to their lesser complexity use L1 cache. L1 cache is built in temporary fast, closer memory present on the microprocessor chip itself.
- This is called as the one level cache, since only one type or only one level of cache is involved. Few processors performing complex tasks require more cache memory. This requirement is satisfied by using L2 cache along with L1 cache. L2 cache is present as an external memory chip but accessed and addressed as if it were a part of internal cache. Since both L1 and L2 are involved, it is termed as two-level cache.

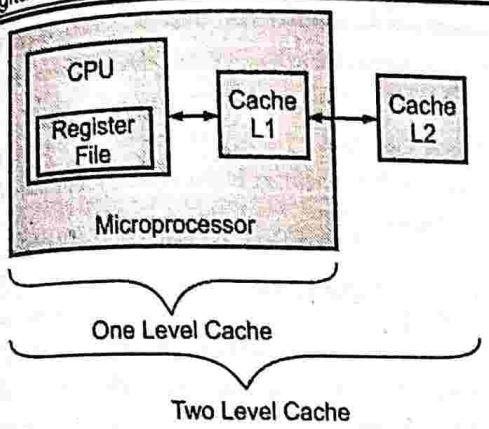


Fig. 10.11.12 : One and Two level Cache

B. Performance Characteristics of Two Level Cache - Locality and Operation

1. Locality

- The L1 and L2 cache memory forms a buffer between the main memory and the CPU forming a two-level internal memory. This two-level architecture uses the property of locality known as **locality of reference**, to improve its performance over one level cache. Locality of reference works on the principle that the memory references tend to cluster. When memories accessed for a long period of time, the clusters change, but when shorter time span is considered, the memories accessed usually tend be closer and hence form a cluster.
- The principle of locality is based on the following observations :
  1. Except the branching and interrupt instructions, most of the instructions to be fetched and executed are in sequence. Hence usually, the next instruction to be executed follows immediately after the one last fetched.
  2. It is very uncommon to have continuous function calls and returns. Instead few function calls are usually seen, hence there is localization of memory location over shorter spans before memory fetches return to the main flow.
  3. Looping functions usually have small number of instructions repeated several number of times. This also leads to confinement of memory access to a small contiguous portion of the program or memory.
  4. In some cases, computation involves processing data structures like arrays, sequences, etc. This leads to successive memory access.
- Literature suggests two types of locality or references, spatial and temporal. **Spatial locality** refers to the tendency to execute instructions involving memory locations that are clustered. It can also mean tendency to execute instructions accessing sequential data locations.
- **Temporal locality** refers to the tendency of the processor to access those memory locations which were accessed very recently. Iterative loops are example of temporal locality. Traditionally, these recently used data and instructions are captured in cache memory exhibiting temporal locality. Spatial locality is implemented by using large cache size and prefetching mechanism.

2. Operation of two - level Memory

- The locality property can be used in the formation of the two - level memory. Let the upper level memory (L1) be referred as M1 having small size, faster execution and costly. Also, M2 be the lower level (L2) memory having bigger size, slower execution and less costly as compared. M1 is used to partially and temporarily store the contents of memory M2.
- Whenever a reference is made, an attempt is made to find that reference in the M1 memory first. If it is available, it can be accessed quickly. If not, then a block of memory is transferred from M2 to M1 and then data is accessed via M1. Once a block is moved in M1, due to locality, there will be number of accesses that will happen in M1 for that block which results in faster execution.
- If H is the hit rate for M1 then (1 – H) will be the hit rate for M2. Let T1 and T2 be the access times for memories M1 and M2 respectively. Then the average access time, Ts can be given as,

$$\begin{aligned} Ts &= H \times T1 + (1-H) \times (T1 + T2) \\ &= HT1 + T1 + T2 - HT1 - HT2 \\ &= HT1 + (1-H) \times T2 \end{aligned}$$

- It can be observed that for a high percentage of hits, the total access time is much closer to T1 than T2.

☞ Performance

To evaluate the performance of the two level cache, cost per bit is also an important parameter. Let Cs be the average cost per bit for M1 and M2 together,

C1 be the average cost per bit for M1,

C2 be the average cost per bit for M2,

S1 be the size of M1 and

S2 be the size of M2. Then,

$$Cs = \frac{C1S1 + C2S2}{S1 + S2}$$

Thus, if C1 >> C2 and S1 < S2, then Cs would be equal to C2.

Module 5

10.11.6 Cache Coherency

UQ. 10.11.10 Explain in brief cache coherency problem.
MU - Q. 6(b), Dec. 17, 10 Marks

UQ. 10.11.11 Write short notes on Cache Coherency.
MU - Q. 6(b), Dec. 16, 10 Marks

UQ. 10.11.12 Explain in details cache coherency.
MU - Q. 1(c), Dec. 14, 3 Marks, Q. 4(c), May 15, 7 Marks

- Consider a system in which two or more processors, having their own cache sharing a single memory as shown in Fig. 10.11.13.
- If data word in any of the cache (e.g. Cache 1) is modified, and the same block of memory is present in multiple cache (e.g. Cache 2, 3), that data word in main memory and every other cache becomes invalid, even if write through policy is implemented. This problem is termed as Cache Coherency.

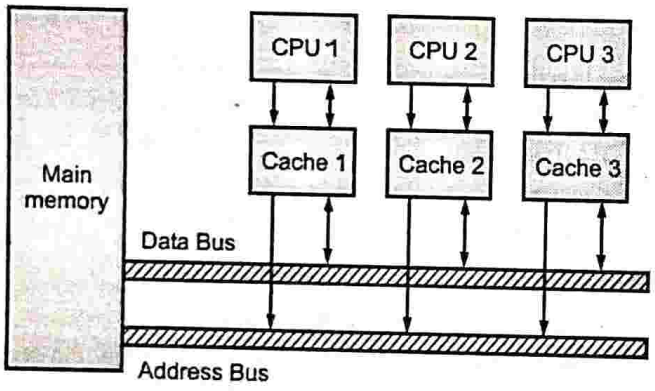


Fig. 10.11.13 : Cache Coherence

- Different approaches to prevent cache coherency are as follows :
  - (i) **Bus Watching with Write Through :** In this case, each cache controller keeps a watch on address bus. If there is write operation performed on main memory block, by some cache, which is also shared with other cache, the cache controller will invalidate that entry on main memory. The policy has to be decided by all cache controllers in advance.
  - (ii) **Hardware Transparency :** In this case, an extra hardware is used to update all the caches and main memory any change happening in the shared data code. Hence, if any cache updates a word, it is immediately updated in main memory. Also, every cache gets updated with the same.
  - (iii) **Non-cacheable Memory :** In this approach, if a block of memory is required by multiple caches at same time, then that memory will not be shared with any of the caches. That memory block is called as **non-cacheable memory.**

10.11.7 Line Size

- Line size is a critical design element. When a block of data from main memory is copied into cache, along with the required word, adjacent words are also moved into cache.
- If the size of block increases, the possibility of the next word contained in the same block also increases. Hence the hit ratio increases. But size of block should increase beyond a limit else two effect comes into play :
- If block size increases, the total number of blocks that fit into cache decreases. Each block fetch replaces older cache contents, a small number of blocks results in data being overwritten shortly.
- As block size increases, the distance current word and requested word increases, hence it is less likely to be needed in near future.

10.12 INTERLEAVED MEMORY SYSTEM

UQ. 10.12.1 Explain Memory Interleaving Techniques.
MU - Q. 1(D), May 19, 5 Marks

UQ. 10.12.2 Explain the Interleaved memory.
MU - Q. 2(b), May 16, 10 Marks

UQ. 10.12.3 Explain various high speed memories such as interleaved memories and caches.
MU - Q. 5(a), Dec. 14, 10 Marks

- The memory address space of processor or CPU is very large, and the size of individual memory devices is relatively smaller. Therefore, the physical memory address space is partitioned or segmented into smaller memory blocks or units using addressing techniques. Such mechanism is called as **Memory Partitioning** or **Memory Segmentation.**
- An important technique by which the memory space can be divided is called as **Memory Interleaving.**

Memory Interleaving and Inter leaved memory system

- The size of the memory address space is determined by the number of address lines provided by the CPU or processor on its address bus. K address lines provide the memory address space of 2^K locations.

Using a few address lines separately for the selection of memory block and connecting the remaining address lines in the address bus to the memory device used in that memory block makes the memory partitioning or segmentation possible.

Such technique of using address lines in the address bus partially for selecting the memory block and partially for accessing the memory block, is called as **Memory Interleaving** and the memory devices configured are said to be in **Interleaved Memory System.**

Let the CPU or processor have address bus of K bits and Therefore, the address lines are numbered from 0 to K-1 (A_{K-1} to A_0). Let j lines out of these K lines be used for the selection of the memory block. This leaves K-j lines for accessing or addressing each memory block. Based on whether the j lines used for the selection of memory blocks, are from the MSB aside or LSB side, there are two types of memory interleaving. Mainly, **Higher Order Interleaved** or **Lower Order Interleaved.**

10.12.1 Higher Order Interleaved (HOI) Memory

- In the Higher Order Interleaved memory system, j address lines from the MSB side are used for memory block selection, leaving the lower K-j address lines for memory block access.
- In this situation, address lines from 0 to K–j–1 (A_{K-j-1} to A_0) are kept for the memory block access and address lines from K–j to K–1 (A_{K-1} to A_{K-j}) are kept for memory block selection. HOI system divides the memory into equal sized blocks with each block having the consecutive address range.

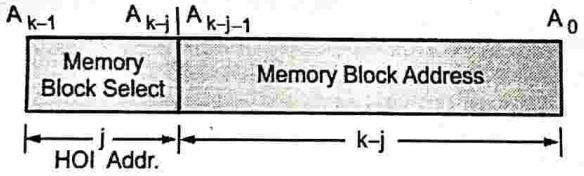


Fig. 10.12.1 : Higher Order Interleaved (HOI) Memory

- This method is suitable for dividing the memory into different memory areas using these HOI memory blocks.

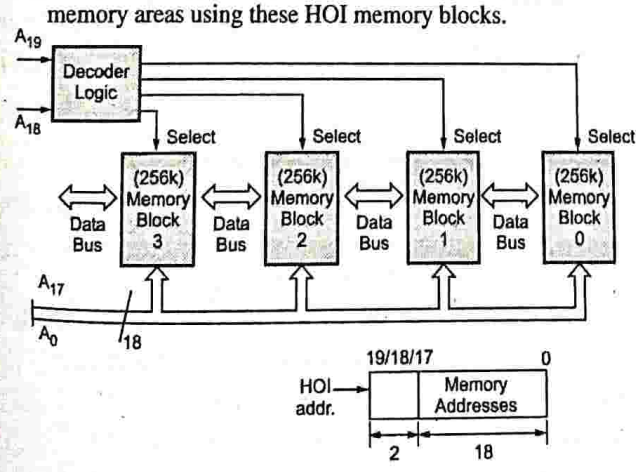


Fig. 10.12.2 : Four blocked HOI memory

- E.g. Let us consider a two bit HOI memory system for a CPU having 20-bit address bus (such as 8086). Such systems provide two bits for HOI from the MSB side ($A_{19} - A_{18}$) and the memory block is addressed by the remaining 18 bits on the lower side of the address bus ($A_{17} - A_0$).
- Therefore, the system divides the total of 1 MB (1024 KB) memory address space into four memory blocks of 256 KB sized and having consecutive address range. The blocks are selected with the help of decoder using the HOI address lines, A_{19} and A_{18}.

10.12.2 Lower Order Interleaved (LOI) Memory

- In the Lower Order Interleaved memory system, j address lines from the LSB side are used for memory block selection, leaving the higher K–j address lines for memory block access. In this situation, address lines from 0 to j-1 (A_{j-1} to A_0) are kept for the memory block access and address lines from j to K–1 (A_{K-1} to A_j) are kept for memory block selection.

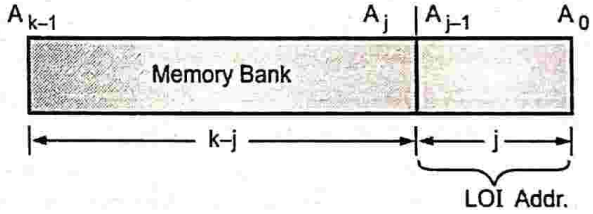


Fig. 10.12.3 : Lower Order Interleaved (LOI) Memory

- LOI system divides the memory into equal sized blocks with each block having the non-consecutive address range i.e. the address in consecutive blocks are consecutive but addresses in the same block are interleaved. This method is suitable for dividing the memory into memory banks required to be used for byte-addressable systems having data bus width of 16 or 32 or 64 bits.
- E.g. Let us consider a one-bit LOI memory system for a CPU having 20 bit address bus (such as **8086**). Such systems provide one bit for LOI from the LSB side (A_0) and the memory bank is addressed by the remaining 19 bits on the higher side of the address bus ($A_{19} - A_1$).
- Therefore, the system divides the total of 1 MB (1024 KB) memory address space into two memory banks of 512 KB sized called as EVEN and ODD bank. They have non-consecutive address range with all odd addresses in Bank 1 and all even addresses in Bank 0. The banks are selected with the help of decoder using the LOI address line, A_0.

Module 5

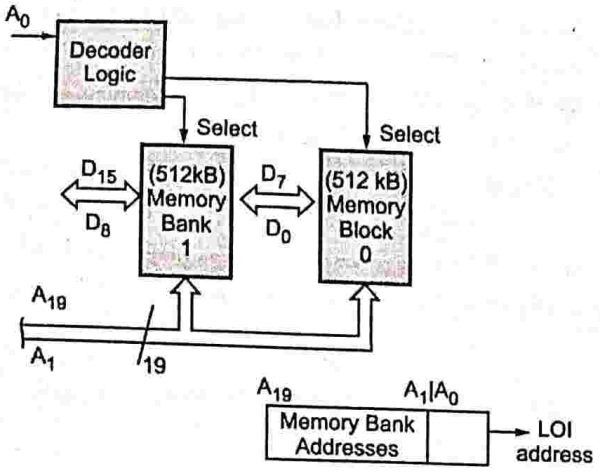


Fig. 10.12.4 : Odd and Even Banked LOI memory

10.13 ASSOCIATIVE MEMORY

UQ. 10.13.1 What is Associative memory ?
MU - Q. 1(e), May 15, 4 Marks

- An associative memory is the memory unit or device whose stored data can be identified for access by the content of the data itself rather than by providing any address or specifications of the memory location. Associative memory is also referred to as **Content Addressable Memory (CAM).**
- When a write operation is performed on this associative memory, no address or specifications of the memory location is given to the word. The memory itself can find the location to store the word.
- For this purpose, k bits of from the contents of the data to be written are taken as Argument and loaded in the Argument register which is used as Key of the finding one of the m locations of the memory unit and it is referred by Key register.
- As m locations are addressed by k bits, they are related as $m = 2^k$. Now as one of the possible m locations is chosen, remaining portion of the data (n bits) is written into the location, the data is transferred through the Map register.
- Therefore, total data width is k + n bits in size of which k bits are used for contents addressability to select one of $m = 2^k$ location and remaining n bits are actual stored in the addressed location.
- On the other hand, when the word is to be read from an associative memory, the reverse process is carried out. The contents addressable portion of k bits is supplied and memory accesses that specific location retrieving the n bits of data and total data output of k + n bits is the supplied. Fig. 10.13.1 shows the organization of the Associative memory.
- From the block diagram, we can say that an associative memory consists of a memory array and logic for 'm' words with 'n' bits per word.

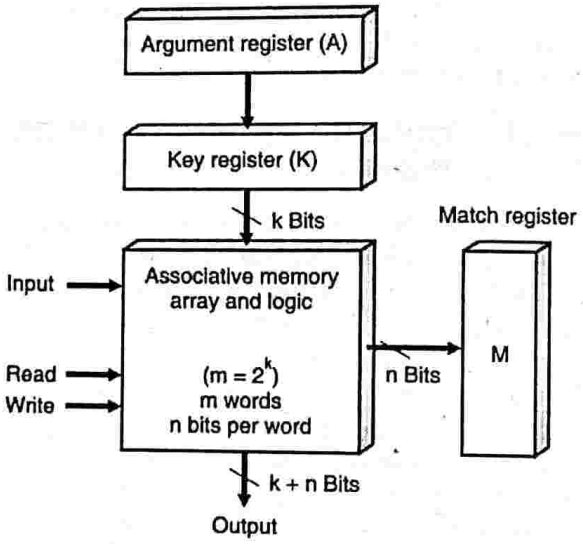


Fig. 10.13.1 : Associative Memory organization

...Chapter Ends

❑❑❑

CHAPTER 11

Principles of Advanced Processor and Buses

Module 6

University Prescribed Syllabus

Principles of Advanced Processor and Buses

6.1 Basic Pipelined Data path and control, data dependencies, data hazards, branch hazards, delayed branch, and branch prediction, Performance measures-CPI, Speedup, Efficiency, throughput, Amdhal's law.

6.2 Flynn's Classification, Introduction to multicore architecture.

6.3 Introduction to buses : ISA, PCI, USB. Bus Contention and Arbitration.

11.1 BASIC PIPELINED DATA PATH AND CONTROL

UQ. 11.1.1 Explain six stage instruction pipeline with suitable diagram.

MU - Q. 3(b), Dec. 15, Q. 5(a), Dec. 17, 10 Marks

Five stages in the MIPS pipelined architecture are shown in the Fig. 11.1.1 :

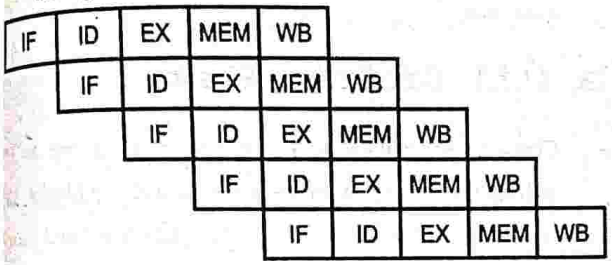


Fig. 11.1.1 : 5 Stage MIPS Pipeline

- The MIPS instruction set is highly regular, and all instructions are classified into R-type instructions, Load-Store Instructions and Branch instructions.
- R-Type instructions are used for the arithmetic and logical operations and are on R(Register) operands only.
- Load -Store instructions carry out memory references and
- Branch instructions facilitate program control transfers.

▶ Stage 1 - IF : Instruction Fetch

- In this stage for all three type of instructions, R-type instructions, Load-Store Instructions and Branch instructions, the memory location instruction contents pointed by PC are received over the external buses and loaded into the instruction register (IR). PC is internally incremented by 4 to point to the next instruction. (As all instructions are 32 bits or 4 bytes wide). The register transfer representation of Instruction Fetch stage is given as :

$$IR \leftarrow \text{Instr-Mem}[PC]$$
$$PC \leftarrow PC + 4$$

▶ Stage 2 - ID : Instruction Decode

- This stage is used for decoding the instruction and fetching the register operands.

R-type instructions

Register A ← Register [IR[25:21]]

Register B ← Register [IR[20:16]]

- Register A and B now contain 2 source data operands received from MIPS register.

Load-Store Instructions

Register A ← Register [IR[25:21]]

Register B ← Register [IR[20:16]]

- Register A and B store the component needed for generating the address for the memory reference as per the memory addressing.

Branch instructions

Register A ← Register [IR[25:21]]

Register B ← Register [IR[20:16]]

- Register operand contents are loaded in to register A for computing the target (the address to which the branch would be taken) branch address of the Instruction.

▶ Stage 3 - EX : Execute Stage

- In this stage the instructions are executed.

R-type instructions

$$ALU_{OUT} \leftarrow \text{Register A } (ALU_{Function}) \text{ Register B}$$

- The specified arithmetic or logical operation is performed on the source data operands contained in register A and B, and the result of operation is ready at the ALU output.

Load-Store Instructions

$$ALU_{OUT} \leftarrow \text{Register A} + \text{SignExtend}[IR[15:0]]$$

- The 16 bits value is sign extended and added to the contents of Register A and the result is ready at the ALU output. This is the physical address generated for the memory reference.

Branch instructions

$$\text{Target-Address} \leftarrow PC + \text{Sign-Extend}[R[15\text{-}0]] << 2$$

- Target address for the Branch is computed as the signed relative address (on account of the sign extension) added to the existing contents of PC. Therefore, MIPS branches are relative branches.

▶ Stage 4 - MEM : Memory Access

- In this stage the memory references are carried out and the memory is accessed at the specified address.

R-type instructions

NOP

- No operation takes place in this stage.

Load-Store Instructions

Load Instructions

$$\text{Mem-Data} \leftarrow \text{Data-Mem}[ALU_{OUT}]$$

- The contents of the Data Memory location pointed by ALU Output is loaded into the MIPS processor as Mem-Data.

Store Instructions

$$\text{Data-Mem}[ALU_{OUT}] \leftarrow B$$

- The Data specified by B register is stored to the Data Memory location pointed by the ALU Output.

Module 6

Branch instructions

IF ALU(Z) : PC ← Target-Address

- If the ALU Zero output (Z) is asserted, indicating the Zero result or equality, then the Target address computed in the previous stage is written in the PC. Hence effectively, the branch or jump taken as next instruction would fetched from this changed PC address contents.

▶ **Stage 5 - WB : Write Back**

- In this stage the destination operands are written back.

R-type instructions

Register [IR[15:11]] ← ALU_{OUT}

- ALU result is stored to the register designated for the destination data operand.

Load-Store Instructions

Load Instructions

B ← Mem-Data

- The Mem-Data received is stored in the Register B. The Load from mem location transfer operation is completed

Store Instructions

NOP

- Nothing happens on the MIPS processor. However, the Data sent out by store operation gets actually written into the Data Memory at the specified location.

Branch instructions

NOP

- No operation is done for branch instruction in WB (Write-Back) stage.

11.1.1 Data Path for Pipeline

- Creating and building a single data path for the MIPS architecture, consist of the logic circuit building blocks and connecting them together into the complete functional block diagram. The data connections and multiplexers are added in the design where ever required.
- The single data path for MIPS architecture consists of Instruction fetching logic, ALU and Execution logic, Data Memory and Sign extend logic, discrete components like adders and multiplexers. It is as shown in the Fig. 11.1.2.

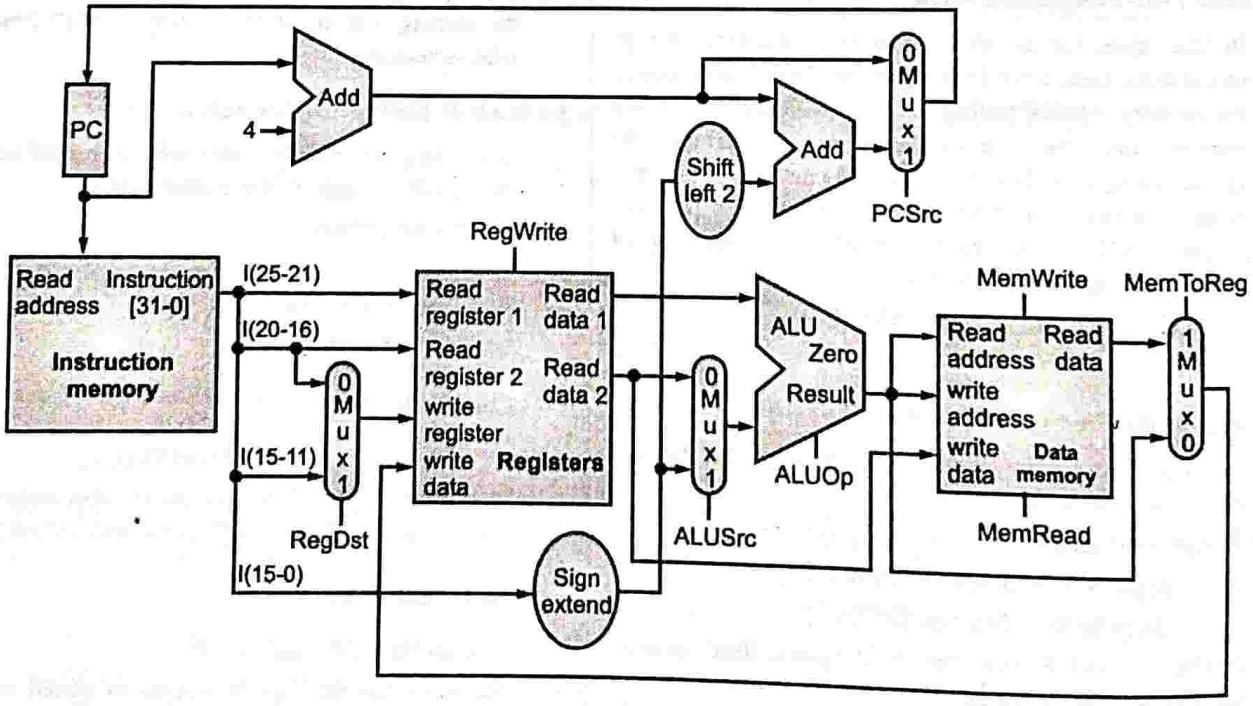


Fig. 11.1.2 : Single data path for MIPS architecture

The single data path diagram is explained by the following functional blocks :

- **Instruction fetching logic :** It consists of PC, Instruction memory and a simple adder. Contents of PC are supplied to the instruction memory. The instruction memory accesses the instruction that is to be executed. It generates the control signals for the source and destination register operands, ALU operation function, sign extensions, Read and write signals for the data memory.
- **ALU and Execution Logic :** It consists of 32 General purpose registers of 32-Bits and ALU. The register data path connects to the ALU. The two source register operands provide two inputs of the ALU used for ALU instructions (R-Type Instructions) and an optionally multiplexed input from Sign Extend logic used for branch instruction. The ALU output can be fed back to the destination register operand of the registers (for ALU instructions) or as physical address generated for the data memory(for Load Store instruction)
- **Data Memory and Sign Extend Logic :** It consists of the Data memory and logic used for sign extension. The data memory contains can be Read or written to and from registers in case of Load Store instructions for which additional multiplexer is used at the output of the data memory.
- **Adders and Multiplexers :** The adders are used for incrementing the PC contains by 4 (for every instruction) and for generating the branch target address (for branch instructions). Multiplexers are used.

11.1.2 Data Path with Controls

- The control logic depends on the details of the devices in the control path, and on the individual bits in the op code for the instructions. The arithmetic and logic operations for the R-type instructions also depend on the *funct* field of the instruction. The data path elements we have used are :
  - A 32-bit ALU with an output indicating if the result is zero.
  - Adders • MUX's (2 line to 1-line)
  - A 32 register × 32 bits/register - Register file
  - Individual 32-bit registers
  - A sign extender
  - Instruction Memory
  - Data Memory
- The Control signals and control logic is added to this Datapath so that the simple implementation of the MIPS processor is complete

ALU Controls

- For generating the ALU control signals, there are three control bits; the single bit *Binvert*, and the two-bit input to the MUX, labelled *Operation*. The ALU performs the operations and, or, add, and subtract. ALU Function Controls lines are as follows :

Table 11.1.1 : ALU function Control Lines

ALU Control Lines	ALU Function implemented
000	And
001	Or
010	Add
110	Subtract
111	Set on less than
1000	Nor

- The control design implementation of control logic is for implementing limited number of instructions. Other instructions can be added in the similar fashion. The *funct* field will also have to be decoded to produce the required control signals for the ALU.A separate decoder will be used for the main control signals and the ALU control. This approach is sometimes called local decoding. Its main advantage is in reducing the size of the main controller.

The Set of Control Signals

- The signals required to control the Datapath are the following :
  - **Jump** - set to 1 for a jump instruction.
  - **Branch** - set to 1 for a branch instruction.
  - **MemtoReg** - set to 1 for a load instruction.
  - **ALUSrc** - set to 0 for r-type instructions, and 1 for instructions using immediate data in the ALU (beq requires this set to 0).
  - **RegDst** - set to 1 for r-type instructions, and 0 for immediate instructions.
  - **MemRead** - set to 1 for a load instruction.
  - **MemWrite** - set to 1 for a store instruction.
  - **RegWrite** - set to 1 for any instruction writing to a register.
  - **ALUOp (k bits)** - encodes ALU operations except for r-type operations, which are encoded by the funct field.
- Additionally, the control signals are generated for the register selection encoding :
  - **rs1** – Source Register Operand 1 – Bits [25 – 21] of the Instruction.
  - **rs2** – Source Register Operand 2 – Bits [20 – 16] of the Instruction.
  - **rd** – Destination Register Operand – Bits [15 – 11] of the Instruction.
- The control signals are generated from the opcode fields of instruction Bits [5 – 0] and instruction Bits [31 – 26] to decode the necessary control signals required for the implementation. The Fig. 11.1.3 shows the generation of control signals.

Module 6

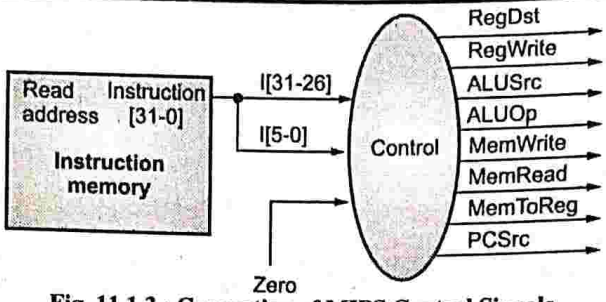


Fig. 11.1.3 : Generation of MIPS Control Signals

- The control Signal Table for simple Arithmetic, Logical, Load & Store and Branch instructions is as per the Table 11.1.2.

Table 11.1.2 : Control signal generation for sample set of MIPS instructions

	Control Signals						
Operation	**RegDst**	**RegWrite**	**ALUSrc**	**ALUOp**	**MemWrite**	**MemRead**	**MemToReg**
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	x	0	1	010	1	0	x
beq	x	0	0	110	0	0	x

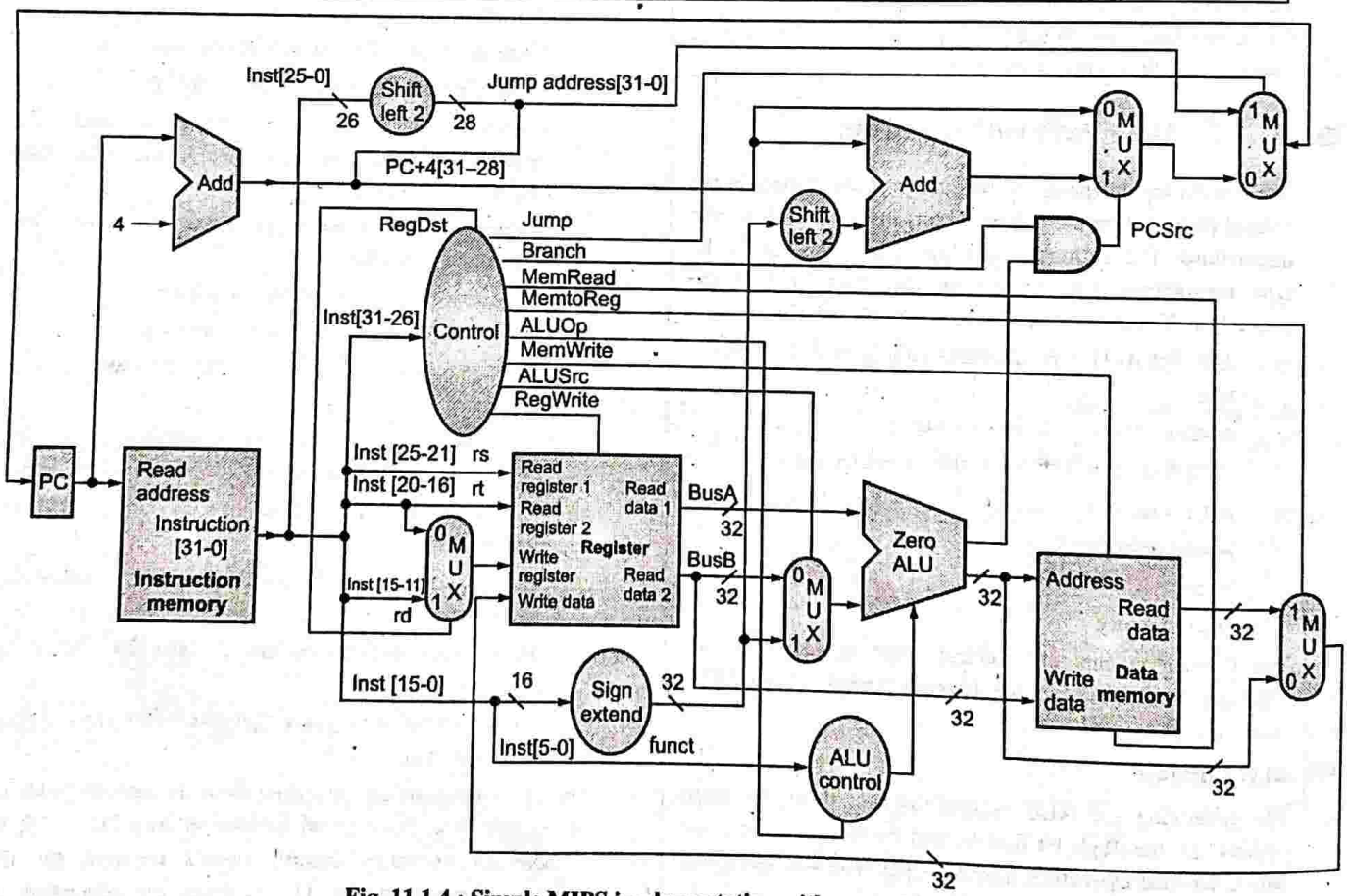


Fig. 11.1.4 : Simple MIPS implementation with control signals

For the consideration of control signal implementation, sample subset of MIPS instructions, is taken.

The instruction meaning are :

(i) **add** and **sub** are Arithmetic Instructions - They fall under the category of R-Type instructions. They access Register Source 1 and 2 operands and Register destination operand as clearly seen from the Table 11.1.2. Also, *ALUOp* function 010 and 110 are selected for add and subtract operations, respectively.

(ii) **and, or** and **slt** are Logical Instructions - They fall under the category of R-Type instructions. They access Register Source 1 and 2 operands and Register destination operand as clearly seen from the Table 11.1.2. Also, *ALUOp* function 000,001 and 111 respectively select and, or and set lest than (slt) ALU operations.

(iii) **lw** and **sw** are Load-Store Instructions - They need register as well as memory access signals as seen from the Table 11.1.2. lw loads the Data memory contents in to the Register operand and sw stores the Register operand contents to Data memory.

(iv) **beq** instruction is a Branch type instruction - It means branches if equal. It is not needing any register or memory access and so no control signal is needed. The Table 11.1.2 clearly indicates this set.

- Implementing the above - said control signals on the Data path designed in the previous section and showing the respective connection controls for Register access, Data memory access and ALU functions are as shown in the Fig. 11.1.4.

11.2 INSTRUCTION PIPELINE

UQ. 11.2.1 What is instruction pipelining?

MU - Q. 5(c), May 14, Q. 5(a), May 15, 6 Marks, Q. 6(b), May 18, 10 Marks

- Instruction cycle of the processor or CPU consists of many functions and is operationally divided into its functional units. A single instruction in the processor life cycle at a time is an inefficient process.
- When a specific instruction completes its work on the specific functional unit and goes to the next functional unit then the earlier functional unit is free, and it can process subsequently next instruction in the same time frame.
- Therefore, at any given instance, different instructions are in the different stages of advancement at the same time allows a greater number of instructions to be executed in the same time. This feature is called as instruction Pipeline.
- Consider a case of a non-pipelined processor as against a pipelined processor as shown in the Fig. 11.2.1. We consider two functional operations, fetching of the instructions (F) and execution of the instruction (E).
- In a non-pipelined processor case, it is seen that instructions require two time slots each two complete its fetching and execution.
- On the other hand, in the case of pipelined processor, it is possible to fetch the next instruction at the same time while the previous instruction is being executed.
- This functional overlap allows a greater number of instructions to be executed in the same time frame (the number of instructions executed would be one less than double).

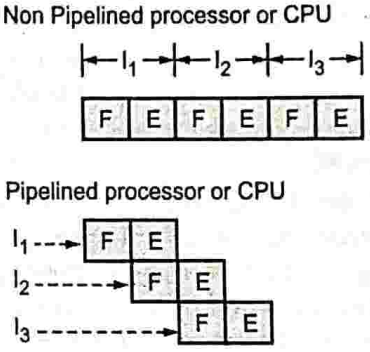


Fig. 11.2.1 : Pipeline Hazards

- To increase the performance of the processor instructions are simultaneously fetched and executed this technique of fetching while executing is called as instruction pipelining.

11.2.1 Pipelining Strategy

- Every instruction in processor is divided into smaller task, some task in processor are independent to each other.
- For e.g. If the processor is performing arithmetic operation with register operands, then the buses are free to fetch next instruction. Therefore, the task of execution and fetching can be performed simultaneously.
- This will reduce the waiting time also two tasks are completely in the same clock cycle which will reduce the required time to execute the next instruction.
- This technique of processing simultaneously is called as pipelining.
- Let us consider a two-stage pipeline in which there is a fetch cycle and execute cycle. There are many instances when during the execution cycle when memory is not being accessed the buses can perform fetch cycle.
- Therefore, the execution cycle and fetch cycle work simultaneously.

Module 6

11.2.2 Pipelined Performance

GQ. 11.2.2 Explain the Performance metrics for Instruction pipelines.

- As explained above, the instruction pipeline improves the performance of the system as it allows the simultaneous execution of instruction and thereby increasing the number of instructions executed per unit time as against any non-pipelined processor.
- The performance characterises of pipeline are determined by different matrix. Consider a generic instruction pipeline that is having k stages in the pipeline. Let this pipeline be subjected to execution of a sequence of n instructions then, we can deduce that
- A non-pipelined processor would require k clock periods each to execute one entire instruction and therefore to execute a sequence of n instructions it would require $k \times n$ number of clock periods.
- An equivalent k stage pipelined processor would require k clock periods to execute first instruction however the remaining n – 1instructions would require only one clock period each to execute. Therefore, total number of clock period required to execute are $k + n - 1$ which is much smaller as compared to $(k \times n)$. This clearly indicates that the pipeline performance will be improved.

Speed Up (s)

- It is defined as the ratio of the amount of time taken by a non-pipelined processor to the amount of time taken by an equivalent k stage pipelined processor evaluated over the execution of program sequence of n instructions.
- Now we have time taken by non-pipeline processor to be $(k \times n)$ and time taken by k stage pipeline processor to be $(k + n - 1)$. Therefore, Speed Up

$$s = \frac{k \times n}{k + n - 1}$$

- Speed up is an important metric for evaluating of performance of instruction pipelines.
- If n >>k then quantities k and 1 becomes insignificant as compared to n and therefore $s \cong \frac{k \times n}{n} = k$.
- Therefore, a k stage pipeline in its best case provides a speed up by a factor of k executing the programs nearly k times faster.

Clocks per instruction (CPI)

- CPI is another important performance matrix of pipeline. It gives an indication that how many clock periods are needed to complete an instruction on an average.
- It is defined as the number of clock period required per instruction on an average taken over the sequence of n instructions in an instruction pipelined processor.
- We know that, to complete the execution of n instruction, a k stage pipelined processor requires $(k + n - 1)$ clock periods. Therefore

$$CPI = \frac{k + n - 1}{n}$$

- If n >>k then quantities k and 1 becomes insignificant as compared to n and therefore. Therefore,

$$CPI \cong \frac{n}{n} = 1$$

- This indicates that for a pipelined processor CPI value is near unity i.e. one instruction completes execution every clock period.

Throughput : Instructions Per Second (IPS) and Million Instructions Per Seconds (MIPS)

- Number of instructions completed in one second provides another good matrix for evaluating the performance of processors implementing the instruction pipelines.
- Since most of the processor operates in frequency range above megahertz, a more convenient unit is the million instructions executed in one second (MIPS). A higher value of MIPS indicates a better performing processor.

Latency

- The latency is time delay. In the context of Instruction pipeline, latency is defined as the time delay between the time at which instruction is fetched and loaded in the processor or CPU and the time at which the same specific instruction completes its execution on the processor. All latencies are measured in Sec. (or more appropriately in Micro-seconds or Nano-seconds).
- Issue Latency is the latency or time delay in issuing the instruction for execution.
- Execution Latency is the latency or time delay in completing the execution of an instruction over the processor.

11.2.3 Pipeline Hazards

UQ. 11.2.3 Explain different type of pipeline hazards.

MU - Q. 1(a), Dec. 14, 3 Marks, Q. 1(a), May 16, Q. 1(b), Dec. 16, 5 Marks, Q. 6(a), May 17, Q. 6(a), Dec. 17, Q. 6(b), May 18, Q. 6(A), May 19, 10 Marks

- It assumes the pipeline is running absolutely smooth and every instruction requires exactly same time in each functional stage of the pipeline and therefore after the first instruction, every instruction is completing its execution in exactly one clock period.

- However, in the practical situation, it is not the case. There are many resource bottle necks and operational difficulties that doesn't allow the pipeline to operate in ideal condition.
- All such practical issues and resource bottle necks that make the instruction pipeline to deviate from its ideal characteristics are called as Hazards.
- Hazards introduce inefficiency and therefore increase the execution time of instructions in certain cases. Therefore, hazards in general degrade the instruction pipeline performance.
- The hazards can be classified into three types as follows :

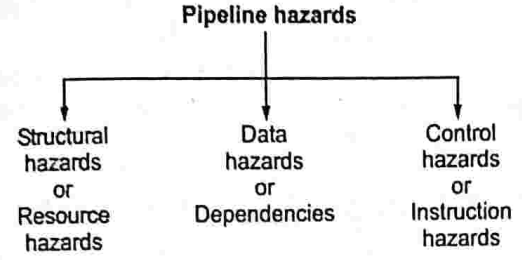


Fig. 11.2.2 : Pipeline hazards

11.2.3(A) Structural Hazards (or Resource Hazards)

- Structural Hazards or resource hazards are the hazards introduced in an instruction pipeline by virtue of structural problems or resource bottle necks.

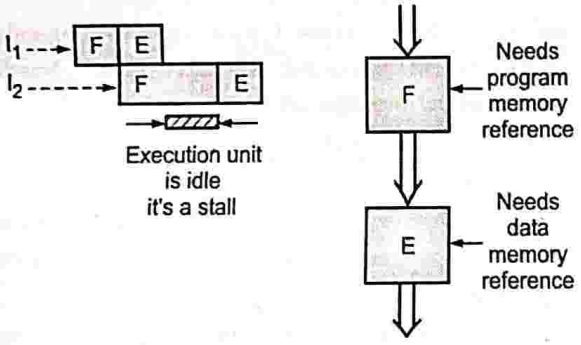


Fig. 11.2.3 : Structural or resource hazards

- Consider a situation where the fetch and execute stages of the pipeline are of dissimilar durations.
- As shown in the Fig. 11.2.3, as the instruction I_2 fetch stage requires more time to complete as compared to the execute stage of Instruction I_1, the execution unit is ideal as it has already completed execution of instruction I_1, but cannot execute instruction I_2 which is not yet being completely fetched.
- Such situation is called as pipeline stall which pushes the timings forward making instructions to take longer times to execute and therefore degrading the performance.
- Alternately, if the current instruction that is under execution requires a memory reference (e.g. MOV AX, [BX]) and at the same time fetch stage also requires access to the memory for fetching the next instruction in the program.
- In this case both fetch stage and execute stage are requiring the memory reference over the busses.
- However, the busses can be given to only one of them at a time.
- This forces the two operations to be performed one after the other due to the resource bottle on the busses.
- The result is that the instruction execution takes longer time and pipeline performance is degraded

Solutions to the Structural Hazards (or Resource Hazards)

- Make the pipeline deeper i.e. the pipeline should have a greater number of stages. This reduces the probability of resource bottleneck and the pipeline stalls and therefore reducing the negative impact of structural hazards.
- To have bigger register file inside the CPU. This makes the number of memory references to reduce intern reducing the resource bottle neck on the buses and improving the pipeline performance.
- To have dual cache. A dual cache implementation at level 1 splits the cache memory into a separate data cache and instruction cache. The execution stage memory references access data and there they are routed to the data cache.
- On the other hand, pre-fetch or fetch stage memory references always requires instructions and therefore they are routed to the instruction cache.
- This drastically reduces resource the bottle neck on the system bus and therefore structural hazards are also reduced.

11.2.3(B) Data Hazards (or Dependencies)

UQ. 11.2.4 *List and explain various data dependencies, data and branch hazards that occur in the computer system.*

MU - Q. 2(B), Dec. 18, 10 Marks

- Data Hazards or Dependencies are introduced into instruction pipeline due to the consecutive instruction beings dependent.
- The two consecutive instructions are said to be dependent if at least one of the source or destination operand referred in these two instructions are common. There are three types of dependencies observed in data hazards.

Module 6

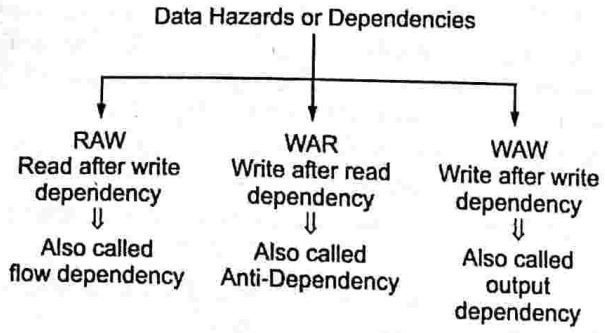


Fig. 11.2.4 : Data hazards or dependencies

☞ Read after Write (RAW) dependency or flow dependency

- In this type of data dependency hazard, the destination operand of the previous instruction is one of the source operands in the subsequent instruction.
- Therefore, it is termed as read after write dependency. This type of dependency always leads to the data hazard in instruction pipeline.
- This happens because the subsequent instruction cannot fetch its source operands unless the previous instruction has stored its destination operand and generally results in one clock period stall as a subsequent instruction execution waits.

e.g. I_1: MOV BX, DX
I_2: ADD AX, BX

☞ Write after Read (WAR) dependency or Anti-Dependency

- In this type of data dependency hazard, the one of the source operands of the previous instruction is the destination operand in the subsequent instruction. Therefore, it is termed as Write after Read (WAR) dependency.
- This type of dependency does not lead to the data hazard in instruction pipeline. Therefore, usually WAR dependency is not needed to be treated further.

e.g. I_1: MOV DX, BX
I_2: ADD BX, AX

☞ Write after Write (WAW) dependency or Output Dependency

- In this type of data dependency hazard, the destination operands of the previous instruction are same as destination operand in the subsequent instruction. Therefore, it is termed as Write after Write dependency.
- This type of dependency usually does not lead to the data hazard for single scalar instruction pipeline. However, it may lead to a data hazard in superscalar execution as multiple instructions are executed simultaneously.

e.g. I_1: MOV DX, BX
I_2: ADD DX, AX

☞ Solution to Data Hazards

- The instruction pipeline stalls and resultant clock period delays generated due to data dependencies are resolved by a technique that is called as data forwarding or operand forwarding.
- In this technique, output of the execute stage is fed back at the input of the execute stage. Therefore, the destination operand of the previous instruction is made instantly available as the source operand for the subsequent instruction even before it gets actually written into the destination register or location.
- Therefore, the delay caused due to waiting for the destination operand of previous instruction is avoided.
- As a result of this, there is no stall observed on the instruction pipeline and the negative impact of the data hazard (RAW or WAW) is minimized.

11.2.3(C) Control Hazards (Instruction Hazards)-Dealing with Branches

GQ. 11.2.5 List and explain various ways in which an instruction pipeline can deal with conditional branch instructions.

- Control Hazards or Instruction Hazards are caused due to the control transfers encountered in the user programs or due to the system activity or events. Control transfers are generally caused by the conditional and unconditional branch instructions and subroutine calls in the user programs.
- They can also be caused by system events such as interrupts, exceptions and task switches.
- When the control transfer, say typically a branch instruction appears in the user program, a wide-ranging instruction pipeline stall takes place as shown in the Fig. 11.2.5.
- The branch instruction comes for execution, next few instructions are already in the pipe line but, since the next instruction is to be executed from the branch address (target address), all the instructions in the pipeline are flushed. Similarly, since the pipeline is now built afresh, some clock periods are wasted till such time the pipeline is fully built again (right hand side triangle of the Fig. 11.2.5).
- Therefore, in total heavy penalty in terms of pipeline stalls and delay is required to be paid.
- Control hazards are the hazards that contribute maximum to the pipeline stalling and degradation of the system performance.

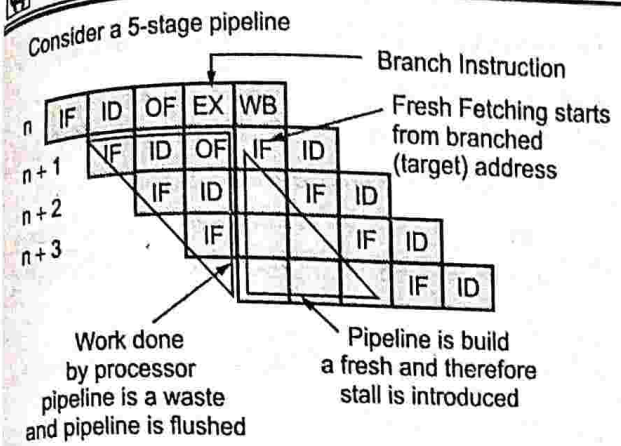


Fig. 11.2.5 : Control or instruction Hazards

Solutions to control hazards

- **Branch Predictions :** The best way to mitigate the control hazard is to predict in advance the branches that are occurring in the user program. Unconditional branches can be predicted accurately as the branch is always taken.
- This allows the branch prediction logic to start fetching subsequent addresses form the branched location (target address). As the branches are always predicted correctly, the penalty in terms of stalls is nearly reduced to zero.
- The more challenging task is to predict conditional branches.
- The conditional branch prediction may not always be correct as the condition code evaluated while executing the instruction may or may not take the branch and knowing the condition code in advance while decoding the branch instruction is not possible.
- There are two different techniques of branch prediction :

  (i) Static branch prediction logic : In this logic, the branch predictions are performed by a fixed static rule which predicts all unconditional branches to be taken, all forward branches (where the target address is bigger than the source address where branch instruction appears) are predicted not to be taken and all backward branches (target address smaller than source address) are predicted to be taken. This branch prediction logic is good enough but not very efficient

  (ii) Dynamic branch prediction logic : This is more complex logic where branch predictions are made dynamically on the basis of branch history.

  For this purpose, the CPU maintains a branch history table (BHT) or a Branch Target Buffer (BTB).

  The branch target buffer maintains the branch history for the most recently used branches. Based on the status of branch history, the dynamic branch prediction is carried out.

- **Delayed Branch Technique :** In this technique, one of the instructions before the branch instruction is shifted to a slot immediately subsequent to the branch instruction.
- Now, when the conditional branch is taken, the branching is affected delayed making this specific instruction subsequent to the branch instruction to be executed in any case before actually branching to the target address.
- Therefore, utilizing some time out of the time wasted in the pipeline stall due to pipeline flushing. This instruction which is shifted to the delayed branch slot must be an instruction that is an independent instruction that does not affect the outcome of the branch directly or indirectly.

UEx. 11.2.1

MU - Q. 2(a), Dec. 18, Q. 2(B), May 19, 10 Marks

A program having 10instructions (without Branch and Call instructions) is executed on non-pipeline and pipe line processors. All instructions are of same length and having 4pipeline stages and time required to each stage is 1 nsec.

i) Calculate time required to execute the program on Non-pipeline and Pipeline processor.

ii) Calculate Speedup

Soln. :

The number of instructions executed are 10.

Therefore n = 10

For pipeline stages are given as 4. Therefore k = 4

Now the time required for Non-pipelined processor is given as

$$S_{np} = n * k = 10 * 4 = 40 \text{ nS}$$

And time required for Pipelined Processor (4 Stage) is given as

$$S_p = n + k - 1 = 10 + 4 - 1 = 13 \text{ nS}$$

Further the Speed-up is given as Ratio of times for Non-Pipelined and Pipelined processors –

$$S = (n*k) / (n + k - 1) = S_{np} / S_p = 40/13$$

Therefore, Speedup S = 3.08

11.3 BRANCH PREDICTIONS

UQ. 11.3.1 *Explain Branch Prediction Logic and delayed branch.*

MU - Q. 2(a), Dec. 18, Q. 2(A), May 19, 10 Marks

- Branch Predictions are the best way to mitigate the control hazard is to predict in advance the branches that are occurring in the user program.
- Unconditional branches can be predicted accurately as the branch is always taken.

Module 6

- This allows the branch prediction logic to start fetching subsequent addresses form the branched location (target address).
- As the branches are always predicted correctly, the penalty in terms of stalls is nearly reduced to zero.
- The more challenging task is to predict conditional branches.
- The conditional branch prediction may not always be correct as the condition code evaluated while executing the instruction may or may not take the branch and knowing the condition code in advance while decoding the branch instruction is not possible.
- There are two different techniques of branch prediction.

☞ Static branch prediction logic

In this logic, the branch predictions are performed by a fixed static rule which predicts all unconditional branches to be taken, all forward branches (where the target address is bigger than the source address where branch instruction appears) are predicted not to be taken and all backward branches (target address smaller than source address) are predicted to be taken. This branch prediction logic is good enough but not very efficient.

☞ Dynamic branch prediction logic

- This is more complex logic where branch predictions are made dynamically on the basis of branch history.
- For this purpose, the CPU maintains a branch history table (BHT) or a Branch Target Buffer (BTB).
- The branch target buffer is a caching address buffer maintains the branch history for the most recently used branches. For this purpose, it stores the Branch source addresses (address at which the branch instruction appears), target addresses (address to which the branch is to be taken) and history state bits which keep track of what happened to that specific branch in the past.
- Based on the status of branch history, the dynamic branch prediction is carried out.

Source Address	Target Address	State Bits
..	..	..
..	..	..
..	..	..
..	..	..

Fig. 11.3.1 : Branch Target Buffer (BTB)

- Let us consider the Dynamic branch prediction logic of Pentium processor from Intel :
  - Branch Target Buffer (BTB) is used in Pentium Architecture to store the addresses (Source as well as Target) of the most frequently and recently Branch statements and refer to these addresses when branch instruction is encountered.
  - Source Address – The address at which the branch Instruction appears.
  - Target address – The address to which the branch is made.
  - State Bits – 2 Bit Branch State field 00,01,10,11(asST, WT, WNT and SNT)
  - Pentium Maintains 2 Pre-fetch buffers of 2 Cache Lines (32 Bytes X 2) i.e. Having total of 128 Bytes of Pre-fetch Buffer Memory.
  - If the Branch is predicted not to be taken, then The Pre-fetching continues in Current Buffer sequentially.
  - If the Branch is predicted to be taken, then The Pre-fetch buffer is switched and Pre-fetching is carried out from branch target address in the switched Buffer.
  - If the branch prediction is correct, the instruction pipeline continues smoothly.
  - If the branch mis-prediction occurs, then instruction pipeline stalls for 3 Clocks in U-Pipeline / 4 Clocks in V-Pipeline.

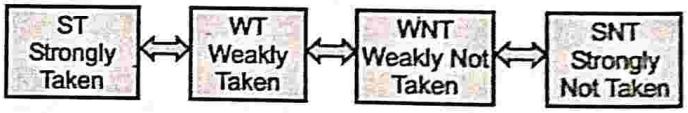


Fig. 11.3.2 : Dynamic Branch Prediction logic

- Prediction Logic (Applied at D1 Pipeline stage)
- Branch is Predicted to be Taken if State is ST or WT
- Branch is Predicted Not to be Taken if state is WNT or SNT
- State Change (Applied at EX Pipeline stage)
- Initially, ST state is always allocated for each branch
- State is altered rightward if the Branch is Actually not Taken.
- State is altered leftward if the Branch is Actually Taken.

11.4 COMPUTER PERFORMANCE MEASUREMENT - BENCHMARKS (SPEC)FOR EVALUATION

UQ. 11.4.1 Write short note on Performance measures.

MU - Q. 6(a), Dec. 18, 10 Marks

- Computers today are extensively used in many complex applications. This demands that the computers must be capable of processing large amount of data in minimal time i.e. better performance.
- The above requirement imposes redesigning of computing machines. But exactly which parameters to be redesigned is a big question.
- In this section we highlight certain factors to design for performance.

11.4.1 Metrics such as CPU Time, Throughput, etc.

- The speed of the processor can be increased by improving the mechanism in which the data and instructions are fed to it.
- There are various architectural improvements reported which has increased the performance of processors significantly. Following are few methods:

1. Pipelining
2. Branch Prediction
3. Superscalar execution
4. Speculative execution

1. Pipelining

- An instruction is executed in multiple phases of operations for e.g. fetching the instruction, decoding the instruction, fetching the data operands, data processing, etc.
- Using pipelining the processor can work with multiple instructions at different phases. In this way the multiple instructions are executed simultaneously in the same clock cycle.
- Therefore the overall speed of the processor seems to be increased.

2. Branch Prediction

- A branching instruction can cause complete fall of pipelining technique.
- To counteract this problem designers of processors have incorporated a hardware mechanism to predict the branch instruction and its target address well in advance so that the set of new instructions can be prefetched into pipeline.
- Therefore branch prediction avoids the fall of pipeline and the processor always works improving the performance.

3. Superscalar execution

- The performance of the processor can be further improved by increasing the execution of multiple instructions in every clock cycle.
- This can be achieved by incorporating multiple parallel pipelines.

4. Speculative execution

- In this process the processor alters the flow of program speculatively.
- Therefore now the processor can work on multiple instructions which are independent to each other.

11.4.2 Aspects and Factors Affecting Computer Performance

- Even though the speed of the processor is increased exponentially but adjoining components like memory and input output devices have not kept with speed therefore there is a need to look into performance balance i.e. tuning the architectural structure of computer to compensate the mismatch.
- Let us first consider the mismatch at memory-processor interface. This interface affects the speed of processor drastically as it fetches data and instructions from memory which is comparably slow and hence the operation is lagged. Following are few ways to counteract :
  - (i) Increasing the bus width.
  - (ii) More level intermediate faster memory can be implemented between processor and main memory (Cache concept).
  - (iii) Reducing the access to the main memory.
  - (iv) Increase the interconnect bandwidth.
- The second mismatch occurs at processor – Input/Output (I/O) interface, as most of the data is moved in and out of peripherals. The strategies to overcome this problem are :
  - (i) Using cache and buffering mechanism.
  - (ii) Higher speed interconnection buses.

Module 6

11.4.3 Comparing Computer Performances

GQ. 11.4.2 State the evolution of processors.

Processor	Year	Addressable memory	Bus Width	Clock Speed	Cache Memory
4004	1971	640Bytes	4 bits	108kHz	--
8008	1972	16KB	8 bits	108 kHz	--
8080	1974	64KB	8 bits	2 MHz	--
8086	1978	1MB	16 bits	5MHz,8MHz,10 MHz	--
8088	1979	1MB	8 bits	5 MHz, 8MHz	--
80286	1982	1GB	16 bits	6-12.5MHz	--
80386DX	1985	4GB	32 bits	16-33MHz	--
80486DX	1989	4GB	32 bits	25-50MHz	--
Pentium	1993	4GB	64 bits	60-166MHz	8KB
Pentium Pro	1995	64GB	64 bits	150-200MHz	512KB L1 and 1MB L2
Pentium-II	1997	64GB	64 bits	200-300MHz	512KB L2
Pentium-III	1999	64GB	64 bits	450-660MHz	512KB L2
Pentium 4	2000	64GB	64 bits	1.3-1.8GHz	256KB L2
Core-2 Duo	2006	64GB	64 bits	1.06-1.2GHz	2MB L2
Corei7	2013	64GB	64 bits	4GHz	1.5MB L2/1.5MB L3

11.4.4 Marketing Metrics - MIPS & MFLOPS

- **Base Metric:** These are required for all reported results and have strict guidelines for compilation. In essence, the standard compiler with more or less default settings should be used on each system under test to achieve comparable results.
- **Peak Metric :** This enables users to attempt to optimize system performance by optimizing the compiler output.
- **Speed Metric :** This is simply a measurement of the time it takes to execute a compiled benchmark. The speed metric is used for comparing the ability of a computer to complete single tasks.
- **Rate metric :** This is a measurement of how many tasks a computer can accomplish in a certain amount of time; this is also called as throughput, capacity or rate measure.
- Commercially, following metrics are used in the Market to benchmark and compare the performances of the computer systems –

Million Instruction Per Second (MIPS)

It a performance parameter for the processor which indicates the rate at which instructions are executed. MIPS is defined as the number of Million Instructions executed Per Second. The MIPS rate is computed by the formula :

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

Millions (of) floating point operations per second (MFLOPS)

It is also a performance parameter of the processor which indicates the floating-point instructions execution rate. MFLOPS is defined as the number of Floating-Point Operations completed or executed Per Second, It is given as

$$\text{MIFLOPS rate} = \frac{\text{Number of exectured floating} - \text{point operations in a program}}{\text{Execution time} \times 10^6}$$

11.4.5 Speedup and Amdahl's Law

The speedup is a performance parameter of the computer system. To improve this system parameter various solution are proposed by system designers :

- Parallel Processing – By increasing the number of parallel processing element handling the execution of the task and trying to process or execute in parallel; the parallelable fraction of the program code.
- Cache Memory hierarchy – By implementing faster memories such that the average memory access cycle time is reduced, average wait states in the memory accesses are reduced and thereby increasing the system performance.
- Faster memory and I/O cycles – By introducing techniques where memory as well as I/O data transfers are performed in a faster way. The quicker is the access to these system components, faster the completion of their cycles and improvement in the performance is achieved.
- But increase in any of the above techniques does not increase the Speedup significantly. The limiting factor on speedup is given by Amdahl's Law.
- The Amdahl's law provides the speedup comparison of program running on multiprocessor versus single processor.
- Let us consider there are N parallel processors running on with complete parallel portion of program. Then the speedup formula is given as :

$$\text{Speedup} = \frac{\text{Time to execute program on a single processor}}{\text{Time to execute program on N parallel processors}}$$

$$\text{Speedup} = \frac{T(1-f)+Tf}{T(1-f)+g(Tf,N)}$$

Where

T is the total execution time of the program using single processor

f is the fraction of the code which is parallelizable with no scheduling

and (1-f) is the fraction of code which executes sequentially

- From above equation we can conclude that for exploiting the complete parallelism of processor, the program must have high parallel execution capabilities.

The overall speedup of the system is given as:

$$\text{Speedup} = \frac{1}{(1-f)+\frac{f}{SU_f}}$$

Where SU_f is the enhancement factor.

f is the fraction of the code which is parallelizable with no scheduling

and (1-f) is the fraction of code which executes sequentially.

11.5 FLYNN'S CLASSIFICATION

UQ. 11.5.1 *Write short note on Flynn's Classification.*

MU - Q. 6(c), May 18, 10 Marks

UQ. 11.5.2 *Explain Flynn's classification in detail.*

MU - Q. 1(f), May 14, 3 Marks, Q. 4(b), Dec. 15, Q. 2(a), May 16, 10 Marks, Q. 1(d), Dec. 16, 5 Marks, Q. 4(b), Dec. 17, 10 Marks

UQ. 11.5.3 *List the Flynn's classification of parallel processing systems.*

MU - Q. 1(c), May 15, 3 Marks

- The processor or CPU has primary functionality of processing the data by executing programs. Program is sequence of instructions. Both programs and data are stored in the memory. When the processor or CPU executes the program, it is needed to fetch instructions in the program from the memory. Once the instruction is fetched, it is decoded and executed in the processor or CPU. While executing it reads the input data operands from the memory and it stores output data operands to the memory.
- The flow or movement of instructions from memory to the CPU is called as **instruction stream.**
- The two-way flow of input and output data operands between the CPU and the memory is called as **data stream.**

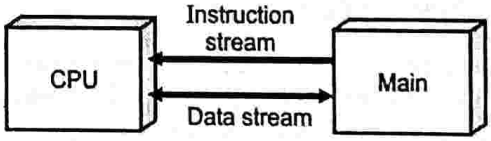


Fig. 11.5.1 : Instruction and Data stream

- Therefore, there are two types of streams observed: Instruction stream and Data stream as shown in the Fig. 11.5.1.
- Based on the multiplicity of the instruction and data streams, the computers are classified into four different types. This classification or taxonomy is called as **Flynn's Classification** or **Taxonomy**. The four types are :

(I) SISD : Single Instruction Stream, Single Data Stream

(II) SIMD : Single Instruction Stream, Multiple Data Stream

(III) MISD : Multiple Instruction Stream, Single Data Stream

(IV) MIMD : Multiple Instruction Stream, Multiple Data Stream

Module 6

(I) Single Instruction Stream, Single Data Stream (SISD)

In this organization, there is only one instruction stream and one data stream. The CPU can be visualized to be made up of two parts namely :

- **Data processing unit (DPU)** and **control unit (CU).** The DPU deals with data and consists of ALU and the data path. Therefore, DPU accesses the data stream from the memory bidirectionally.
- The control unit deals with the instruction stream as it receives instructions from the memory and decodes them. Subsequently, the CU gets these instructions executed from the DPU.
- The typical organization of SISD computer is as shown in the Fig. 11.5.2. It is clearly seen that it has only one data stream and one instruction stream. This is the most widely used computer type.

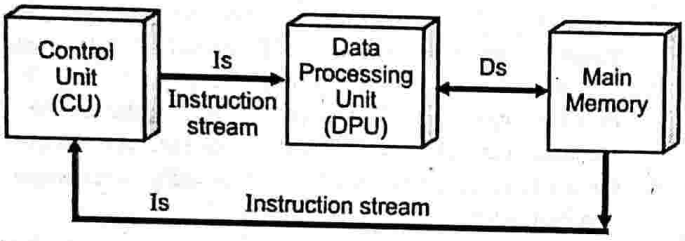


Fig. 11.5.2 : Single Instruction Stream, Single Data Stream (SISD)

(II) Single Instruction Stream, Multiple Data Stream (SIMD)

- In this type of organization of computer systems, there are multiple processing elements or Data processing units (DPUs) in the computer architecture.
- They are controlled by a single control unit.

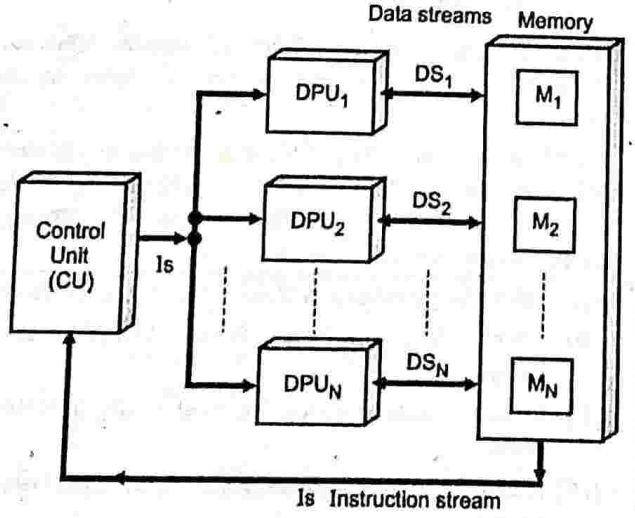


Fig. 11.5.3 : Single Instruction Stream, Multiple Data Stream (SIMD)

- The CU receives instructions from the memory over a single instruction stream and generates controls on the basis of decoded instruction to different DPU and each DPU processes different data element.
- The data element is accessed from the memory over different data streams.
- Therefore, SIMD processors are capable of executing the same instruction on multiple data elements simultaneously.
- The Fig. 11.5.3 shows organization of SIMD computers.
- These type of computer systems are widely used in vector processor and find their applications in multimedia and signal processing.

(III) Multiple Instruction Stream, Single Data Stream (MISD)

- In the organization of this type of computer system, there are multiple data processing units and multiple control units.
- The data processing units are arranged in a sequential manner in such a way that output of one DPU is input of the next DPU in order.
- Therefore, all DPUs together handle only one data element, which is accessed from the memory over single data stream.
- Each DPU receives controls from its independent CU and each CU processes it's different instruction. Each of these instructions is fetched from the memory over different instruction streams.
- Therefore, this type of processor processes the same data elements sequentially by multiple instructions. Organization of MISD is as shown in the Fig. 11.5.4.

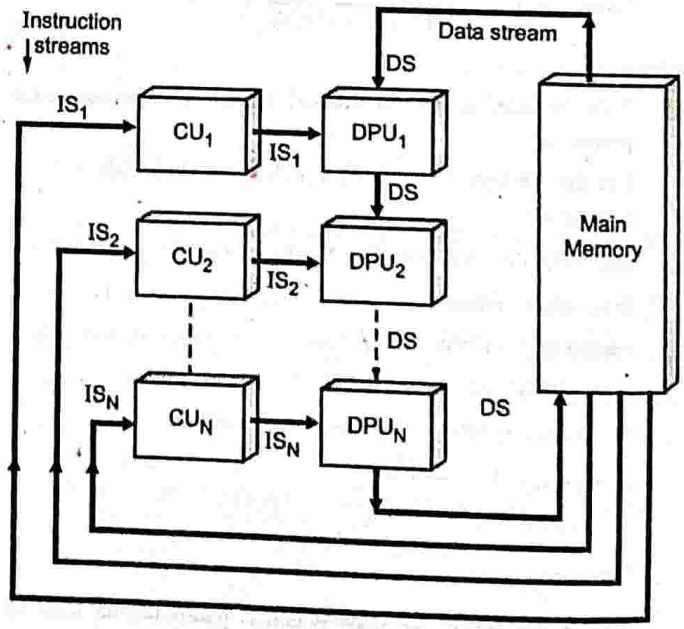


Fig. 11.5.4 : Multiple Instruction Stream, Single Data Stream (MISD)

- These type of computer systems are widely used in the data pipelined processors which are also called as Systolic arrays.

(IV) Multiple Instruction Stream, Multiple Data Stream (MIMD)

- In this type of computer systems there are multiple control units and multiple data processing units. Each DPU processes different data element using different data streams.
- Each DPU receives controls from its independent CU and each CU processes it's different instruction. Each of these instructions is fetched from the memory over different instruction streams.
- Therefore, these kinds of processors are capable of handling multiple independent instructions on multiple different data elements simultaneously.

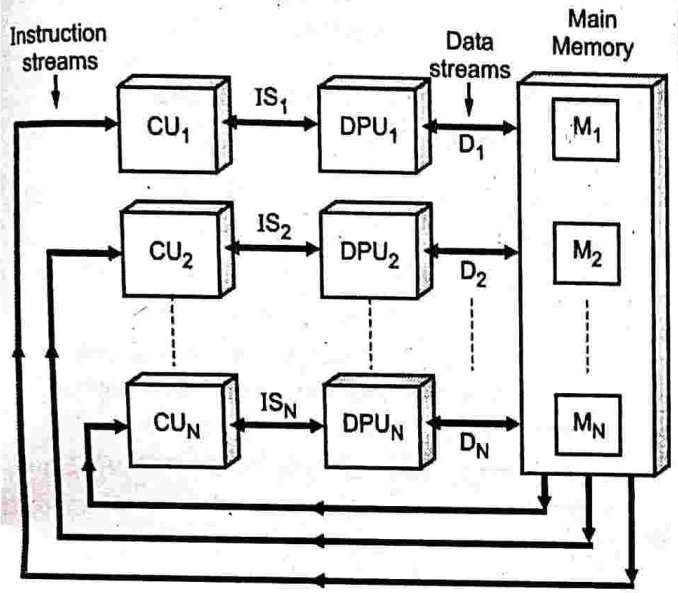


Fig. 11.5.5 : Multiple Instruction Stream, Multiple Data Stream (MIMD)

- These type of computer systems are widely used in supercomputers and certain advanced GPGPU (General Purpose Graphic Processing Unit)

11.6 INSTRUCTION PIPELINE

The finest level of parallelism or parallel processing is carried out by way of instruction pipelining. The pipelining concepts are further advanced by way of superscalar execution and out of order execution. Following sections discuss these parallel processing approaches.

11.6.1 Pipeline Concepts

- Instruction cycle of the processor or CPU consists of many functions and is operationally divided into its functional units. A single instruction in the processor life cycle at a time is an inefficient process.
- When a specific instruction completes its work on the specific functional unit and goes to the next functional unit then the earlier functional unit is free, and it can process subsequently next instruction in the same time frame.
- Therefore, at any given instance, different instructions are in the different stages of advancement at the same time allows a greater number of instructions to be executed in the same time. This feature is called as instruction Pipeline.
- Consider a case of a non-pipelined processor as against a pipelined processor as shown in the Fig. 11.6.1. We consider two functional operations, fetching of the instructions (F) and execution of the instruction (E).
- In a non-pipelined processor case, it is seen that instructions require two time slots each two complete its fetching and execution.
- On the other hand, in the case of pipelined processor, it is possible to fetch the next instruction at the same time while the previous instruction is being executed.
- This functional overlap allows a greater number of instructions to be executed in the same time frame (the number of instructions executed would be one less than double).

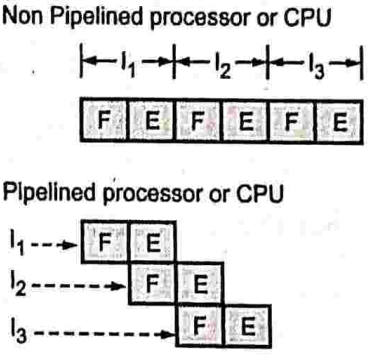


Fig. 11.6.1 : Instruction Pipeline

- To increase the performance of the processor instructions are simultaneously fetched and executed this technique of fetching while executing is called as instruction pipelining.

11.6.2 Pipeline Stages

On a processor, instructions are executed in a pipeline architecture. A typical instruction pipeline is a multistage pipeline having five functional stages. The pipeline stages are completed in single clock period in majority of the cases. If all pipeline stages complete in one clock, the pipeline would execute in a streamlined manner minimizing the CPU stalls and therefore maximizing the system performance for the given clock speed.

IF	ID	OF	EX	WB

Module 6

The five pipeline stages are :

- **Instruction Fetch (IF) :** In this stage the processor fetches instructions and loads them into an instruction register for decoding.
- **Instruction Decode (ID) :** In this stage the processor decodes the instructions and the control unit of the processor generates the necessary internal and external control signals for the controller.
- **Operand Fetch (OF) :** In this stage the processor fetches the data operands required for instruction execution from either the registers or memory locations.
- **Execute (EX) :** In this stage, the processor executes the instruction.
- **Write Back (WB) :** In this stage, the processor stores or writes back the results of the executed instruction to registers or memory locations.

11.7 CONCEPTS OF SUPERSCALAR ARCHITECTURE

UQ. 11.7.1 Explain Superscalar Architecture.

MU - Q. 5(a), May 18, Q. 1(e), Dec. 18, 10 Marks, Q. 1(e), May 19, 5 Marks

11.7.1 Superscalar Concept

- The instruction pipelines usually implemented are essentially single pipelines operating on single set of function units in the processor providing functional overlap.
- It means that multiple instructions are handled inside the processor or CPU in different functional stages. However, in each functional stage there is only one instruction.
- Meaning, suppose we consider Execute (EX) stage, at any instance, there is only one instruction executing on the processor.
- Such single set of resource is called as a single scalar pipeline. If we visualize that the processor possesses such multiple units of scalar pipelines, then such implementation is called as Superscalar execution of Instructions in the processor. Processing in such CPU is called as Superscalar processing.
- Meaning, suppose we consider Execute (EX) stage, processor provides sufficient resources to actually execute 2 or more instructions simultaneously.
- The Fig. 11.7.1 shows superscalar execution schematic, with 2-issue Super-Scalar pipelines A and B.

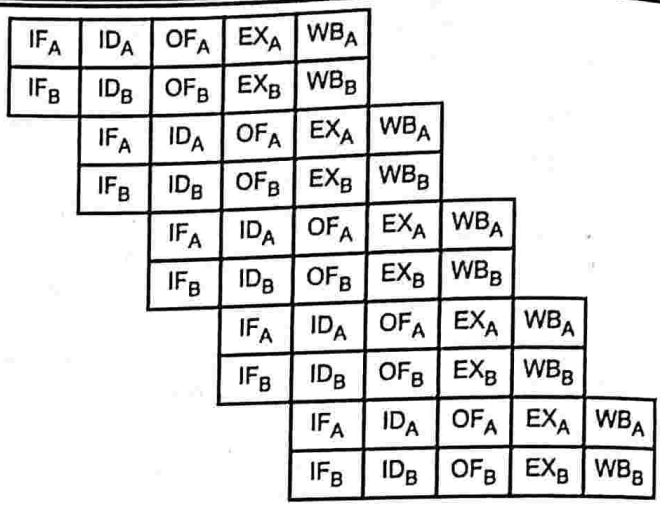


Fig. 11.7.1 : Superscalar Execution with 2 scalar Instruction pipelines

11.7.2 Superscalar Processors or Superscalar Architecture

- Superscalar processing or Superscalar execution is the multiple scalar pipelined execution where more than one scalar pipeline exist in the processor architecture, each scalar pipeline equivalent to a simple multi-stage pipeline.
- The processor that use and implement the necessary resources for such Superscalar execution are called as Superscalar processors.
- If a Superscalar processor is having 2 scalar pipelines, then it is called as a 2-issue Superscalar processor. Similarly 3-issue and 4-issue Superscalar processor possess 3 or 4 parallel scalar instruction pipelines. Such processors allow 2,3 or 4 instructions simultaneously. They provide higher order performances. Their speed-up values and MIPS value for instruction throughput are also much higher that the simple single multi-stage pipelined processors. Under ideal condition, a 2-issue superscalar processor provides CPI value of 0.5 and a 3-issue superscalar provides CPI value of 0.33.
- However, increased complexity of simultaneous instruction execution in Superscalar processing poses various design issues and operating constraints. These issues are needed to be tackled in order to make superscalar processing to run smoothly and provide the desired performance improvements.

11.8 OUT-OF-ORDER EXECUTION

To understand Out-of-Order (OoO) execution, it is required to understand the concept of Instruction scheduling in normal pipelined architecture that is not supporting OoO.

Instruction Scheduling

- In standard pipeline, the instructions are fetched, decoded, executed. The decode stage is the stage where it identifies the structural, data, and control hazards. The processor hardware architectural features try to minimize the impact of these hazards. The compiler can also make use of information available about the hardware to minimize the hazards by scheduling the dependent instructions away from each other. Also, compiler tries to avoid putting together instructions that complete for some limited system resource to avoid a structural hazard. But there are limitations to these efforts taken at compiler end. In certain sequences of instructions, therefore, hazards become unavoidable.
- Out-of-Order execution is based on the concept of carrying out the Instruction scheduling at the processor hardware architecture level rather that getting performed at the compiler level.

Out-of-order Execution

- The pipelines studied earlier has the instructions statically scheduled and therefore, the execution is carried out in-order in these pipelines. That is, instructions are executed in program order. If a hazard causes stall cycles in the processor, then all instructions subsequent to the instruction causing stall, are also stalled until the hazard is resolved and the instruction executes. There are methods such as data forwarding, branch prediction, and other techniques minimize it but sometimes a stall is unavoidable. For instance, consider the following code :

```
ld    r1, (r6) // load r1 from memory location pointed by r6
add   r7, r1, r3      // r7 ← r1 + r3
sub   r4, r3, r2      // r4 ← r3 – r2
```

- Now the register contents r2 and r3 are already available in the register file. Now consider that the load instruction *ld* that causes the memory reference, generates L1 data cache miss, so the load unit takes large number of clock cycles to retrieve the data from the L2 cache or main memory.
- During the time the load unit is working on this memory reference, the pipeline is stalled. Notice, however, that the third instruction *sub* is actually not dependent on the value of r1. Therefore, it can be executed, but the standard in-order pipeline is not capable of executing this instruction unless all the previous instructions in the program order are executed. The function unit (typically say ALU)that is used for subtracting r2 from r3, is currently idle. This is exactly where the OoO becomes useful.
- ***Out-of-order execution***, or ***dynamic scheduling***, is a technique used to execute instructions non necessarily in-program-order (i.e. the order or sequence in which they appear in the user program). With out-of-order execution (OoO), the processor would ***issue*** each of the instructions in program order, and then evaluate which instructions whose operands are available (i.e. ready) would be despatched for the execution stage, irrespective of their order in the program, however, subject to the fact that it is not dependent any of the prior instruction that have not yet completed their execution.
- Therefore, in Out-of-Order execution, the instructions in the program order and received, decoded but they are executed in the data order (i.e. even prior to the instructions preceding them in the program order), subject to the conditions –
  1. The data operands needed to execute the instruction, are ready.
  2. The instruction is not dependent on any preceding instruction whose execution is not yet completed.

Out-of-order Execution Implementations

- To appropriately implement the OoO processor, the pipeline is needed to be modified to keep track of the extra complexity arising due to this feature. Additional architectural features and functional units are required.
- **Register renaming :** Registers that are the destinations of instruction results are renamed to avoid unnecessary dependency clashes, therefore more than one version of that register name may be used in the hardware. This is done by processor pipeline stage that allocates physical registers to instances of logical renamed registers using a Register Alias Table. It also keeps track of free available physical register resources.
- **Modified Instruction issue logic :** The instruction issue logic must be modified to issue instructions out of order depending on their readiness to execute i.e. in Data order rather than Program order.
- **Reservation Stations :** Each functional unit has a set of reservation stations associated with it. Each station is containing information about instructions that are either waiting or ready to issue. The reservation stations can also be used for facilitating register renaming.
- **Score boarding :** These are algorithms that keeps track of the details of the pipeline, deciding when and what to execute. The scoreboard knows when results will be available from instructions, so it knows when dependent instructions are able to be executed, and when they can write their results into destination registers. It ensures that no dependent instruction is executed and writes its results prior to the instruction on which it is dependent, thus maintaining the flow and logic of the program.
- **Retiring Unit :** It is the additional logic needed to collect the results of all the instructions upon their execution. Since the execution is Out-of-Order, the results are also received Out-of-Order. The retiring unit collects and arranges them to restore them in-order. Therefore, instructions complete or retire in the original program order.

Module 6

11.9 SPECULATIVE EXECUTION

- The Speculative execution relates to the execution of instructions speculatively i.e. with the speculation that the instructions may or may not be executed in the program sequence due to conditional branches or structures appearing in the program.
- It is the tool that is used along-with Out-of-Order execution (i.e. Dynamic instruction scheduling) and Static/Dynamic branch prediction logic, to minimize the pipeline hazards and therefore improve the processor and system performance.
- First the either the ***Static or Dynamic branch prediction*** logic is applied for the conditional branch and loop instructions to identify the alternative instruction sequences that would be followed by the program logic and the one that would be followed, is predicted. Then the ***Speculative execution*** runs the processors' pipeline to fetch, decode and execute the instructions in the program order speculatively, either in predicted path or in all alternative paths. This utilizes the processor's hardware resources and minimizes penalties due to stalls created by data and control (i.e. Instruction) hazards. Finally, the technique of ***Out-of-Order execution*** reduces the stalls and time wastage by executing instructions by data order. Three techniques together provide optimized results and deliver high performance of the processor and system.
- The ***Speculative execution***, therefore, uses system resources to execute instruction sequence in predicted (speculatively) path or all alternate paths. If the speculative prediction is correct, the results generated by execution of the instruction segment, are committed. If the prediction goes wrong, then these results are rolled back and penalty due to stalls is incurred.
- The Speculative execution is of two different types as follows :
  1. **Eager Execution :** In this type of speculative execution, every conditional branch is encountered, there are 2 paths of instruction sequences (one when the condition is satisfied and the other one when it is not) The processor resources are used to execute **both** the instruction paths. When the branch outcome is known, the correct path results are committed. This method never incurs penalties due to stalls but requires large amount of processor resources. The need for such processor resource requirement exponentially increases when multiple branches are considered.
  2. **Predictive Execution :** In this type, every conditional branch is predicted speculatively and instruction sequence only in the predicted path, is executed. In this type of execution there is no penalty if prediction goes correct. Otherwise, penalty due to stall is incurred and execution along the other path is needed to be taken up. However, since only one instruction sequence path is in execution, this method doesn't require higher amount of processor resources.

11.10 INTRODUCTION TO MULTICORE PROCESSOR ARCHITECTURE

One of the advances in parallel processing is providing Multiple cores of the processor in a single CPU package. Multi-core processing is becoming increasingly popular. Following section discusses Multicore processor architectures.

11.10.1 Multicore Processor Architecture

- **Multicore processor architecture** refers to an architecture in which a single physical processor incorporates the core logic of more than one processor. These processor cores are packaged into a single Integrated Circuit.
- These single integrated circuits are known as a die. Multicore architecture implements multiple processor cores and packages them as a single physical processor.
- The primary objective of building multiple cores is that of exploiting parallelism at the hardware level that can be used in multitasking, multithreading and multi- user environment and thereby achieve overall greater system performance.

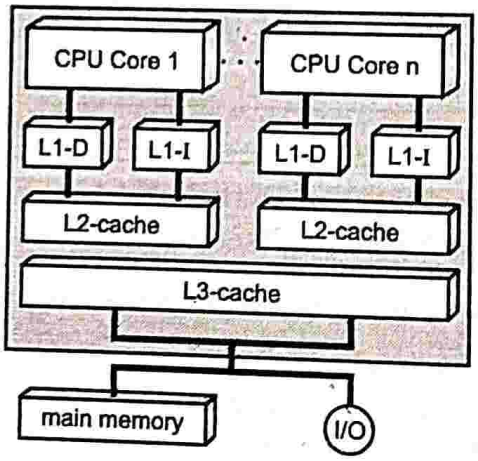


Fig. 11.10.1 : Multi-core Architecture

- This technology is most commonly used in multicore processors, where two or more processor chips or cores run simultaneously and concurrently-in-time; as a single system.
- Multicore-based processors find applications in every field and domain. They are used in mobile devices, desktops, workstations and servers.
- The Fig. 11.10.1 shows a typical Multi-core processor architecture.

- A typical Multi-core processor consists of two or more (two shown in the figure) processor cores built along with a split L1 cache (separate Instruction cache called L1-I and separate Data cache called L1-D) and a tightly coupled local unified L2 cache memory. Usually a On-board tightly coupled unified L3 cache is also implemented.
- The concept of multicore technology is mainly revolving around the concepts of parallel computing, which can significantly boost computer speed and efficiency by including two or more central processing units (CPUs) in a single chip. This reduces the system's heat and power consumption. This means much better performance with less or the same amount of energy.
- The architecture of a multicore processor enables communication between all available cores to ensure that the processing tasks are appropriately distributed and assigned among the cores, accurately.
- At the time of task completion, the processed data from each core is delivered back to the system bus by means of a single shared gateway.
- This technique significantly enhances performance compared to a single-core processor of the equivalent clock speed.
- Multicore technology is very useful in computation heavy tasks and applications, such as decoding/encoding, 3-D gaming, multimedia streaming and video editing.

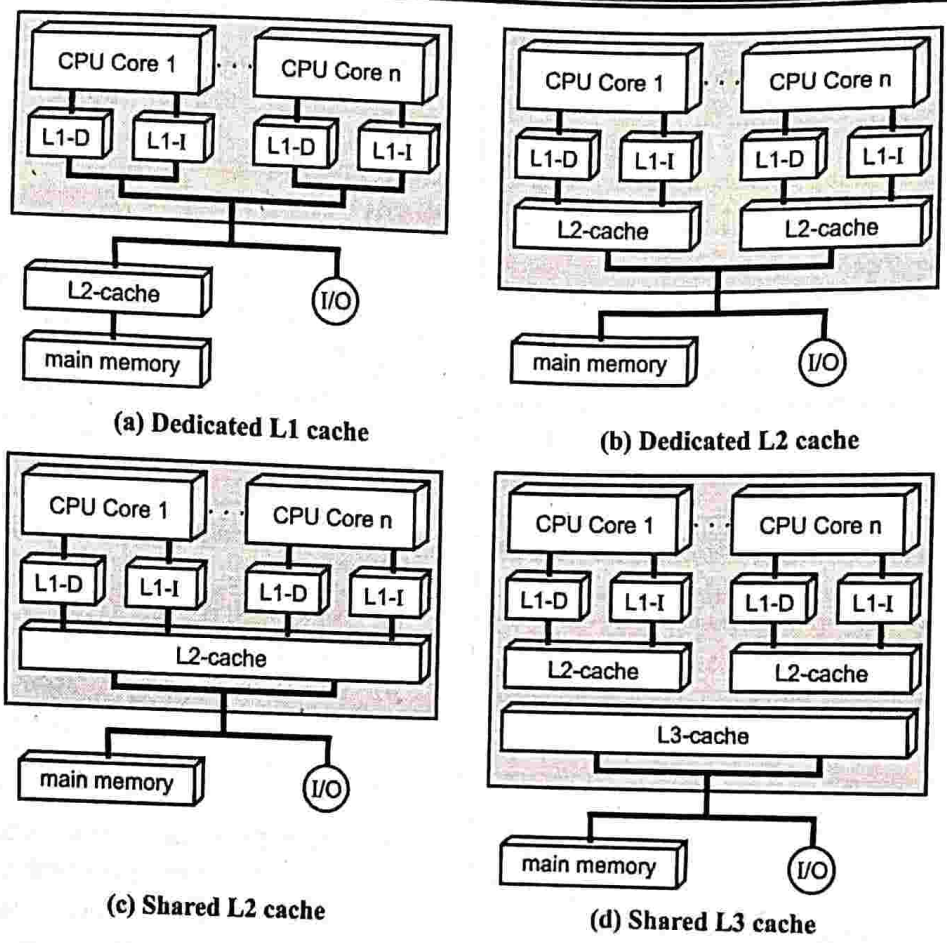
11.10.2 Hardware and Software Issues in Multicore Organization

- Multi Core Architectures are used most widely in the current dated processors.
- With saturation reaching in , the growth and frequency reaching a maximum for single CPU chip designs, manufacturers are putting up implementations of multi-cores, where each core is much smaller and relative lighter functionality then a CPU.
- These cores work together for processing a job in a better manner and with improved performance.
- The use of multiple cores allows for the frequency of the processor to be reduced, thus reducing the temperature of the system and with multi cores, instructions are allowed to run on individual cores simultaneously, which increases the amount of parallelism.
- However, issues are being faced in the designs and implementations of the multi-core processors :
  - **Thermal issues in multi core Processors :** Higher density of components and higher switching speeds make the processor chip to heat excessively putting the limitation on the frequency of operation.
  - Using higher number of cores allows the system to typically be less computation heavy and frequency of operation can be lowered to reduce the heating of the chip.
  - **Interconnection network related issues :** Bottlenecks on interconnection networks are evident in the multi-core processors due to complexity of the communication traffic between processor cores, local cache memories, main memory and other system components.
  - Newer types and design of buses such as in fib and are necessary to control this issue.
  - **Effects of parallelism -** Multi-core allows multiple parallel operations on the processors. Different core executes different threads of a processor or completely different processes.
  - This makes handling of instructions streams and flow in the parallel processing environment, a challenging task.
  - **Cache and memory issues -** Multi-core processors have privately held or tightly coupled L1 and L2 cache memories, multi-level cache memory systems with multiple cores and either a loose or tight coupling poses cache coherency and data integrity issues and elaborate protocols of substantial complexity are necessary to carry out efficient cache coherent management.

11.10.3 Multicore Organizations

- Different Multicore organization alternative are worked out on the basis of following criteria :
  - Number of cores in the Multicore Processor organization
  - Number of levels of cache memories in the Multicore Processor organization
  - Coupling of the cache memories
- The Fig. 11.10.2 shows different types of Multicore Processor organizations with different memory cache levels.

Module 6



(a) Dedicated L1 cache

(b) Dedicated L2 cache

(c) Shared L2 cache

(d) Shared L3 cache

Fig. 11.10.2 : Multi-core Organizations

(a) Multicore processor organization with dedicated L1 cache

This type of organization provides multiple processor cores with split dedicated loosely coupled L1 cache memory and external tightly coupled unified L2 cache memory.

This organization is commonly prevalent in processor architectures having 2-level cache systems. Example, Pentium Processor

(b) Multicore processor organization with dedicated L2 cache

This type of organization provides multiple processor cores with split dedicated loosely coupled L1 cache memory and internal loosely coupled dedicated unified L2 cache memory.

This organization is commonly prevalent in processor architectures having 2-level cache systems having improved performance over processors mentioned in a) above.

Example : Xeon and i-Core Processors.

(c) Multicore processor organization with Shared L2 cache

This type of organization provides multiple processor cores with split dedicated loosely coupled L1 cache memory and internal tightly coupled shared unified L2 cache memory.

This organization is commonly prevalent in processor architectures having 2-level cache systems having improved performance over processors mentioned in a) above. The Example: Core Duo Processors

(d) Multicore processor organization with Shared L3 cache

This type of organization provides multiple processor cores with split dedicated loosely coupled L1 cache memory and internal loosely coupled unified L2 cache memory.

It is added with external tightly coupled shared unified L3 cache memory for additional improvement in the system performance.

This organization is commonly prevalent in processor architectures having 2-level cache systems having improved performance over processors mentioned in (a), (b) and (c) above. The Example : Zeon and i-Core-3, 5, 7 (i3, i5, i7) Processors.

11.11 RISC AGAINST CISC

UQ. 11.11.1 Differentiate between RISC and CISC processor.

MU - Q. 4(a), May. 17, 10 Marks.
Q. 5(b)(i), Dec. 17, 5 Marks.

UQ. 11.11.2 What are the differences between RISC and CISC processors?

MU - Q. 3(a), May 14, Q. 3(b), May 15, 5 Marks.
Q. 2(b), Dec. 15, 10 Marks.
Q. 1(e), Dec. 16, 5 Marks

UQ. 11.11.3 Compare RISC and CISC processors.

MU - Q. 1(d), May 16, 5 Marks

UQ. 11.11.4 Explain features of RISC and CISC processors.

MU - Q. 4(a), Dec. 14, 10 Marks

Reduced Instruction Set Computing (RISC)

Reduced Instruction Set Computing (RISC) is based on simplification of architecture by reducing the number of instructions in the instruction set.

Complex Instruction Set Computing (CISC)

Complex Instruction Set Computing (CISC) is based on architecture that is, capable of executing complicated or complex and large number of instructions in the instruction set.

Comparison of RISC and CISC

Sr. No.	Reduced Instruction Set Computing (RISC)	Complex Instruction Set Computing (CISC)
1.	Limited or Reduced number of instructions in the Instruction set (Less than 256)	Complex and Large number of instructions in the Instruction set (much more than 256)
2.	Limited number of Addressing Modes (usually less than 5)	Large number of Addressing Modes (usually mode than 5)
3.	Limited number of Instruction formats (usually less than 5)	Large number of Instruction formats (usually much more than 5)
4.	Fixed Instruction format. All instructions are of same size and usually that of Data bus width	Variable Instruction format. Instruction sizes are varied and has no correlation with Data bus width
5.	Large size of Register file (No of CPU registers always exceed 32)	Small size of Register file (No of CPU registers are usually less that 32)
6.	Most of the instructions (usually 80-90%) execute in single clock	Different instructions require different amount of times in terms of clock periods
7.	ALU Instructions work only with Register operands. Only Load/Store Instructions support memory addressing modes	ALU Instructions work with all types (Register, memory, I/O) of operands. All Instructions usually support memory addressing modes
8.	Usually implemented by Hardwired control unit	Usually implemented by Micro-programmed (i.e. Softwired) control unit

11.12 INTRODUCTION TO BUSES

Single Bus CPU

UQ. 11.12.1 Explain Bus Contention and different method to resolve it.

MU - Q. 4(a), Dec. 18, 10 Marks

- Components of a CPU could be interconnected in different ways. In a single bus CPU organization, all components of a CPU are connected to a single internal bus. This makes the interconnection structure quite simple.
- The external bus, the bus connecting the CPU to the memory and I/O devices, is connected to the CPU via Memory Data Register (MDR) and the memory address register (MAR).
- The advantages of using internal bus arrangements are :
  (i) Simple structure / architecture allows simple controls.
  (ii) Ease of Inter-register organization and CPU space minimization is facilitated.

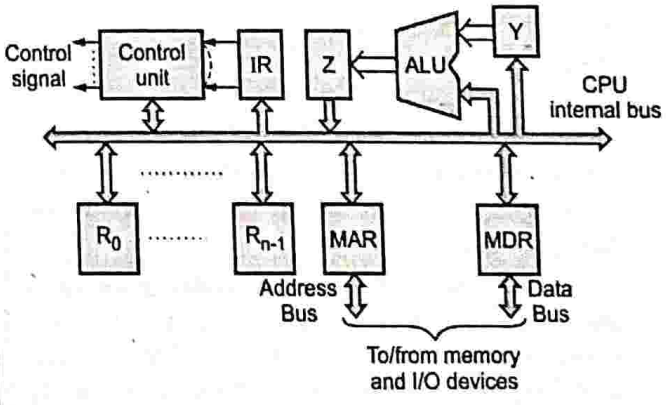


Fig. 11.12.1 : Single Bus CPU Architecture

Module 6

- In this organization
  - R0 to Rn-1 are General Purpose registers.
  - **PC - Program Counter :** Register that keeps track of current Program execution point.
  - **IR - Instruction Register :** Register that stores the Instruction Opcode received from the Memory.
  - **MAR - Memory Address Register :** Register that stores the Memory Address to be accessed, can be seen as the Front end of the External Address Bus.
  - **MDR - Memory Data Register :** Register that stores the Memory Date (read from memory or to be written to the memory), can be seen as Front end of the External Data Bus.
- Two temporary registers Y (ALU Input Temporary Register) and Z (ALU Output Temporary Register) are added to the ALU. In their absence, the output of ALU would be shorted with its input, which is to be avoided in order to keep the Data Integrity over the single bus system.

11.13 BUS ARBITRATION AND MULTIPLE BUS HIERARCHY

UQ. 11.13.1 What is bus arbitration? Explain any two techniques of bus arbitration.

MU - Q. 6(a), Dec. 14, Q. 6(b), Dec. 15, Q. 4(a), May 16, Q. 6(d), May 18, 10 Marks

UQ. 11.13.2 Explain Bus contention and different method to resolve it.

MU - Q. 4(a), Dec. 18, Q. 4(A), May 19, 10 Marks

- **Bus Arbitration** refers to the process by which the current bus master accesses and then leaves the control of the bus and passes it to another bus requesting processor unit. The controller that has access to a bus at an instance is known as **Bus master.**
- A conflict may arise if the number of DMA controllers or any other controllers or processors (In fact any Bus master) try to access the common bus at the same time, but access can be given to only one of those. Only one processor or controller can be Bus master at the same point of time. If there are multiple bus masters assigned to the bus simultaneously, multiple signals would be mixed on the buses. This would lead to noise and hardware damage. This situation is called as **Bus Contention.**
- To resolve these conflicts, Bus Arbitration procedure is implemented to coordinate the activities of all devices requesting memory transfers. The selection of the bus master must take into account the needs of various devices by establishing a priority system for gaining access to the bus.
- The **Bus Arbiter** decides who would become current bus master.
- There are two approaches to bus arbitration :
  - **Centralized bus arbitration :** A single bus arbiter performs the required arbitration.
  - **Distributed bus arbitration :** All devices participate in the selection of the next bus master.

11.13.1 Methods of Bus Arbitration / Types of Bus Arbitration / Techniques of Bus Arbitration

There are three bus arbitration methods :

(i) Daisy Chaining method

It is a centralized bus arbitration method. During any bus cycle, the bus master may be any device – the processor or any DMA controller unit or any bus master, connected to the bus. The bus mastering device checks for whether the common bus is currently occupied or not, by sampling the ***Bus busy*** line. If the ***Bus busy*** line is not asserted, one or many devices may ask for the buses by raising the ***Bus Request***. The bus arbiter, in response generates ***Bus Grant*** signal which is daisy-chained through the devices. The first requesting device that gets the ***Bus Grant*** signal gets the buses.

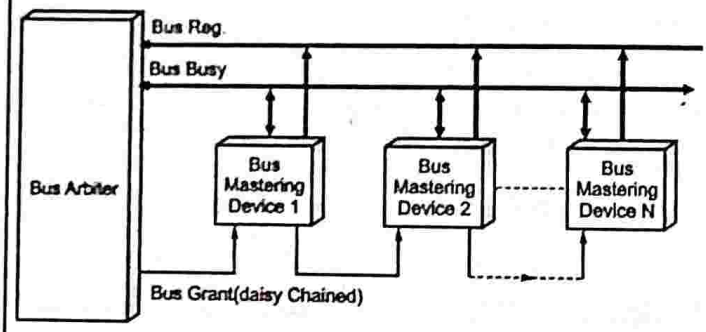


Fig. 11.13.1

Advantages

1. It is simple to use and easy to scale. No. of lines required are always low and equals 3.
2. The user can add more devices anywhere along the chain, up to a certain maximum value.

Disadvantages

1. The value of priority assigned to a device is depending on the position of master bus. i.e. Devices closer to Bus arbiter in daisy-chain get higher priority and hog the bus, on the other hand, farther devices are starved of the bus.
2. Propagation delay arises across the daisy chain, in this method.
3. If one device fails, then entire system stops working.

(ii) Polling or Rotating Priority method

In this method, the devices are assigned unique priorities and complete to access the bus, but the priorities are dynamically changed to give every device an opportunity to access the bus.

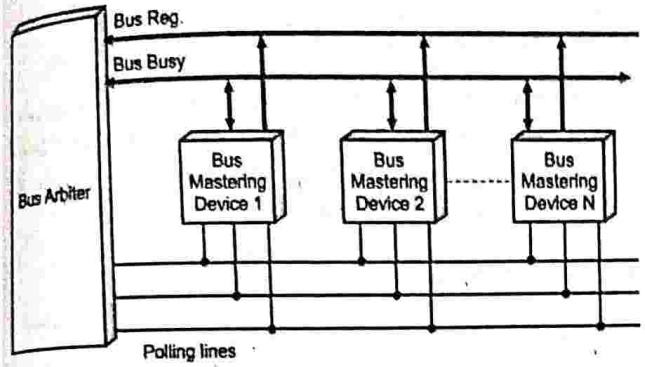


Fig. 11.13.2

When ***Bus busy*** signal is not asserted, one or more devices may request for the bus by raising ***Bus request*** signal. The bus arbiter operates the polling sequence (it is a counter), the device having matching device id gets the chance to occupy the bus. Therefore, polling applies rotating priority.

☞ Advantages

1. This method does not favour any particular device and processor.
2. The method is also quite simple.
3. If one device fails, then entire system will not stop working.

☞ Disadvantage

1. Adding bus masters is different as increases the number of address lines of the circuit.

(iii) Fixed priority or Independent Request method

In this method, the bus control passes from one device to another only through the centralized bus arbiter. Each device has an independent ***Request*** and ***Grant*** lines and they communicate with the bus arbiter through these pair of lines.

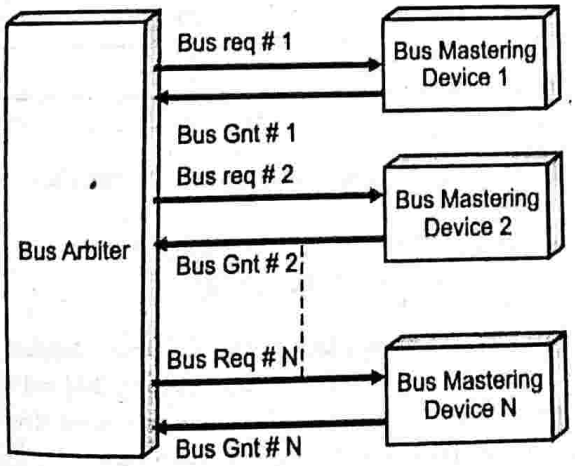


Fig. 11.13.3

☞ Advantages

1. This method generates fast response.
2. It is the most flexible method. It can implement masking and fixed or rotating priorities.

☞ Disadvantages

1. Hardware cost is high as large number of control lines are required.

11.13.2 Multiple Bus (Multibus) Hierarchy

- Generally, single bus architecture has processor or CPU connected to memory and I/O devices over the single system bus which has address bus data bus and control bus as its constituents
- In order to improve the performance of the system it is advicable to have multiple buses forming the multibus hierarchy this is done in order to have all higher speed devices connected to one bus and lower speed devices connected on different bus. Therefore allowing high speed communication on one bus and low speed communication on another bus.
- There are two widely used multibus hierarchy architectures.

(a) Architecture with local bus and system bus

- In this architecture the processor or CPU with its internal L1 cache memory interfaces to the processor local bus through the front side bus(FSB). It connects to L2 cache memory through the backside bus(BSB). The processor local bus is high speed bus and it connects to high speed devices such as video, and main memory of the system.
- The system bus (i.e. PCI bus) is operating at relatively lower speed and it connects to the processor local bus through the north bridge. The system bus interfaces storage devices, networking devices, communication devices and all other medium and low speed devices

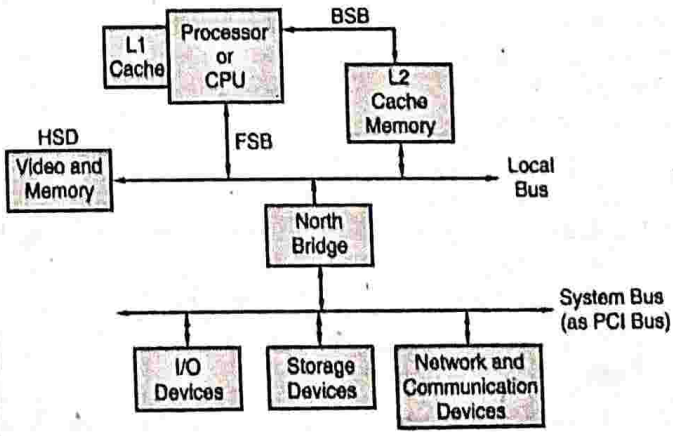


Fig. 11.13.4 : Architecture with local bus and system bus

(b) Three bus architecture (Three bus hierarchy)

- This architecture has interfaces to the processor local bus and the system bus with north bridge connecting the two buses; is very same the two bus architecture explained above.
- There is a third bus added which is a low speed bus it is called as the I/O bus(e.g. ISA bus) this bus connects to relatively low speed devices such as human interaction devices and serial and parallel ports.
- Therefore system gives a structured appearance with high speed devices connected to the processor local bus, medium speed devices connected to system bus and low speed devices connected to IO bus.

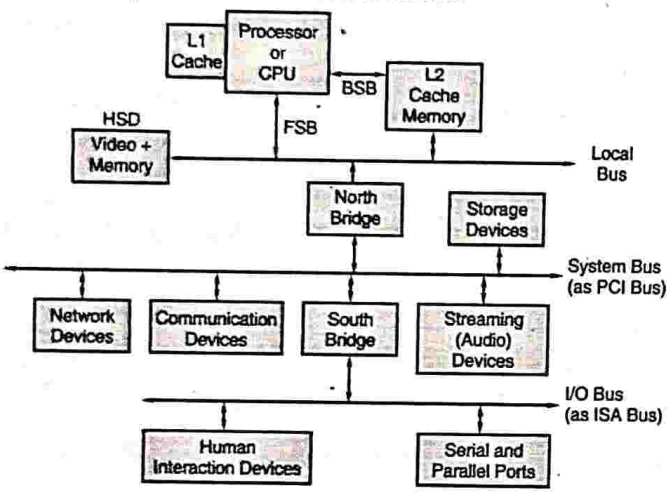


Fig. 11.13.5 : 3-Bus system architecture (3-Bus hierarchy)

11.14 ISA, PCI AND USB STANDARDS

11.14.1 Industry Standard Architecture (ISA)

Industry Standard Architecture (ISA) is the Parallel bus standard designed by IBM for providing internationally standardised interfacing capabilities to the System boards (usually named as Motherboards) for it's original 1st generation Personal Computers (PCs). The bus is based on 8-bit architecture and offers following features –

- The 8 bit ISA bus used in PC / PC-XT motherboard consists of a single card-edge connector with 62 contacts. There are 2-sides or edges to the connector and are called Side A and Side B. The Pins are standardized to have nomenclature as A1-A31 (for Side A) and B1-B31 (for Side B).
- This bus provides 8 data lines and 20 address lines. Thus addressing capability of 1 MB.
- The bus supports connections for six Independent device interrupts requests (IRQ2 - IRQ7) and 3-DMA Request/Acknowledge Channels (DRQ0,2,3) channels.
- The ISA bus was designed to operate at system speed of 4.77 MHz and was later on upgraded to 8 / 8.33 MHz.

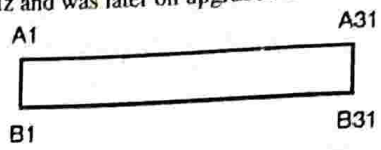


Fig. 11.14.1 : 8-Bit ISA Bus – 62 Pin Interface

- The ISA Bus was originally designed for 8-Bit architectures but later on upgraded to house 8 as well as 16-Bit Architectures. Therefore the bus is available in 2 versions 8-bit version and 16-bit version.

Features of 8 Bit ISA

- Eight data lines
- Eight interrupt request levels
- 20 address lines
- Enables to handle 1 MB of memory

Feature of 16 Bit ISA

- This bus was developed for PC-AT type of motherboard in order to provide for more interrupts, DMA channels.
- The 16 bit ISA bus used in PC-AT motherboard consists of two card-edge connectors with 62 pin and 36 pin each. The Side A and Side B connectors are pin compatible and function compatible with 8-bit ISA. It has pin connections A1-A31 (for Side A) and B1-B31 (for Side B). The additional 36 connections are available as Side C and Side D having Pin connections C1-C18 (for Side C) and D1-D18 (for Side D)
- It supports 16 bit data and 24 bit addresses. It has addressing capacity of 16 Mb.
- It has five additional independent device interrupt requests (IRQ9, IRQ11-12, IRQ14-15) and 4 more DMA Request/Acknowledge channels (DRQ4-DRQ7).
- Four more address lines are also provided in addition to several more control signals. Clock speed is increased on AT bus to 8 / 8.33 MHz.

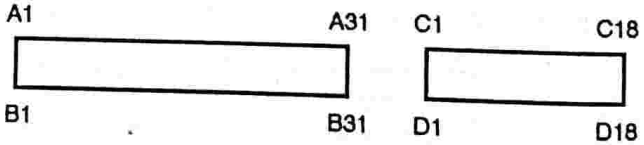


Fig. 11.14.2 : 16-Bit ISA Bus – 62 + 36 Pin Interface

11.14.2 Peripheral Component Interconnect (PCI)

- Peripheral Component Interconnect (PCI) was designed by the committee of computer scientists set up by IBM and other computer companies as the Universal international standard for 32-Bit interface for 32-Bit System Architecture.

It replaced the earlier 16-Bit bus standards such as 16-Bit ISA and Extended ISA (EISA) buses. It had many unique features and was designed to be green, environment friendly bus standard.

It connected to the high-speed processor local bus through a bridging device component called as the North Bridge and it connected to legacy I/O devices and slower peripherals operating on then ISA bus architecture through another bridge called as the South Bridge. This architecture of PCI Bus allows it to be interfaced seamlessly with almost all types of system devices. The PCI bus architecture connecting different system devices appears as shown in the Fig. 11.14.3.

Feature of PCI Bus

- PCI bus adds another tier to the traditional interface between CPU and I/O devices.
- This tier bypasses the standard I/O bus by inserting another bus between CPU and I/O devices. The tier is termed as PCI bus.
- The PCI bus is driven using a special component called as PCI bridge.
- Due to the above design and bypassing standard I/O bus, the system bus is used to increase bus clock speed and makes efficient use of CPU's data bus.
- Standard bandwidth of PCI bus at clock speed of 33 / 66 MHz and 32-bit capacity is equal to 33.33 MHz × 32 bits (4 Bytes) = 133 Mbytes/s.
- PCI slot is a 188 pin capable of transferring data at 133 MB/s as it is much faster to then standardized 5 MB/s of ISA bus.
- It has autoconfiguration capabilities for switchless/ jumperless peripherals.
- This capabilities take care of all addresses, IRQ and DMA used by PCI type of peripheral.
- This bus supports linear bursts type of data transfer which ensures that data bus is continually filled with data.
- This bus supports bus mastering, thereby allowing one of a number of intelligent peripheral to take control bus to accelerate high throughput, high priority task.
- PCI architecture supports concurrency-a technology which ensures that microprocessor operates simultaneously with these masters, instead of waiting for them. Another key feature of PCI is that it is dual voltage architecture.
- Normally, the bus operates at +5 volt dc supply, like other buses, however it can also operate in +3.3 volt dc mode.
- Note that there are two major segments to the +5 volt dc connector. A +3.3 volt connector adds a key in 12/13 position to prevent accidental insertion of +5 volt dc PCI board into +3.3 volt dc slot.
- Similarly, the +5 volt dc slot is keyed in 50/51 position to prevent placing +3.3 volt dc board into +5 volt dc slot.

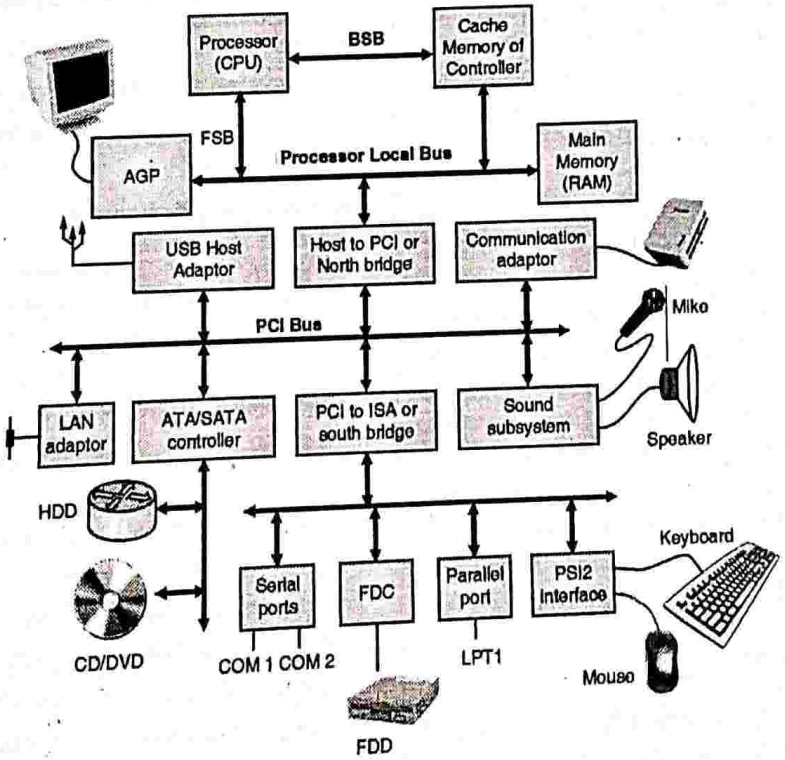


Fig. 11.14.3 : PCI Bus Architecture

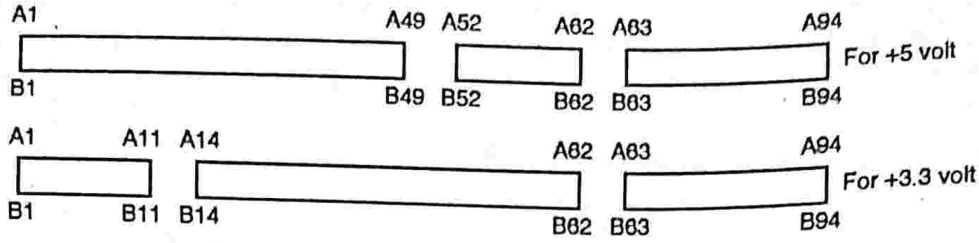


Fig. 11.14.4 : 188 Pin PCI Bus Interface

- Peak transfer rate of 133 MB/s for 32-Bit and 266 MB/s for 64-bit bus width.
- It can work with 32-bit or 64-bit bus width.
- Each PCI bus have 256 devices and maximum of 256 PCI buses per system.
- It uses 3.3 V or 2.2 V for operations and consumes less power.
- PCI bus is device independent which means it can be used to connect different types of devices such as hard disk controllers, sound cards, multimedia controllers, LAN cards etc.
- It supports techniques such as Green Machine concept, Reflected wave switching and Fairness algorithm for Bus mastering.

11.14.3 Universal Serial Bus (USB)

- Universal Serial Bus (USB) is the international universal bus/port interface standard for connecting peripherals and I/O devices with the computer system and works on the synchronous serial data communication protocols & techniques. Simple design with only 4-wire interface and its capabilities of interfacing to host of and most of the peripherals seamlessly makes it a very popular serial interface port. Currently, it has become the de-facto interface standard for all modern peripherals and I/O devices.
- Computer comes with one or more Universal Serial Bus connectors on the back. All the operating systems supports USB so that the installation of the device drivers is quick and easy. Compared to other ways of connecting devices to computer (including parallel ports, serial ports and special cards), USB devices are very simple to interface. The Universal Serial Bus provides a single, standardized, easy-to-use way to connect up to 127 devices to a computer.
- It is capable of interfacing to slow and high-speed input devices as well as output devices at the same time with external mass storage devices and streaming Multi-media (Audio & Video) devices.

USB Cables and Connectors

- Connecting a USB device to a computer is simple - you find the USB connector on the back of your machine and plug the USB connector into it. If it is a new device, the operating system auto-detects it and asks for the device driver. If the device has already been installed or in the system database of drivers, the computer auto-detects it and starts communicating to it. USB devices can be connected and disconnected at any time.
- USB has two types of connectors, namely Type A and Type B connectors. Type A connectors interface to the system side and are called as Upstream connector. Type B connectors interface on the device side and are called as Downstream connector.
- Many USB devices come with their own built-in cable, and the cable has an "A" connection on it. If not, then the device has a socket on it that accepts a USB "B" connector.

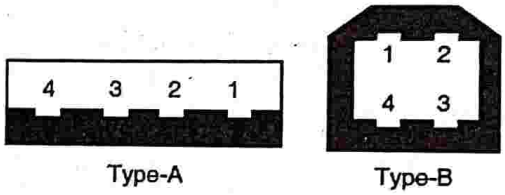


Fig. 11.14.5 : USB Type A and Type B Connections

The USB standard uses "A" and "B" connectors :

- "A" connectors head "upstream" toward the computer.
- "B" connectors head "downstream" and connect to individual devices.

Types of USB data transfers

USB works on Synchronous Serial Data communication technique with default half duplex mode. It employs Packet transfer protocol and therefore has different types of data transfers for facilitating different class of data.

When the host powers up, it queries all of the USB devices connected to the bus through the USB Root Hub and assigns each one an address. This process is called as USB enumeration. Therefore, devices are said to be enumerated when they connect to the bus. The USB host adapter also finds out from each device what type of data transfer it wishes to perform. There are Four different Data Transfers supported by USB standard :

- **Control Transfer :** This transfer has small packet size and lower data transfer rates. It is used for sending and receiving Control and Status Data packets respectively.

Interrupt Transfer : This transfer has small packet size and lower data high transfer rate. It is suitable of an input device like a mouse or a keyboard, which will be sending very little data (and needs an interrupt to draw processor's attention toward the device activity) would choose the interrupt mode.

Bulk Transfer : This transfer has large packet size and very high data transfer rate. It is suitable for bulk or large data transferring device like a printer or external Hard disk, which receives data in one big packet, uses the bulk transfer mode. A block of data is sent and verified to make sure it is correct.

Isochronous Transfer : This transfer has large packet size and very high data transfer rate. A streaming device such as Multi-media device uses the isochronous mode. Data streams between the device and the host in real-time, and there is no error correction; thus achieving the highest data transfer rates.

Features of USB Ports

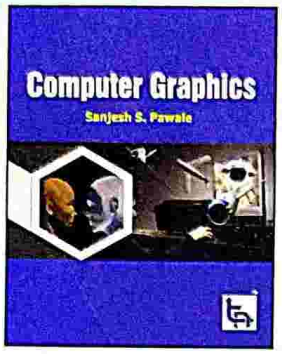
- The computer has the host adapter.
- **Number of devices :** Up to 127 devices can connect to the host, either directly or by way of USB hubs.
- **Maximum distance :** Individual USB cables can run as long as 5 meters; with hubs, devices can be up to 30 meters (six cables cascaded) away from the host.
- **Speed of data transfer :** With USB 2.0, the bus has a maximum data rate of 480 megabits per second and USB 3.0 allows maximum data rates up to 1 gbps.
- **Power requirement and types of cable :** A USB cable has two wires for power (+5 volts and ground) and a twisted pair of wires to carry the data. On the power wires, the computer can supply up to 500 milliamps of power at 5 volts. Low-power devices (such as mice) can draw their power directly from the bus. High-power devices (such as printers) have their own power supplies and draw minimal power from the bus. Hubs can have their own power supplies to provide power to devices connected to the hub.
- **Hot pluggability :** A USB device can be connected without powering off a PC. The 'plug and play' feature in the BIOS together with the intelligence in the USB devices takes care of detection, device recognition and handling.
- **Hot swappable :** USB devices are hot swappable; means you can plug them into the bus and unplug them any time.
- **Hub architecture :** The devices are not daisy chained. Each device is connected to an USB hub. The USB hub is an intelligent device interacting to the PC on one side and the USB peripheral devices on other sides.
- **Host centric :** The CPU/software initiates every transaction on the USB bus. Hence, the overhead on the PC software increases when large number of peripherals, that too, involving the large number of transactions, are connected.
- **Power saving :** Many USB devices can be put to sleep by the host computer when the computer enters a power-saving mode.

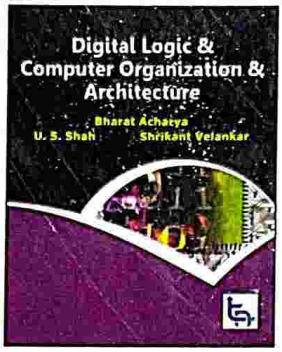
...Chapter Ends

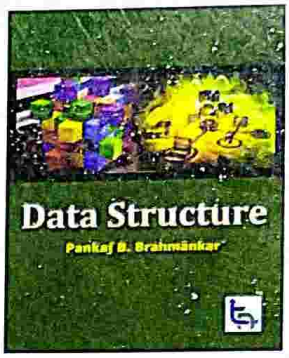
❑❑❑

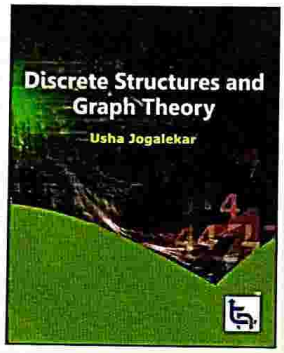
Note

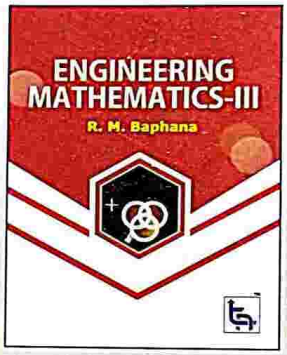
Semester 3 - Computer Engineering











TECH-NEO PUBLICATIONS
Where Authors Inspire Innovation
A Sachin Shah Venture

Tech-Neo Books
Tech-Neo Publications
Tech-Neo Publications
Tech-Neo Publications

For Orders Contact :
Krishna Book Collections
Ground Floor, Krishna Niwas Building, Behind BEST Niwas Building,
Near Napoo Hall, Chandavarkar Road, Matunga East, Mumbai 400019.
E-mail : dharmeshsota05@gmail.com
Mobile No.: Dharmesh Sota - 9820741455
Tulsidas Sota - 9833133921 / 9833082745 / 9833082761

ISBN : 978-81-947354-9-6
Price ₹ 285/-
(B1258)

SCAN TO VISIT
www.techneobooks.com

info@techneobooks.com
www.techneobooks.com