

VSM COLLEGE OF ENGINEERING

RAMACHANDRAPURAM

1-2 Semester

COMPUTER ORGANIZATION

UNIT I: Digital Computers and Data Representation: Introduction ,Numbering Systems, Decimal to Binary Conversion, Binary Coded Decimal Numbers, Weighted Codes, Self Complementing Codes, Cyclic Codes, Error Detecting Codes, Error Correcting Codes, Hamming Code for Error Correction, Alphanumeric Codes, ASCII Code Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation. Boolean Algebra and Logical gates: Boolean Algebra :Theorems and properties, Boolean functions, canonical and standard forms , minimization of Boolean functions using algebraic identities; Karnaugh map representation and minimization using two and three variable Maps ;Logical gates ,universal gates and Two-level realizations using gates : AND-OR, OR-AND, NAND-NAND and NOR-NOR structures.

UNIT II: Digital logic circuits: Combinatorial Circuits: Introduction, Combinatorial Circuit Design Procedure, Implementation using universal gates, Multi-bit adder, Multiplexers, Demultiplexers, Decoders Sequential Switching Circuits: Latches and Flip-Flops, Ripple counters using T flip-flops; Synchronous counters: Shift Registers; Ring counters

UNIT III: Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Booth multiplication algorithm, Division Algorithms, Floating – point Arithmetic operations. Register Transfer language and microinstructions :Bus memory transfer, arithmetic and logical micro-operations, shift and rotate micro-operations Basic Computer Organization and Design: Stored program concept, computer Registers, common bus system, Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input–Output configuration and program Interrupt.

UNIT IV: Microprogrammed Control: Control memory, Address sequencing, microprogram example, design of control unit. Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control: conditional Flags and Branching

UNIT V: Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory. Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt Direct memory Access.

Text Books: 1. Digital Logic and Computer Design, Moriss Mano, 11th Edition, Pearson Education.
2. Computer System Architecture, 3rd ed., M. Morris Mano, PHI

Reference Books: 1. Digital Logic and Computer Organization, Rajaraman, Radhakrishnan, PHI, 2006
2. Computer Organization, 5th ed., Hamacher, Vranesic and Zaky, TMH, 2002
3. Computer Organization & Architecture :Designing for Performance, 7th ed., William Stallings, PHI, 2006

Course Outcomes: By the end of the course the student will be able to

- Demonstrate and understanding of the design of the functional units of a digital computer system.
- Relate Postulates of Boolean algebra and minimize combinational functions.
- Recognize and manipulate representations of numbers stored in digital computers.
- Build the logic families and realization of logic gates.
- Design and analyze combinational and sequential circuits.
- Recall the internal organization of computers, CPU, memory unit and Input/Outputs and the relations between its main components .
- Solve elementary problems by assembly language programming.

VSM COLLEGE OF ENGINEERING
RAMACHANDRAPURAM
DEPARTMENT OF HUMANITIES AND BASIC SCIENCES

Course Title	Year/Sem	Branch	Periods per Week
COMPUTER ORGANIZATION	I / II	CSE BRANCH	6

Course Outcomes:

By the end of the course the student will be able to

- Demonstrate and understanding of the design of the functional units of a digital computer system.
- Relate Postulates of Boolean algebra and minimize combinational functions.
- Recognize and manipulate representations of numbers stored in digital computers.
- Build the logic families and realization of logic gates.
- Design and analyze combinational and sequential circuits.
- Recall the internal organization of computers, CPU, memory unit and Input/Outputs and the relations between its main components .
- Solve elementary problems by assembly language programming.

Unit No	Outcomes	Name of the Topic	No. of Periods required	Total Periods	Reference Book	Methodology to be adopted
I	CO 1: Demonstrate and understanding of the design of the functional units of a digital computer system.	Unit-1 Digital Computers and Data Representation		14	T1, T2 R20	
		Introduction ,Numbering Systems	1			Black Board
		Decimal to Binary Conversion, Binary Coded Decimal Numbers, Weighted Codes, Self Complementing Codes,	2			Black Board
		Cyclic Codes, Error Detecting Codes, Error Correcting Codes,	2			E-Class Room
		Hamming Code for Error Correction, Alphanumeric Codes, ASCII Code, Data Representation: Data types, Complements,	2			E-Class Room
		Fixed Point Representation, Floating Point Representation.	1			E-Class Room
		Boolean Algebra :Theorems and properties, Boolean functions, canonical and standard forms	1			Seminar
		minimization of Boolean functions using algebraic identities;	1			Black Board
		Karnaugh map representation and minimization using two and three variable Maps	2			Black Board
		Logical gates ,universal gates and Two-level realizations using gates : AND-OR, OR-AND, NAND-NAND and NOR-NOR structures	2			E-Class Room

		Unit-2 Digital logic circuits, Sequential Switching Circuits				
II	CO 2: Relate Postulates of Boolean algebra and minimize combinational functions	Combinatorial Circuits: Introduction, Combinatorial Circuit Design Procedure	1	9	T1, T2 R20	Black Board
		Implementation using universal gates, Multi-bit adder	2			E-Class Room
		Multiplexers, Demultiplexers, Decoders	2			E-Class Room
		Latches and Flip-Flops, Ripple counters using T flip-flops	2			Black Board
		Synchronous counters: Shift Registers; Ring counters	2			E-Class Room

		Unit-3 Computer Arithmetic, Register Transfer language and microinstructions, Basic Computer Organization and Design				
III	CO 3 : Recognize and manipulate representations of numbers stored in digital computers	Addition and subtraction, multiplication Algorithms	2	17	T1, T2 R20	Black Board
		Booth multiplication algorithm, Division Algorithms	2			E-Class Room
		Floating – point Arithmetic operations	2			Black Board
		Bus memory transfer, arithmetic and logical micro-operations	2			Black Board
		shift and rotate micro-operations	1			Black Board
		Stored program concept, computer Registers, common bus system	2			E-Class Room
		Computer instructions, Timing and Control	2			Black Board
		Instruction cycle, Memory Reference Instructions	2			Black Board
		Input–Output configuration and program Interrupt	2			E-Class Room

		Unit-4 Micro programmed Control , Central Processing Unit				
IV	CO 4 : Build the logic families and realization of logic gates, Design and analyze combinational and sequential circuits.	Control memory, Address sequencing	2	10	T1, T2 R20	Black Board
		Micro program example, design of control unit	2			Black Board
		General Register Organization, Instruction Formats	2			E-Class Room
		Addressing modes, Data Transfer and Manipulation	2			E-Class Room
		Program Control: conditional Flags and Branching	2			Black Board

		Unit-4 Memory Organization , Input-Output Organization					
V	CO5: Recall the internal organization of computers, CPU, memory unit and Input/Outputs and the relations between its main components , Solve elementary problems by assembly language programming		2	10	T1, T2 R20	Black Board	
		Memory Hierarchy				2	E-Class Room
		Main Memory , Auxiliary memory				2	Black Board
		Associate Memory , Cache Memory				2	E-Class Room
		Input-Output Interface, Asynchronous data transfer				2	Black Board
		Modes of Transfer, Priority Interrupt Direct memory Access				2	Black Board

		TOTAL	60		
--	--	--------------	-----------	--	--

Text Books: 1. Digital Logic and Computer Design, Moriss Mano, 11th Edition, Pearson Education.
2. Computer System Architecture, 3rd ed., M. Morris Mano, PHI

Reference Books: 1. Digital Logic and Computer Organization, Rajaraman, Radhakrishnan, PHI, 2006
2. Computer Organization, 5th ed., Hamacher, Vranesic and Zaky, TMH, 2002
3. Computer Organization & Architecture : Designing for Performance, 7th ed.,
William Stallings, PHI, 2006

Faculty Member

Head of the Department

Principal

UNIT - 1

Introduction :- 'Computer' is a machine that can store and process information. Most computers rely on a binary system that uses two variables 0 and 1 to complete tasks such as storing data calculating algorithms and displaying information.

Organization :- Group of people who work together and to reach a goal by using proper system.

System :- A set of things working together to accomplish particular goal.

Ex :- School system, college system and railway system.

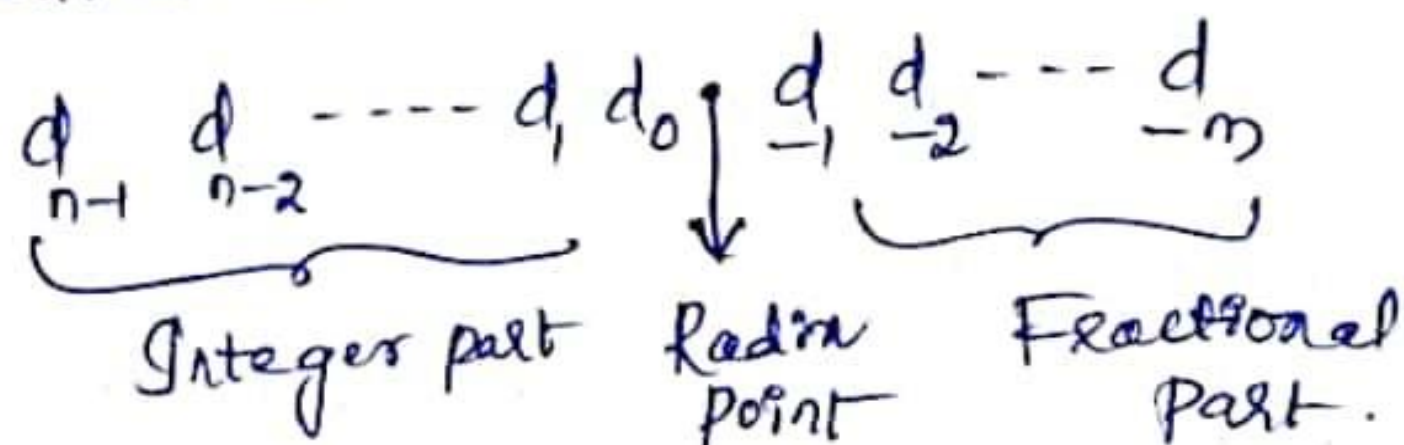
Number system :- Represent data in digital form.

Binary Number system :- A number is made up of a collection of digits and it has two parts.

(a) Integer part

(b) Fraction part

Both are separated by a radix point (.). The number is represented as



Number systems are classified as

- 2
- (a) Binary Number System
 - (b) Decimal Number System
 - (c) Octal Number System
 - (d) Hexadecimal Number System.

(a) The Binary number system is a radix-2. This is represented in terms of 0 and 1. The radix point is known as the binary point and 0 and 1 is a binary bits.

(b) The decimal number system is a radix-10. These are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 (0-9). The radix point is known as decimal point.

(c) Octal number system is a radix 8. They are 0, 1, 2, 3, 4, 5, 6, and 7 (0-7). The radix point is known as Octal point.

(d) The Hexadecimal number system is a radix 16. They are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15. The radix point is known as Hexadecimal point. The decimal equivalent of A, B, C, D, E and F are 10, 11, 12, 13, 14 and 15.

Radix :- Number of symbols presented in a respective number system is called Radix.

$$\text{Sa: } \underline{\quad} 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0$$

\downarrow
 MSB
 (Most significant bit)

 \downarrow
 LSB (Least significant bit)

Decimal number system to Binary number system conversion

$$\textcircled{1} (25)_{10} = (?)_2$$

$$\begin{array}{r} 2 \overline{) 25} \\ 2 \overline{) 12} \text{ - } 1 \\ 2 \overline{) 6} \text{ - } 0 \\ 2 \overline{) 3} \text{ - } 0 \\ 1 \text{ - } 1 \end{array}$$

From Bottom to top

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

(Successive Division
method)

$$\textcircled{2} (72)_{10} = (?)_2$$

$$\begin{array}{r} 2 \overline{) 72} \\ 2 \overline{) 36} \text{ - } 0 \\ 2 \overline{) 18} \text{ - } 0 \\ 2 \overline{) 9} \text{ - } 0 \\ 2 \overline{) 4} \text{ - } 1 \\ 2 \overline{) 2} \text{ - } 0 \\ 1 \text{ - } 0 \end{array}$$

From
Bottom
to
Top

$$\therefore (72)_{10} = (1001000)_2$$

⇒ Fractional part :-

$$\textcircled{3} (0.25)_{10} = (?)_2 \quad \begin{array}{l} \text{Top to} \\ \text{Bottom} \end{array}$$

$$\begin{array}{l} 0.25 \times 2 = 0.5 \rightarrow 0 \\ 0.5 \times 2 = 1.0 \rightarrow 1 \\ 0.0 \times 2 = 0.0 \rightarrow 0 \\ 0.0 \times 2 = 0 \rightarrow \text{ignore it} \end{array}$$

Whenever we get repeated numbers just ignore it

$$\therefore (0.25)_{10} = (0.010)_2$$

$$\textcircled{4} (0.8125)_{10} = (?)_2$$

$$\begin{array}{l} 0.8125 \times 2 = 1.6250 \rightarrow 1 \\ 0.6250 \times 2 = 1.2500 \rightarrow 1 \\ 0.2500 \times 2 = 0.5000 \rightarrow 0 \\ 0.5000 \times 2 = 1.0000 \rightarrow 1 \\ 0.0000 \times 2 = 0.0000 \rightarrow 0 \end{array}$$

Top to Bottom

$$\therefore (0.8125)_{10} = (0.11010)_2$$

⑤ $(10.625)_{10} = (?)_2$

2	10	
2	5-0	↑
2	2-1	
	1-0	
(1010)		

$0.625 \times 2 = 1.250 \rightarrow 1$	↓	(010)
$0.250 \times 2 = 0.50 \rightarrow 0$		
$0.50 \times 2 = 1.0 \rightarrow 1$		
$0.0 \times 2 = 0.0 \rightarrow 0$		

$\therefore (10.625)_{10} = (1010.1010)_2$

⑥ $(25.125)_{10} = (?)_2$

2	25	
2	12-1	↑
2	6-0	
2	3-0	
	1-1	
(11001)		

$0.125 \times 2 = 0.250 \rightarrow 0$	↓	(0010)
$0.250 \times 2 = 0.50 \rightarrow 0$		
$0.5 \times 2 = 1.0 \rightarrow 1$		
$0.0 \times 2 = 0.0 \rightarrow 0$		

$\therefore (25.125)_{10} = (11001.0010)_2$

Binary Number system to Decimal number system conversion:-

① $(1101)_2 = (?)_{10}$

(Successive multiplication method)

1	1	0	1
2^3	2^2	2^1	2^0

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13$ $\therefore (1101)_2 = (13)_{10}$

(2) $(101011)_2 = (?)_{10}$

1	0	1	0	1	1
2^5	2^4	2^3	2^2	2^1	2^0

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 32 + 0 + 8 + 0 + 2 + 1 = 43$$

$$\therefore (101011)_2 = (43)_{10}$$

(3) $(101.10)_2 = (?)_{10}$

1	0	1	.	1	0
2^2	2^1	2^0		2^{-1}	2^{-2}

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} = 4 + 0 + 1 + 0.5 + 0 = 5.5$$

$$\therefore (101.10)_2 = (5.5)_{10}$$

(4) $(11010.010)_2 = (?)_{10}$

1	1	0	1	0	.	0	1	0
2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}

$$= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} = 16 + 8 + 0 + 2 + 0 + 0 + 0.25 + 0 = 26.25$$

$$\therefore (11010.010)_2 = (26.25)_{10}$$

(5) $(10010)_2 = (?)_{10}$

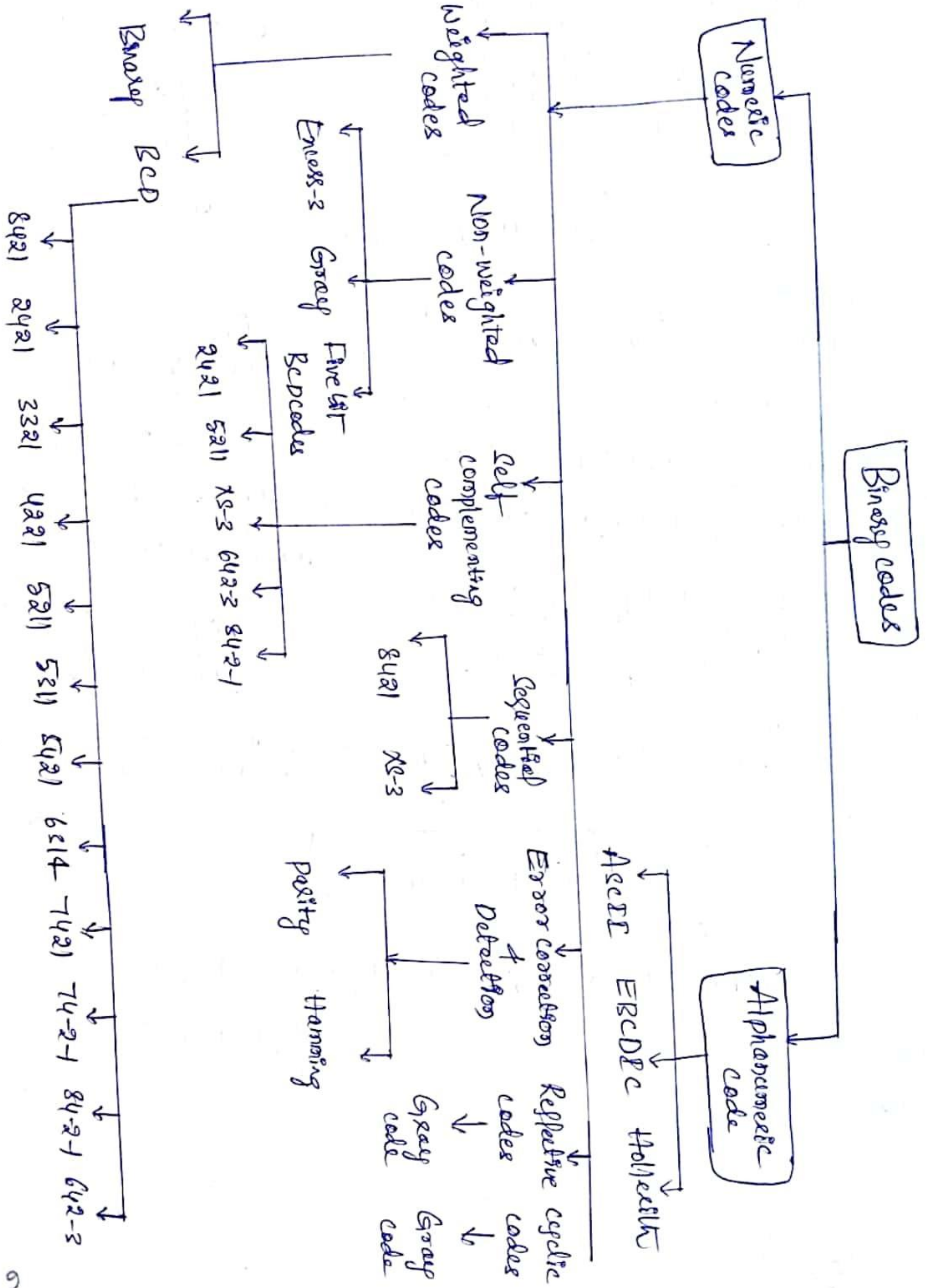
1	0	0	1	0
2^4	2^3	2^2	2^1	2^0

$$1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 16 + 0 + 0 + 2 + 0 = (18)_{10}$$

(6) $(011.01)_2 = (?)_{10}$

0	1	1	.	0	1
2^2	2^1	2^0		2^{-1}	2^{-2}

$$0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 0 + 2 + 1 + 0 + 0.25 = (3.25)_{10}$$



7

→ Binary coded decimal Numbers (BCD) :-

→ BCD code uses four bits to represent the decimal numbers. i.e (0-9). Each single decimal number can be represented by a four bit pattern.

→ 8421 is also known as Natural BCD.

Ex:- 8421, 2421, 3321, 4221, 5211, 5311, 5421, 6314, 7421
84-2-1, 642-3.

→ Representation of BCD code

Ex:- (1) $\begin{matrix} 12 \\ \swarrow \downarrow \\ 0001 \ 0010 \end{matrix}$ (Each digit is represented by four bits)

Ex:- (2) $\begin{matrix} 14 \\ \swarrow \downarrow \\ 0001 \ 0100 \end{matrix}$ (Each digit is indicated by group of 4-bits)

There are 6 unused states in BCD (1010, 1011, 1100, 1101, 1110, 1111)
 $\begin{matrix} \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 10 & 11 & 12 & 13 & 14 & 15. \end{matrix}$

<u>Decimal number</u>	<u>BCD</u>
14	0001 0100
234	0010 0011 0100
239.56	0010 0011 1001 . 0101 0110
653.96	0110 0101 0011 . 1001 0110

* Represent 356 in BCD format

ans:- $\begin{matrix} 3 & 5 & 6 \\ \downarrow & \downarrow & \downarrow \\ 0011 & 0101 & 0110. \end{matrix}$

⇒ Pure binary representation :-

Ex: 14 —

8	4	2	1
1	1	1	0

Only 4-bits

Ex: 12 —

8	4	2	1
1	1	0	0

Only 4-bits

In BCD

1	4
↓	↓
0001	0100

8-bits required

1	2
↓	↓
0001	0010

8-bits required.

Q To represent 12 in binary what are the minimum no. of digits we required.

Q To represent 12 in BCD what are the minimum no. of digits we required.

Binary →

8	4	2	1
1	1	0	0

Only 4-bits

BCD →

↓	↓
0001	0010

8-bits.

Note :- From the above concept, we can conclude one thing that BCD is simple to represent decimal numbers but some times it takes more no. of bits. So, it occupies memory. Arithmetic operations are more complex than the binary.

Ex:

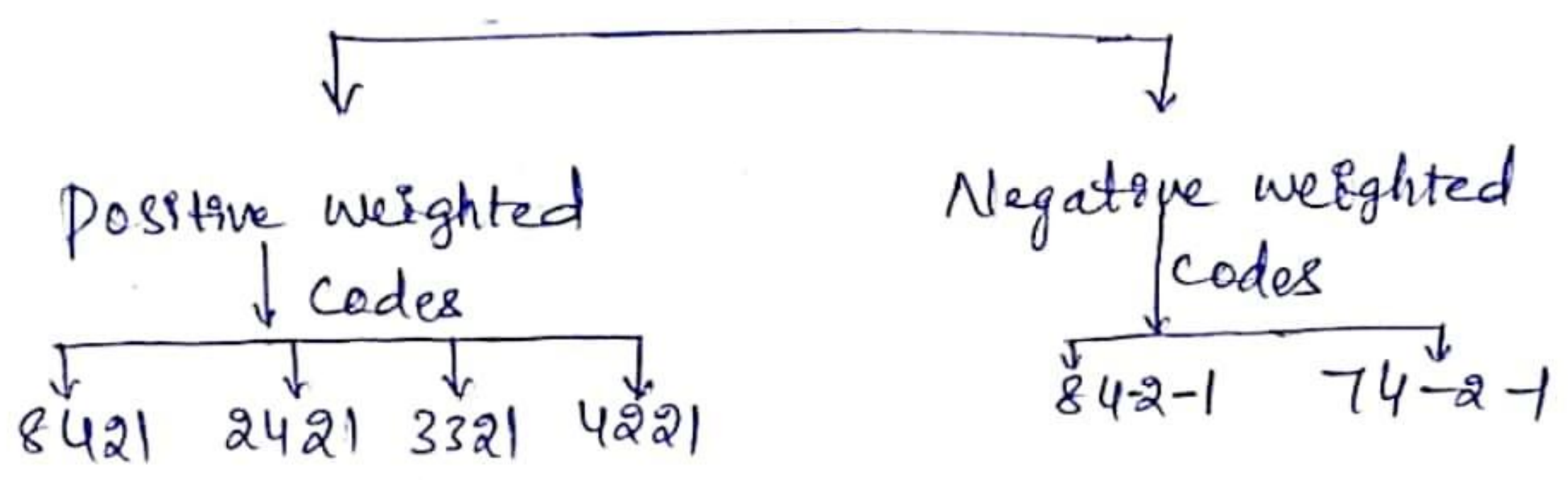
92 in Binary representation

64	32	16	8	4	2	1
1	0	1	1	1	0	0

92 in BCD

↓	↓
1001	0010

⇒ Weighted codes :- The weighted codes are those which obey the position weighting principle. Each position of the number represents a specific weight.



Ex^o:-

Decimal	8421	2421	3321	4221
0	0000	0000	0000	0000
1	0001	0001	0001	0001
5	0101	{ 1010 2 0101	{ 1010 2 0110	{ 1001 2 0111
7	0111	{ 1101 2 0111	1101	{ 1101 2 1011

All these are positive weighted codes

Ex^o:-

Decimal	84-2-1	74-2-1
0	0000	0000
5	1011	1010
7	1001	1000
9	1111	1110

⇒ Non-weighted codes → Excess-3 code
 → Gray code

→ Self-complementing code :- It is said to be self-complementing if the code word of the 9's complement of N i.e. $9-N$ can be obtained from the code word of N by interchanging all the 0's and 1's.

Ex: 2421, 5211, 642-3, 84-2-1 & Excess-3.

$$\begin{array}{cccc}
 2+4+2+1 & 5+2+1+1 & 6+4+2-3 & 8+4-2-1 \\
 =9 & =9 & =9 & =9
 \end{array}$$

Decimal digital	8421	2421	Excess-3
0	0000	0000	0011
1	0001	0001	0100
2	0010	0010	0101
3	0011	0011	0110
4	0100	0100	0111
5	0101	1011	1000
6	0110	1100	1001
7	0111	1101	1010
8	1000	1110	1011
9	1001	1111	1100

Ex: ①

5 in Excess-3 is $5 + 3 = 8 = 1000$ ⁸⁴²¹

It can be obtained by adding 3 to that binary number

$1000 \xrightarrow[\text{1's complement}]{} 0111$ [Ex-3 code of decimal number 4]

4 is the 9's complement of 5 ($9 - 5 = 4$)

\therefore It is a self-complementing code

Ex: ②

4 in 2421 = 0100 ²⁴²¹

$0100 \xrightarrow[\text{1's complement}]{} 1011$ (This is 2421 code for decimal number 5)

5 is the 9's complement of 4.

\therefore It is a self-complementing code.

Ex: ③

BCD code for 6 is 0110 ⁸⁴²¹

$0110 \xrightarrow[\text{1's complement}]{} 1001$ (This is BCD code for decimal number 9)

9 is not the 9's complement of 6.

\therefore BCD is not a self-complementing code.

Ex: ④ ^{pure} Binary (8421) code for 7 is 0111

$0111 \xrightarrow[\text{1's complement}]{} 1000 = 8$ (Binary code ^{9 decimal} 8)

8 is not the 9's complement of 7

\therefore Binary code (8421) is not a self-complementing code.

⇒ Cyclic codes :- cyclic codes are those in which each successive code word differs from the preceding one in only 1-bit position. cyclic codes are also called as unit distance codes Ex: Gray code.

* Gray code is also called as Reflective code. Reflective code means in 8421 code 0-7 is the mirror image of 8-15. Gray code is not a sequence code. That's why we can't do arithmetic operation by using

Ex: this code.

<u>Decimal No.</u>	<u>Binary</u>	<u>Gray</u>
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Ⓐ Convert $(1010)_2$ to gray code

Sol 1010
↓
ans 1111

Ⓑ Convert $(0110)_2$ to gray code

0110
↓
ans 0101

X-OR Truth Table

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

⇒ Alphanumeric codes :- These are the codes which represent alphanumeric information i.e letters of the alphabet and decimal numbers as a sequence of 0's and 1's.

Eg:- ASCII, EBCDIC codes

→ ASCII → American standard code for information interchange

→ EBCDIC → Extended binary coded decimal interchange code.

→ Alphanumeric codes consists of numbers as well as alphabetic characters.

→ It contains 26 Alphabets with Capital & Small letters, numbers (0-9), punctuation marks and other symbols.

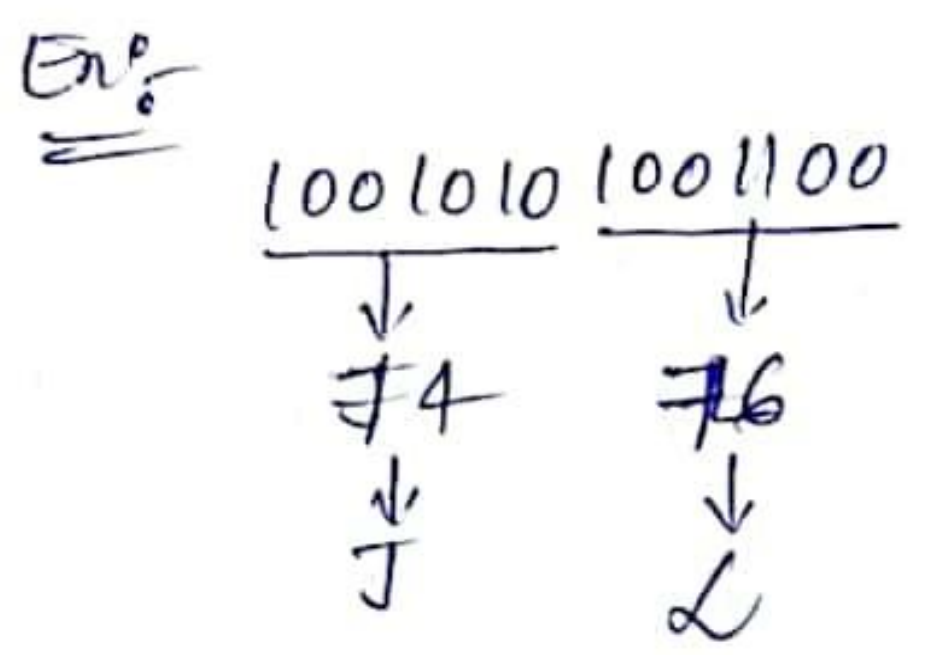
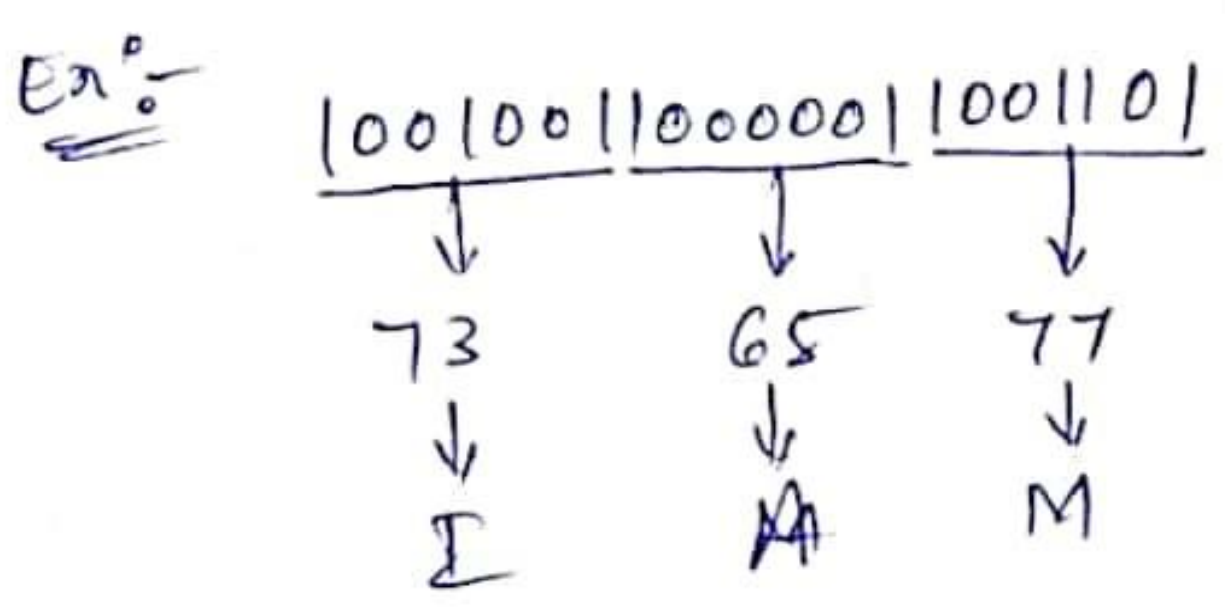
→ ASCII code is a 7-bit code and more commonly used worldwide.
∴ $2^7 = 128$ symbols are used to represent info's.

→ EBCDIC code is a 8 bit code and used in large IBM computers.
∴ $2^8 = 256$ symbols are used
International Business machine.

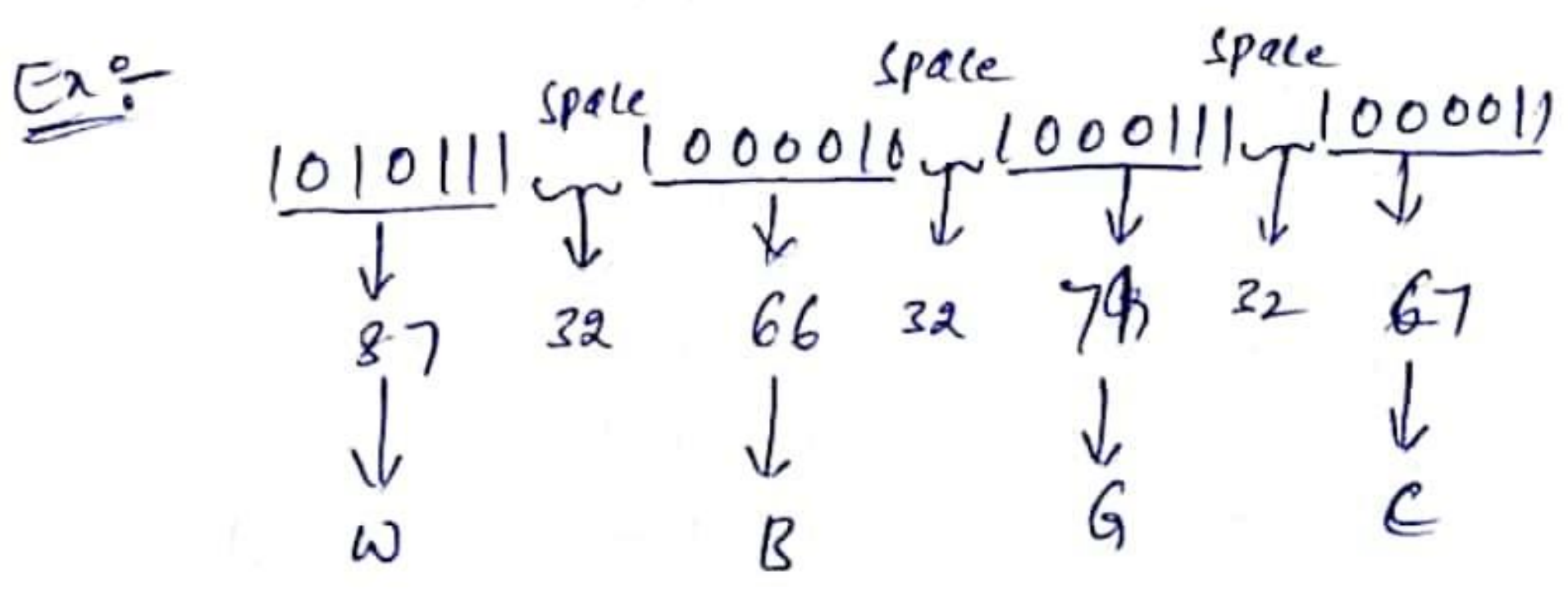
- | | | |
|------------------|-----------------|-----------------------------|
| 32 → space | 42 → * | 61 → = |
| 33 → ! | 43 → + | 62 → > |
| 34 → " " | 44 → , | 63 → ? |
| 35 → # | 45 → - | 64 → @ |
| 36 → \$ | 46 → . | 65-90 (A-Z) capital letters |
| 37 → % | 47 → / | 91 → [|
| 38 → & | 48 → 57 → (0-9) | 92 → \ |
| 39 → ' ' (left) | 58 → : | 93 →] |
| 40 → ' ' (right) | 59 → ; | 94 → ^ |
| | 60 → < | 95 → _ (underscore) |

- 96 → `
- 97-122 (a-z) small letters
- 123 → {
- 124 → | (vertical bar)
- 125 → }
- 126 → ~
- 0-31 → character control

Note:- Binary - Decimal - ASCII (Basic phenomena to do ASCII problems).



- ASCII codes are used in micro computers (or) personal computer
- EBCDIC codes are used in large computers.
- Hollerith code:- This code is used in system to represent alphanumeric information.
- It consists of 80 columns and 12 rows
- It is a 12-bit code.



→ Error correcting codes :- Codes which allow error detection and correction are called Error correcting codes.

Eg:- Hamming code.

→ Hamming code is a specific type of error correcting code that allows the detection and correction of single bit transmission errors. Hamming code algorithm can solve only single bit issues. These are used in satellite communication.

Ex:- Encode the data (or) message bits 0011 into the 7-bit even parity Hamming code.

Sol Given message = 0011
Number of message bits $M = 4$

Number of parity bits required is calculated using the formula

$$2^p \geq m+p+1$$

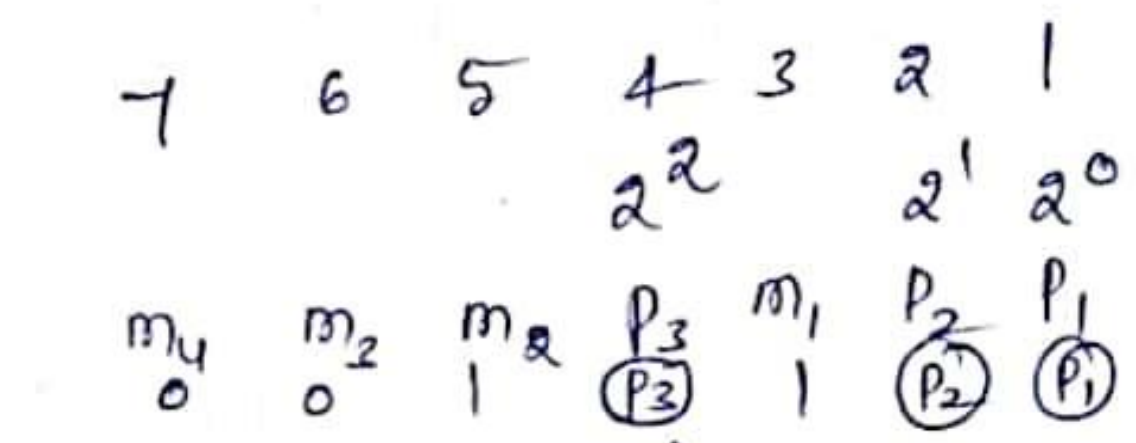
$$2^p \geq 4+p+1$$

$$2^3 \geq 4+3+1$$

$$8 \geq 8$$

Number of parity bits $P = 3$
Total no of bits $m+p = 4+3 = 7$

Decimal no	2^2	2^1	2^0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1



$$P_1 = 1 \ 3 \ 5 \ 7 = P_1 \ 110$$

$$= 0110 \ (\because P_1 = 0; \text{ to maintain the even parity})$$

$$\therefore P_1 = 0.$$

$$P_2 = 2, 3, 6, 7 = P_2 100 = 1100$$

To become the even parity ($\because P_2 = 1$)

$$\therefore P_2 = 0$$

$$P_3 = 4, 5, 6, 7 = P_3 100 = 1100$$

$$\therefore P_3 = 1$$

Error position = By combining the parity bits

$$P_3 P_2 P_1 = P_3 P_2 P_1 = 0110 = (6)_{10}$$

Error is located at 2nd position

$$\text{Total message bits} = 0 \textcircled{1} 1 1 1 0$$

$$\text{After correcting} = 011110.$$

Sol:

Generate Hamming code for the message 1110

$$2^p \geq p+m+1$$

$p =$ parity bits
 $m =$ message bits

$$2^p \geq p+4+1$$

$$2^p \geq p+5$$

p should be atleast 3 to satisfy the condition

$$2^3 \geq 3+5 \therefore 8 \geq 8 \text{ (true)}$$

1	2	3	4	5	6	7
2^0	2^1		2^2			
P_1	P_2	m_1	P_3	m_2	m_3	m_4
P_1	P_2	1	P_3	1	1	0

(The code may be any length the process will be same)

(For even parity)

$$P_1 \Rightarrow 1, 3, 5, 7 \rightarrow P_1 110 \rightarrow 0110 \quad (P_1 = 0)$$

$$P_2 \rightarrow 2, 3, 6, 7 \rightarrow P_2 110 \rightarrow 0110 \quad (P_2 = 0)$$

$$P_3 \rightarrow 4, 5, 6, 7 \rightarrow P_3 110 \rightarrow 0110 \quad (P_3 = 0)$$

Total message bits
 = 0 0 1 0 1 1 0

odd parity :-

$$P_1 \Rightarrow 1, 3, 5, 7 \rightarrow P_1 110 \rightarrow 1110 \quad (P_1 = 1)$$

$$P_2 \Rightarrow 2, 3, 6, 7 \rightarrow P_2 110 \rightarrow 1110 \quad (P_2 = 1)$$

$$P_3 \Rightarrow 4, 5, 6, 7 \rightarrow P_3 110 \rightarrow 1110 \quad (P_3 = 1)$$

Total message bits

$$= P_1 P_2 m_1 P_3 m_2 m_3 m_4$$

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

⇒ Error correction in Hamming code

Q3 A (7,4) hamming code is received as 1110000 determine the corrected code when even and odd parity.

sol

1	2	3	4	5	6	7
1	1	1	0	0	0	0

To ensure that error is there are not

$$E_1 \rightarrow 1, 3, 5, 7 \rightarrow 1100 \rightarrow \text{to make it even parity } E_1 = 0$$

(even parity) $E_2 \rightarrow 2, 3, 6, 7 \rightarrow 1100 \Rightarrow E_2 = 0$

$$E_3 \rightarrow 4, 5, 6, 7 \rightarrow 0000 \Rightarrow E_3 = 0$$

$$\text{Error} = E_3 E_2 E_1 = 000 \quad (0^{\text{th}} \text{ position})$$

odd parity

$$E_1 \rightarrow 1, 3, 5, 7 \rightarrow 1100 \rightarrow E_1 = 1 \quad (\text{to make it odd parity})$$

$$E_2 \rightarrow 2, 3, 6, 7 \rightarrow 1100 \rightarrow E_2 = 1 \quad (\text{to make it odd parity})$$

$$E_3 \rightarrow 4, 5, 6, 7 \rightarrow 0000 \rightarrow E_3 = 1$$

$$\text{Error} = E_3 E_2 E_1 = 111 \quad \text{7th position error is there}$$

Corrected code may be 1110001

Q Determine the single error-correcting code for the information code 10111 for odd parity

STEP 1

Sol Given message bit $m = 10111$

By using trial and error method we should find parity bits

$2^p \geq n+p+1$ $\therefore n = m$

$2^1 \geq 5+1+1$ Let $p=1$

$2 \geq 7$ ✗

$2^2 \geq 5+2+1$ Let $p=2$

$2^2 \geq 8$ ✗

$2^3 \geq 5+3+1$ Let $p=3$

$8 \geq 9$ ✗

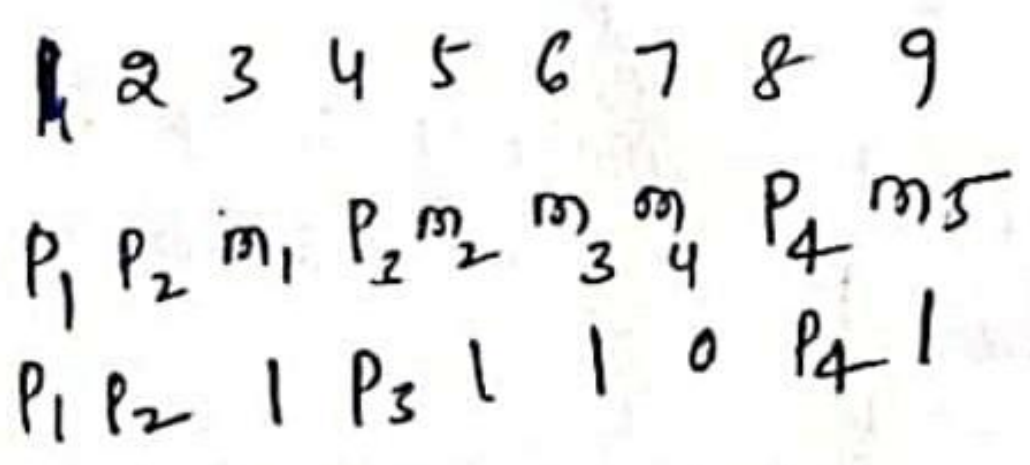
$2^4 \geq 5+4+1$ Let $p=4$

$16 \geq 10$ ✓

So, we need 4 parity bits. We should take parity bits always powers of 2.

$2^0 = 1; 2^1 = 2, 2^2 = 4, 2^3 = 8$
 $2^4 = 16; 2^5 = 32$ and soon.

Find the value to the parity



$\therefore m = 10111$
 $m_5 m_4 m_3 m_2 m_1$

Bit destination	m_5	P_8	m_4	m_3	m_2	P_4	m_1	P_2	P_1
Bit location	9	8	7	6	5	4	3	2	1
Information bits	1001	1000	0111	0110	0101	0100	0011	0010	0001
Parity bits	1	?	0	1	1	?	1	?	?
Received code	1	0	0	1	1	1	1	1	0

$P_1 \rightarrow P_1 m_1 m_2 m_4 m_5 = P_1 1101$

To make it become odd we kept $P_1 = 0$

$P_2 \rightarrow P_2 m_1 m_3 m_4 = P_2 110$ To make it become odd

we kept $P_2 = 1$; so $P_2 = 1$

$P_4 \rightarrow P_4 m_2 m_3 m_5 = P_4 110$ To make it odd

we kept $P_4 = 1$; $P_4 = 1$

$P_8 \rightarrow P_8 m_5 = P_8 1$ To make it odd

we kept $P_8 = 0$; $P_8 = 0$

Error position $P_8 P_4 P_2 P_1$
 $= 0110 = 6^{th}$ position

like is 9 bit hamming code

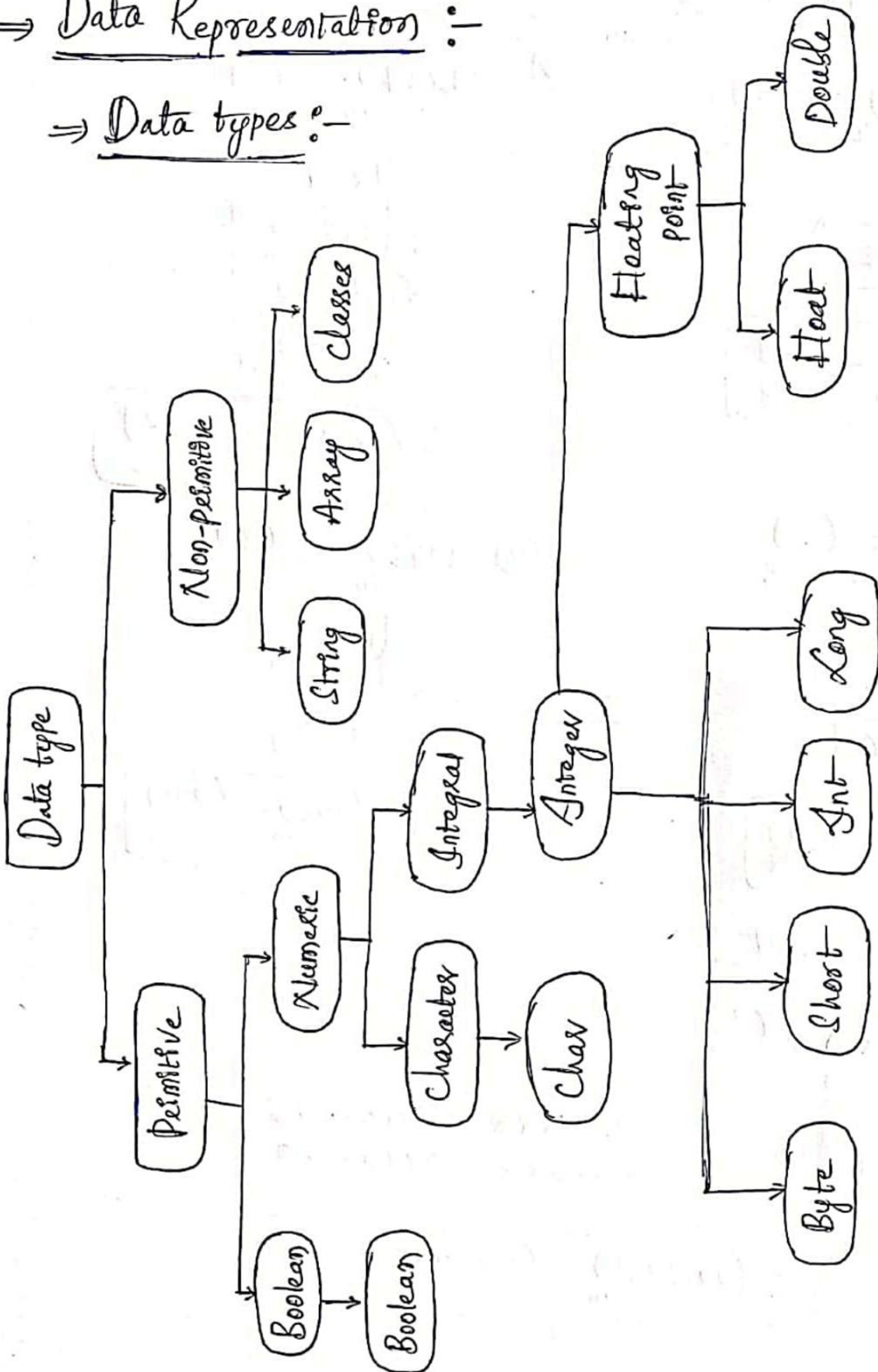
9 8 7 6 5 4 3 2 1
 100111110

error in 6th position

100011110

⇒ Data Representation :-

⇒ Data types :-



⇒ Decimal to octal :-

$$\textcircled{1} \quad (20)_{10} = (?)_8$$

$$\begin{array}{r} 8 \overline{) 20} \\ \underline{2-4} \end{array} \uparrow$$

$$\therefore (20)_{10} = (24)_8$$

$$\textcircled{2} \quad (1234)_{10} = (?)_8$$

$$\begin{array}{r} 8 \overline{) 1234} \\ \underline{154-2} \\ 8 \overline{) 19-2} \\ \underline{2-3} \end{array} \uparrow$$

$$\therefore (1234)_{10} = (2322)_8$$

$$\textcircled{3} \quad (183)_{10} = (?)_8$$

$$\begin{array}{r} 8 \overline{) 183} \\ \underline{22-7} \\ 2-6 \end{array} \uparrow$$

$$\therefore (183)_{10} = (267)_8$$

$$\textcircled{4} \quad (145)_{10} = (?)_8$$

$$\begin{array}{r} 8 \overline{) 145} \\ \underline{18-1} \\ 2-2 \end{array} \uparrow$$

$$\therefore (145)_{10} = (221)_8$$

⇒ Fractional part :-

$$\textcircled{1} \quad (27.625)_{10} = (?)_8$$

$$\begin{array}{r} 8 \overline{) 27} \\ \underline{3-3} \end{array} \uparrow$$

$$\begin{aligned} 0.625 \times 8 &= 5.000 \rightarrow 5 \\ 0.000 \times 8 &= 0.000 \rightarrow 0 \end{aligned}$$

$$\therefore (27.625)_{10} = (33.50)_8$$

② $(3287.5100098)_{10} = (?)_8$

Sol

Integer part

$$\begin{array}{r} 8 \overline{) 3287} \\ \underline{410} \\ 8 \overline{) 410} \\ \underline{51} \\ 8 \overline{) 51} \\ \underline{6} \end{array}$$

$$\begin{array}{l} 0.5100098 \times 8 = 4.0800 \rightarrow 4 \\ 0.0800 \times 8 = 0.640 \rightarrow 0 \\ 0.640 \times 8 = 5.125 \rightarrow 5 \\ 0.125 \times 8 = 1.000 \rightarrow 1 \end{array}$$

$\therefore (3287.5100098)_{10} = (6327.4051)_8$

③ $(20.5)_{10} = (?)_8$

$$\begin{array}{r} 8 \overline{) 20} \\ \underline{2} \end{array}$$

Fractional part

$$\begin{array}{l} 0.5 \times 8 = 4.0 \rightarrow 4 \\ 0.0 \times 8 = 0.0 \rightarrow 0 \end{array}$$

$\therefore (20.5)_{10} = (24.40)_8$

④ $(60.7)_{10} = (?)_8$

$$\begin{array}{r} 8 \overline{) 60} \\ \underline{7} \end{array}$$

$$\begin{array}{l} 0.7 \times 8 = 5.6 \rightarrow 5 \\ 0.6 \times 8 = 4.8 \rightarrow 4 \\ 0.8 \times 8 = 6.4 \rightarrow 6 \\ 0.4 \times 8 = 3.2 \rightarrow 3 \\ 0.2 \times 8 = 1.6 \rightarrow 1 \end{array}$$

$\therefore (60.7)_{10} = (74.54631)_8$

⇒ Decimal to Hexadecimal :- (H=16)

① $(20)_{10} = (?)_H$

$$\begin{array}{r} 16 \overline{) 20} \\ \underline{16} \\ 4 \end{array}$$

$$\therefore (20)_{10} = (14)_H$$

② $(1234)_{10} = (?)_H$

$$\begin{array}{r} 16 \overline{) 1234} \\ \underline{1120} \\ 114 \\ \underline{112} \\ 20 \\ \underline{16} \\ 4 \end{array}$$

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

③ $(20.5)_{10} = (?)_H$

$$\begin{array}{r} 16 \overline{) 20} \\ \underline{16} \\ 4 \end{array}$$

$$\begin{array}{l} 0.5 \times 16 = 8.0 \rightarrow 8 \\ 0.0 \times 16 = 0.0 \rightarrow 0 \end{array}$$

$$\therefore (20.5)_{10} = (14.8)_{16}$$

④ $(675.625)_{10} = (?)_{16}$

$$\begin{array}{r} 16 \overline{) 675} \\ \underline{640} \\ 35 \\ \underline{32} \\ 3 \\ \underline{2} \\ 1 \end{array}$$

(or) A

$$\begin{array}{l} 0.625 \times 16 = 10.000 \rightarrow 10 \text{ (or) } A \\ 0.000 \times 16 = 0.000 \rightarrow 0 \end{array}$$

$$\therefore (675.625)_{10} = (2A3.A)_{16}$$

⇒ Binary to octal :-

To convert binary to octal, starting from binary point make group of 3 bits and write its equivalent

$$\textcircled{1} (101)_2 = (?)_8$$

$$\begin{array}{c} 421 \\ 101 \rightarrow 5 \end{array}$$

$$\therefore (101)_2 = (5)_8$$

$$\textcircled{2} (1101)_2 = (?)_8$$

$$\begin{array}{c} 001101 = 15 \\ \underbrace{\quad} \quad \underbrace{\quad} \\ 1 \quad 5 \end{array}$$

$$\therefore (1101)_2 = (15)_8$$

$$\textcircled{3} (10.11001)_2 = (?)_8$$

$$\leftarrow 10.11001 \rightarrow$$

$$\begin{array}{c} 010.110010 \\ \downarrow \quad \downarrow \quad \downarrow \\ 2 \quad 6 \quad 2 \end{array}$$

$$\therefore (10.11001)_2 = (2.62)_8$$

$$\textcircled{4} (011010110.11)_2 = (?)_8$$

$$\leftarrow 011010110.11 \rightarrow$$

$$\begin{array}{c} 011010110.110 \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ 3 \quad 2 \quad 6 \quad 6 \end{array}$$

$$\therefore (011010110.11)_2 = (326.6)_8$$

$$\textcircled{5} (1101101.01101)_2 = (?)_8$$

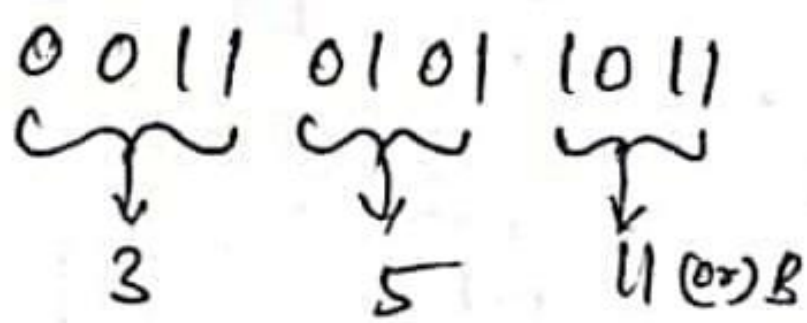
$$\begin{array}{c} \text{append zero.} \quad \text{append zero} \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ 001101 \quad 101. \quad 011010 \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ 1 \quad 5 \quad 5 \quad 3 \quad 2 \end{array}$$

$$\therefore (1101101.01101)_2 = (155.32)_8$$

⇒ Binary to Hexadecimal :-

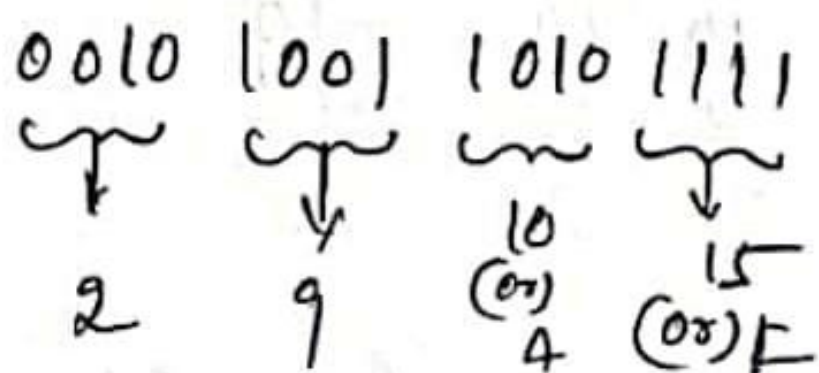
To convert binary to hexadecimal, Group 4-bits of binary and write its equivalent hexadecimal digit.

$$\textcircled{1} (1101011011)_2 = (?)_{16}$$



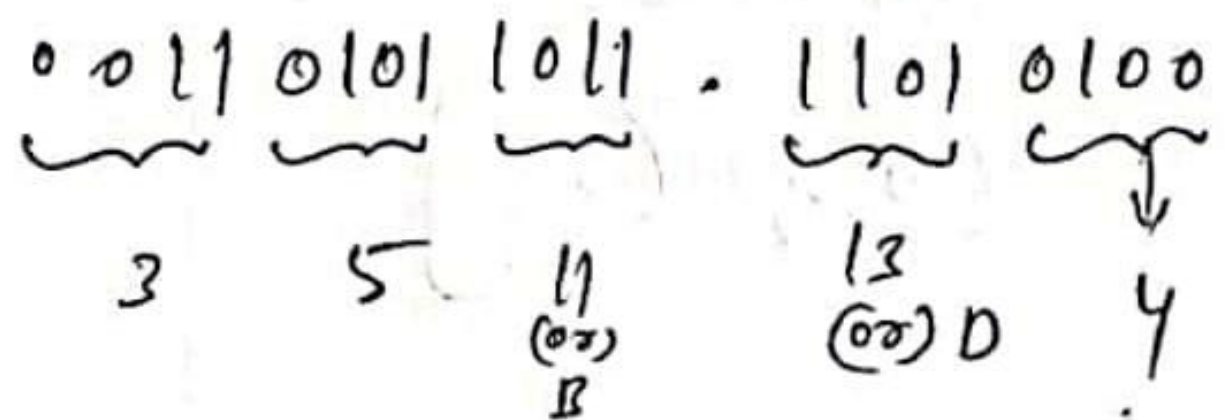
$$\therefore (1101011011)_2 = (35B)_{16}$$

$$\textcircled{3} (10100110101111)_2 = (?)_{16}$$



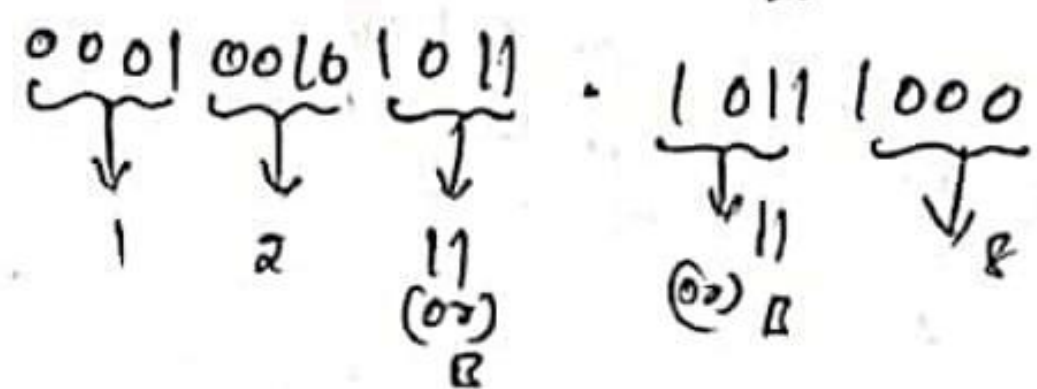
$$\therefore (10100110101111)_2 = (29AF)_{16}$$

$$\textcircled{2} (1101011011.110101)_2 = (?)_{16}$$



$$\therefore (1101011011.110101)_2 = (35B.D4)_{16}$$

$$\textcircled{4} (100101011.101110)_2 = (?)_{16}$$



$$\therefore (100101011.101110)_2 = (12B.B8)_{16}$$

⇒ Octal to Other Number Systems :-

⇒ Octal to Decimal :-

① $(24)_8 = (?)_{10}$

2	4
8^1	8^0

$$2 \times 8^1 + 4 \times 8^0$$

$$= 16 + 4 = 20$$

$$\therefore (24)_8 = (20)_{10}$$

② $(24.4)_8 = (?)_{10}$

2	4.	4
8^1	8^0	8^{-1}

$$= 2 \times 8^1 + 4 \times 8^0 + 4 \times 8^{-1}$$

$$= 16 + 4 + 4 \times \frac{1}{8}$$

$$= 16 + 4 + 0.5 = 20.5$$

$$\therefore (24.4)_8 = (20.5)_{10}$$

③ $(6327.4051)_8 = (?)_{10}$

6	3	2	7	.	4	0	5	1
8^3	8^2	8^1	8^0	.	8^{-1}	8^{-2}	8^{-3}	8^{-4}

$$= 6 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times \frac{1}{8}$$

$$+ 0 \times \frac{1}{8^2} + 5 \times \frac{1}{8^3} + \frac{1 \times 1}{8^4}$$

$$= 3072 + 192 + 96 + 7 + 0.5100098$$

$$= (3367.5100098)_{10}$$

④ $(1234.242)_8 = (?)_{10}$

1	2	3	4	.	2	4	2
8^3	8^2	8^1	8^0	.	8^{-1}	8^{-2}	8^{-3}

$$1 \times 8^3 + 2 \times 8^2 + 3 \times 8^1 + 4 \times 8^0 + 2 \times \frac{1}{8} + 4 \times \frac{1}{8^2}$$

$$+ 2 \times \frac{1}{8^3}$$

$$= 512 + 128 + 24 + 4 + 0.375 + 0.062 + 0.003$$

$$= (668.440)_{10}$$

⇒ Octal to Binary :- To convert octal to binary just replace each octal digit by its 3-bit binary equivalent.

$$\textcircled{1} \quad (15)_8 = (?)_2$$

$$1 \rightarrow 001$$

$$5 \rightarrow 101$$

$$\therefore (15)_8 = (001101)_2$$

$$\textcircled{2} \quad (736)_8 = (?)_2$$

$$7 \rightarrow 111$$

$$3 \rightarrow 011$$

$$6 \rightarrow 110$$

$$\therefore (736)_8 = (111011110)_2$$

$$\textcircled{3} \quad (563)_8 = (?)_2$$

$$5 \rightarrow 101$$

$$6 \rightarrow 110$$

$$3 \rightarrow 011$$

$$\therefore (563)_8 = (101110011)_2$$

$$\textcircled{4} \quad (725)_8 = (?)_2$$

$$7 \rightarrow 111$$

$$2 \rightarrow 010$$

$$5 \rightarrow 101$$

$$\therefore (725)_8 = (111010101)_2$$

$$\textcircled{5} \quad (326)_8 = (?)_2$$

$$3 \rightarrow 011$$

$$2 \rightarrow 010$$

$$6 \rightarrow 110$$

$$(326)_8 = (011010110)_2$$

$$\textcircled{6} \quad (452)_8 = (?)_2$$

$$4 \rightarrow 100$$

$$5 \rightarrow 101$$

$$2 \rightarrow 010$$

$$\therefore (452)_8 = 100101010$$

⇒ Octal to Hexadecimal :- There is no direct conversion available for octal to hexadecimal. To convert octal number into a hexadecimal number by converting octal to binary then binary to hexadecimal (or) octal to decimal then decimal to hexadecimal.

Note: $()_8 \rightarrow ()_2 \rightarrow ()_{16}$
 $()_8 \rightarrow ()_{10} \rightarrow ()_{16}$

① $(356.63)_8 = ()_{16}$

Step ① Octal to Binary

Step ② Binary to Hexadecimal

$$\begin{array}{cccccc} 3 & 5 & 6 & . & 6 & 3 \\ \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow \\ 011 & 101 & 110 & . & 110 & 011 \end{array}$$

$$\begin{array}{cccccc} 0000 & 110 & 1110 & . & 1100 & 1100 \\ \hline 0 & E & E & . & C & C \\ & =14 & =14 & & =12 & =12 \end{array}$$

∴ $(356.63)_8 = (0EE.CC)_{16}$

② $(247.52)_8 = ()_{16}$

Step ① Octal to Binary

Step ② Binary to Hexadecimal

$$\begin{array}{cccccc} 2 & 4 & 7 & . & 5 & 2 \\ \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow \\ 010 & 100 & 111 & . & 101 & 010 \end{array}$$

$$\begin{array}{cccccc} 0000 & 1010 & 0111 & . & 1010 & 1000 \\ \hline 0 & A & 7 & . & A & 8 \\ & (or) A & & & (or) A & \end{array}$$

∴ $(247.52)_8 = (0A7.A8)_{16}$

⇒ Hexadecimal to other Number system :-

⇒ Hexadecimal to binary :-

① $(2F9A)_{16} = ()_2$

$$\begin{array}{l} 2 \rightarrow 0010 \\ F \rightarrow 1111 \\ 9 \rightarrow 1001 \\ A \rightarrow 1010 \end{array}$$

∴ $(2F9A)_{16} = (0010111110011010)_2$

$$(2) (6A3)_{16} = ()_2$$

$$6 \rightarrow 0110$$

$$A \rightarrow 1010$$

$$3 \rightarrow 0011$$

$$\therefore (6A3)_{16} = (011010100011)_2$$

$$(3) (58C)_{16} = ()_2$$

$$5 \rightarrow 0101$$

$$8 \rightarrow 1000$$

$$C \rightarrow 1100$$

$$\therefore (58C)_{16} = (010110001100)_2$$

$$(4) (7DE3)_{16} = ()_2$$

$$7 \rightarrow 0111$$

$$D \rightarrow 1101$$

$$E \rightarrow 1110$$

$$3 \rightarrow 0011$$

$$\therefore (7DE3)_{16} = (0111110111100011)_2$$

Hexadecimal to Decimal

$$(1) (3A.2F)_{16} = ()_{10}$$

3	A	.	2	F
16^1	16^0	.	16^{-1}	16^{-2}

$$3 \times 16^1 + 10 \times 16^0 + 2 \times 16^{-1} + 15 \times 16^{-2}$$

$$= 48 + 10 + \frac{2}{16} + \frac{15}{16^2}$$

$$\therefore (3A.2F)_{16} = (58.1836)_{10}$$

$$(2) (5E.7A)_{16} = ()_{10}$$

5	E	.	7	A
16^1	16^0	.	16^{-1}	16^{-2}

$$5 \times 16^1 + 14 \times 16^0 + 7 \times \frac{1}{16^1} + 10 \times \frac{1}{16^2}$$

$$= 90 + 14 + 0.43 + 0.03$$

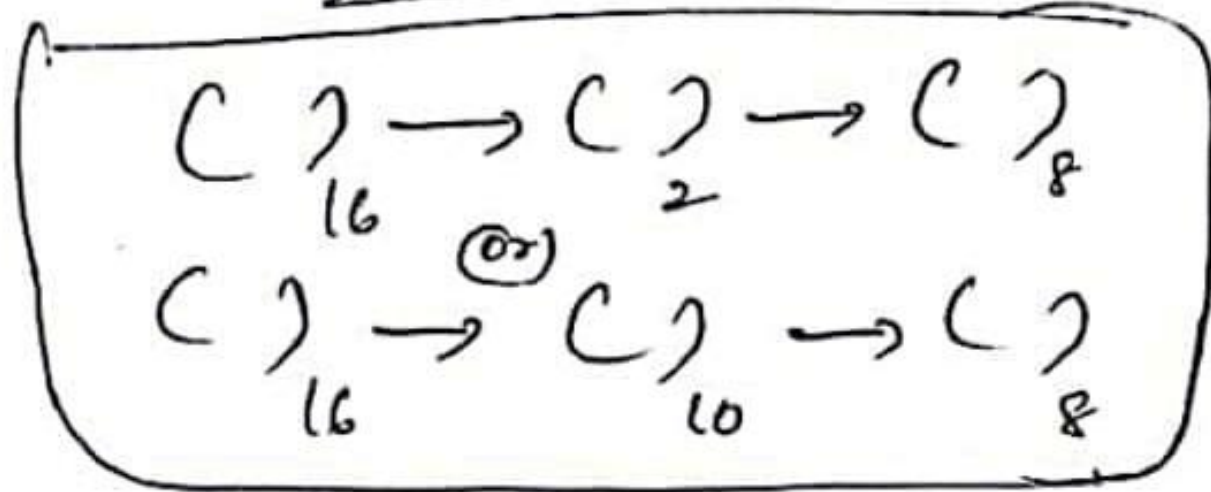
$$= (104.46)_{10}$$

$$\therefore (5E.7A)_{16} = (104.46)_{10}$$

⇒ Hexadecimal to Octal Conversion :-

No direct conversion available, to convert hexadecimal to octal first convert given hexadecimal number into decimal/binary then into octal system.

Note :-



① $(B9F.AE)_{16} = ()_8$

- ✓ Hexadecimal to binary
- ✓ Binary to Octal

$$\begin{array}{c|c|c|c|c|c} B & 9 & F & . & A & E \\ \hline 1011 & 1001 & 1111 & . & 1010 & 1110 \end{array} = \begin{array}{cccccc} \underline{101} & \underline{110} & \underline{011} & \underline{111} & \underline{101} & \underline{011} & \underline{100} \\ 5 & 6 & 3 & 7 & 5 & 3 & 4 \end{array}$$

$\therefore (B9F.AE)_{16} = (5637.534)_8$

② $(A8C.BC7)_{16} = ()_8$

- ✓ Hexadecimal to binary
- ✓ Binary to Octal

$$\begin{array}{c|c|c|c|c|c} A & 8 & C & . & B & C & 7 \\ \hline 1010 & 1000 & 1100 & . & 1011 & 1100 & 0111 \end{array} =$$

$$\begin{array}{cccccc} \underline{001} & \underline{010} & \underline{100} & \underline{011} & \underline{100} & \underline{101} & \underline{111} & \underline{000} & \underline{111} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 2 & 4 & 3 & 4 & 5 & 7 & 0 & 7 \end{array}$$

$\therefore (A8C.BC7)_{16} = (12434.5707)_8$

Complement of Numbers :

(or) $(r-1)$'s complement and r 's complement

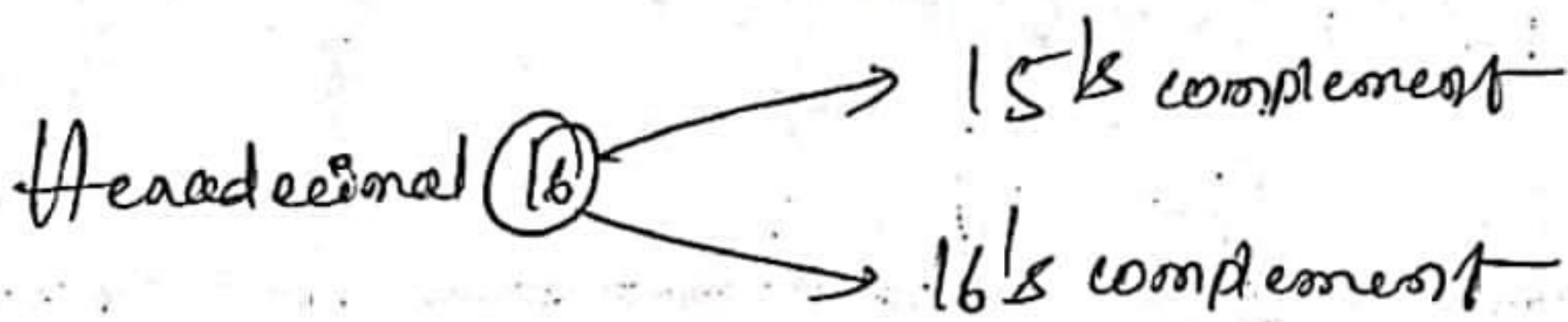
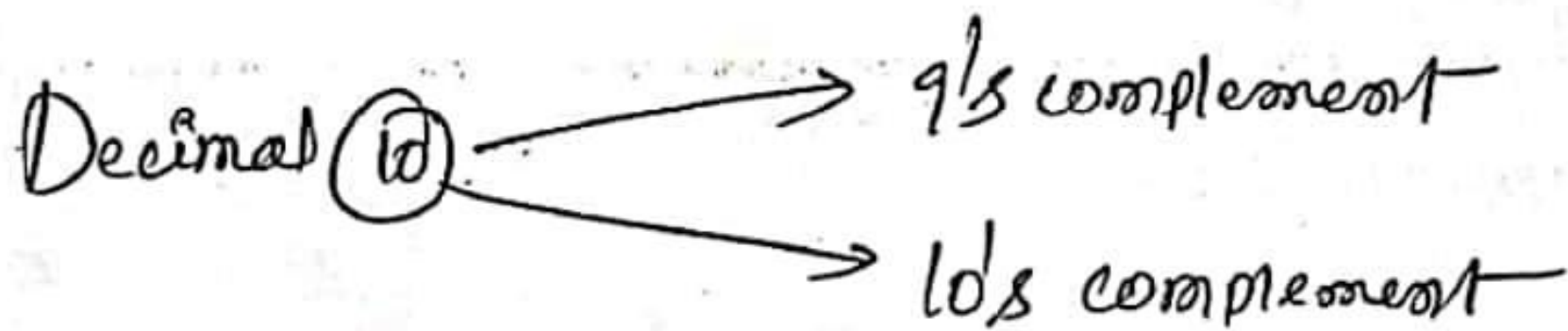
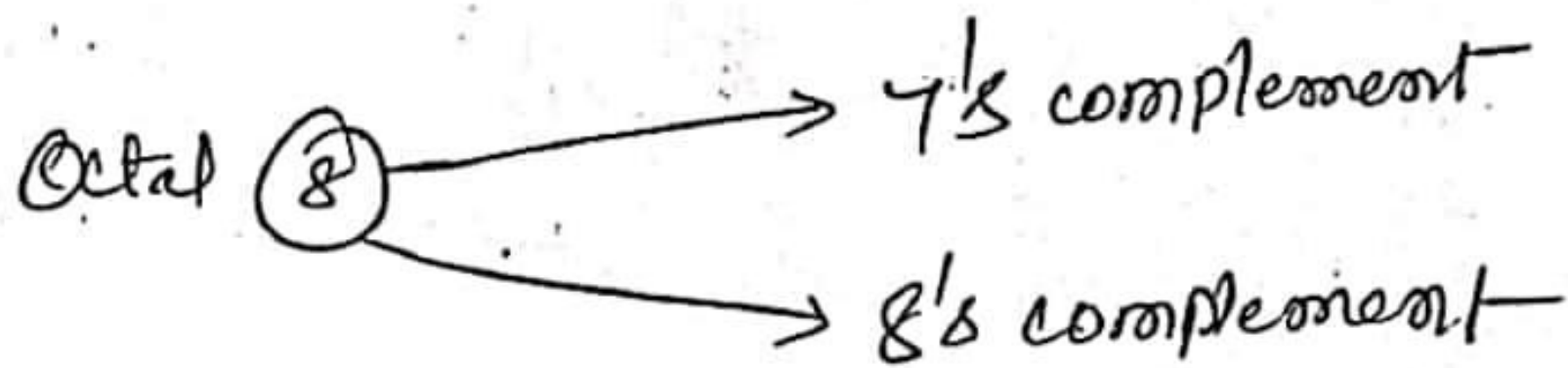
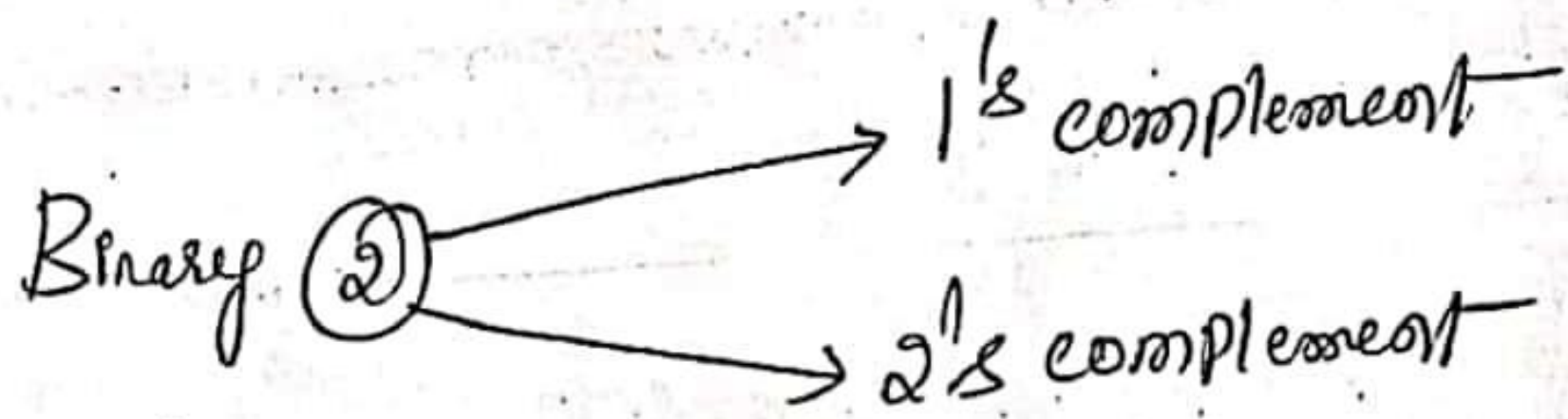
Complements are used in systems to simplify the subtraction operation base (radix) r system there are two useful types of complements, r 's complement (Radix Complement) and $(r-1)$'s complement (Diminished Radix Complement).

$(r-1)$'s complement :-

For a given number 'N' have the no. of digits 'n' belonging to 'r' number system, then $(r-1)$ complement is given by $(r^n - N) - 1$

r 's complement :-

For a given number 'N' have the no. of digits 'n' belonging to 'r' number system, then r 's complement is given by $r^n - N$



⇒ 1's and 2's complements

The 1's complement of a binary number is obtained complementing all its bits, that is by replacing all 0's by 1's and all 1's by 0's.

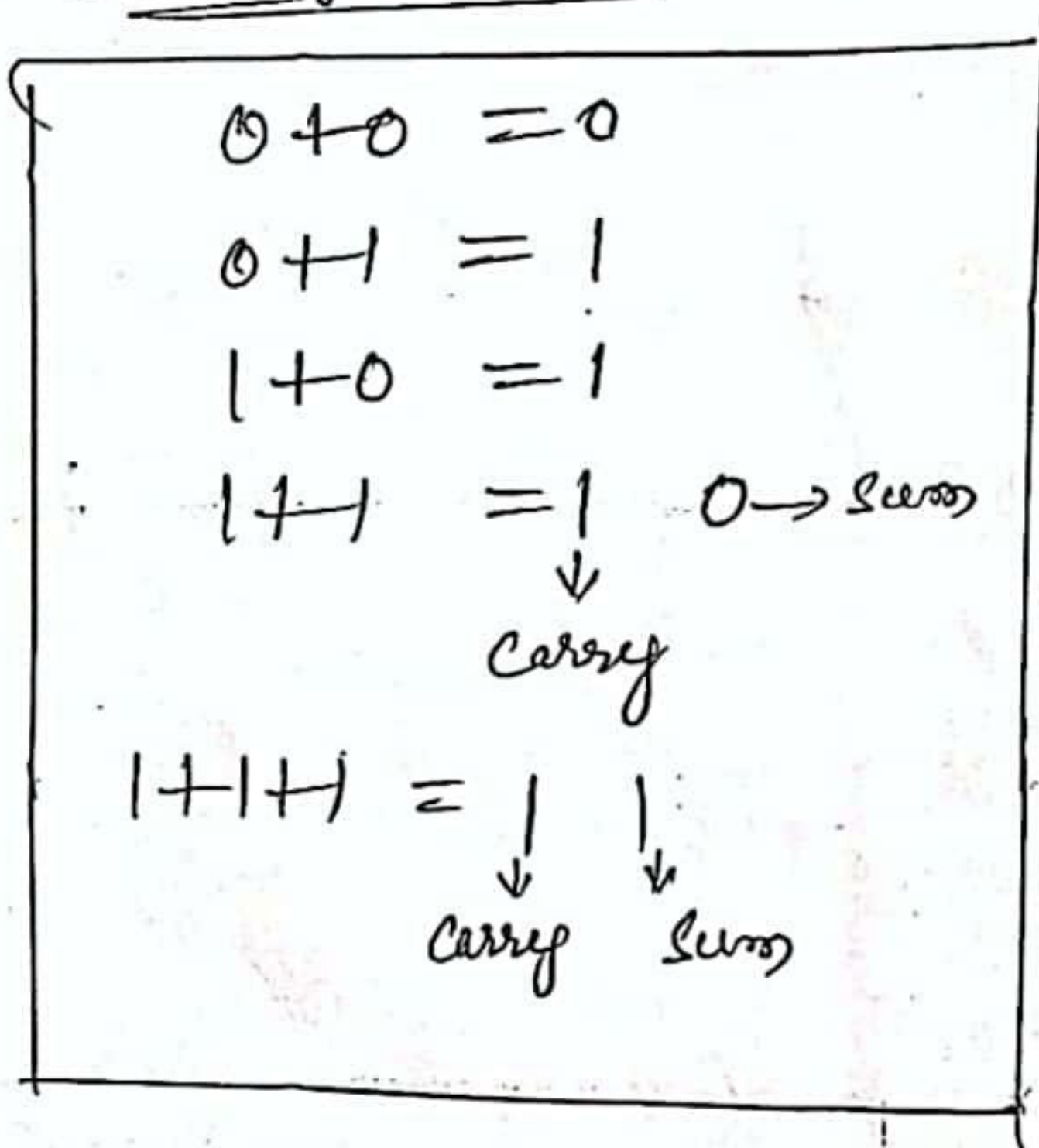
Ex: (1) 0101101000 → (Given Binary Number)
 1010010111 → (1's complement form)

The 2's complement of a binary number is obtained by adding '1' to its 1's complement.

Ex: ①

$$\begin{array}{r}
 0101101000 \rightarrow \text{(Given binary number)} \\
 1010010111 \rightarrow \text{(1's complement form)} \\
 \text{ } \downarrow \text{(adding 1)} \\
 \hline
 1010011000 \rightarrow \text{2's complement}
 \end{array}$$

⇒ Binary addition



②

$$\begin{array}{r}
 011010101 \rightarrow \text{Given binary number} \\
 100101010 \rightarrow \text{1's complement form} \\
 + 1 \rightarrow \text{adding '1'} \\
 \hline
 100101011 \rightarrow \text{2's complement form}
 \end{array}$$

⇒ 1's complement Arithmetic

- ⇒ In 1's complement subtraction, add the 1's complement of the subtrahend to the minuend.
- ⇒ If there is a carryout, bring the carry around and add it to the LSB. This is called "End around carry".
- ⇒ Look at the sign bit (MSB). If MSB is a '0', the result is positive and is in true binary. If MSB is '1', the result is negative and is in its 1's complement form. Take its 1's complement and put -ve sign to get magnitude in binary.

Ex ①

① Subtract 14 from 25 using 8-bit 1's complement method.

Sol Normally

$$\begin{array}{r} 25 \rightarrow 00011001 \\ -14 \rightarrow 11110001 \\ \hline +11 \end{array}$$

$$\begin{array}{r} 14 \rightarrow 00001110 \\ 1's \text{ complement} \rightarrow 11110001 \end{array}$$

$$\hline \boxed{1}00001010$$

1 → (adding of End around carry)

End around carry

$$\hline 00001011$$

The MSB is 0, so the result is positive and is in pure binary. Therefore, the result is $00001011 = +11_{10}$

② Add -25 to +14 using 8-bit 1's complement method

$$\begin{array}{r}
 +14 \rightarrow 00001110 \\
 -25 \rightarrow 11100110 \\
 \hline
 -11 \rightarrow 11101000 \rightarrow \text{NO carry}
 \end{array}$$

$25 \rightarrow 00011001$
 $1's \rightarrow 11100110$
 Complement

⇒ There is no carry. The MSB is a '1'. So, the result is negative and is in the 1's complement form. Take 1's complement indicates -ve sign

⇒ The complement of 11101000 is 00010111. The result is -11₁₀

③ Add -25 to -14 using 8-bit 1's complement method

$$\begin{array}{r}
 -25 \rightarrow 11100110 \text{ (1's complement)} \\
 -14 \rightarrow 11110001 \text{ (1's complement)} \\
 \hline
 -39 \rightarrow 11101001
 \end{array}$$

$25 \rightarrow 00011001$ (Normal form)
 $14 \rightarrow 00001011$ (Normal form)

End around carry \rightarrow 11010111
 Adding 9 and around carry \rightarrow 1111
 \rightarrow 11011000
 MSB \rightarrow 00100111 \rightarrow 1's complement

→ The MSB is '1'. So the result is negative and we should find 1's complement above answer. The 1's complement of 11011000 is 00100111 therefore the result is -39.

① Add +25 to +14 using 8-bit 1's complement arithmetic.

$$\begin{array}{r}
 +25 \rightarrow 00011001 \\
 +14 \rightarrow 00001110 \\
 \hline
 +39 \rightarrow 00100111
 \end{array}$$

There is no carry. The MSB is '0'. So, the result is positive and is in pure binary. Therefore, the result is +39₁₀.

Ex: ②

1) Subtract 20 from 36 using 8-bit 1's complement form

$$\begin{array}{r}
 36 \rightarrow 00100100 \\
 -20 \rightarrow 11101011 \text{ (1's complement)} \\
 \hline
 +16 \rightarrow 00001111 \\
 \hline
 \text{End around carry} \leftarrow 11111111 \text{ (adding of end around carry)} \\
 \hline
 00010000 \\
 \hline
 \text{MSB is 0}
 \end{array}$$

⇒ The MSB is zero. The result is positive and it is in True Binary form.

② Add +36 to +20 using 8-bit 1's complement form

$$\begin{array}{r}
 +36 \rightarrow 000100100 \\
 +20 \rightarrow 00010100 \\
 \hline
 \text{MSB is 0} \rightarrow 00100100
 \end{array}$$

MSB is zero. the result is positive and it is in true binary form

3 Add -36 to -20 using 8-bit 1's complement form.

$ \begin{array}{r} -36 \rightarrow 11011011 \rightarrow \text{1's complement} \\ -20 \rightarrow 11101011 \\ \hline -56 \rightarrow 11000111 \\ \text{End around carry} \leftarrow \boxed{1} \rightarrow \text{adding of end around carry} \\ \hline 11000111 \\ \hline \downarrow \\ \text{MSB} = 1 \end{array} $	$ \begin{array}{r} 36 \rightarrow 00100100 \\ 20 \rightarrow 00010100 \end{array} $
---	--

MSB is 1 the result is negative and it is in 1's complement form. To get the correct result take 1's complement to the result and put -ve sign before the result.

$$11000111 \xrightarrow{\text{1's complement}} -00111000 = -(56)_{10}$$

4 Add -36 to +20 using 8-bit 1's complement form

$ \begin{array}{r} -36 \xrightarrow{\text{1's comple}} 11011011 \\ +20 \xrightarrow{\text{Normal}} 00010100 \\ \hline -16 \\ \hline 11101111 \\ \text{MSB} = 1 \end{array} $	$ \begin{array}{r} 36 \rightarrow 00100100 \\ 20 \rightarrow 00010100 \end{array} $
--	--

the MSB is '1'. the result is negative and it is in 1's complement form.

Take 1's complement to the result and put -ve sign before the result

$$11101111 \rightarrow -00010000 = (-16)_{10}$$

\Rightarrow 2's complement :- In 2's complement subtraction, add 2's complement of the subtrahend to the minuend. If there is a carryout, ignore it. If the MSB is '0', the result is positive and is in true binary form. If MSB is '1' the result is negative and is in its 2's complement form.

Ex:
 ① Subtract 14 from 25 using 8-bit 2's complement arithmetic

$$\begin{array}{r}
 +14 = 00001110 \text{ (normal format)} \\
 \quad \quad 11110001 \text{ (1's complement)} \\
 \quad \quad \quad \quad 11 \text{ (2's complement)} \\
 \hline
 11110010 \rightarrow \text{2's complement form}
 \end{array}$$

$$\begin{array}{r}
 25 \rightarrow 00011001 \\
 -14 \rightarrow 11110010 \\
 \hline
 100001011
 \end{array}$$

Ignore carry MSB

\rightarrow There is a carry ignore it. The MSB is '0', so, the result is positive and is in normal binary form. Therefore, the result is $+0001011 = +11_{10}$

② Add -25 to +14 using 8-bit 2's complement arithmetic

$$\begin{aligned}
 +25 &\rightarrow 00011001 \\
 -25 &\rightarrow 11100110 \rightarrow \text{1's complement form} \\
 &\quad \underline{00111} \rightarrow \text{2's complement form}
 \end{aligned}$$

$$\begin{aligned}
 +14 &\rightarrow 00001110 \\
 -25 &\rightarrow 11100111 \\
 &\quad \underline{00111} \\
 &\quad \underline{0101} \quad (\text{No carry}) \\
 &\quad \uparrow \\
 &\quad \text{MSB} = 1
 \end{aligned}$$

There is no carry, the MSB is '1'. So, the result is negative and is in 2's complement form. The magnitude is 2's complement of 1110101, that is $\underline{0001010} - \underline{0001011} = (-11)_{10}$

Imp Ex

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform subtraction (a) $X - Y$ (b) $Y - X$ using 2's complements

$$\begin{aligned}
 X = 1010100 \quad Y = 1000011 &\rightarrow \text{subtrahend} \\
 &\quad \underline{100} \rightarrow \text{1's complement} \\
 &\quad \underline{101} \rightarrow \text{2's complement}
 \end{aligned}$$

- ① Find 2's complement of subtrahend
- ② Add subtrahend to the minuend.

$$X = 1010100$$

$$Y = 0111101 \rightarrow 2's \text{ complement of } Y$$

$$\begin{array}{r} 0111101 \\ \hline \end{array}$$

$$\boxed{1}000001$$

Discard carry
MSB = 0

The MSB = 0 the result is positive and it is in true binary form.

(b) $Y - X$

$$Y = 1000011 \quad X = 1010100$$

$$0101011 \rightarrow 1's \text{ complement}$$

$$\begin{array}{r} 0101011 \\ \hline \end{array}$$

$$0101100 \rightarrow 2's \text{ complement}$$

$$Y = 1000011$$

$$0101100 \rightarrow 2's \text{ complement of } X$$

$$\begin{array}{r} 1101111 \\ \hline \end{array}$$

$$MSB = 1$$

There is no carry. And the MSB is '1' so the answer is 2's complement form. So find 2's complement of the result to get the correct answer.

$$1101111$$

$$0010000 \rightarrow 1's \text{ complement}$$

$$\begin{array}{r} 1101111 \\ 0010000 \\ \hline 0010001 \end{array} \rightarrow \text{Correct answer}$$

2's complement form.

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction (a) $X - Y$ and (b) $Y - X$ using 1's complement.

(a) $X - Y$

$$X = 1010100$$

$$= 0111100 \rightarrow \text{1's complement of } Y$$

$$\begin{array}{r} 1111 \\ \hline 10010000 \end{array}$$

End around carry

$$\begin{array}{r} 1 \\ \hline 0010001 \end{array}$$

MSB = 0 so, it is in true binary form

(b) $Y - X$

$$Y = 1000011$$

$$X = 0101011 \rightarrow \text{1's complement of } Y$$

$$\begin{array}{r} 11 \\ \hline 1101110 \end{array}$$

MSB = 1

There is no carry. And the MSB is 1 so the result is in its complement form. So, find its complement of answer

$$\text{1's complement of } 1101110 \text{ is } \underline{\underline{0010001}}$$

⇒ 9's and 10's complement :-

→ In 9's complement subtraction just follow the below rules

- ① Find the 9's complement of subtrahend and Add 9's complement of subtrahend to minuend.
- ② If there is a carry it indicates that the answer is true then add carry to the LSD of the result to get true answer.
- ③ If there is no carry, it indicates that the answer is negative and the result obtained is its 9's complement.

④ Find the 9's complement of the following decimal number

① 3465

Sol

$$\begin{array}{r} 9999 \\ 3465 \\ \hline 6534 \end{array}$$

↓
 (9's complement of 3465)

By using formula

$$\left(10^4 - 3465 \right) - 1 = 6534$$

② 782.54

Sol

$$\begin{array}{r} 999.99 \\ 782.54 \\ \hline 217.45 \end{array}$$

③ 4526.075

Sol

$$\begin{array}{r} 9999.999 \\ 4526.075 \\ \hline 5473.924 \end{array}$$

9's complement Method of Subtraction :

Subtract the following numbers using 9's complement method :

① $745.81 - 436.62$

Step ①

$$\begin{array}{r} 999.99 \\ - 436.62 \\ \hline 563.37 \end{array} \rightarrow \text{9's complement of } 436.62$$

Step ②

$$\begin{array}{r} 745.81 \\ + 563.37 \\ \hline \boxed{-} 309.18 \end{array}$$

Carry indicates the answer is +ve

Carry indicates the answer is +ve

adding of undrawn carry

$$\begin{array}{r} 309.19 \end{array} \rightarrow \text{final answer.}$$

② $436.62 - 745.81$

Step ①

$$\begin{array}{r} 999.99 \\ - 745.81 \\ \hline 254.18 \end{array} \rightarrow \text{9's complement of } 745.81$$

Step ②

$$\begin{array}{r} 436.62 \\ + 254.18 \\ \hline 690.80 \end{array}$$

There is no carry, so it indicates that the answer is negative. So, take 9's complement of the intermediate result and put a minus sign before it.

$$\begin{array}{r}
 998.99 \\
 690.80 \\
 \hline
 - 309.19 \rightarrow \text{Therefore the answer is } \underline{-308.18}
 \end{array}$$

⇒ 10's complement method of subtraction

The 10's complement of a decimal number is obtained by adding a '1' to its 9's complement.

Q Find the 10's complement of the following decimal numbers.

① 3465

Sol

$$\begin{array}{r}
 9999 \\
 3465 \\
 \hline
 6534 \rightarrow \text{9's complement} \\
 \hline
 6535 \rightarrow \text{10's complement}
 \end{array}$$

② 782.54

Sol

$$\begin{array}{r}
 999.99 \\
 782.54 \\
 \hline
 217.45 \rightarrow \text{9's complement} \\
 \hline
 217.46 \rightarrow \text{10's complement}
 \end{array}$$

③ 4526.075

Sol

$$\begin{array}{r}
 9999.999 \\
 4526.075 \\
 \hline
 5473.924 \\
 \hline
 5473.925
 \end{array}$$

10's complement method of subtraction

- ① To perform decimal subtraction using 10's complement method, obtained the 10's complement of the subtrahend and add it to the minuend.
- ② If there is a carry, ignore it. The presence of the carry indicates that the answer is positive.
- ③ If there is no carry, it indicates the answer is negative and the result obtained in its 10's complement form and put negative sign in front of the answer.

① $745.81 - 436.62$

Step ①

$$\begin{array}{r}
 999.99 \\
 436.62 \\
 \hline
 563.37 \\
 \hline
 563.38 \rightarrow \text{10's complement form}
 \end{array}$$

Step ②

$$\begin{array}{r}
 745.81 \\
 563.38 \\
 \hline
 1309.19
 \end{array}$$

Ignore the carry
Carry indicates result is positive

② $436.62 - 745.81$

Step ①

$$\begin{array}{r}
 999.99 \\
 745.81 \\
 \hline
 254.18 \\
 \hline
 254.19
 \end{array}$$

Step ②

$$\begin{array}{r}
 436.62 \\
 254.19 \\
 \hline
 690.81 \rightarrow \text{no carry answer is negative}
 \end{array}$$

Step ③

$$\begin{array}{r}
 999.98 \\
 690.81 \\
 \hline
 309.18 \\
 \hline
 -309.18
 \end{array}$$

15's complement Method :-

Q Find the 15's complement of the following numbers

(a) 6A36

$$\begin{array}{r} \text{sol} \quad \text{FFFF} \\ (-) \quad 6A36 \\ \hline 95C9 \end{array} \quad (\text{or}) \quad \begin{array}{r} 15 \ 15 \ 15 \ 15 \\ 6 \ A \ 3 \ 6 \\ \hline 9 \ 5 \ C \ 9 \end{array}$$

(b) 9AD.3A

$$\begin{array}{r} 15 \ 15 \ 15 \ 15 \ 15 \\ (-) \quad 9 \ A \ D \ . \ 3 \ A \\ \hline 6 \ 5 \ 2 \ . \ C \ 5 \rightarrow (\text{15's complement}) \end{array}$$

⇒ 15's complement method of subtraction

(a) 69B - C14

Step ① 15's complement of (-C14)

$$\begin{array}{r} 15 \ 15 \ 15 \\ (-) \quad C \ 1 \ 4 \\ \hline 3 \ E \ B \rightarrow (\text{15's complement of } (-C14)) \end{array}$$

Step ②

$$69B - C14 = 69B + (\text{15's complement of } (-C14))$$

$$\begin{array}{r} 6 \ 9 \ B \\ (+) \quad 3 \ E \ B \\ \hline A \ 8 \ 6 \end{array} \quad \begin{array}{l} (22)_{10} = (16)_H \\ (24)_{10} = (18)_H \end{array}$$

There is no carry, result is (\rightarrow) ve

Step 3 15's complement of intermediate result is given by

$$\begin{array}{r} 15 \ 15 \ 15 \\ A \ 8 \ 6 \\ \hline 5 \ 7 \ 9 \\ \hline \end{array}$$

\therefore Final result is $-(579)_{16}$

(B) $-69B + C14$ or $C14 - 69B$

Step 1 15's complement of $(-69B)$

$$\begin{array}{r} 15 \ 15 \ 15 \\ - \ 6 \ 9 \ B \\ \hline 9 \ 6 \ 4 \rightarrow \text{15's complement} \\ \hline \end{array}$$

Step 2

$$C14 - 69B = C14 + (\text{15's complement of } (-69B))$$

$$\begin{array}{r} C14 \\ + \ 964 \\ \hline \textcircled{1} \ 578 \\ \hline \end{array} \quad (21)_{10} = (15)_{16}$$

carry

There is a carry, so the result is +ve

$$\begin{array}{r} 578 \\ + \ 1 \text{ (end around carry)} \\ \hline 579 \\ \hline \end{array}$$

\therefore Final result = $+(579)_{16}$

16's complement method :-

First find the 15's complement and then add 1

Q Find the 16's complement of the following number

(A) A8C

15's complement is given by

$$\begin{array}{r} 15 \ 15 \ 15 \\ \rightarrow A \ 8 \ C \\ \hline 5 \ 7 \ 3 \rightarrow 15's \ complement \end{array}$$

$$\begin{array}{r} 16's \ complement \rightarrow 573 \\ \quad \quad \quad \oplus 1 \\ \hline 574 \end{array}$$

⇒ 16's complement method of subtraction :-

Find the 16's complement subtraction of the following numbers :-

(a) C9B - C14

Step 1

15's complement of (-C14)

$$\begin{array}{r} 15 \ 15 \ 15 \\ C \ 1 \ 4 \\ \hline 3 \ E \ B \rightarrow 15's \ complement \end{array}$$

$$\begin{array}{r} 16's \ complement \rightarrow 3EB \\ \quad \quad \quad \oplus 1 \\ \hline 3EC \end{array}$$

Step 2 :-

$$C9B - C14 = C9B + (16\text{'s complement of } (-C14))$$

$$\begin{array}{r} C9B \\ + 3EC \\ \hline \textcircled{1}087 \\ \hline \end{array}$$

Carry

$$(23)_{10} = (17)_{16}$$

$$(24)_{10} = (18)_{16}$$

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

There is a carry ignore it. Since the carry is 1. The result is (+ve).

∴ Final result is $(087)_{16}$.

(B) $2A4.2D - 3B2.3C$

Step 1 :- $15\text{'s complement of } (-3B2.3C)$

$$\begin{array}{r} 15\ 15\ 15\ 15\ 15 \\ - 3\ B\ 2\ 3\ C \\ \hline C\ 4\ D\ C\ 3 \rightarrow 15\text{'s complement} \\ \hline \end{array}$$

16's complement is given by,

$$\begin{array}{r} C4D.C3 \\ 1 \\ \hline C4D.C4 \rightarrow 16\text{'s complement} \\ \hline \end{array}$$

Step 2

$$2A4.2D - 3B2.3C = 2A4.2D + (16\text{'s complement of } (-3B2.3C))$$

$$\begin{array}{r} 2A4.2D \\ + C4D.C4 \\ \hline EF1.F1 \rightarrow \text{intermediate result} \\ \hline \end{array}$$

there is no carry. so the result is (-ve)

Step 3 15's complement of intermediate result is given by

$$\begin{array}{r}
 15 \ 15 \ 15 \ 15 \ 15 \\
 E \ F \ 1 \ F \ 1 \\
 \hline
 1 \ 0 \ E \ 0 \ E \rightarrow 15's \ complement \\
 \oplus \ 1 \\
 \hline
 1 \ 0 \ E \ 0 \ F \rightarrow 16's \ complement
 \end{array}$$

∴ Final result is $-(0E.0F)_{16}$

⇒ 7 and 8's complements:

Subtract the following numbers using 7's complement method.

(a) $234.65 - 135.74$

Sol 7's complement of (-135.74) is given by

$$\begin{array}{r}
 777.77 \\
 135.74 \\
 \hline
 642.03 \rightarrow 7's \ complement
 \end{array}$$

∴ $234.65 + (7's \ complement \ of \ (-135.74))$

$$\begin{array}{r}
 234.65 \\
 642.03 \\
 \hline
 \text{Carry} \leftarrow \textcircled{1} 076.70
 \end{array}$$

$(8)_{10} = (10)_8$
(Carry ⇒ result is positive)

$$\begin{array}{r} 076.70 \\ (+) 1 \\ \hline 76.71 \text{ (End around carry)} \end{array}$$

∴ Result is +76.71

ⓑ $135.74 - 236.65$

7's complement of (-236.65) is given by,

$$\begin{array}{r} 777.77 \\ 236.65 \\ \hline 541.12 \rightarrow 7's \text{ complement} \end{array}$$

$$\therefore 135.74 - 236.65 = 135.74 + (7's \text{ complement of } (-236.65))$$

$$\Rightarrow \begin{array}{r} 135.74 \\ 541.12 \rightarrow (7's \text{ complement}) \\ \hline 677.06 \rightarrow \text{Intermediate result} \end{array}$$

There is no carry. Hence the final result is (-)ve.

Final result is the 7's complement of the above intermediate result

$$\begin{array}{r} 777.77 \\ \ominus 677.06 \\ \hline 100.71 \end{array}$$

∴ Final result is -100.71

Subtract the following using 8's complement method:

(29)

(a) $246.31 - 162.45$

7's complement of (-162.45) is given by,

$$777.77$$

$$162.45$$

$$\hline 615.32 \rightarrow (7's \text{ complement})$$

$$+ 1$$

$$\hline 615.33 \rightarrow (8's \text{ complement})$$

$$\therefore 246.31 - 162.45 = 246.31 + (8's \text{ complement of } (-162.45))$$

$$\Rightarrow 246.31$$

$$615.33$$

$$\hline 063.64$$

Carry

There is a carry. Hence the result is (+ve) and ignore the carry.

$$\therefore \text{Final result is } +63.64$$

(b) $162.45 - 246.31$

8's complement of (-246.31) is given by,

$$777.77$$

$$(-) 246.31$$

$$\hline 531.46 \rightarrow (7's \text{ complement})$$

$$+ 1$$

$$\hline 531.47 \rightarrow (8's \text{ complement})$$

$$\therefore 162.45 - 246.31 = 162.45 + (\text{8's complement of } (-246.31))$$

$$\Rightarrow \begin{array}{r} 162.45 \\ 531.47 \\ \hline 714.14 \end{array} \rightarrow (\text{Intermediate result})$$

there is no carry. Hence the final result is (-)ve. Final result is the 8's complement of the above intermediate result.

$$\begin{array}{r} 777.77 \\ 714.14 \\ \hline 063.63 \rightarrow 7's \text{ complement} \\ (+) 1 \\ \hline 063.64 \rightarrow 8's \text{ complement} \end{array}$$

$$\therefore \text{Final result is } \underline{-63.64}$$

⇒ Floating point Representation :-

The goal of floating point representation is represent a large range of numbers.

Ex :- Given the number -123.154×10^5

Sign = - (Negative) ①

Mantissa = 123.154

Exponent = 5

Base = 10 (Decimal)

Eg :-

① Distance b/w two Planet = 5.9×10^{12} m

② mass of electron = 9.1×10^{-28} gm.

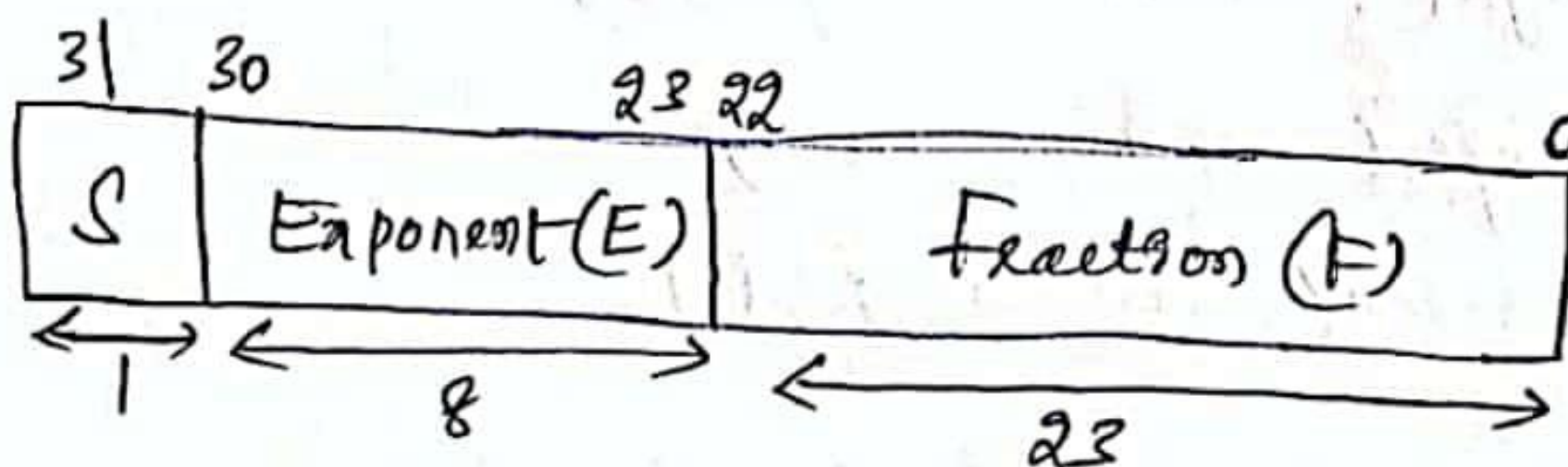
Ex :- Given the number 732.136×10^7

Sign = + (Positive)

Mantissa = 732.136

Exponent = 7

Base = 10 (Decimal)



32-bit single-precision Floating point Number

Ex :-

4.2×10^8 → Exponent
 ↓ ↓
 Mantissa Base

∴ Only the mantissa and exponent are stored. The base is implied (Already known). It will save the memory.

$$\text{Ex}^{\circ} = (11.8)_{10} = (1011.11001\dots)_2$$

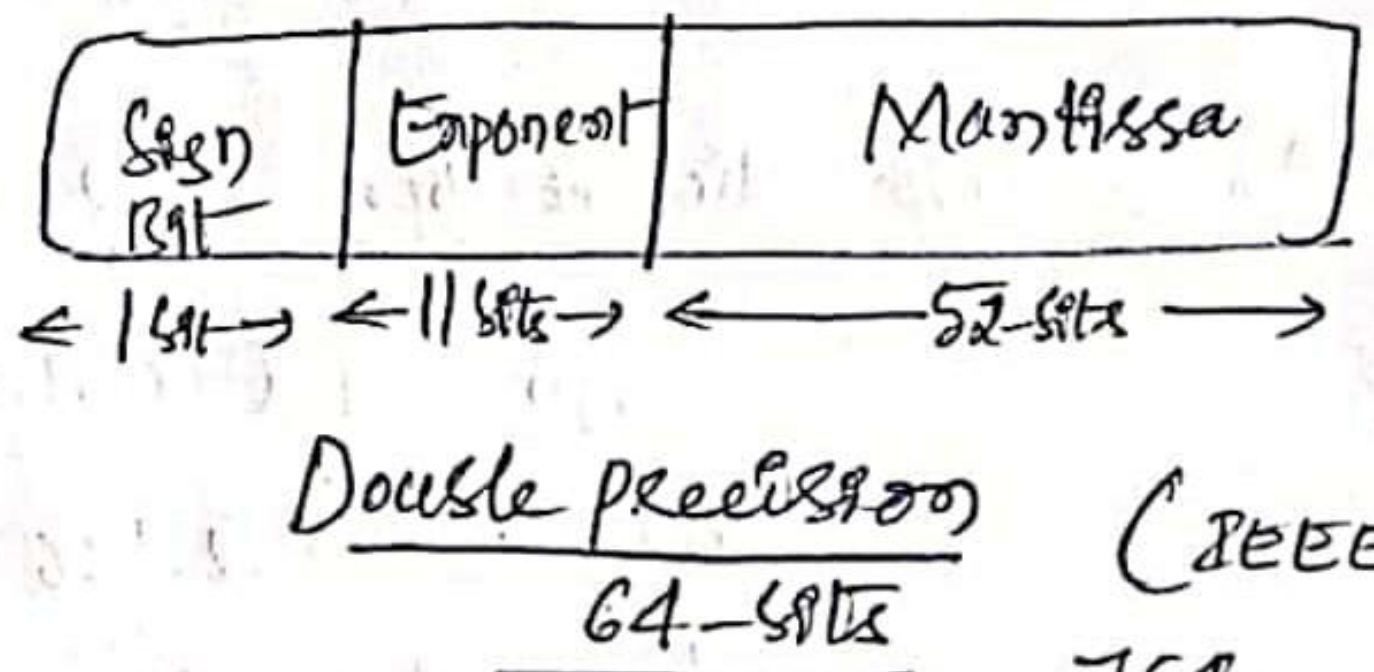
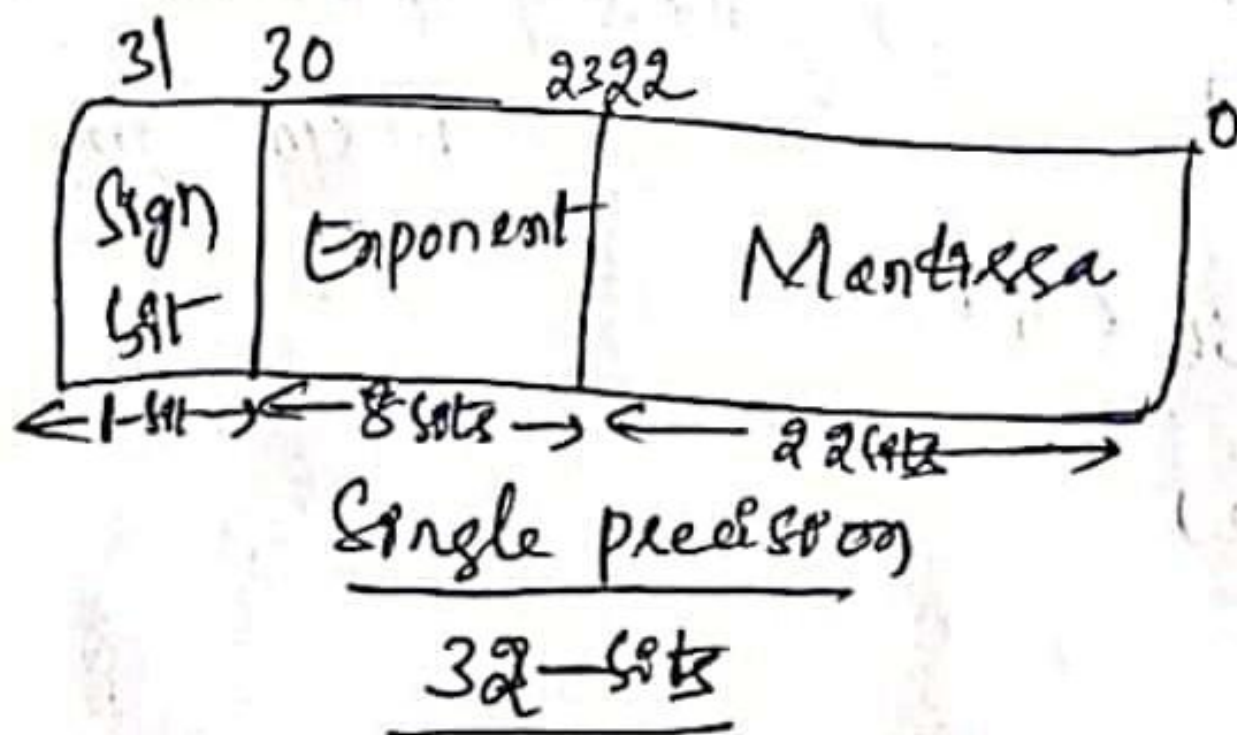
$$= (1.01111001)_2 \times 2^3$$

$$= (1.01111001)_2 \times 2^{(11)_2}$$

Decimal representation

$$12345 = \underbrace{1.2345}_{\substack{\text{mantissa} \\ \text{(or)} \\ \text{Significant}}} \times 10^{\underbrace{4}_{\text{Exponent}}}$$

⇒ We will represent floating point numbers in single precision and double precision formats. They are shown below



(IEEE 754 standard)

- * 1 bit for the sign (positive (or) negative)
- 8 bit for the range (exponent field)
- 23 bit for the precision (fraction field)

$$\left\{ \begin{aligned} N &= (-1)^s \times 1. \text{fraction} \times 2^{\text{exponent}-127}, & 1 \leq \text{exponent} \leq 254 \\ N &= (-1)^s \times 0. \text{fraction} \times 2^{\text{exponent}-126}, & \text{exponent} = 0. \end{aligned} \right.$$

$$\text{Value} = (-1)^s \times (1+F) \times 2^{E-127} \quad \text{(or)} \quad \text{Single precision}$$

$$X = (-1)^s \times 2^{E-1024} \times 1.M$$

↑ double precision

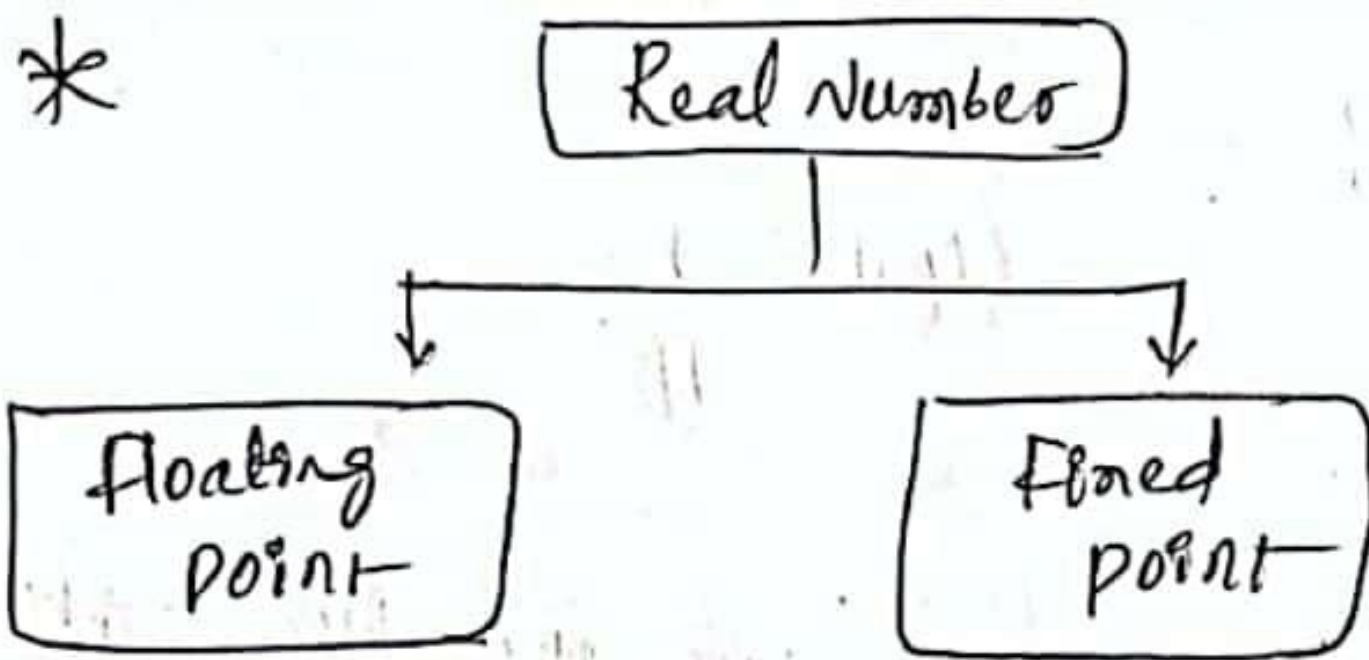
$$N = (-1)^s \times 2^{E-127} \times 1.M$$

Fixed point Representation

- ① A representation of real data type for a number that has a fixed number of digits after the decimal point.
- ② Used to represent a limited range of values.
- ③ High performance
- ④ Less flexible

Floating point Representation

- ① A formulaic representation of real numbers as an approximation so as to support a trade off between range and precision.
- ② Used to represent a wide range of values.
- ③ High performance
- ④ More flexible



problems:-

Q what is the 111.11 in decimal

- ① 7.75
- ② 31
- ③ 7.375
- ④ 15.25

* Limitation of floating points

✓ Size of mantissa is fixed.

Sol. Use a floating point format with a larger mantissa.

double (8 bytes) long double (64 bytes)

Q what is 8.5 in binary

- ① 11111111.1111
- ② 1000.01
- ③ 0.100011
- ④ 1000.10

Ex^o -114.625. represent in binary

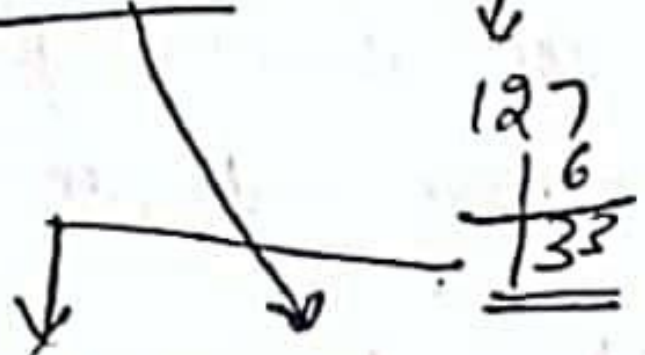
Sol

128 64 32 16 8 4 2 1 0.5 0.25 0.125
 0 1 1 1 0 0 1 0 . 1 0 1

64 + 32 + 16 + 2 = 114 ↑ 0.5 + ~~0.25~~ + 0.125

= 01110010.101

= 1.110010101 × 2⁶



∴ 1 100010101 110010101

Sign bit

Sign Exponent Mantissa

0	1001010	11101000
---	---------	----------

Ex^o

0100101011101000
 ↓ ↓ ↓ ↓
 4 10 14 8
 ↓
 A
 ↓
 E

= (4A E8) H

Ex^o

0.000110011001100110011001100₂

representation floating point in 32-bits.

Sol

1.10011001100110011001100 × 2⁻⁴

128 64 32 16 8 4 2 1
 0 1 1 1 0 1 1

Exponent = -4 + 127 = 123

= 132

Sign bit = 0

Mantissa = 10011001100110011001100

S (bit)	Exponent (bits)	Mantissa (23 bits)
0	01111011	10011001100110011001100.

Fixed point Representation :-

⇒ Representation of signed binary numbers :-

Positive numbers can be represented by unsigned numbers however to represent negative numbers, we need notation for negative numbers.

There are two types of numbers.

① Unsigned numbers

② Signed numbers

① Unsigned numbers :- There is no specific for sign

representation. The numbers without positive (or) negative signs are known as unsigned numbers. The unsigned numbers are always positive numbers.

② Signed numbers :- There is a specific bit for sign

representation. In signed numbers, the numbers may be positive (or) negative. Different formats are used for representation of signed binary numbers. They are

- ① Signed magnitude representation
- ② 1's complement representation
- ③ 2's complement representation

① Sign magnitude representation :-

In signed magnitude form, an additional bit called the 'sign bit' is placed in front of the number. If the sign bit is a '0', the number is positive. If it is a '1', the number is negative.

⇒ For example :-

$$\boxed{0}101001 = +41$$

↑
↑
 sign bit magnitude

$$\boxed{1}101001 = -41$$

↑
↑
 sign bit magnitude

In sign magnitude representation the MSB represents the sign and remaining bits represent the magnitude.

② 1's complement representation :-

In 1's complement representation the positive numbers remain unchanged. 1's complement representation of negative numbers can be obtained by the 1's complement of the binary number.

$$0 \ 110011 = +51; \text{MSB} = 0 \text{ for +ve}$$

↑
Sign bit

$$1 \ 001100 = -51; \text{MSB} = 1 \text{ for -ve}$$

↑
Sign bit

② 2's complement representation

In 2's complement representation, the positive numbers remain unchanged, 2's complement representation of negative numbers can be obtained by

1. Find the 1's complement of the number
2. To find 2's complement of the number adding '1' to its 1's complement number.

$$0 \ 110011 = +51$$

↑
Sign bit magnitude

$$1 \ 001101 = -51 \text{ [In sign 2's complement form]}$$

↑
Sign bit magnitude

Number system	+9	-9
Unsigned	+1001	-1001
Sign magnitude	Sign bit <u>0</u> 1001	Sign bit <u>1</u> 1001
Sign 1's complement form	0 1001	1 0110
Sign 2's complement form	0 1001	1 0111

Decimal	Sign magnitude form	Sign 1's complement form	Sign 2's complement form
+7	0 111	0 111	0 111
+6	0 110	0 110	0 110
+5	0 101	0 101	0 101
+4	0 100	0 100	0 100
+3	0 011	0 011	0 011
+2	0 010	0 010	0 010
+1	0 001	0 001	0 001
+0	0 000	0 000	0 000
-0	1 000	1 111	—
-1	1 001	1 110	1 111
-2	1 010	1 101	1 110
-3	1 011	1 100	1 101
-4	1 100	1 011	1 100
-5	1 101	1 010	1 011
-6	1 110	1 001	1 010
-7	1 111	1 000	1 001

Q Represent $+51$ and -51 in 8-bit magnitude, 8-bit 1's complement and 8-bit 2's complement representation.

Sol

	$+51$	-51
8-bit magnitude	<u>0 110011</u>	<u>1 110011</u>
	8-bit	8-bit

	<u>0 110011</u>	<u>1 001100</u>
8-bit 1's complement	8-bit	8-bit

	<u>0 110011</u>	<u>1 001101</u>
8-bit 2's complement	8-bit	8-bit

Q Represent $+43$ and -43 in 8-bit magnitude, 8-bit 1's complement and 8-bit 2's complement representation.

Sol

	$+43$	-43
8-bit magnitude	<u>0 101011</u>	<u>1 101011</u>
	8-bit	8-bit

	<u>0 101011</u>	<u>1 010100</u>
8-bit 1's complement	8-bit	8-bit

	<u>0 101011</u>	<u>1 010101</u>
8-bit 2's complement	8-bit	8-bit

⇒ Combinational Circuits :-

⇒ Boolean Expressions :- Boolean Algebra is a division of mathematics which deals with operations on logic values and incorporates binary variable. Boolean algebra was invented by great mathematician George Boole in 1854.

⇒ Minimization of logic expressions can be done by using boolean theorems and laws.

⇒ Boolean algebra, Karnaugh map (K-map) are used for boolean minimization.

⇒ The main motto of this concept is to make information simpler, cheaper and low cost.

⇒ Logic Gates :- ① Not gate (or) Inverter :-



Truth table

A	Y = Ā
0	1
1	0

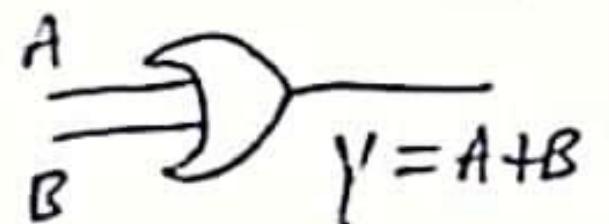
② AND Gate :-



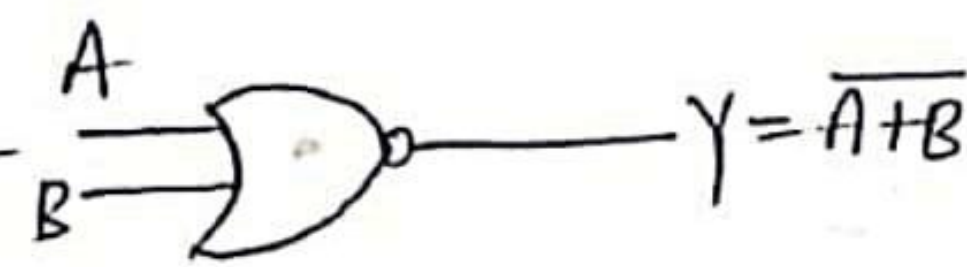
A	B	Y = AB
0	0	0
0	1	0
1	0	0
1	1	1

∴ these 3 gates are primary gates

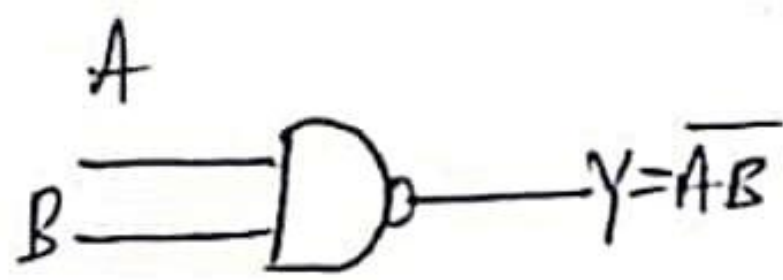
③ OR Gate :-



A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

⇒ NOR Gate :-  $Y = \overline{A+B}$

A	B	$Y = \overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

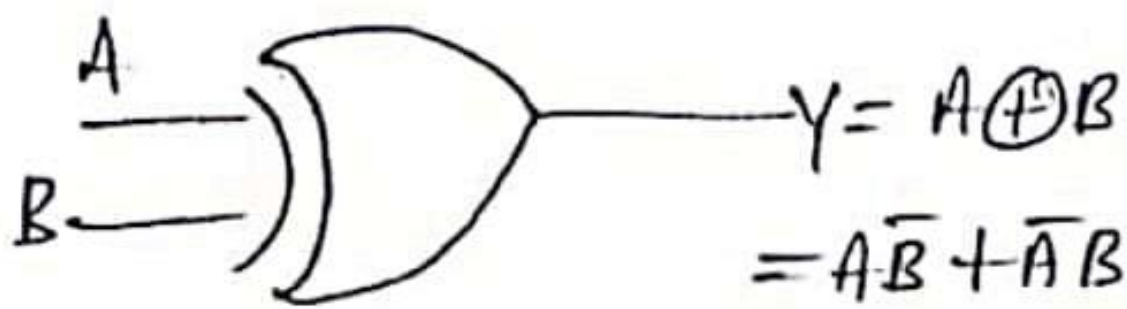
⇒ NAND Gate :-  $Y = \overline{AB}$

A	B	$Y = \overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

∴ The above two gates are universal gates

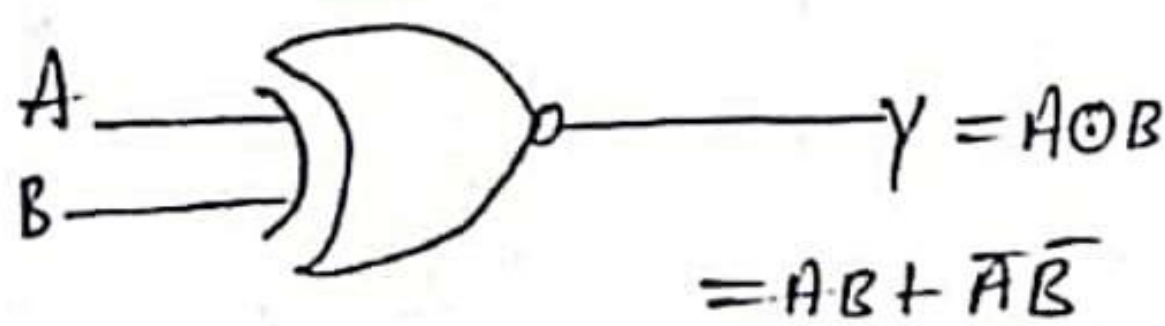
⇒ Special Gates :-

⇒ EX-OR gate :-

 $Y = A \oplus B$
 $= A\bar{B} + \bar{A}B$

A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

⇒ EX-NOR gate :-

 $Y = A \odot B$
 $= AB + \bar{A}\bar{B}$

A	B	$Y = A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

⇒ Laws of Boolean Algebra :-

AND Operation

- 1) $0 \cdot 0 = 0$
- 2) $0 \cdot 1 = 0$
- 3) $1 \cdot 0 = 0$
- 4) $1 \cdot 1 = 1$

OR Operation

- 5) $0 + 0 = 0$
- 6) $0 + 1 = 1$
- 7) $1 + 0 = 1$
- 8) $1 + 1 = 1$

NOT Operation

- 9) $\overline{0} = 1$
- 10) $\overline{1} = 0$

⇒ Complement Law

$$\bar{0} = 1$$

$$\bar{1} = 0$$

$$\text{If } A = 0 \text{ then } \bar{A} = 1$$

$$\text{If } A = 1 \text{ then } \bar{A} = 0$$

$$\overline{\bar{A}} = A$$

AND Law

$$A \cdot 0 = 0$$

$$A \cdot 1 = A$$

$$A \cdot A = A$$

$$A \cdot \bar{A} = 0$$

proof

$$= A \cdot A$$

$$= A \cdot A + 0$$

$$= A \cdot A + A \cdot \bar{A}$$

$$= A(A + \bar{A}) = A(1) = \underline{A}$$

⇒ OR Law :-

$$A + 0 = A$$

$$A + 1 = 1$$

$$A + \bar{A} = 1$$

$$A + A = A$$

proof

$$A + A = A$$

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

$$= (A + A)(A + \bar{A})$$

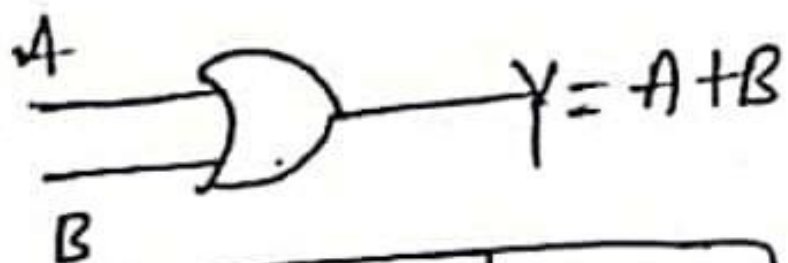
$$= A + A \cdot A + A \cdot \bar{A}$$

$$= A + A \cdot A + 0$$

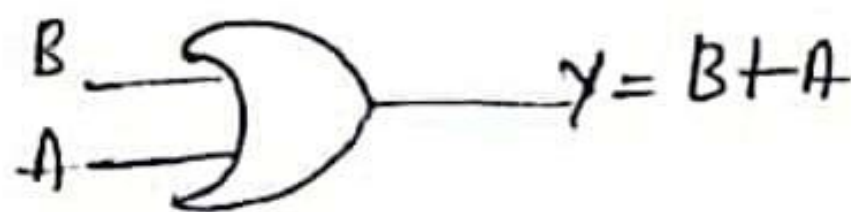
$$= A(1 + A) = A \quad (1 + A = 1)$$

⇒ Commutative Law :-

$$\textcircled{1} \quad A + B = B + A$$

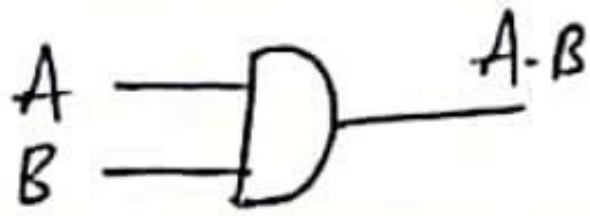


A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

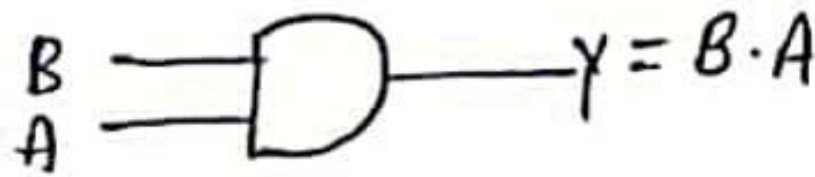


B	A	B+A
0	0	0
0	1	1
1	0	1
1	1	1

Law (2) :- $A \cdot B = B \cdot A$



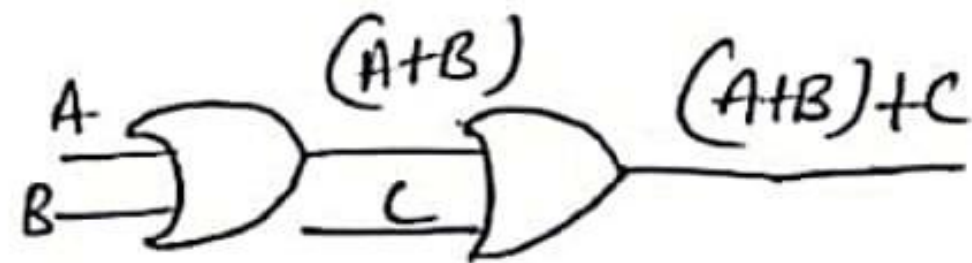
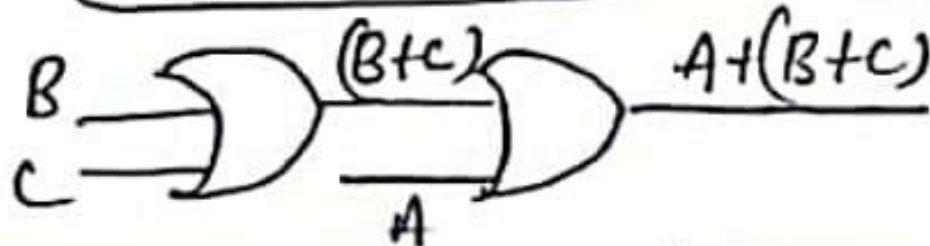
A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1



B	A	B · A
0	0	0
0	1	0
1	0	0
1	1	1

⇒ Associative Law :-

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

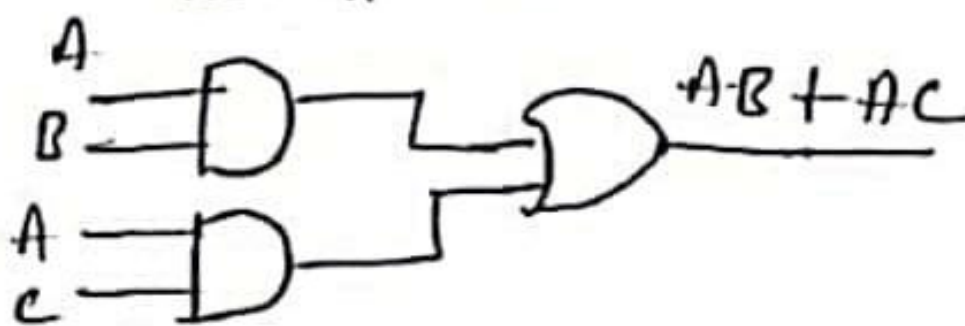
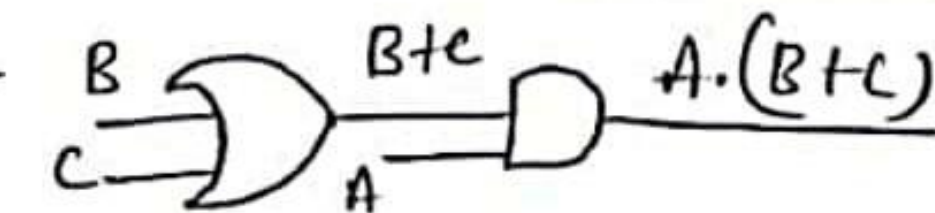
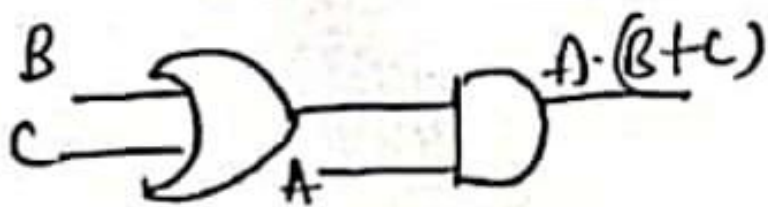


A	B	C	(B + C)	A + (B + C)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

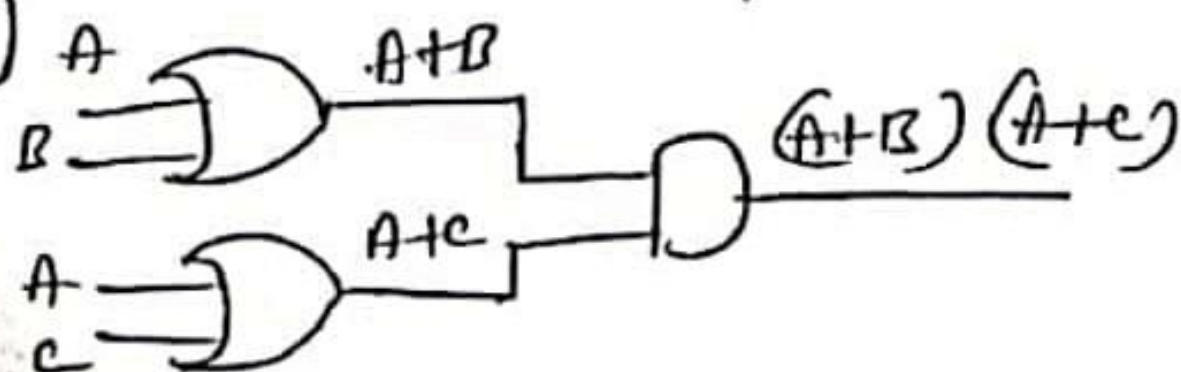
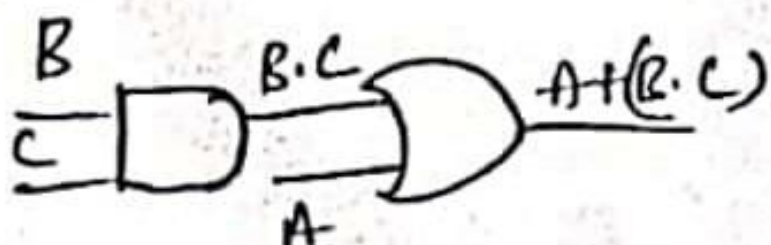
A	B	C	(A + B)	(A + B) + C
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

⇒ Distributive Law :-

① $A \cdot (B + C) = AB + AC$



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



⇒ Consensus Theorem :-

$$AB + \bar{A}C + BC = AB + \bar{A}C$$

$$\begin{aligned} \text{L.H.S} &= AB + \bar{A}C + BC \\ &= AB + \bar{A}C + BC(A + \bar{A}) \\ &= AB + \bar{A}C + BCA + BC\bar{A} \\ &= ABC(1+C) + \bar{A}C(1+B) \\ &= AB(1) + \bar{A}C(1) \\ &= AB + \bar{A}C \\ &= \text{R.H.S} \end{aligned}$$

$$\therefore A + \bar{A} = 1$$

$$1 + C = 1$$

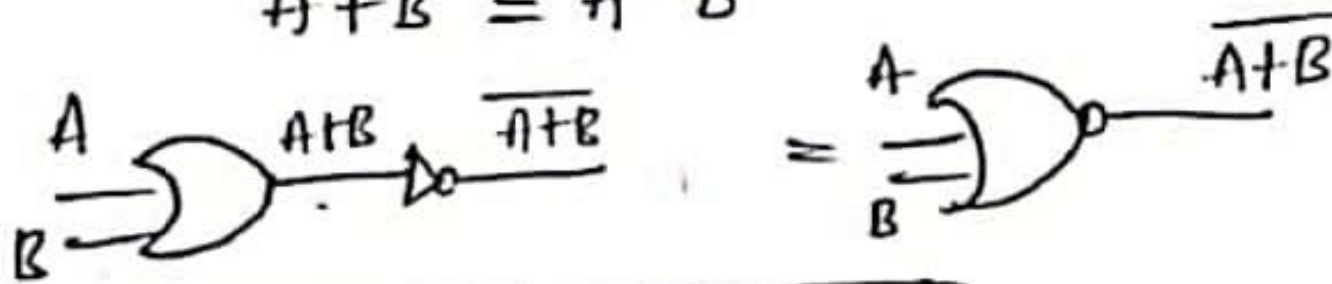
$$1 + B = 1$$

$$\therefore \boxed{\text{L.H.S} = \text{R.H.S}}$$

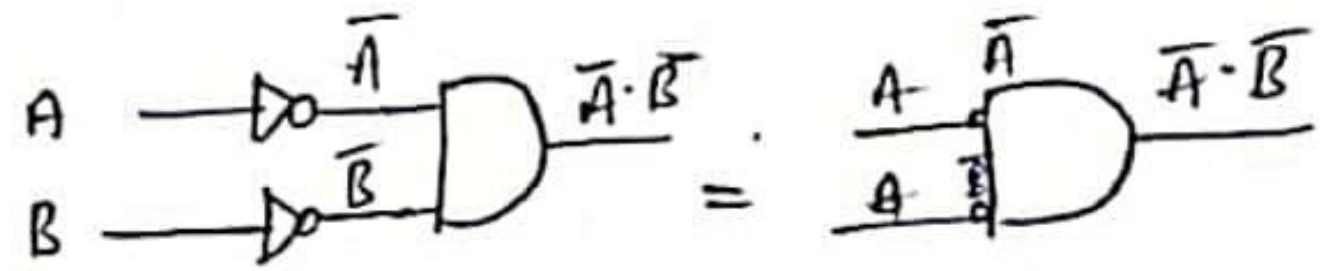
⇒ DeMorgan's Theorem :-

①

$$\overline{A+B} = \bar{A} \cdot \bar{B}$$

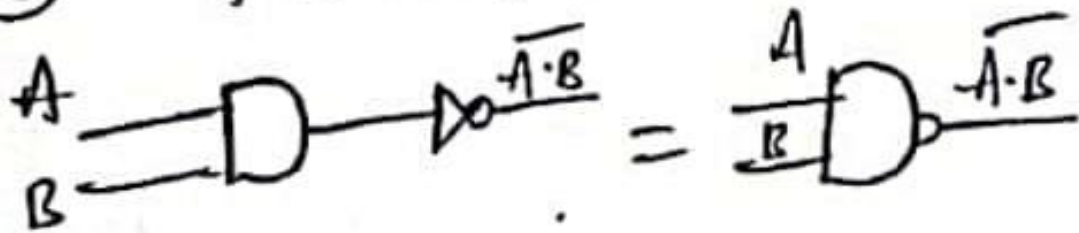


A	B	A+B	$\overline{A+B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

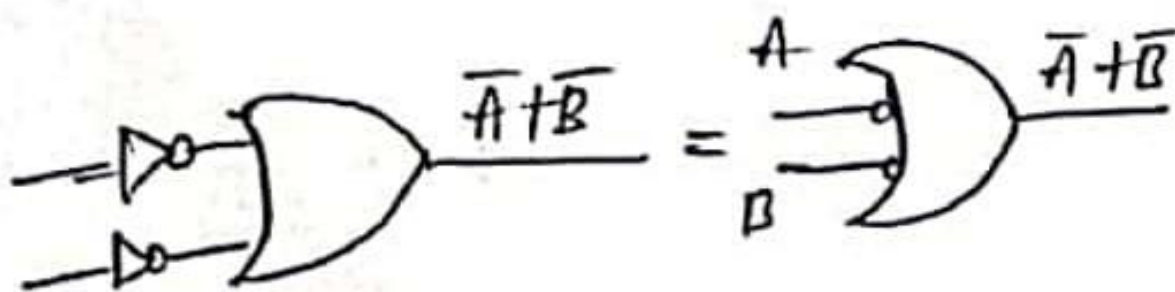


A	B	\bar{A}	\bar{B}	$\bar{A} \cdot \bar{B}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

② $\overline{A \cdot B} = \bar{A} + \bar{B}$



A	B	A \cdot B	$\overline{A \cdot B}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0



A	B	\bar{A}	\bar{B}	$\bar{A} + \bar{B}$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

⇒ Duality :- when changing from one logic system to another system; 0 becomes 1 and 1 becomes 0. An AND gate becomes OR gate and OR gate becomes an AND gate.

⇒ Complement :- If a boolean identity is given, we should change '+' sign to '.' sign and '.' sign to '+' sign. Variables (A, B) also complemented.

Note :-

① $AB + AB + AB + AB = AB$
 parenthesis (Same no of variables are repeated we may consider single term)

② $A \cdot B \cdot \bar{B} = A \cdot 0 = 0$; ③ $ABC\bar{C} = AB \cdot 0 = 0$

③ $AB\bar{C} + ABC + AB\bar{C}D$
 $AB\bar{C}(1+D) = AB\bar{C}(1) = AB\bar{C}$ ④ $AB\bar{C}D + ABC\bar{D}$
 $= AB\bar{C}(D+\bar{D}) = AB\bar{C}(1) (\because D+\bar{D}=1)$

① $f = A + B [AC + (B+\bar{C})D]$
 $= A + B[AC + BD + \bar{C}D]$
 $= A + [ABC + \underbrace{BBD}_{\because BBD=BD \text{ (repeated terms; we can consider as single)}} + B\bar{C}D]$
 $= A(1+BC) + BD(1+\bar{C})$
 $\therefore 1+BC=1 ; 1+\bar{C}=1$
 $f = A + BD$

② $f = (A + \bar{B}\bar{C}) \cdot (A\bar{B} + ABC)$
 $= \bar{A} \cdot \bar{B}\bar{C} (A\bar{B} + ABC)$
 $= \bar{A}\bar{B}\bar{C} (A\bar{B} + ABC)$
 $\because A\bar{A}=0 ; B\bar{B}=0$

$= \bar{A}\bar{B}\bar{C}A\bar{B} + \bar{A}\bar{B}\bar{C}ABC$
 $= \bar{A}\bar{A}\bar{B}\bar{B}\bar{C} + \bar{A}\bar{A}\bar{B}\bar{B}\bar{C}C$
 $= \downarrow_0 \downarrow_0 + 0$ $f = 0$

Hence the boolean expression has been reduced by boolean theorem.

Q Write the duality for the following functions

① $\bar{A}B + \bar{A}B\bar{C} + \bar{A}BCD + \bar{A}B\bar{C}\bar{D}E$

Sol $(A+B)(\bar{A}+B+\bar{C})(\bar{A}+B+C+D)(\bar{A}+B+\bar{C}+\bar{D}+E)$

② $\bar{x}yz + x\bar{y}\bar{z} + xyz + xy\bar{z}$

$(\bar{x}+y+z)(x+\bar{y}+\bar{z})(x+y+z)(x+y+\bar{z})$

Q Find the complements of the following expressions

① $AB + A(B+C) + \bar{B}(B+D)$

Sol $(\bar{A}+\bar{B})(\bar{A}+B+\bar{C})(B+\bar{B}+\bar{D})$

② $\bar{B}\bar{C}D + \overline{(B+C+D)} + \bar{B}\bar{C}\bar{D}E$

$(B+C+\bar{D})(B+C+D) + (\bar{B}+\bar{C}+\bar{D}+\bar{E})$

⇒ Karnaugh Map (K-map) Representation :-

① Sum of product (SOP) $\sum m$:- $\bar{A}B + A\bar{B}$

② product of sum (POS) $\sum M$:- $(A+B)(\bar{A}+C)$

① Sum of product (SOP) :- This is also called as disjunctive normal form (DNF). variables present in this variables are called 'miniterms' (m_0, m_1, m_2, \dots)

$\sum m$:- $f(A, B, C) = m_1 + m_2 + m_3 + m_5 = \sum m(1, 2, 3, 5)$

⇒ Standard SOP form :- (SOP) It is also called as Disjunctive Canonical Form (DCF)

② product of sum (POS) :- It is also called as conjunctive Normal Form (CNF). variables present in this form is called 'maxiterms' (M_1, M_2, M_3, \dots)

$$\text{Ex: } F(A, B, C) = \prod(M_1, M_2, M_6, M_7) \\ = \prod M(1, 2, 6, 7)$$

\Rightarrow Standard pos form i.e. max form is also called as conjunctive Canonical form (CCF)

$$\text{Ex: } f(A, B, C) = (\bar{A} + \bar{B}) (A + B) \\ = (\bar{A} + \bar{B} + C \cdot \bar{C}) (A + B + C \cdot \bar{C}) \quad \because C \cdot \bar{C} = 0 \\ = (\bar{A} + \bar{B} + C) (\bar{A} + \bar{B} + \bar{C}) (A + B + C) (A + B + \bar{C})$$

Ex: Convert SOP to standard SOP form

$$f(A, B, C) = AC + AB + BC$$

$$\text{SOP} = A(B + \bar{B})C + AB(C + \bar{C}) + (A + \bar{A})BC \quad \because B + \bar{B} = 1$$

$$= \underline{ABC} + A\bar{B}C + \underline{ABC} + \underline{ABC} + \underline{ABC} + \bar{A}BC \\ = ABC + A\bar{B}C + \bar{A}BC + A\bar{B}\bar{C}$$

Repeated ABC product is there. we should write only one

Decimal no.	A	B	C	minterms	Maxterms
0	0	0	0	$\bar{A}\bar{B}\bar{C}$ (m ₀)	$A + B + C$ (M ₀)
1	0	0	1	$\bar{A}\bar{B}C$ (m ₁)	$A + B + \bar{C}$ (M ₁)
2	0	1	0	$\bar{A}B\bar{C}$ (m ₂)	$A + \bar{B} + C$ (M ₂)
3	0	1	1	$\bar{A}BC$ (m ₃)	$A + \bar{B} + \bar{C}$ (M ₃)
4	1	0	0	$A\bar{B}\bar{C}$ (m ₄)	$\bar{A} + B + C$ (M ₄)
5	1	0	1	$A\bar{B}C$ (m ₅)	$\bar{A} + B + \bar{C}$ (M ₅)
6	1	1	0	$AB\bar{C}$ (m ₆)	$\bar{A} + \bar{B} + C$ (M ₆)
7	1	1	1	ABC (m ₇)	$\bar{A} + \bar{B} + \bar{C}$ (M ₇)

Rules for K-map minimization :-

- ① Either group zeros & ones
- ② Diagonal mapping is not allowed
- ③ Only power of 2, no. of cells in each group (i.e. 2, 4, 6, 8, ...)
- ④ Group should be as large as possible
- ⑤ Overlapping is allowed.

Q Problems on K-map representation

2-variable K-map (SOP)

i, $f = A\bar{B} + A\bar{B}$

A	B	\bar{B}	B
\bar{A}			
A	1	1	

$f = A$

ii, $f = A\bar{B} + AB + \bar{A}B$

A	B	\bar{B}	B
\bar{A}			
A	1		1
\bar{A}			1

$f = (A+B)$

iii, $f(A,B) = (0,3)$

A	B	\bar{B}	B
\bar{A}	1		
A			1

$= \bar{A}\bar{B} + AB$

Q $f = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$

Sol

A	BC	00	01	11	10
\bar{A}	$\bar{B}\bar{C}$		$\bar{B}C$	BC	$B\bar{C}$
\bar{A}	0		1		1
A	1	1	1		1

$f = \bar{B}C + AC + AB + B\bar{C}$

$f(A,B,C) = \Sigma m(3,4,6,7)$

A	BC	$\bar{B}\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
\bar{A}	0		1	1	
A	1			1	1

$f = BC + A\bar{C}$

3-variable K-map

Q Reduce the below expression using 4 variable K-map

$f(A,B,C,D) = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D} + A\bar{B}CD + A\bar{B}\bar{C}D$

AB	CD	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	1	1	1		
$\bar{A}B$	1				
AB			1		
$\bar{A}\bar{B}$	1	1			

$\bar{A}\bar{B}D$
 $A\bar{B}C\bar{D}$
 $ABCD$
 $\bar{B}\bar{C}$

$\therefore f = \bar{A}\bar{B}CD + \bar{A}\bar{B}D + \bar{A}C\bar{D} + \bar{B}\bar{C}$

Q Convert the following in the SOP form and calculate the minterms

(a) $F(A, B) = \bar{A}B + B$

Sol Given $F(A, B) = \bar{A}B + B$

$= \bar{A}B + B \quad (1)$

$= \bar{A}B + B(A + \bar{A}) \quad \because A + \bar{A} = 1$

$= \bar{A}B + AB + \bar{A}B \quad \because \bar{A}B + \bar{A}B = \bar{A}B$

$= \bar{A}B + AB$

$\begin{matrix} \downarrow \downarrow & \downarrow \downarrow \\ = & 01 & 11 \end{matrix}$

$= m_1 + m_3$

$= \Sigma m(1, 3)$

If same digits are more than two then it becomes one)

(b) $f(A, B, C) = ABC + A\bar{B}C + AB$

$= ABC + A\bar{B}C + AB \quad (1)$

$= ABC + A\bar{B}C + AB(C + \bar{C})$

$= ABC + A\bar{B}C + ABC + A\bar{B}\bar{C} \quad (\because C + \bar{C} = 1)$

$\begin{matrix} \downarrow \downarrow \downarrow & \downarrow \downarrow \downarrow & \downarrow \downarrow \downarrow & \downarrow \downarrow \downarrow \\ 110 & 101 & 111 & 110 \\ 6 & 5 & 7 & 3 \end{matrix}$

$= m_6 + m_5 + m_7 + m_3$

It should be in proper order

$\therefore m_3 + m_5 + m_6 + m_7$

$\Rightarrow \Sigma m(3, 5, 6, 7)$

Q Convert the following in the SPOS form and calculate the minterms.

(a) $F(A, B) = A(\bar{A} + \bar{B})$

$= A + 0(\bar{A} + \bar{B})$

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

$= (A + B)(A + \bar{B})(\bar{A} + \bar{B})$

$\begin{matrix} \downarrow \downarrow & \downarrow \downarrow & \downarrow \downarrow \\ 00 & 01 & 11 \end{matrix}$

$= M_0 \cdot M_1 \cdot M_2$

$= \Pi M(0, 1, 2)$

(b) $f(A, B, C) = A(\bar{A} + \bar{B})B$

$= A + 0(\bar{A} + \bar{B})B + 0$

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

$= (A + B)(A + \bar{B})(A + \bar{B})(B + A)(B + \bar{A})$

$= (A + B)(A + \bar{B})(A + \bar{B})(A + \bar{B})(\bar{A} + B)$

$\begin{matrix} \downarrow \downarrow & \downarrow \downarrow & \downarrow \downarrow & \downarrow \downarrow & \downarrow \downarrow \\ 00 & 01 & 11 & 0 & 0 \\ M_0 \cdot M_1 \cdot M_2 = \Pi M(0, 1, 2) \end{matrix} \quad \begin{matrix} \because (A + B)(A + B) = (A + B) \\ (\bar{A} + \bar{B})(\bar{A} + \bar{B}) = (\bar{A} + \bar{B}) \end{matrix}$

Note :- K-map consist of a no. of squares. Each one of the square is cell. To do K-map minimization the expression should be in SOP form or POS form.

This is extremely useful and extensively used in the minimization of function of 2, variable K-map, 3-variable K-map, 4-variable K-map and so on.

Q Simplify, $f(A, B, C, D) = \sum m(0, 1, 3, 7, 11, 15)$

		CD			
		$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
AB	$\bar{A}\bar{B}$	1	1	1	
	$\bar{A}B$			1	
	$A\bar{B}$			1	
	AB			1	

$\therefore f = \bar{A}\bar{B}\bar{C} + CD$

Q $f(A, B, C, D) = \sum m(0, 5, 1, 2, 3, 7, 8, 10)$

		CD			
		$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
AB	$\bar{A}\bar{B}$	1	1	1	1
	$\bar{A}B$		1	1	
	$A\bar{B}$	1			
	AB				

$\therefore f = \bar{B}\bar{D} + \bar{A}D$

⇒ Minimization of Boolean Expression with help of POS by using K-map

2-variable

		0	1
	B	$\bar{A}B$	$A\bar{B}$
A	\bar{B}	$\bar{A}\bar{B}$	$A\bar{B}$
	B	$\bar{A}B$	$A\bar{B}$

3-variable

		BC			
		$B\bar{C}$	$B\bar{C}$	$B\bar{C}$	$B\bar{C}$
A	1	$A\bar{B}\bar{C}$	$A\bar{B}\bar{C}$	$A\bar{B}\bar{C}$	$A\bar{B}\bar{C}$
	\bar{A}	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}\bar{C}$

		C+D			
		$\bar{C}+\bar{D}$	$C+\bar{D}$	$\bar{C}+D$	$C+D$
A+B	1	$A\bar{B}\bar{C}+\bar{D}$	$A\bar{B}C+\bar{D}$	$A\bar{B}\bar{C}+D$	$A\bar{B}C+D$
	$\bar{A}+\bar{B}$	$\bar{A}\bar{B}\bar{C}+\bar{D}$	$\bar{A}\bar{B}C+\bar{D}$	$\bar{A}\bar{B}\bar{C}+D$	$\bar{A}\bar{B}C+D$
	$\bar{A}+B$	$\bar{A}\bar{B}\bar{C}+\bar{D}$	$\bar{A}\bar{B}C+\bar{D}$	$\bar{A}\bar{B}\bar{C}+D$	$\bar{A}\bar{B}C+D$
	$\bar{A}+B$	$\bar{A}\bar{B}\bar{C}+\bar{D}$	$\bar{A}\bar{B}C+\bar{D}$	$\bar{A}\bar{B}\bar{C}+D$	$\bar{A}\bar{B}C+D$

Q $f(A, B) = (A+B)(\bar{A}+\bar{B})$

		B	\bar{B}
A	1	0	0
	\bar{A}	0	0

\downarrow $B = \bar{B}$

Q $f = \prod M(0, 2, 3, 4, 5, 6, 7, 9, 12, 8)$

		C+D			
		$C+D$	$C+\bar{D}$	$\bar{C}+\bar{D}$	$\bar{C}+D$
A+B	1	0	0	0	0
	$\bar{A}+\bar{B}$	0	0	0	0
	$\bar{A}+B$	0	0		
	$\bar{A}+B$	0	0		

\downarrow $(C+D)$ $(\bar{A}+B+C)$ $(A\bar{B}+C)$ $(A+\bar{C})$

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

Q $f(A, B, C) = (A+B+C)(\bar{A}+\bar{B}+\bar{C})(A+B+\bar{C})(A+\bar{B}+\bar{C})$

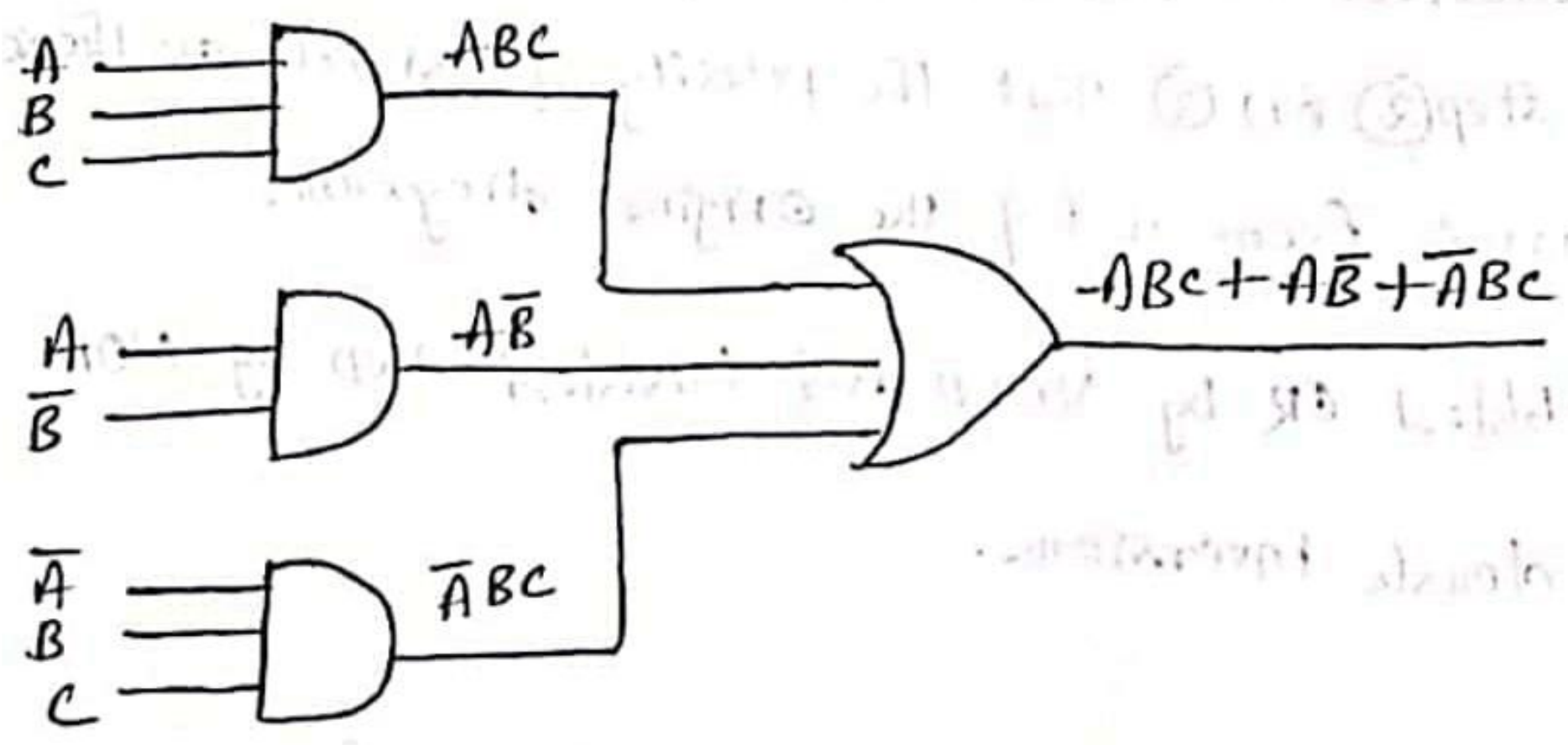
		BC			
		$B\bar{C}$	$B\bar{C}$	$\bar{B}\bar{C}$	$\bar{B}\bar{C}$
A	1	0	0	0	0
	\bar{A}			0	

\downarrow $(A+B)$ \rightarrow $(\bar{B}+\bar{C})$

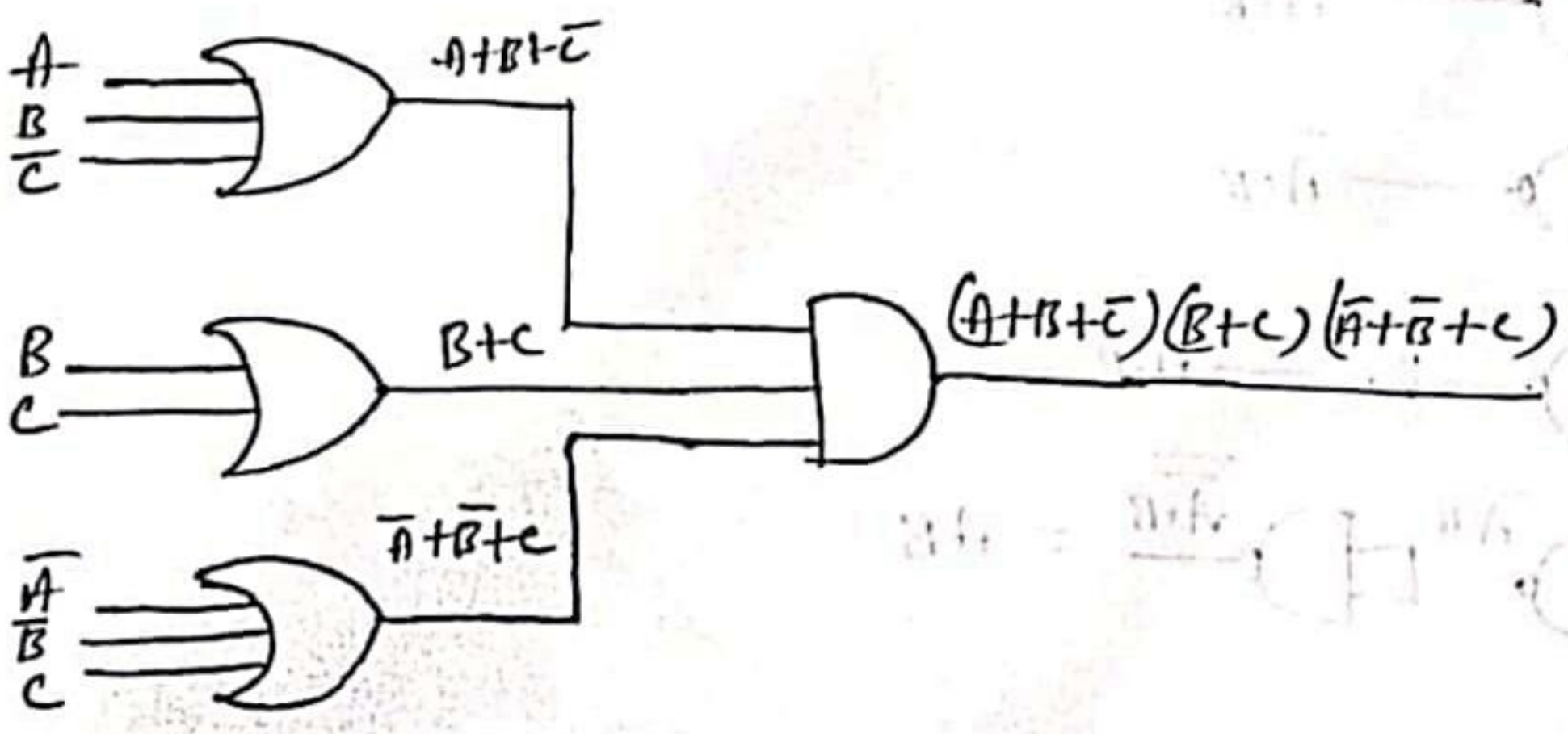
⇒ XNAND and NOR Implementation :-

In the design of digital circuits, the minimal boolean expressions are usually obtained in SOP form (or) POS form. Sometimes the minimal expressions may also be expressed in hybrid form.

For example $f = ABC + A\bar{B} + \bar{A}BC$
Given example is in SOP form. So, SOP expression can be implemented by using AND/OR logic as shown below.



The form of given expression is $f = (A+B+\bar{C})(B+C)(\bar{A}+\bar{B}+C)$.
This can be implemented using OR/AND logic as shown below.

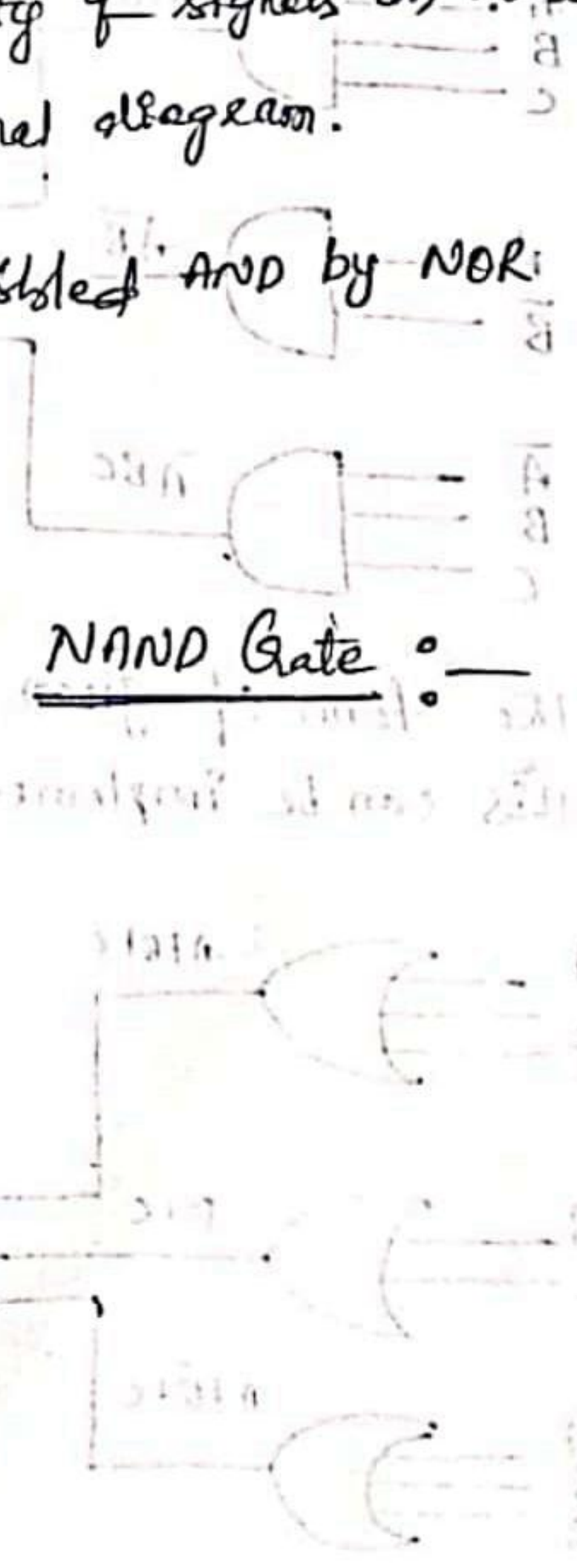
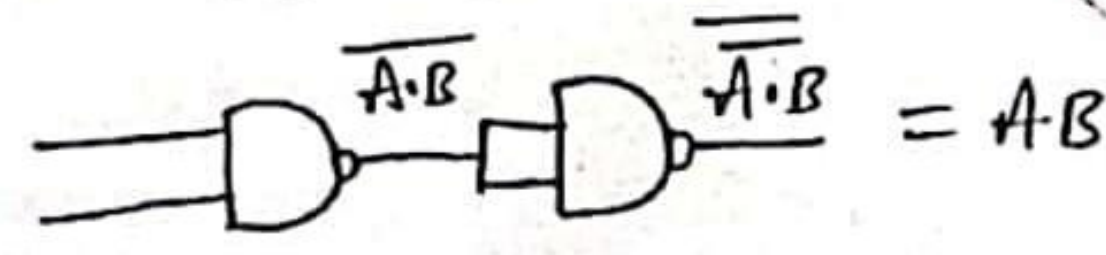
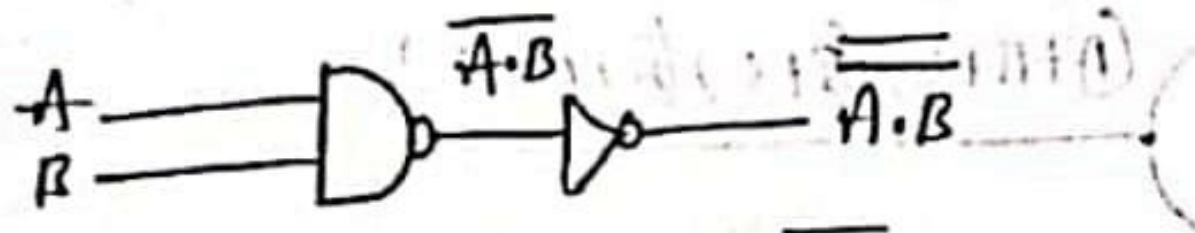
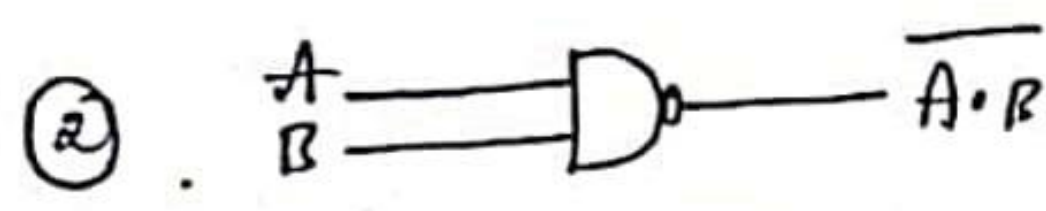
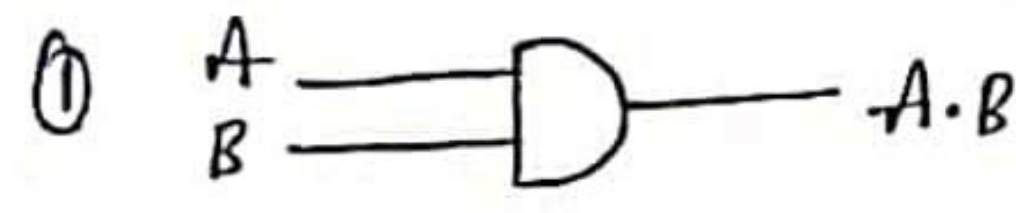


The procedure to convert an AOI logic to NAND logic (or) NOR logic is given below.

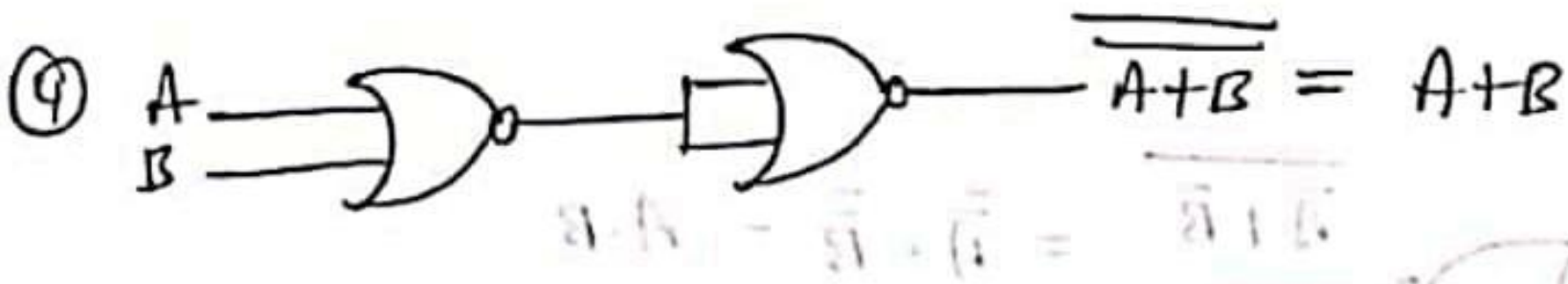
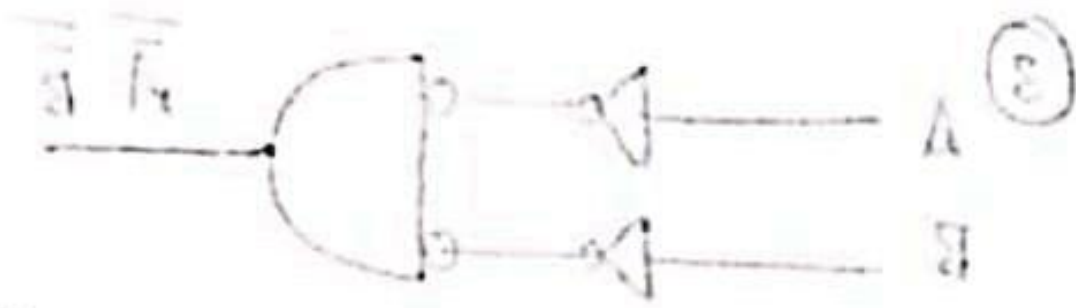
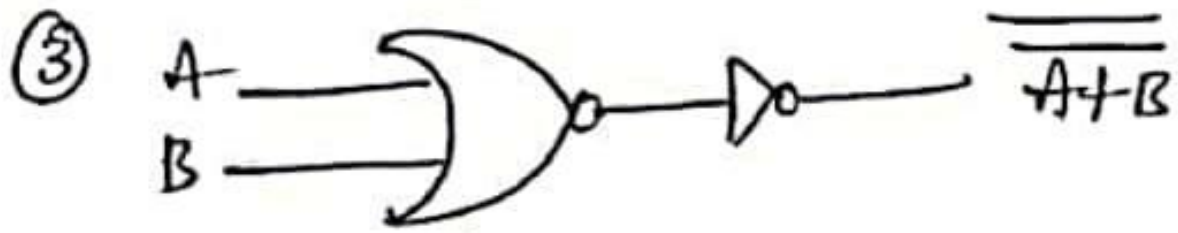
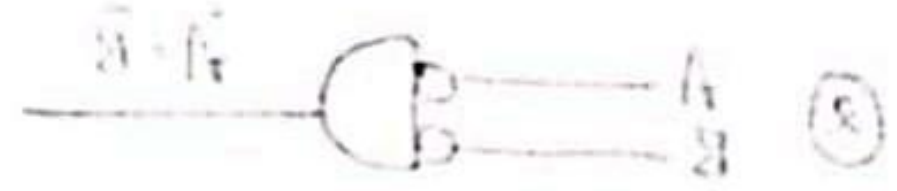
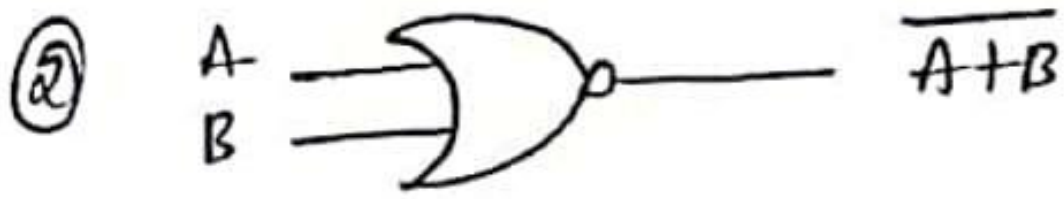
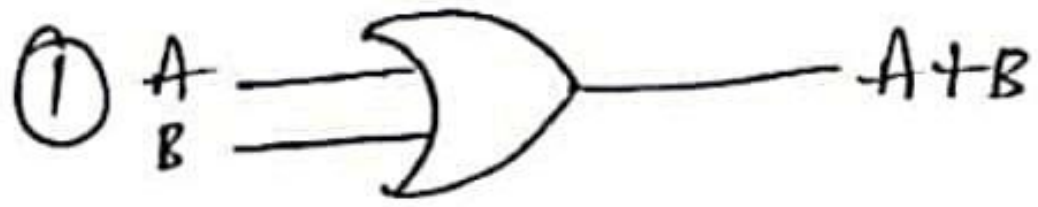
127

- ① Draw the circuit in AOI logic
- ② If NAND hardware is chosen, add a circle at the output of each AND gate and at the inputs to all the OR gates.
- ③ If NOR hardware is chosen, add a circle at the output of each OR gate and at the inputs to all the AND gates.
- ④ Add (or) subtract an inverter on each line that received a circle in step ② (or) ③ that the polarity of signals on those lines remains from that of the original diagram.
- ⑤ Replace bubbled OR by NAND and bubbled AND by NOR.
- ⑥ Eliminate double inversions.

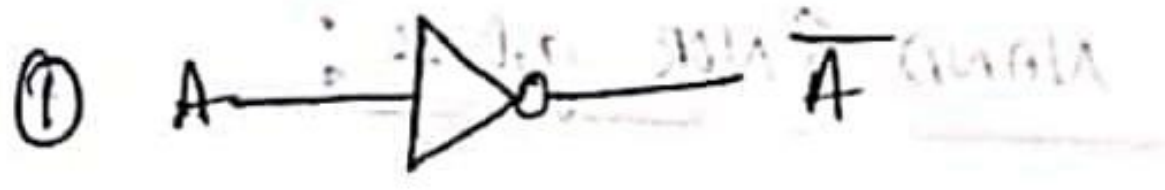
① Implementation of AND gate using NAND Gate :-



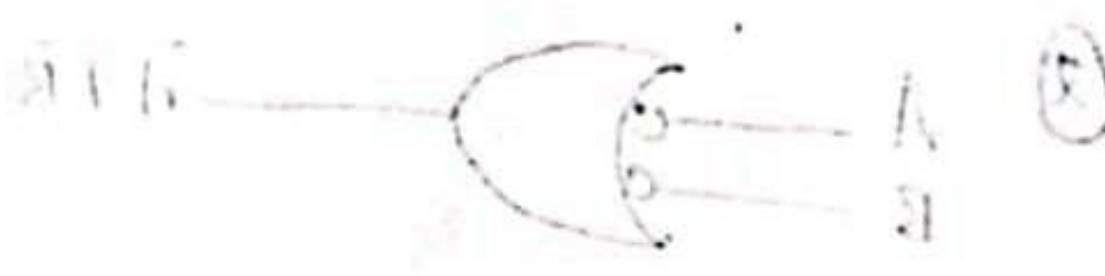
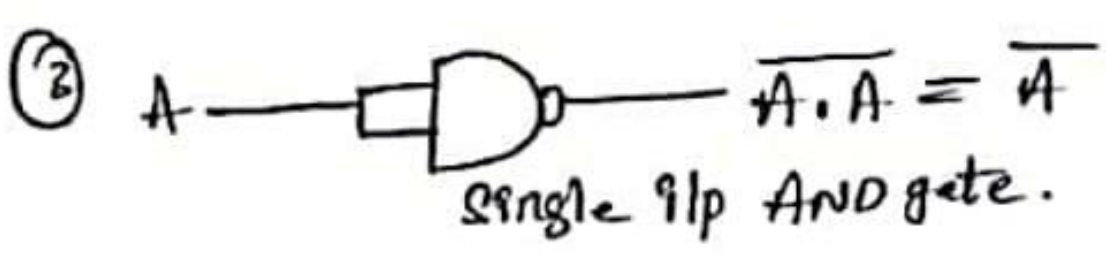
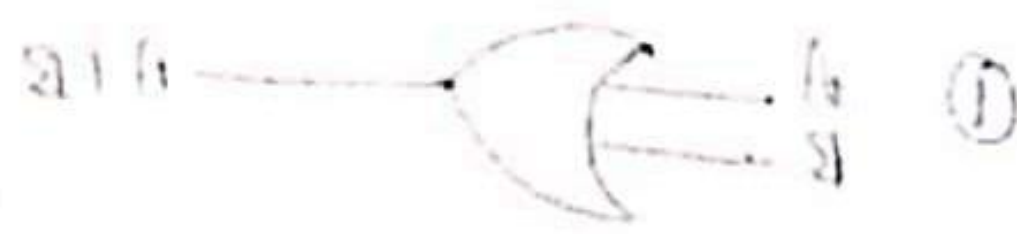
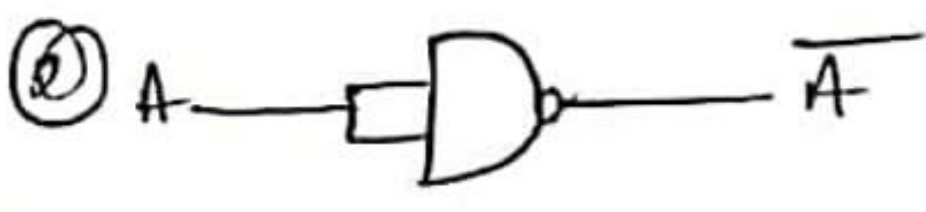
④ OR gate using NOR :- step wise realization. (128)



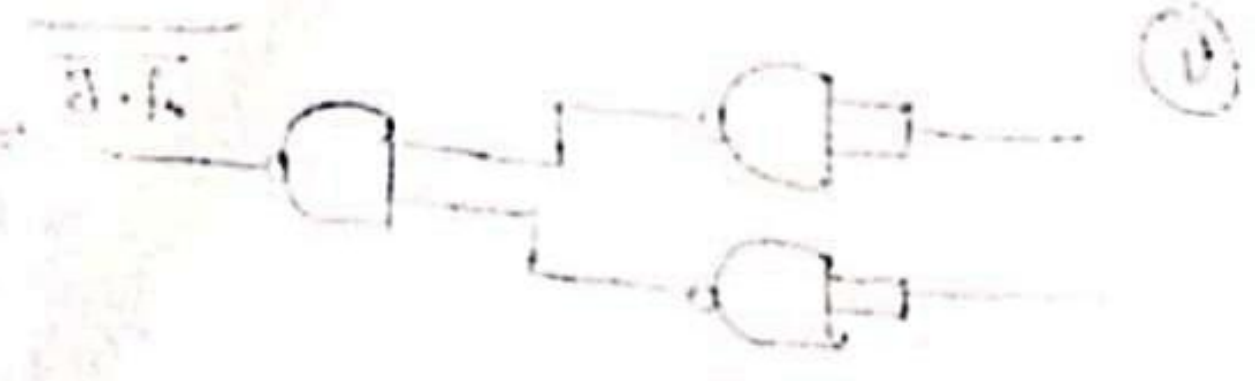
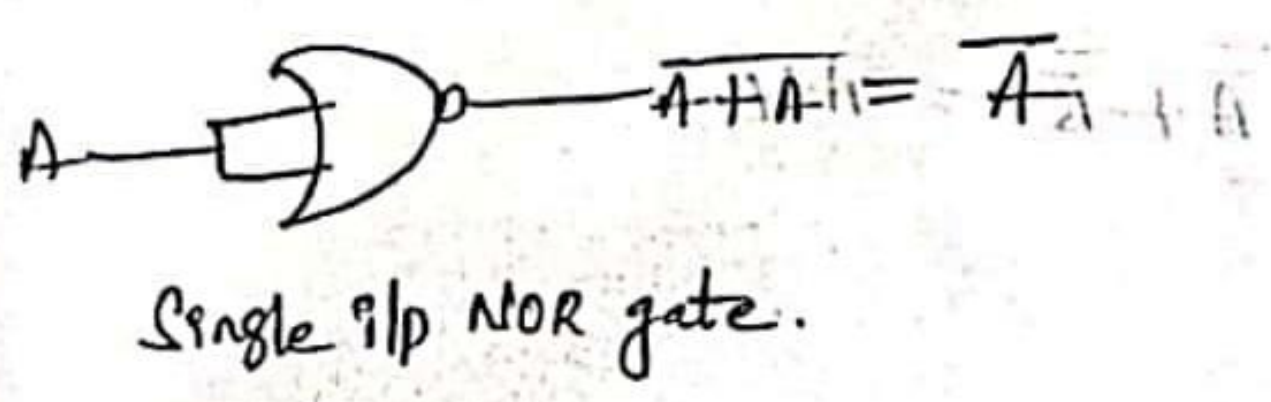
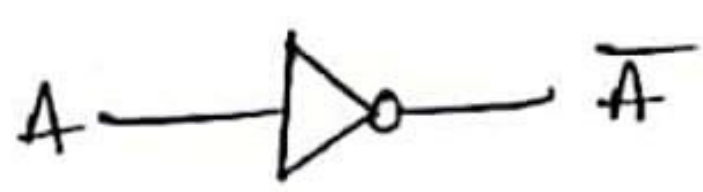
⑤ Realization of NOT gate using NAND & NOR



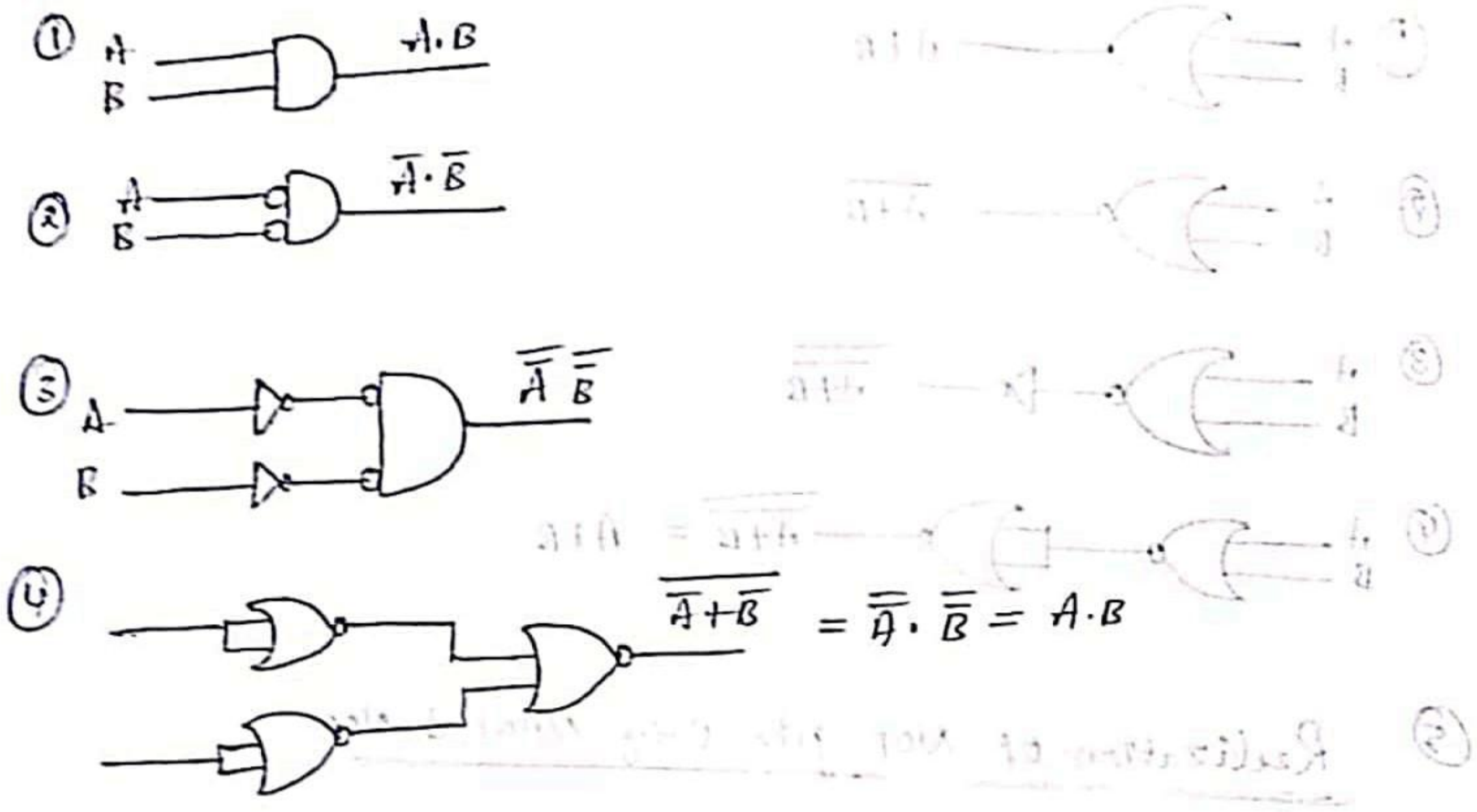
Realization of OR gate using NAND



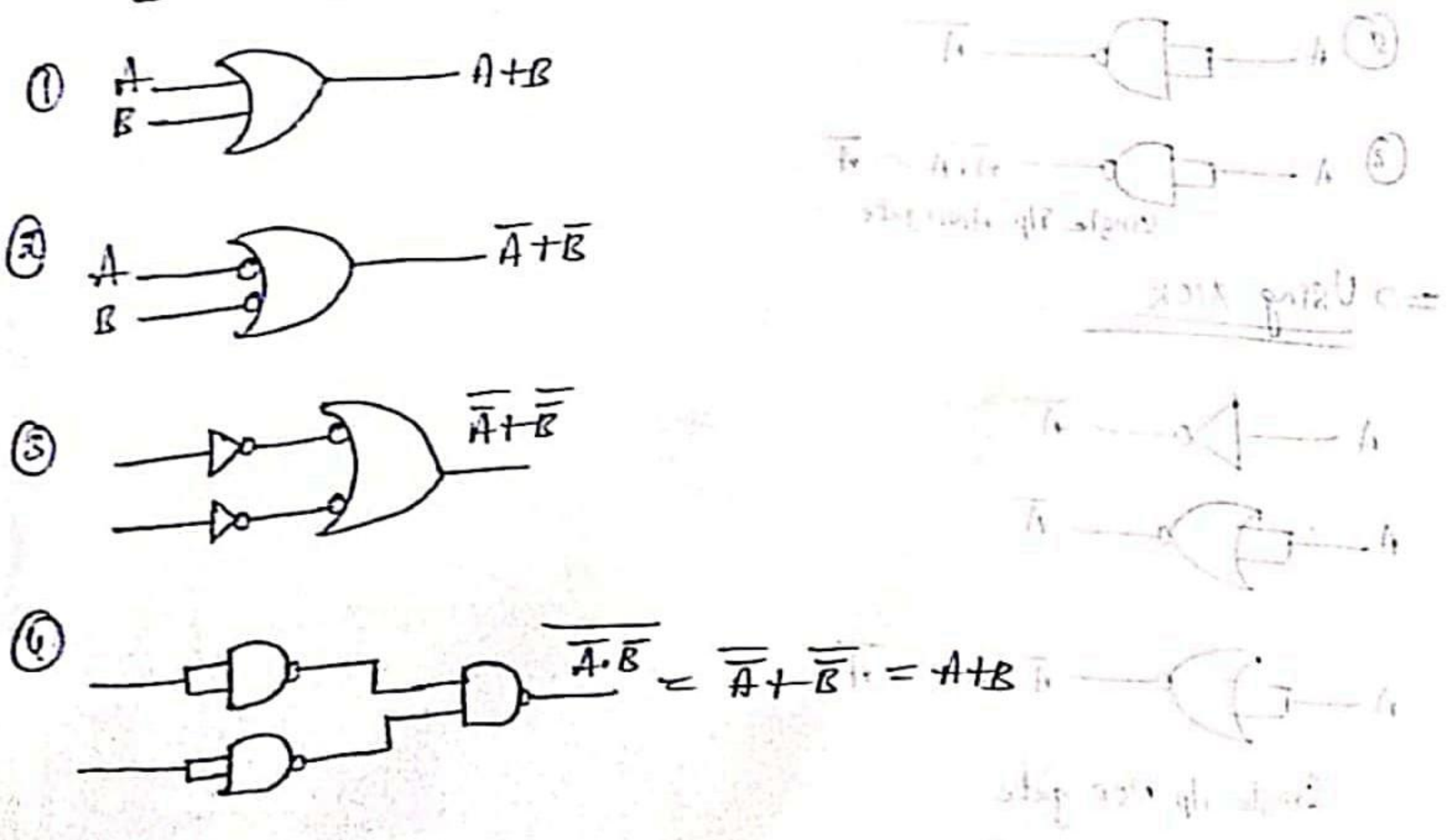
⇒ Using NOR



② Realization of AND gate using NOR Gate:- (129) 26

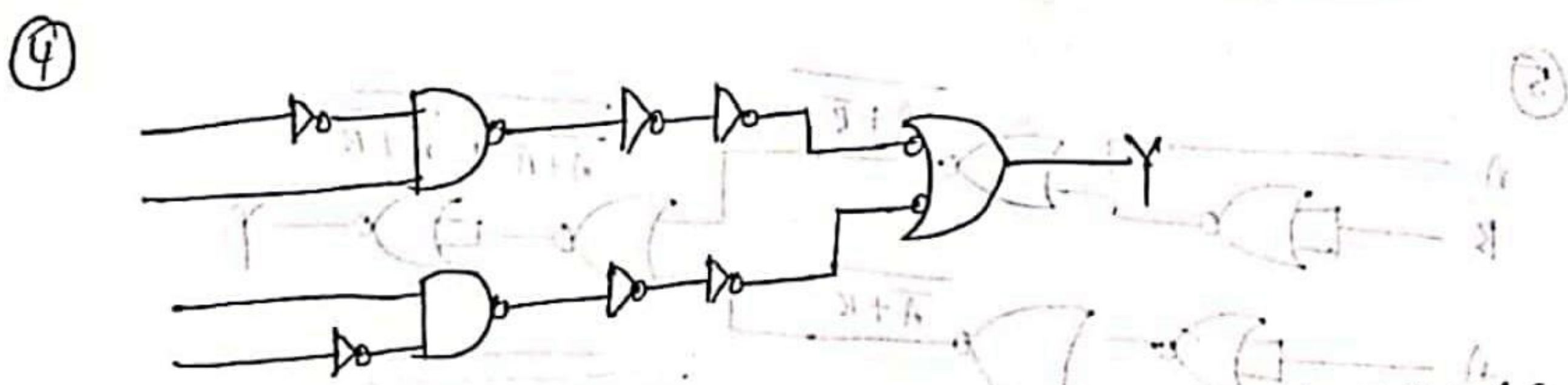
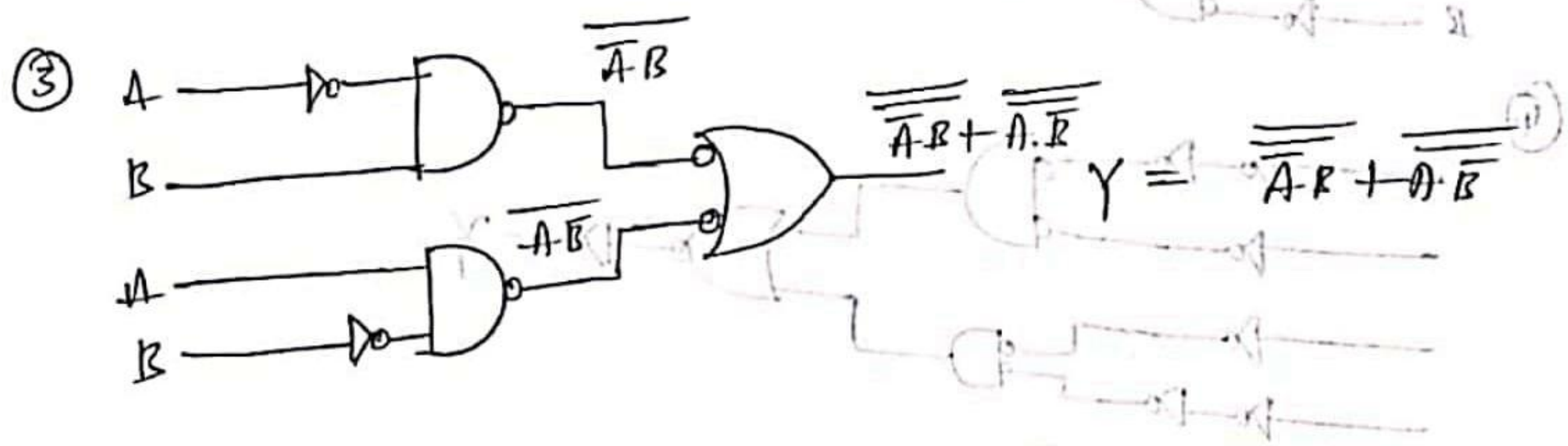
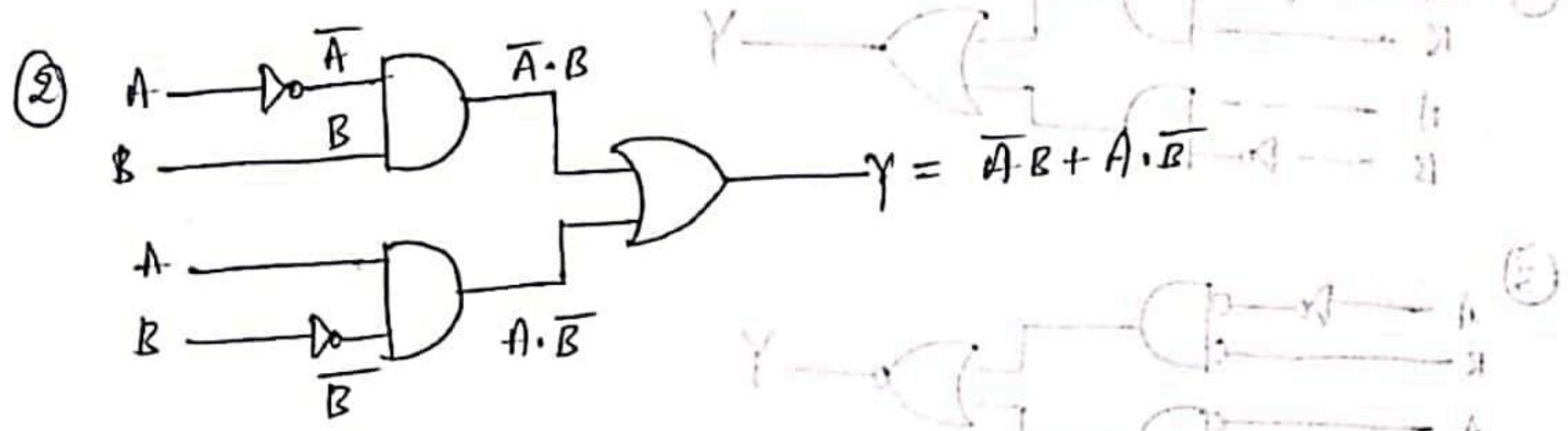
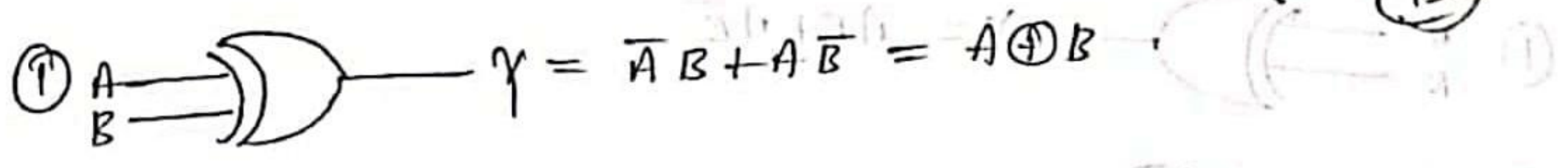


③ Realization of OR gate using NAND & NOR gates: ①

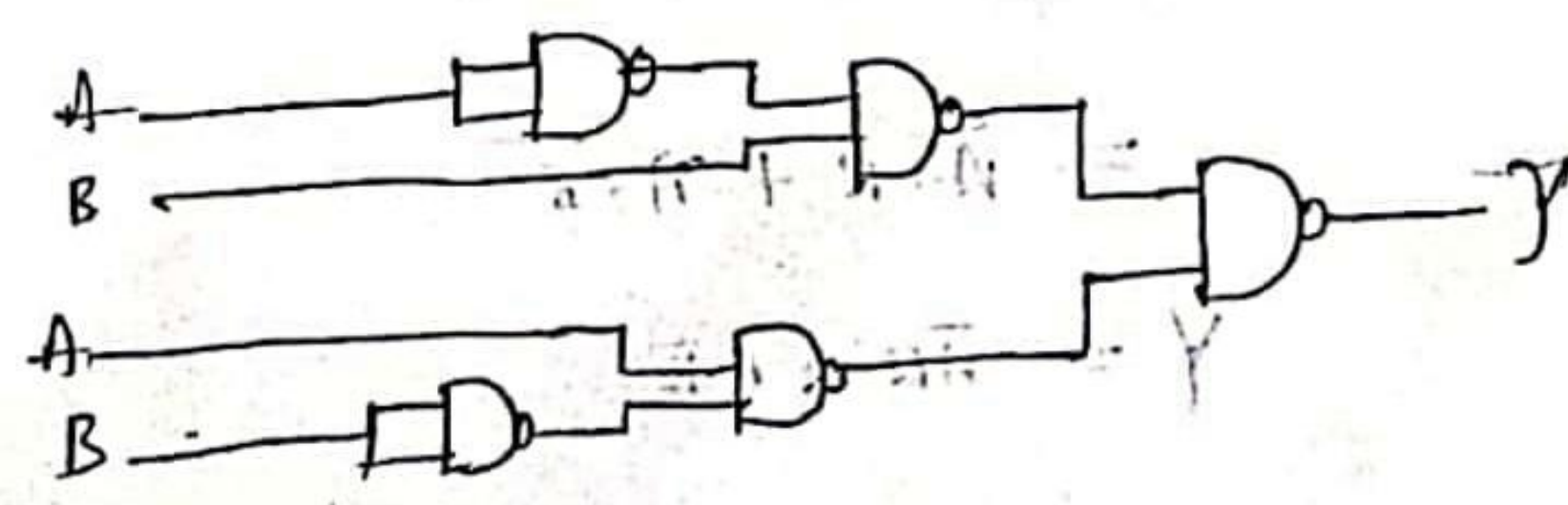


Realization of EX-OR gate using NAND & NOR using NAND gate :- (27)

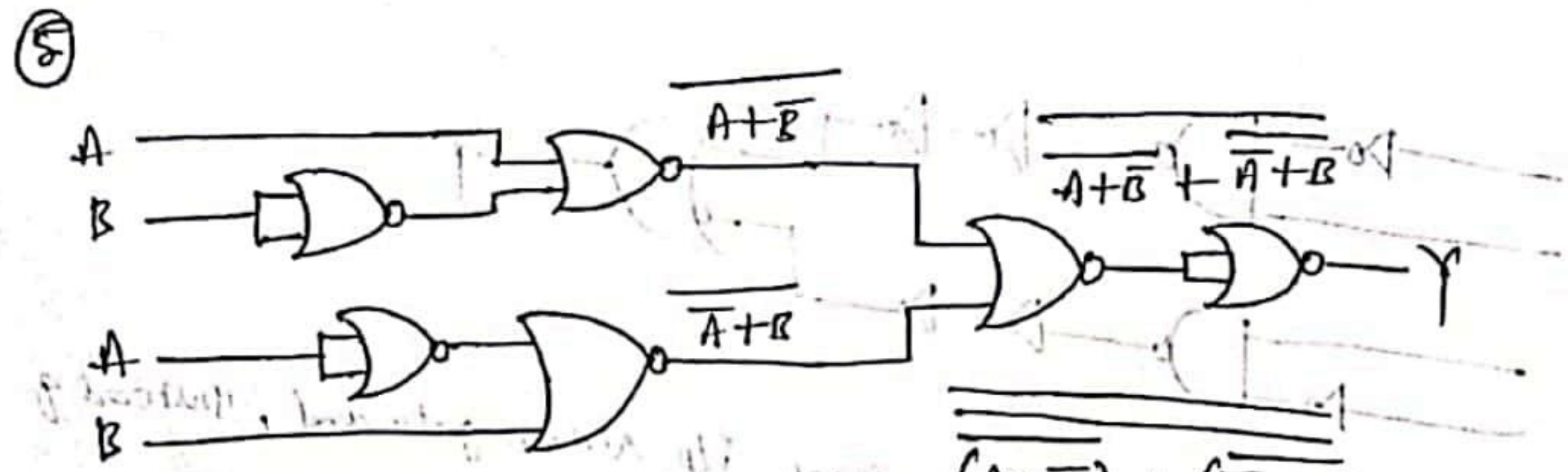
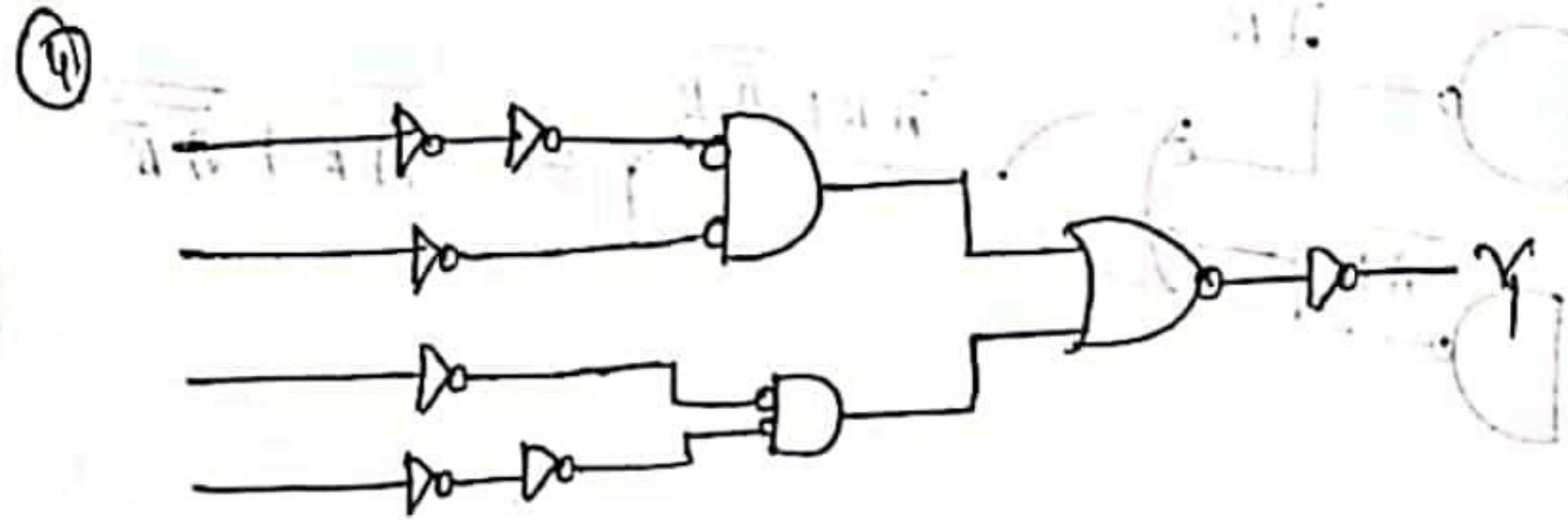
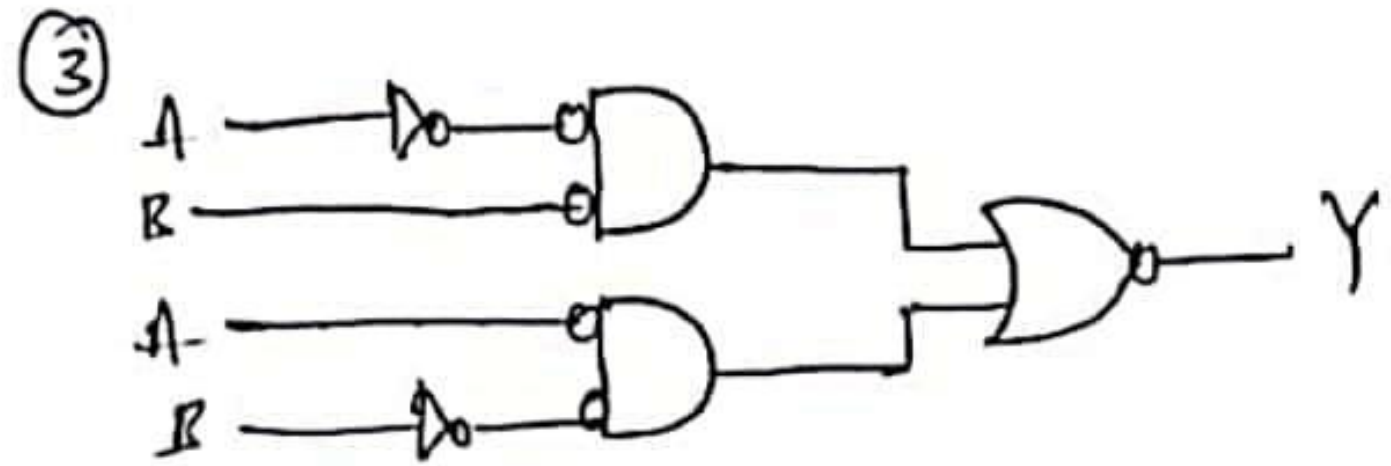
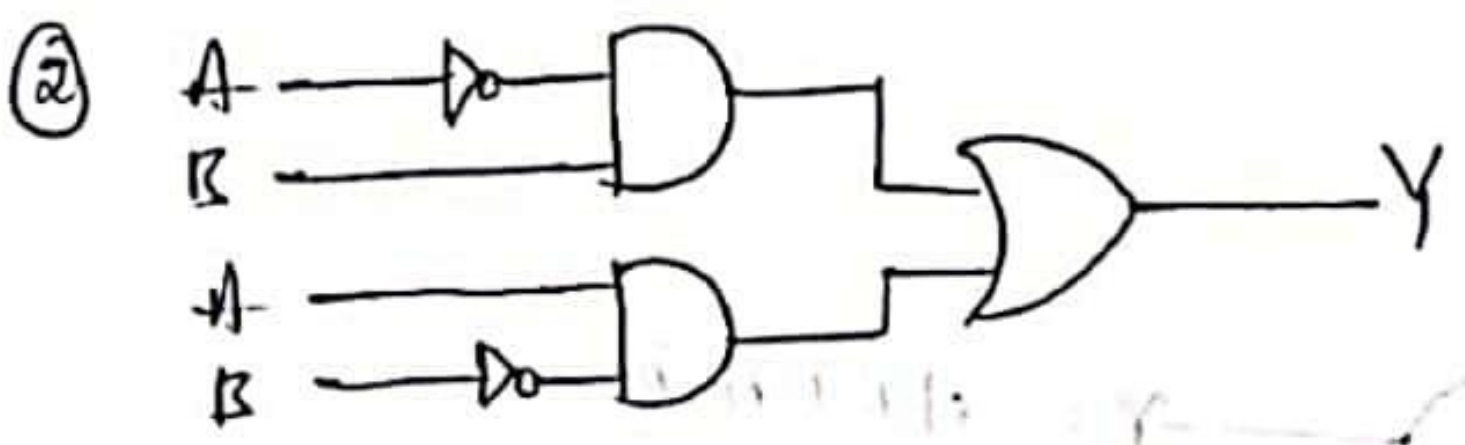
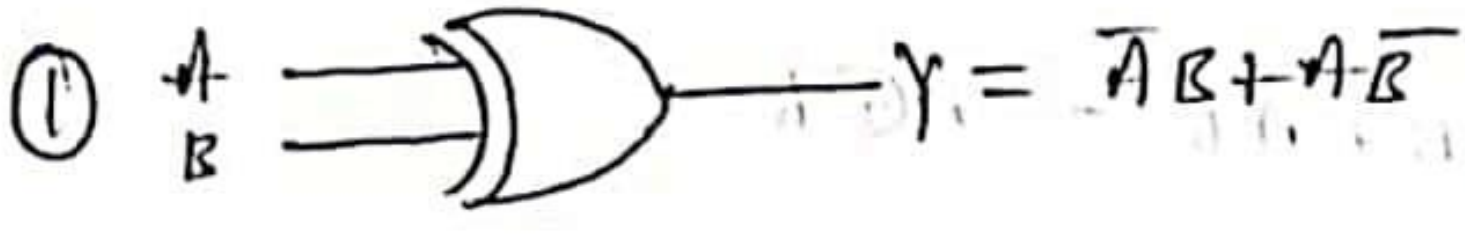
(130)



⑤ Instead of not gate place single 1/p NAND gate and, instead of bubbled OR gate place NAND gate



⇒ Using NOR Gate :-



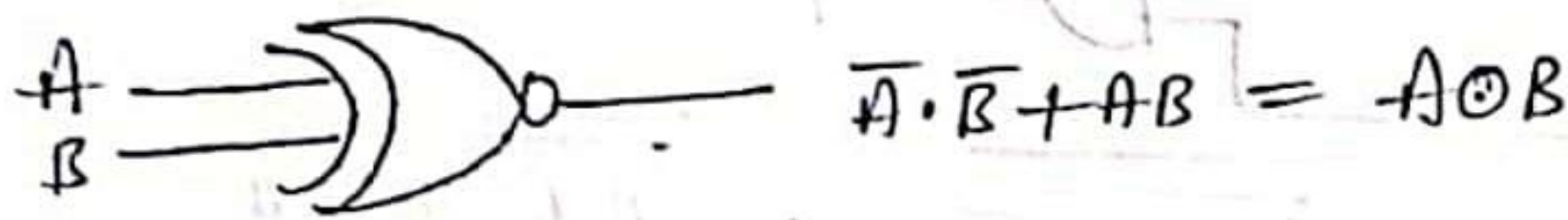
$$= \overline{(A+\bar{B})} + \overline{(\bar{A}+B)}$$

$$= \overline{(A+\bar{B})} + \overline{(\bar{A}+B)}$$

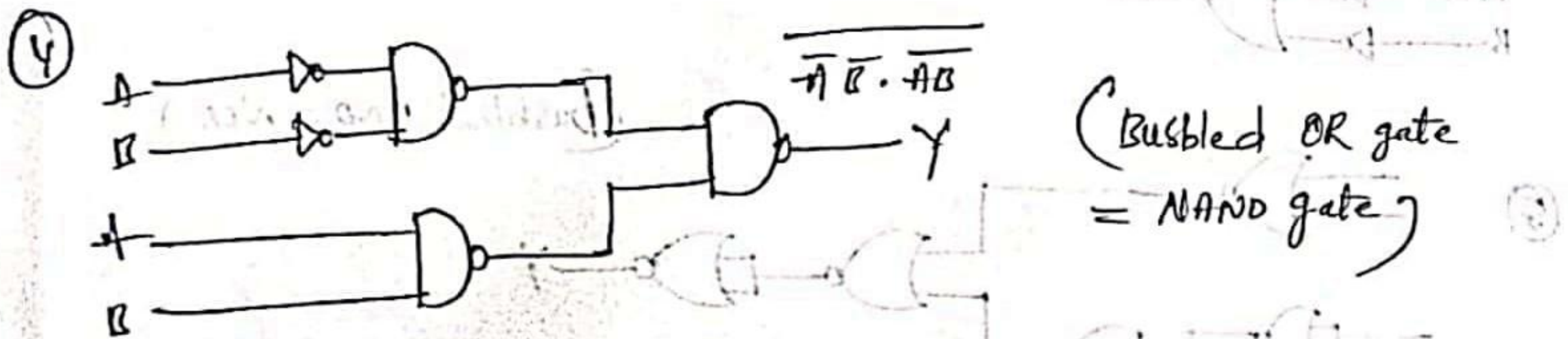
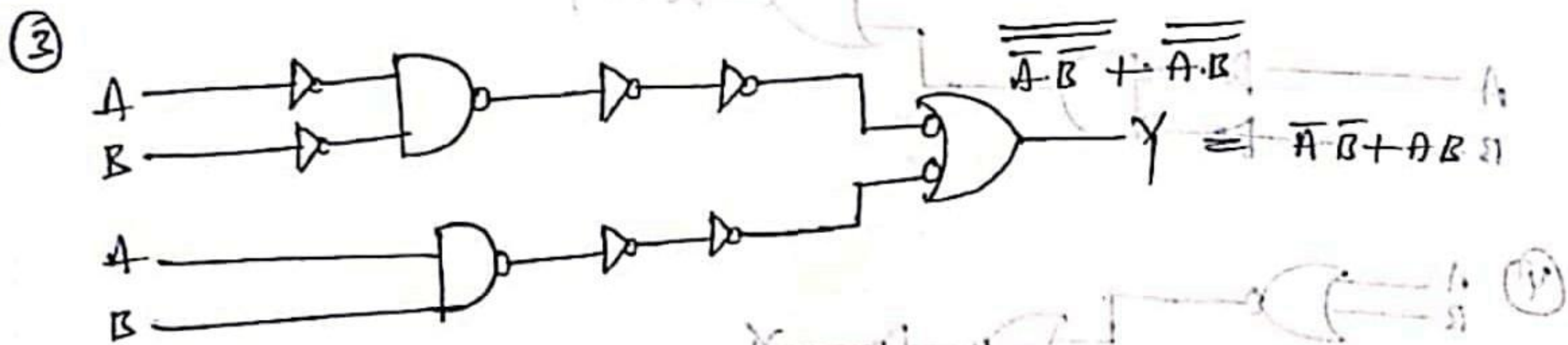
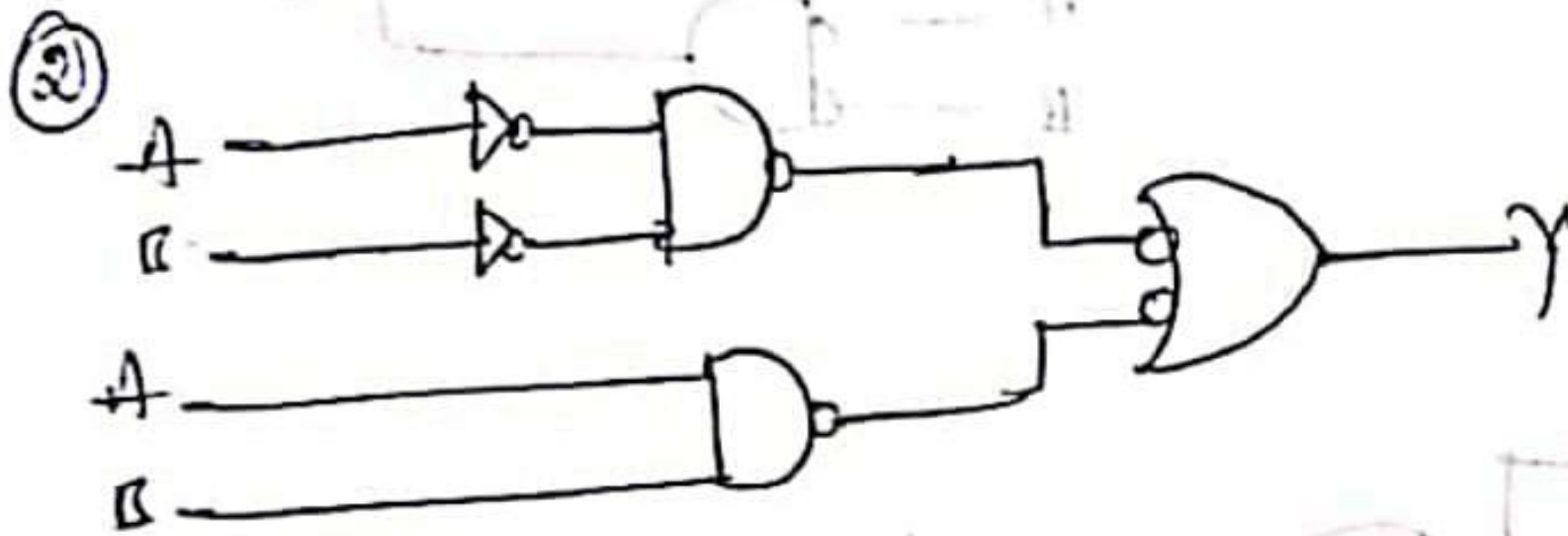
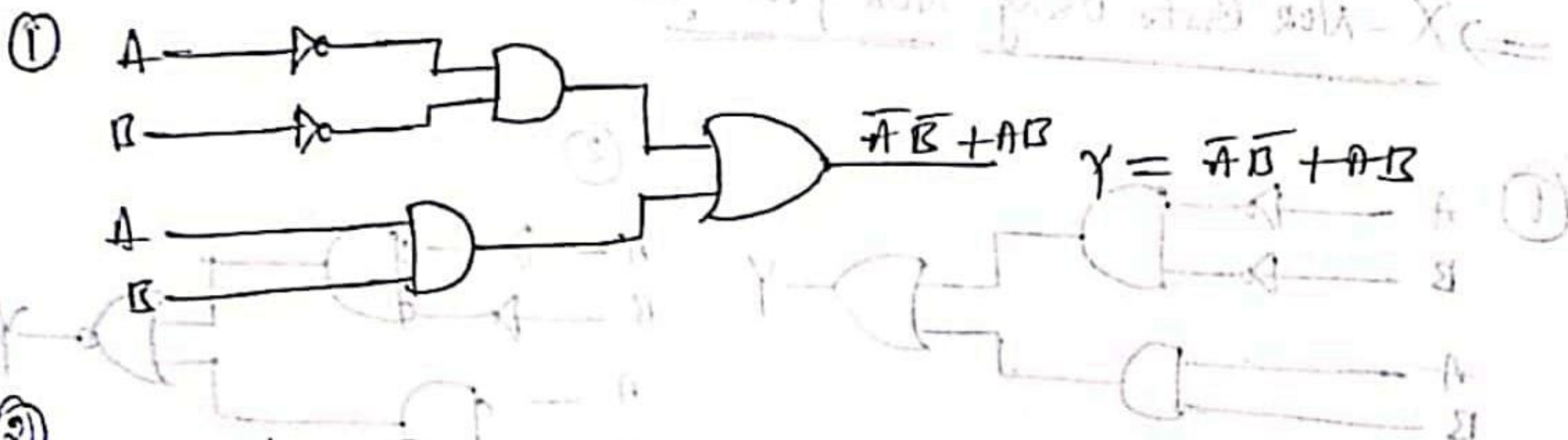
$$= \bar{A} \cdot B + A \cdot \bar{B}$$

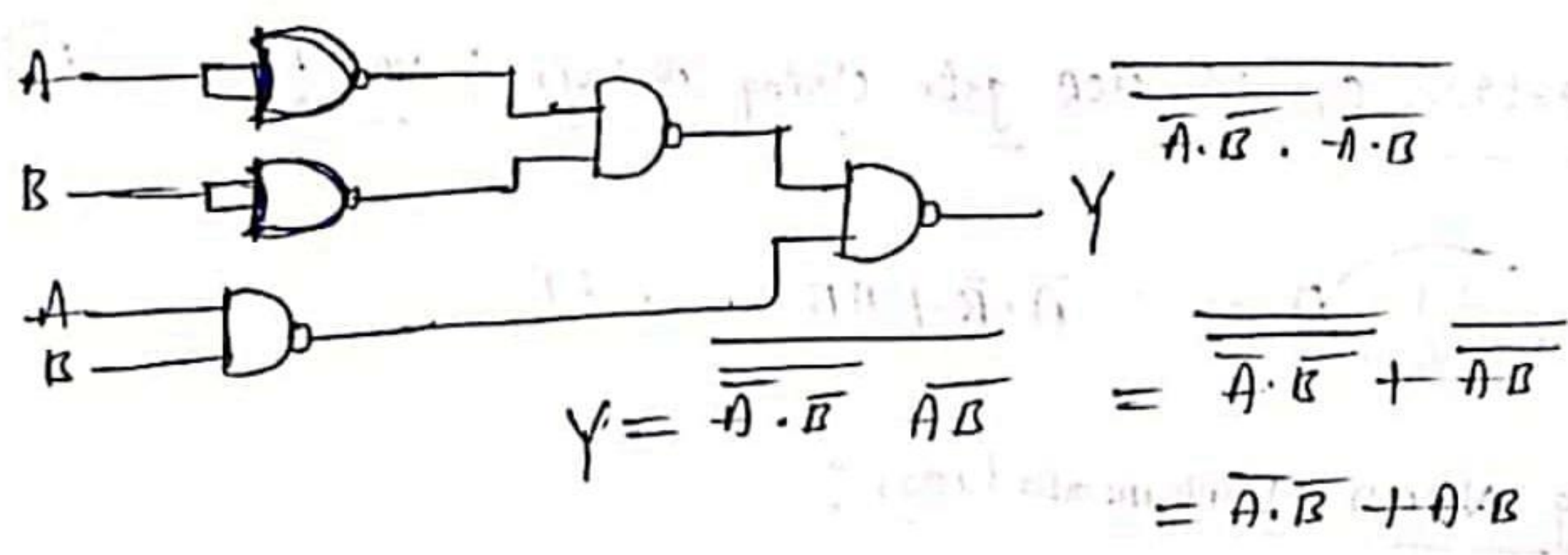
$$Y = \bar{A}B + A\bar{B}$$

⇒ Realization of X-NOR gate using NAND & NOR :- (28)

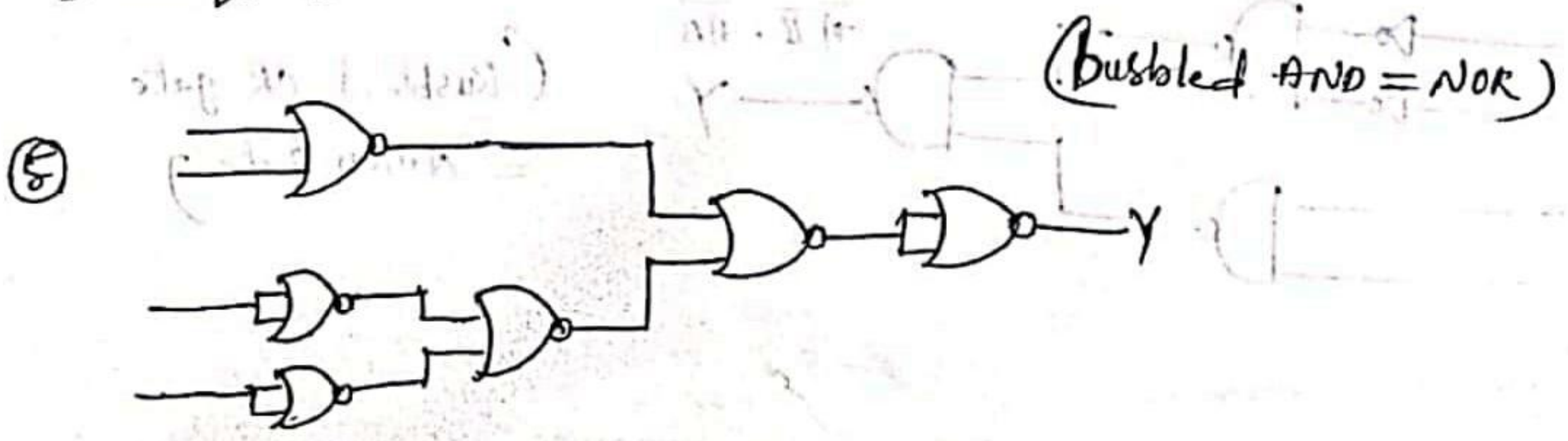
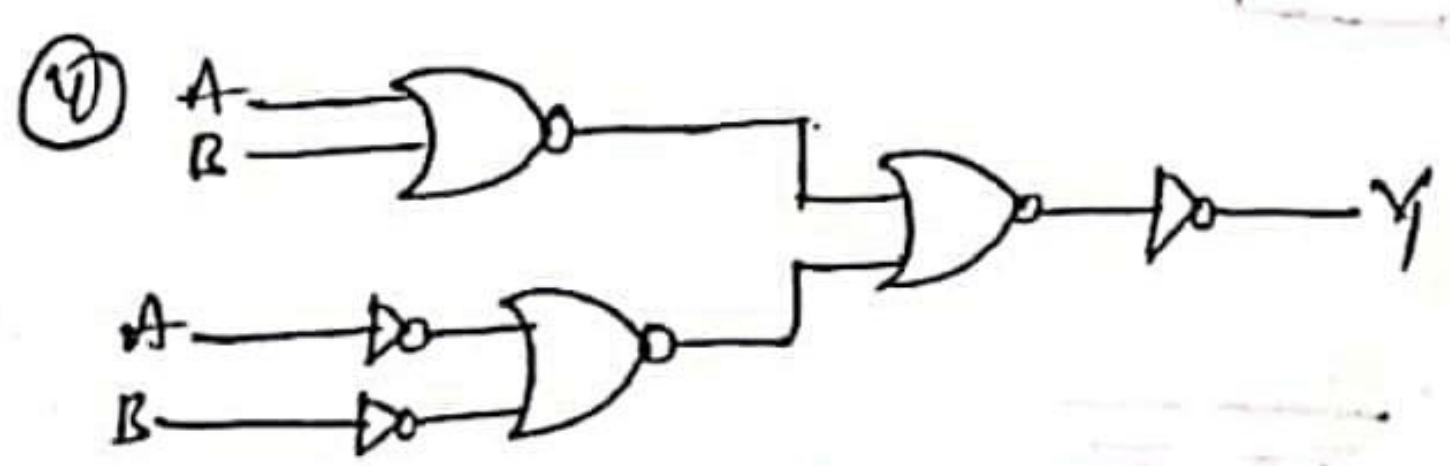
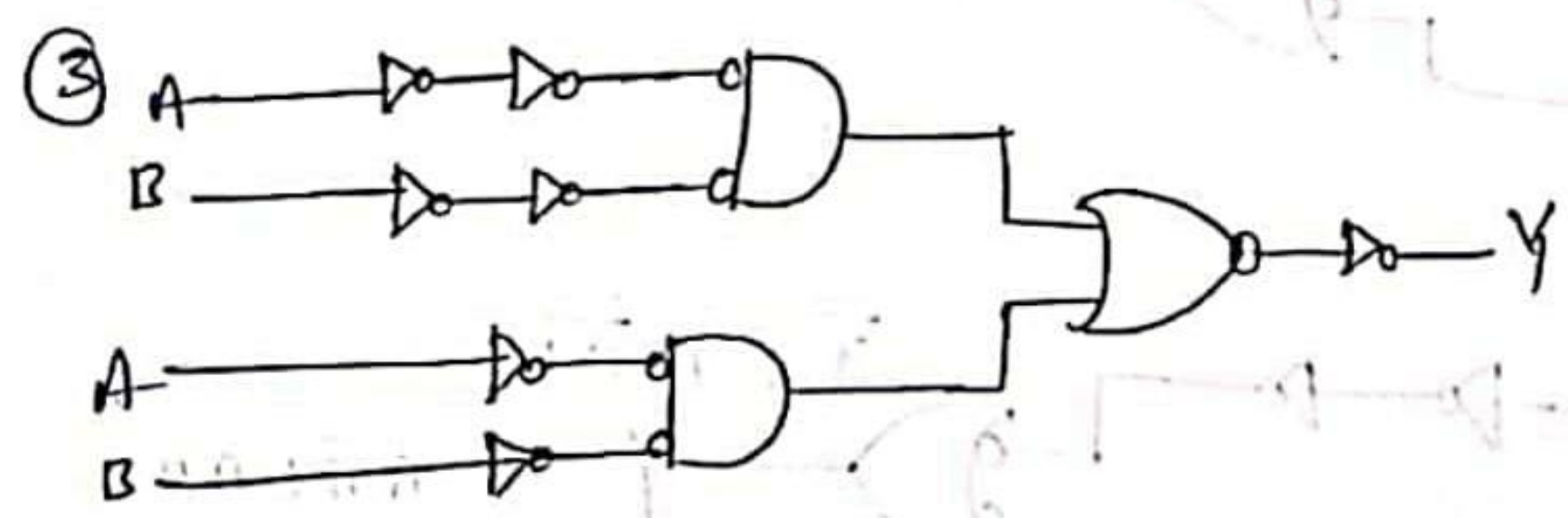
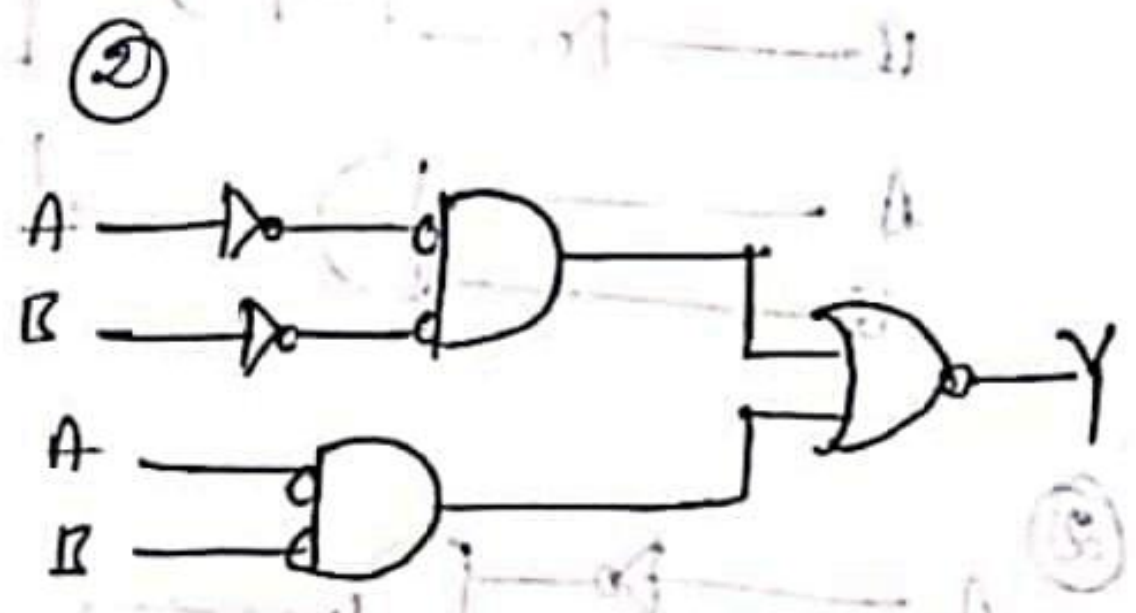
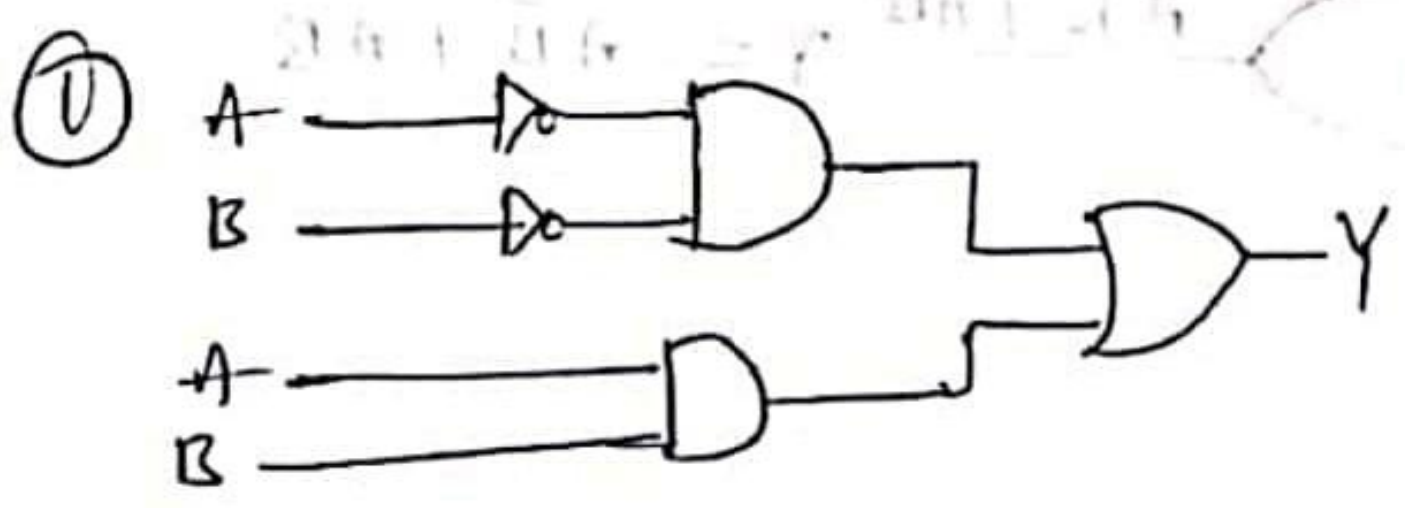


⇒ Using NAND Implementation :-





X-NOR Gate using NOR gate :-



⇒ Implement the following functions using NAND Gates:

134

(a) $F_1 = A(B+CD) + \overline{BC}$

(b) $F_2 = w\overline{x} + \overline{xy}(z+\overline{w})$

sol

(a) $F_1 = A(B+CD) + \overline{BC}$

$= AB + ACD + \overline{BC}$

$= AB + ACD + \overline{B} + \overline{C}$

$= \overline{AB + \overline{B} + \overline{C} + ACD}$

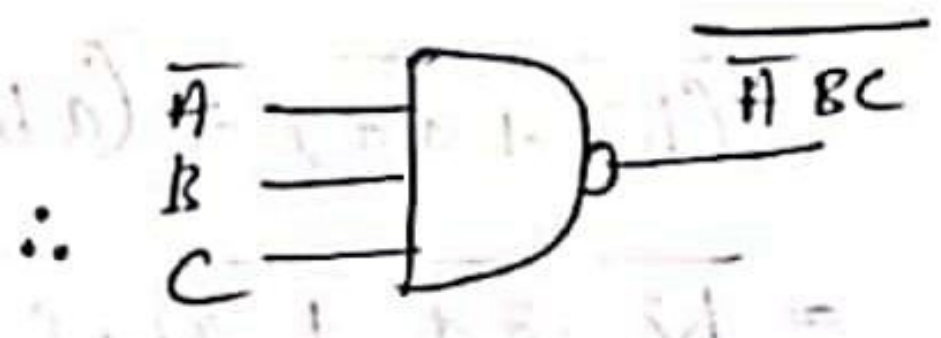
$= \overline{B + A + \overline{C} + AD}$

$= \overline{A(1+D) + \overline{B} + \overline{C}}$

$= \overline{A + \overline{B} + \overline{C}}$

$= \overline{A + \overline{B} + \overline{C}}$

$= \overline{A \cdot B \cdot C}$



$\therefore \overline{B} + AB = \overline{B} + A$

$\overline{C} + ACD = \overline{C} + AD$

Redundant Law) R.K.R

$\therefore 1+D=1$

(b) $F_2 = w\overline{x} + \overline{xy}(z+\overline{w})$

$= w\overline{x} + \overline{xy}z + \overline{xy}\overline{w}$

$= \overline{x}(w + \overline{xy}z) + \overline{xy}\overline{w}$

$= \overline{x}(w + y) + \overline{xy}\overline{w}$

$= \overline{x}w + \overline{xy} + \overline{xy}\overline{w}$

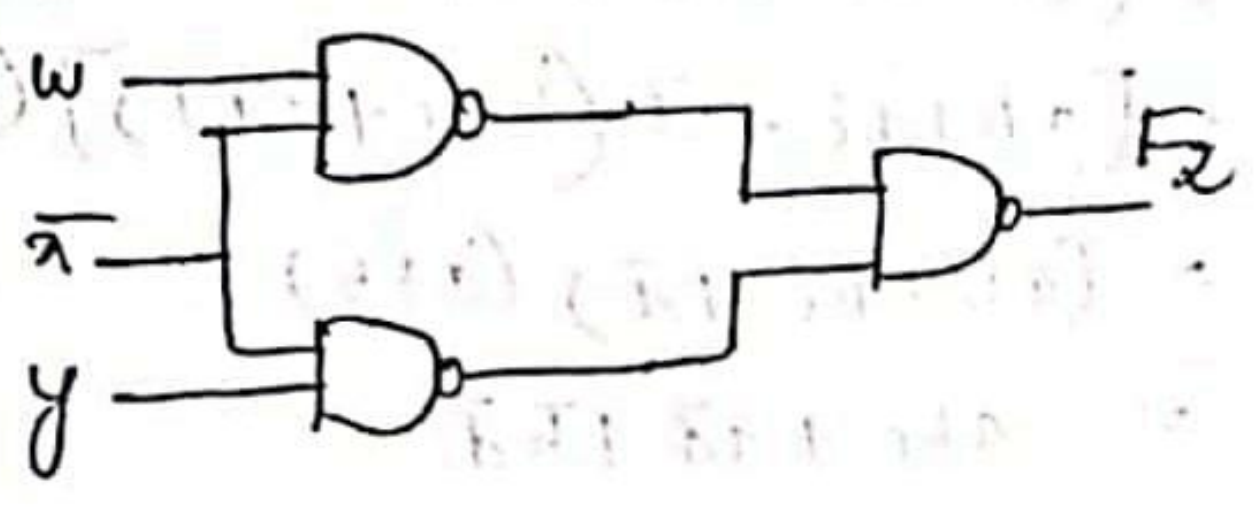
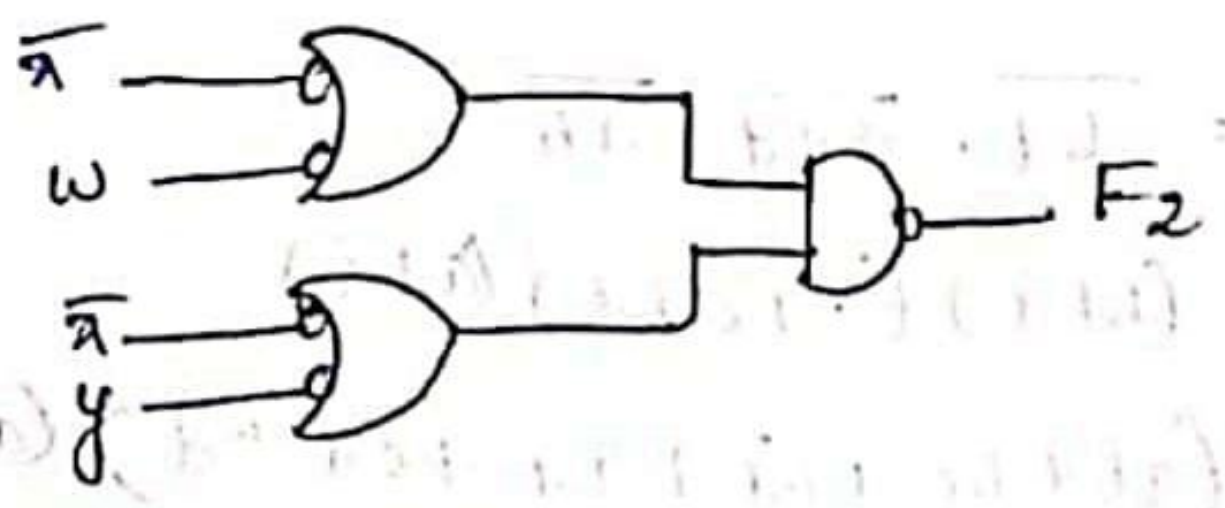
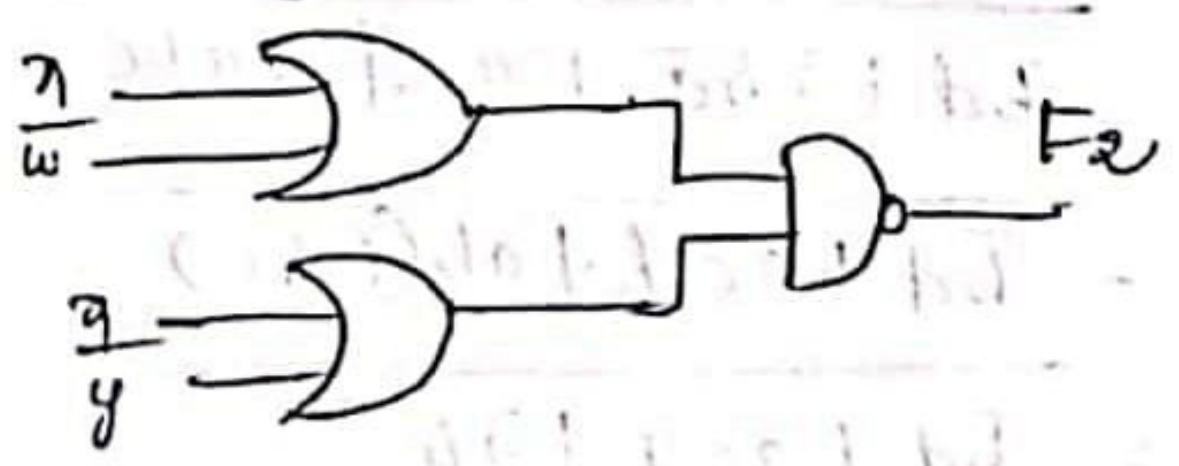
$= \overline{x}w + \overline{xy}(1+z)$

$= \overline{x}w + \overline{xy}$

$= \overline{x}(w+y)$

$= \overline{x}(w+y)$

$\overline{xw + \overline{xy}}$
 $= \overline{\overline{xw} \cdot \overline{\overline{xy}}}$
 $= \overline{(\overline{x+\overline{w}})(\overline{x+y})}$



⇒ Find the complement of the following boolean function & reduce them to minimum no. of literals.

135

(A) $(b\bar{c} + \bar{a}d)(\bar{a}\bar{b} + c\bar{d})$

(B) $(\bar{b}d + \bar{a}b\bar{c} + acd + \bar{a}bc)$

Sol (A) $\overline{(b\bar{c} + \bar{a}d)(\bar{a}\bar{b} + c\bar{d})}$

$$= \overline{(b\bar{c} + \bar{a}d)} + \overline{(\bar{a}\bar{b} + c\bar{d})}$$

$$= \overline{b\bar{c}} \cdot \overline{\bar{a}d} + \overline{\bar{a}\bar{b}} \cdot \overline{c\bar{d}}$$

$$= (\bar{b} + c)(a + d) + (\bar{a} + b)(\bar{c} + d)$$

$$= a\bar{b} + \bar{b}d + ac + c\bar{d} + \bar{a}c + \bar{a}d + b\bar{c} + bd$$

$$= a\bar{b} + ac + \bar{b}d + c\bar{d} + \bar{a}c + \bar{a}d + b\bar{c} + bd$$

$$= 1$$

(B) $\overline{\bar{b}d + \bar{a}b\bar{c} + acd + \bar{a}bc}$

$$= \overline{\bar{b}d + acd + \bar{a}b(\bar{c} + c)}$$

$$= \overline{\bar{b}d + acd + \bar{a}b}$$

$$= \overline{\bar{b}d} \cdot \overline{acd} \cdot \overline{\bar{a}b}$$

$$= (b + \bar{d})(\bar{a} + \bar{c} + d)(a + b)$$

$$= (\bar{a}b + b\bar{c} + b\bar{d} + \bar{a}\bar{d} + \bar{c}\bar{d} + \bar{d})(a + b)$$

$$= [\bar{a}b + b\bar{c} + \bar{d}(b + \bar{a} + \bar{c} + 1)](a + b)$$

$$= (\bar{a}b + b\bar{c} + \bar{d})(a + b)$$

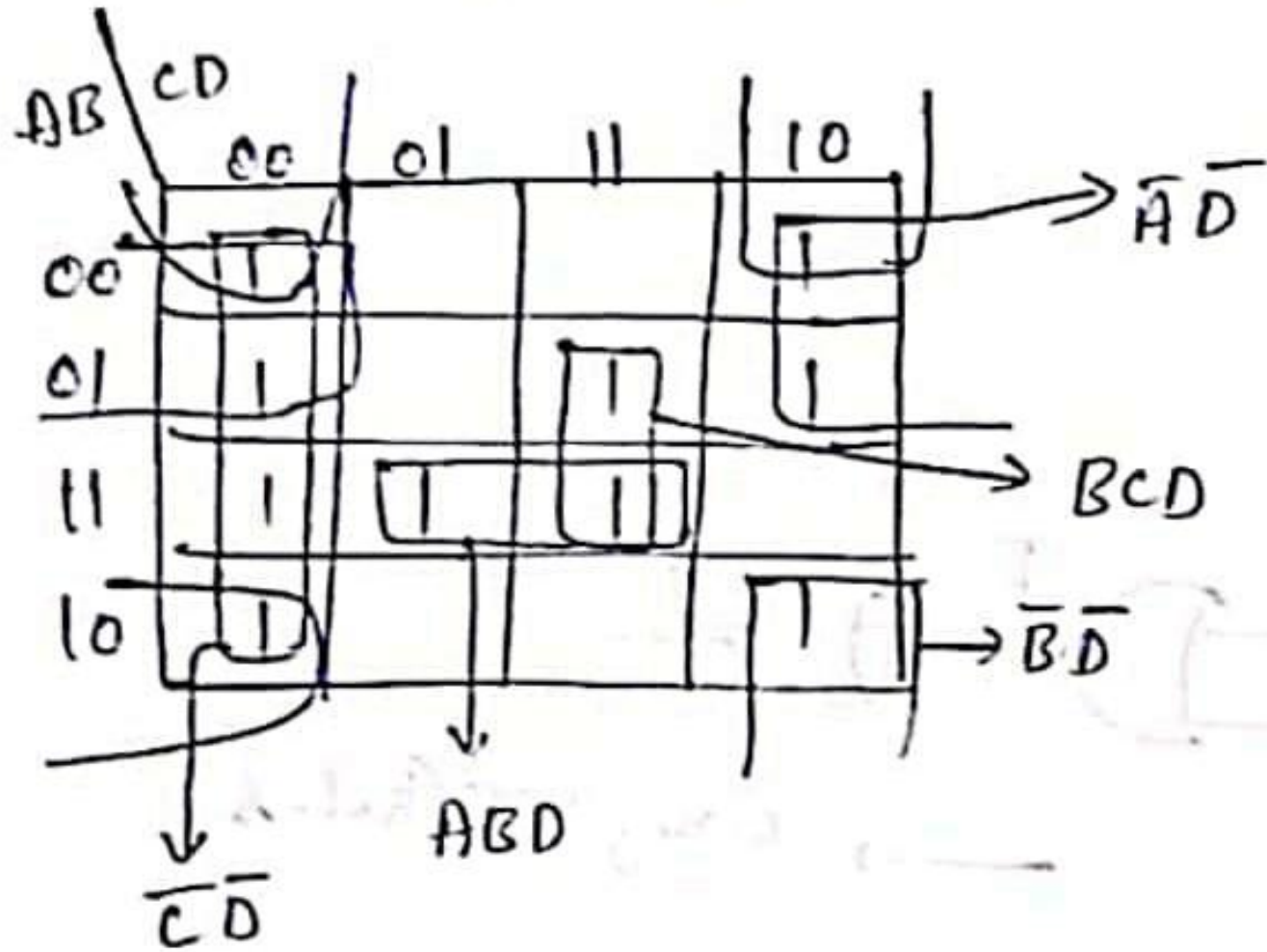
$$= a\bar{b}\bar{c} + a\bar{d} + \bar{b}d$$

Reduce using mapping the following expression and implement the real minimal expression in universal logic

136

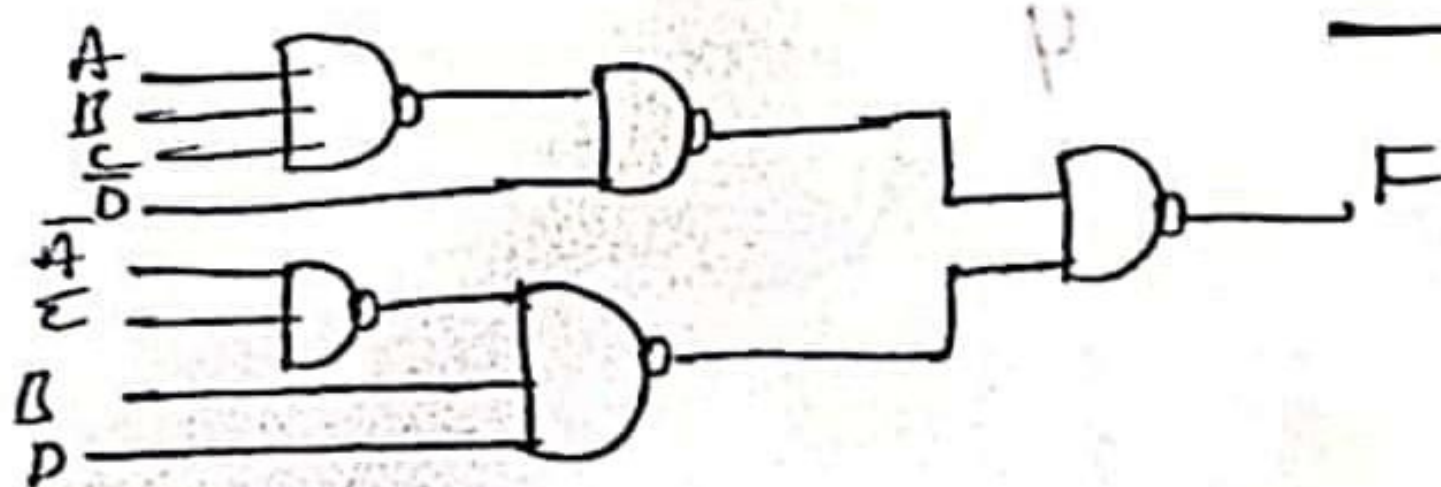
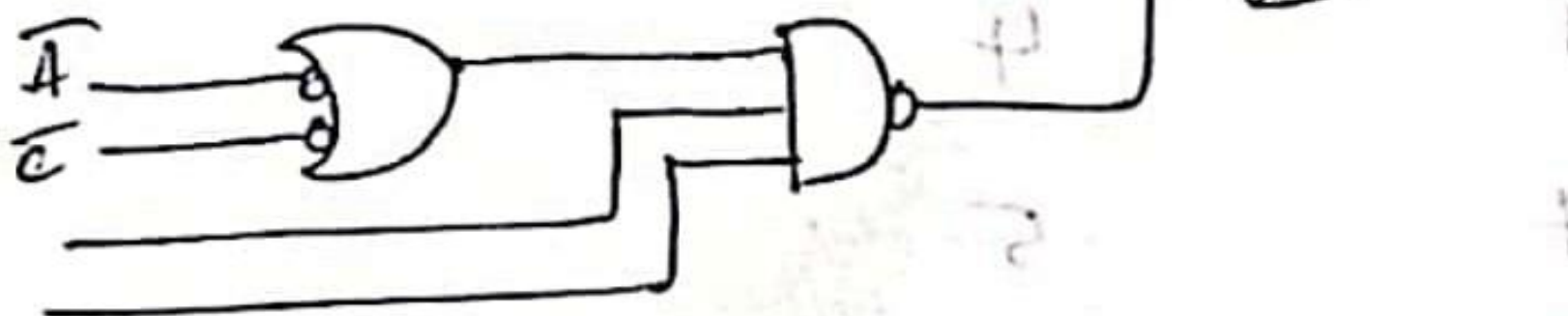
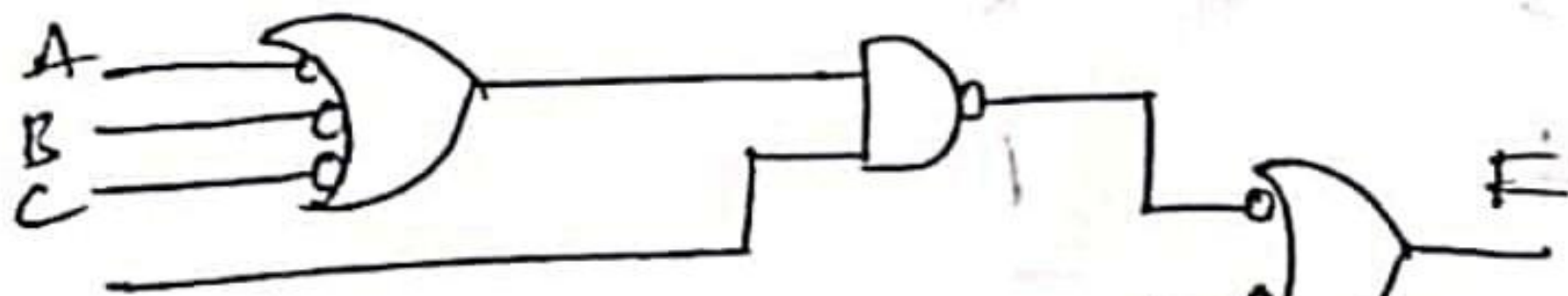
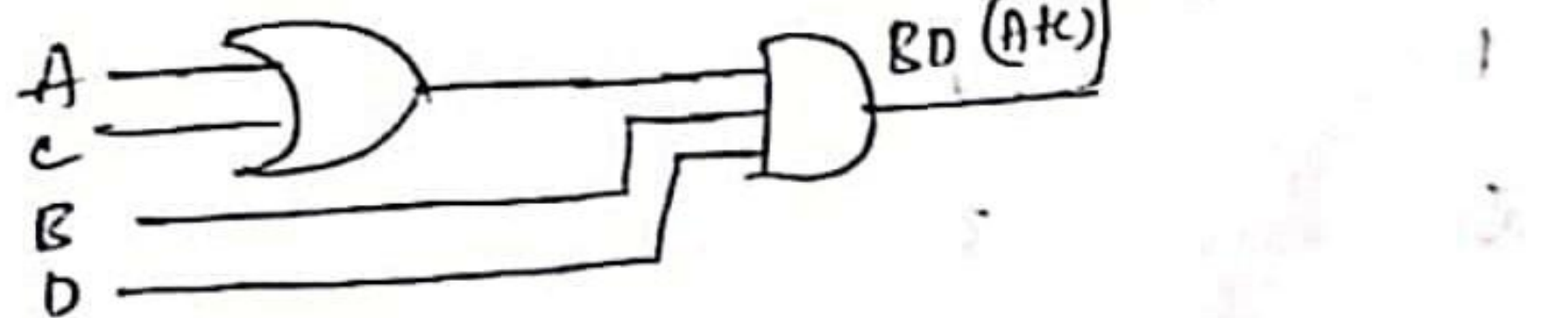
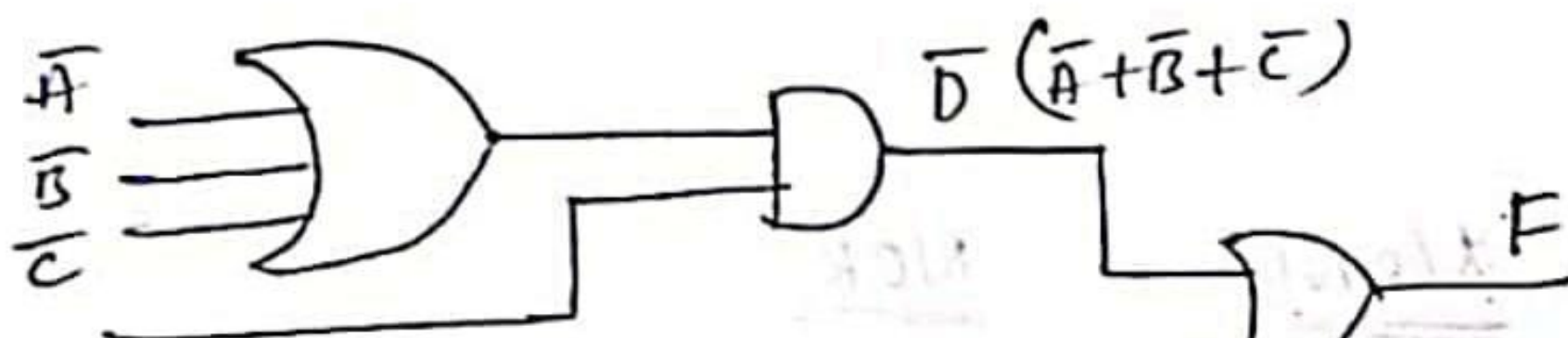
$$F = \sum m(0, 2, 4, 6, 7, 8, 10, 12, 13, 15)$$

Sol

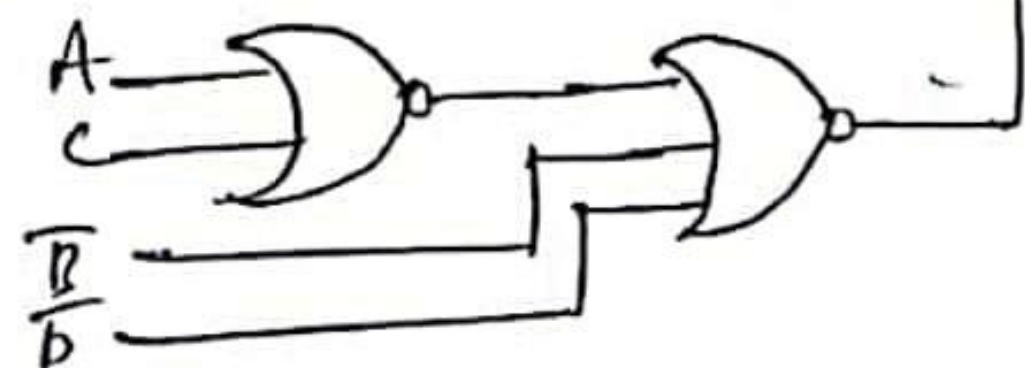
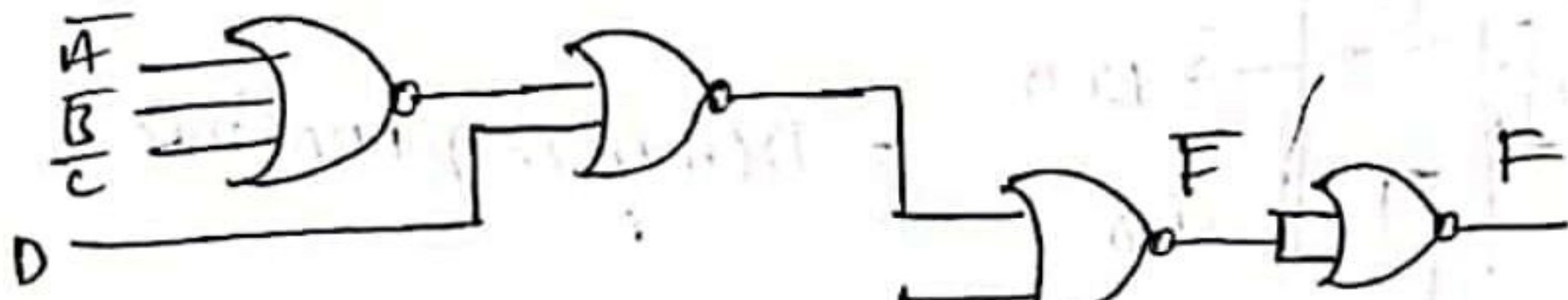
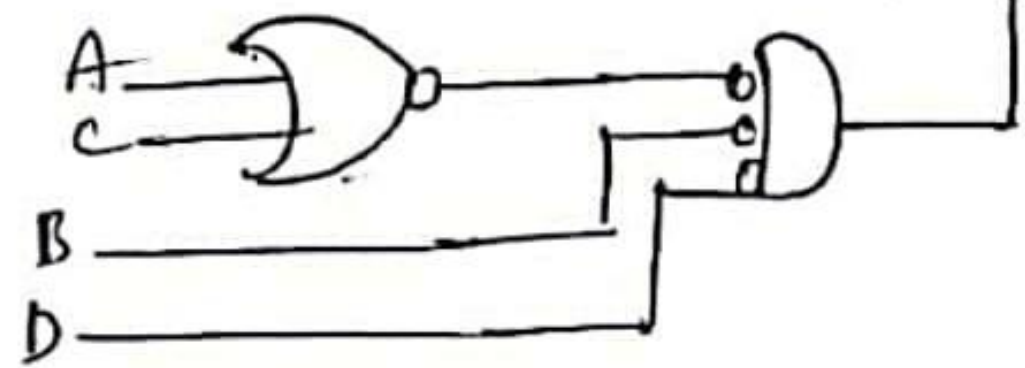
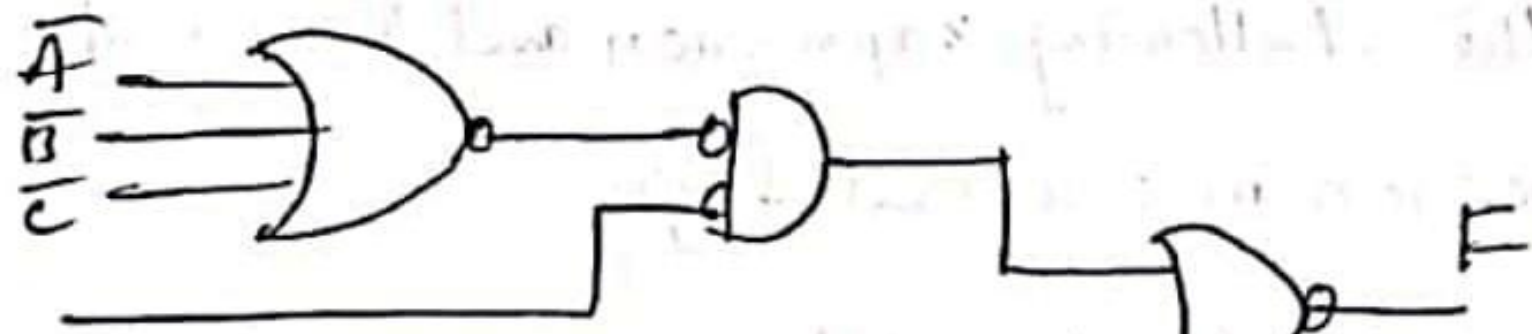


$$F = \bar{C}\bar{D} + \bar{A}\bar{D} + \bar{B}\bar{D} + ABCD + BCD$$

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



(Using NAND gates)



→ using NOR Gates

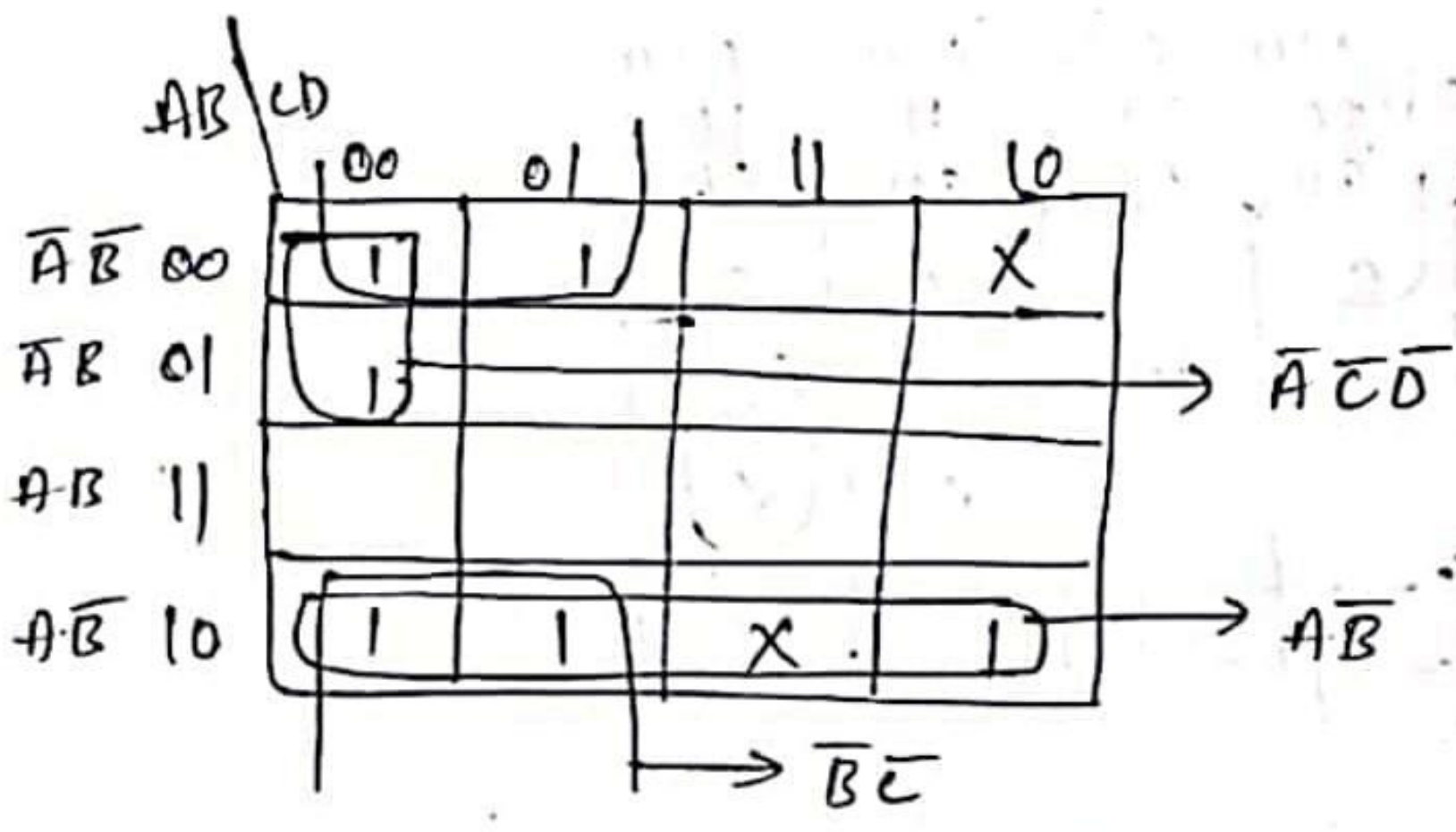
	<u>NAND</u>	<u>NOR</u>
NOT	1	1
AND	2	3
OR	3	2
NOR	4	1
NAND	1	4
X-OR	4	5
X-NOR	5	4

Q. Reduce the following expression using K-map and implement it using NAND gate.

138

$f = \sum m(3, 5, 6, 7, 12, 13, 14, 15) + d(2, 11)$

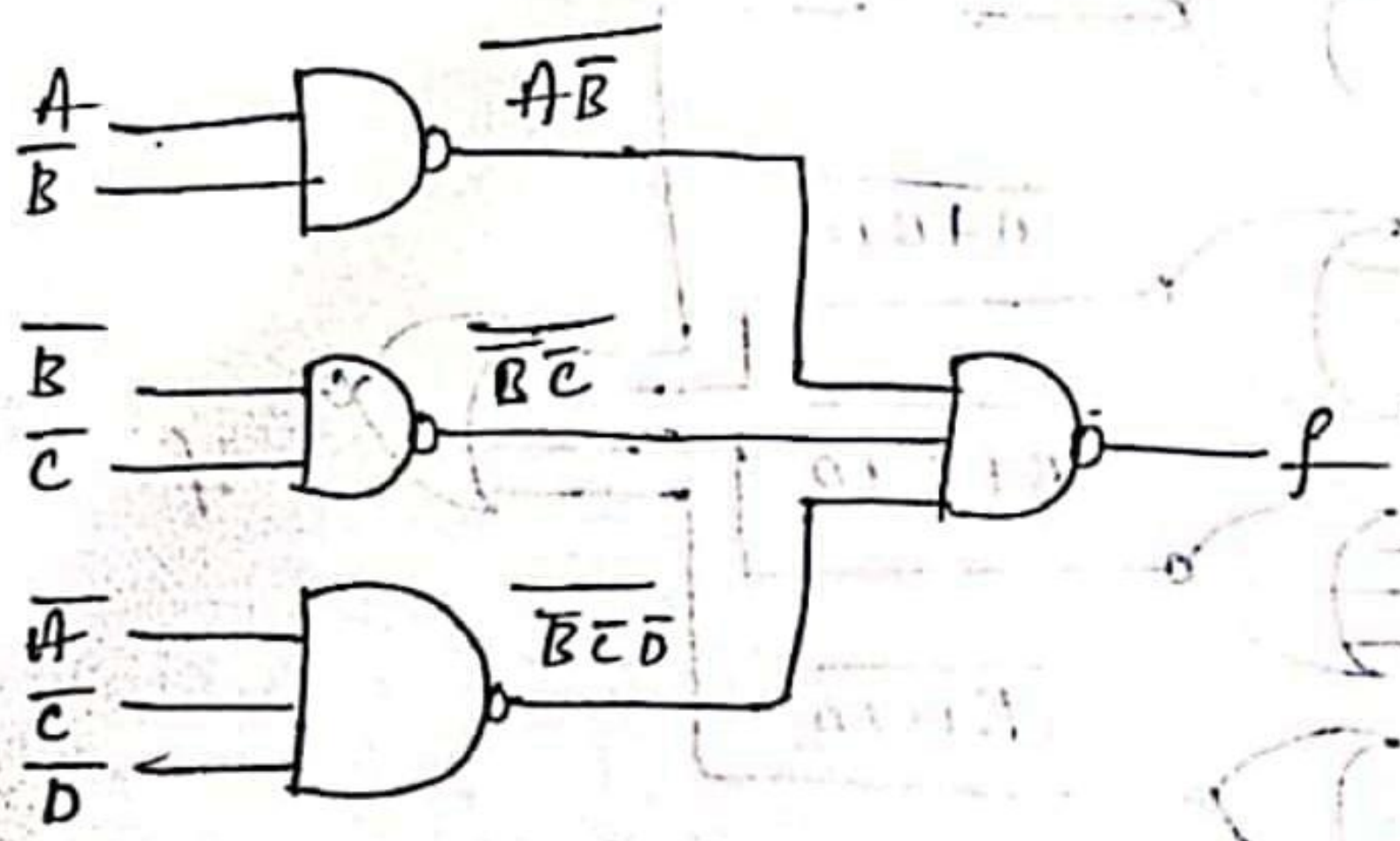
$\Rightarrow \sum m(0, 1, 4, 8, 9, 10) + d(2, 11)$



$f = \bar{A}\bar{B} + \bar{B}\bar{C} + \bar{A}\bar{C}\bar{D}$

W.K.T $f = \overline{\overline{\bar{A}\bar{B} + \bar{B}\bar{C} + \bar{A}\bar{C}\bar{D}}}$

$f = \overline{\bar{A}\bar{B} \cdot \bar{B}\bar{C} \cdot \bar{A}\bar{C}\bar{D}}$



Q

Minimize the expression using K-map and implement it using NOR gate

139

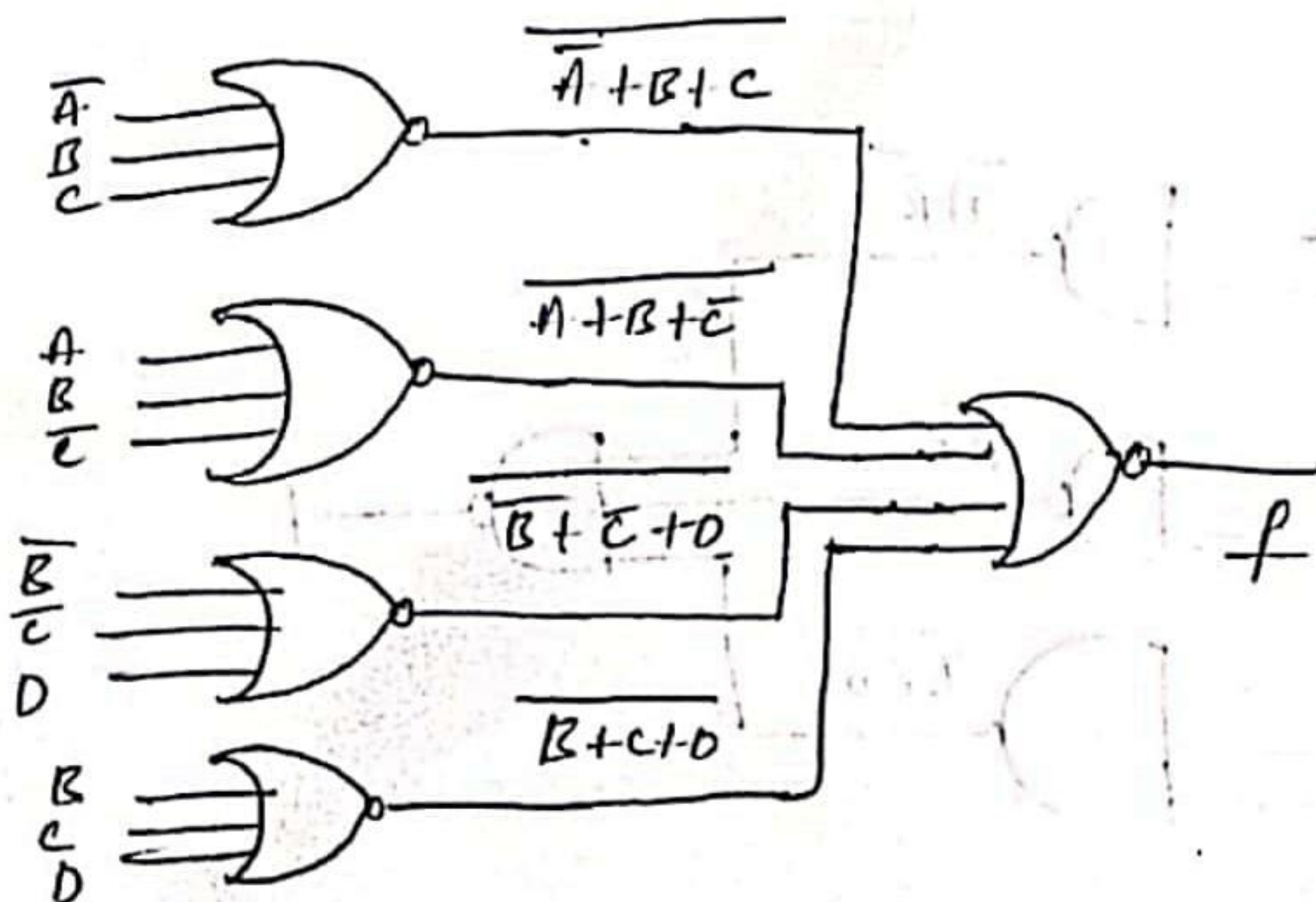
$$f(A, B, C, D) = \sum m(1, 4, 7, 10, 11, 12, 13) + \sum d(5, 14, 15)$$
$$= \prod M(0, 2, 3, 6, 8, 9) + \sum d(5, 14, 15)$$

AB		C			
		D	\bar{D}	D	\bar{D}
A+B	$\bar{A}\bar{B}$	00	01	11	10
A+B	$\bar{A}B$	01	X		0
$\bar{A}+B$	AB	11		X	X
$\bar{A}+B$	$A\bar{B}$	10	0		

$$\Rightarrow (\bar{A}+B+C)(A+B+\bar{C})(\bar{B}+\bar{C}+D)(B+C+D)$$

$$f = \overline{(\bar{A}+B+C)(A+B+\bar{C})(\bar{B}+\bar{C}+D)(B+C+D)}$$

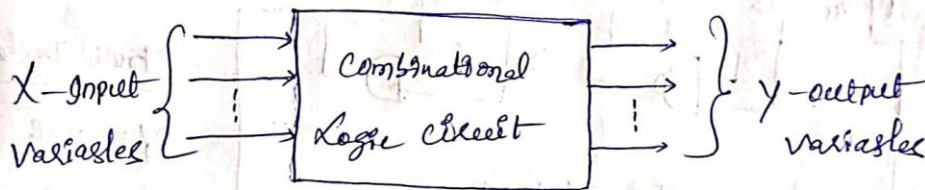
$$f = \overline{(\bar{A}+B+C) + (A+B+\bar{C}) + (\bar{B}+\bar{C}+D) + (B+C+D)}$$



UNIT-II

⇒ Combinational circuits :- Digital Logic circuits

- Logic circuits for digital systems may be combinational (or) sequential.
- In combinational circuits, the output variables at any instant of time are dependent only on the present input variables.
- A combinational circuit consists of input variables, logic gates and output variables.
- Logic gates accept signals from the input and generate signals to the outputs.



⇒ Design procedure of combinational circuit :-

The design of combinational circuits starts from the specification of the problem that can be implemented in a logic circuit diagram or a set of Boolean function.

- ① From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
- ② Derive the truth table that defines the required relationship between inputs and outputs.
- ③ Obtain the boolean function and draw the logic diagram.

⇒ Integrated NAND-NOR Gates :-

NAND gate is actually a series of AND gate with NOT gate. If we connect the output of an AND gate to the i/p of a NOT gate, this combination will work as NOT-AND (or) NAND gate. Its output is 1 when any or all inputs are 0, otherwise o/p is 1.

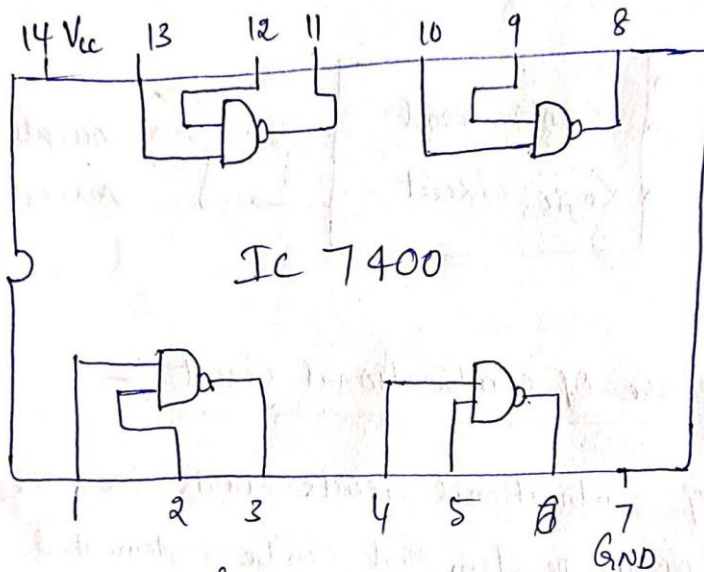


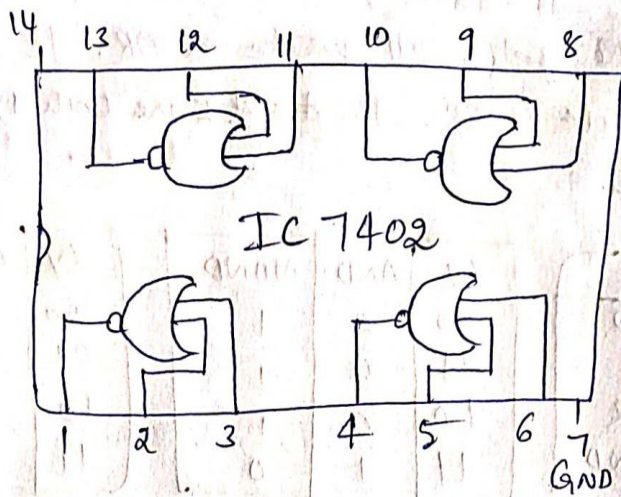
fig IC 7400

Truth Table

$$Y = \overline{A \cdot B}$$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR gate is actually a series of OR gate with NOT gate. If we connect the output of an OR gate to the i/p of a NOT gate, this combination will work as NOT-OR or NOR gate. Its output is 0 when any (or) all inputs are 1, otherwise output is 1.

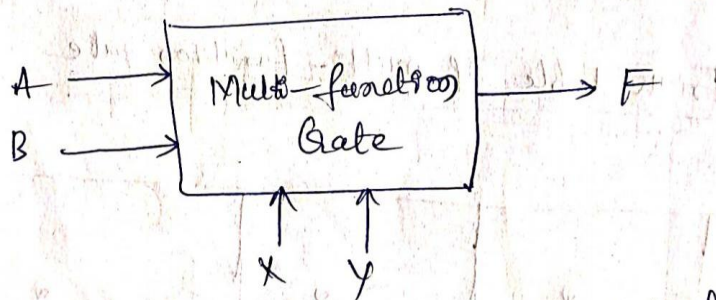


Truth Table

$$Y = \overline{A+B}$$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

⇒ Multifunction gates :- the gates which are doing multi-function, those type of gates are called multifunction gates. Many functions will perform by these gates.



⇒ Design Specification plan :- the idea is to design a multifunction gate that will have sets of inputs, and one output F. The function F will be instructed to perform four different logic operations. A and B are the data inputs. X and Y are control what the gate will do. X and Y are the operation selection lines.

⇒ Design methodology :- A and B tell the gate what operation to perform. If A and B both are 0, the gate will act as an AND gate.

If $A=0, B=1$, the gate will operate as OR. If $A=1, B=0$ the gate will operate as NOR. If A and B are both 1, the gate operates as NAND.

∴

A	B	XY
0	0	AND
0	1	OR
1	0	NOR
1	1	NAND

XY	AND	NAND
00	0	1
01	0	1
10	0	1
11	1	0

XY	OR	NOR
00	0	1
01	1	0
10	1	0
11	1	0

X and Y depend upon what A and B are doing. The truth tables for AND, NAND, OR and NOR can be seen ^{above} for figures. The three above figures can be condensed into a lengthy truth table as shown below figure.

Truth table of multi-function gate

Gates	AND gate	OR gate	NOR gate	NAND gate
Z	0 0 0 1	0 1 1 1	1 0 0 0	1 1 1 0
XY	00 01 10 11	00 01 10 11	00 01 10 11	00 01 10 11
AB	00 00 00 00	01 01 01 01	10 10 10 10	11 11 11 11

It can easily be seen how the truth table can get rather complicated. Karnaugh (K-map) map would be a more convenient way to represent the multi-function gate

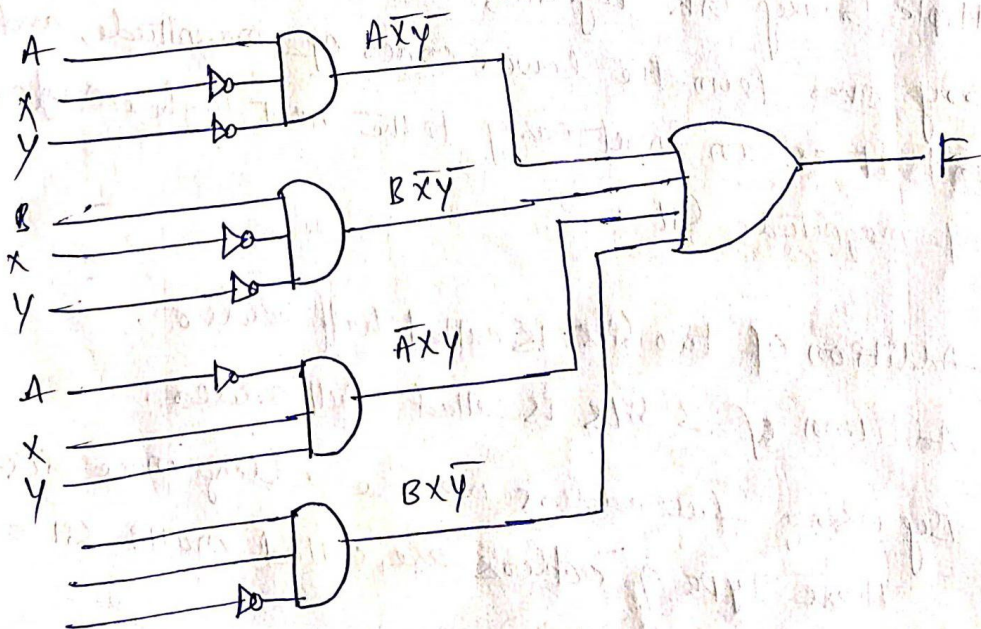
xy	AB	$\overline{A}B$	$A\overline{B}$	$\overline{A}\overline{B}$
	00	01	11	10
$\overline{x}\overline{y}$ 00	0	1	1	0
$\overline{x}y$ 01	0	1	0	0
$x\overline{y}$ 11	1	0	1	0
xy 10	0	1	0	1

$\Sigma m(2, 3, 5, 7, 9, 11, 12, 13)$
 By using the ones on the table, the function can be written as a sum of products (SOP)

To do this you need $\overline{A}Bxy$ to multiply out to equal 1. Therefore, the unsimplified equation for $f(x,y)$

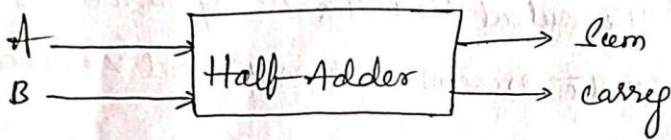
$$F = A\overline{x}\overline{y} + B\overline{x}y + \overline{A}xy + Bx\overline{y}$$

⇒ Schematic diagram :-



⇒ Half-Adder :- A half-adder is a combinational circuit with two binary inputs and two binary outputs (Sum and Carry). It adds the two inputs (A & B) and produces the Sum (S) and Carry (C) as a output bits.

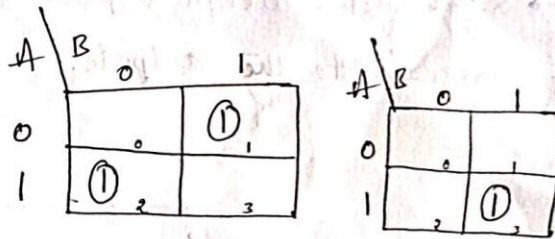
⇒ Block diagram :-



⇒ Truth table :-

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

K-map for (S) K-map for (C)



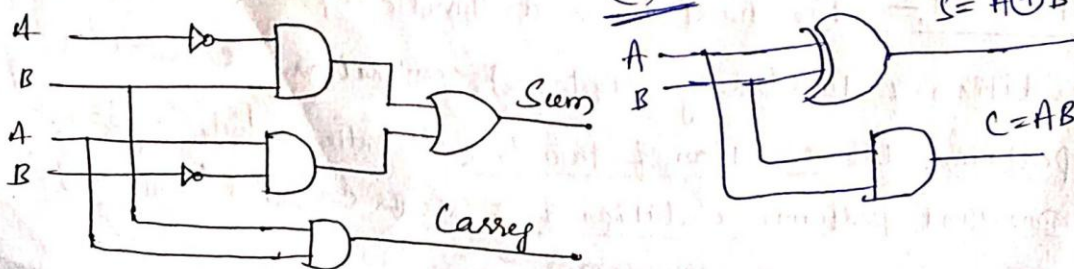
$$S = \bar{A}B + A\bar{B}$$

$$C = AB$$

$$S = A \oplus B$$

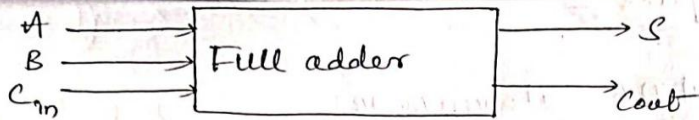
(00)

⇒ Logic diagram :-



⇒ Full Adder :- A full adder is a combinational circuit that adds two bits and a carry and outputs a sum bit and carry bit

→ The full adder adds the bits A and B and the carry from the previous column called the 'carry in' (C_{in}) and outputs the sum bit 'S' and carry bit called (C_{out}).



Block diagram

⇒ Truth table :-

Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

K-map for S

A \ C_{in}	00	01	10	11
0		1		1
1	1		1	

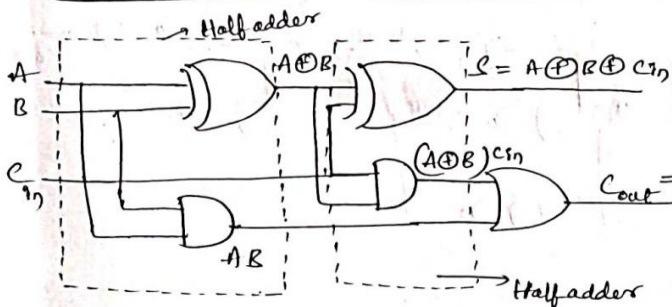
$$\begin{aligned}
 S &= \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in} \\
 &= \bar{A}(B \oplus C_{in}) + A(\bar{B} \oplus \bar{C}_{in}) \\
 &= A \oplus B \oplus C_{in}
 \end{aligned}$$

K-map for C_{out}

A \ C_{in}	00	01	11	10
0			1	
1	1	1	1	1

$$C_{out} = BC_{in} + AC_{in} + AB$$

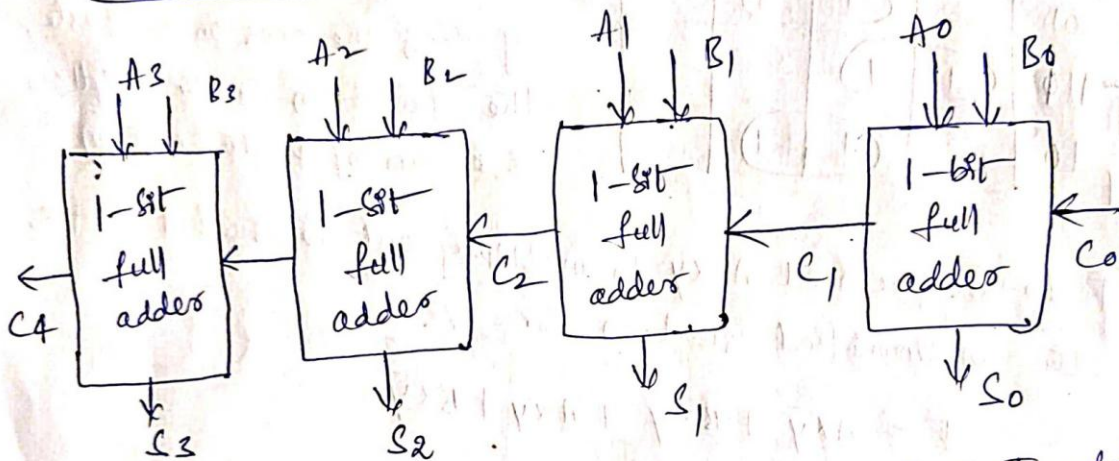
⇒ Logic diagram :- (Using two half adders)



$$\begin{aligned}
 C_{out} &= (A \oplus B) \cdot C_{in} + AB \\
 C &= \bar{A}B C_{in} + A\bar{B} C_{in} + A B C_{in} + A B \bar{C}_{in} \\
 &= C(\bar{A}B + A\bar{B} + AB) + AB\bar{C}_{in} \\
 &= C(A \oplus B) + AB
 \end{aligned}$$

⇒ Multi-bit adder :- The most basic arithmetic operation is the addition of binary digits. A combinational circuit that performs the addition of more bits (multi) is called multi-bit adder.

⇒ Block Diagram :-



A circuit for adding two 4-bit binary numbers. To add multiple binary bits together, we must include a possible carry over from the lower order of magnitude, and send it as an input carry to the next higher order of magnitude bit.

- * Addition of two bits is called half adder.
- * Addition of 3 bits is called full adder.
- * By using full adders we are adding more bits. These type of adders are called multi-bit adders.

→ Multiplexers :- (Multiplexing means sharing)

→ A multiplexer is a combinational circuit that selects binary information from one to many input lines and direct to a single output line.

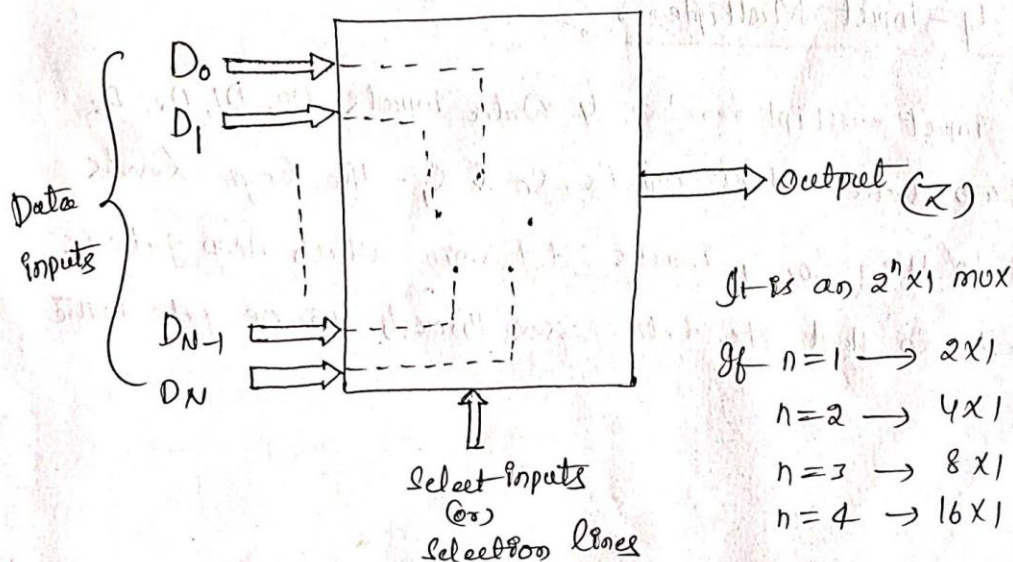
→ The routing of the desired data inputs to the output is controlled by select inputs (or) selection lines.

→ For $2^n \times 1$ multiplexer 2^n input lines, '1' output line, 'n' selection lines.

* Multiplexer is a universal logic circuit.

* It is a parallel to serial converter.

→ As it is selecting one of the input data as a output. It is also called as 'Data selector'

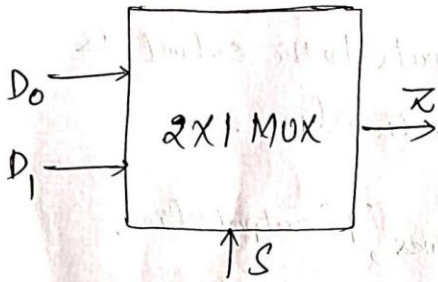


⇒ Basic 2-input multiplexer :-

A 2-input multiplexer has two data inputs D_0 & D_1 , one selection line S and one output Z

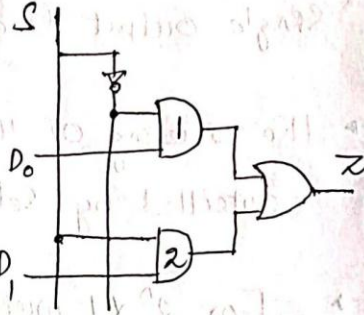
Logic diagram

Block diagram



Truth Table

S	Z
0	D_0
1	D_1



$$Z = \bar{S}D_0 + SD_1$$

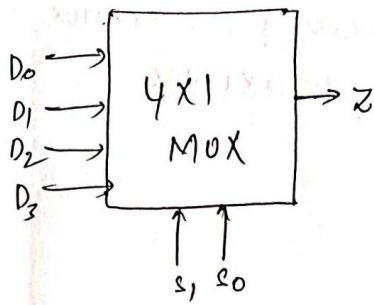
→ When $S=0$, AND gate '1' is enabled & AND gate '2' is disabled
so $Z = D_0$.

→ When $S=1$, AND gate '2' is enabled & AND gate '1' is disabled
so $Z = D_1$.

⇒ Basic 4-input Multiplexer :-

A 4-input multiplexer has 4 data inputs D_0, D_1, D_2, D_3 and two data select inputs S_0 & S_1 . The logic levels applied to the S_0, S_1 inputs determine which AND gate is enabled, so that its data passes through the OR gate to the o/p.

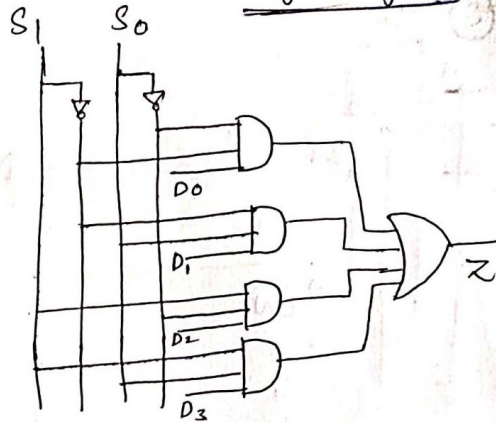
Block diagram



Truth table

S_1	S_0	Z
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Logic diagram

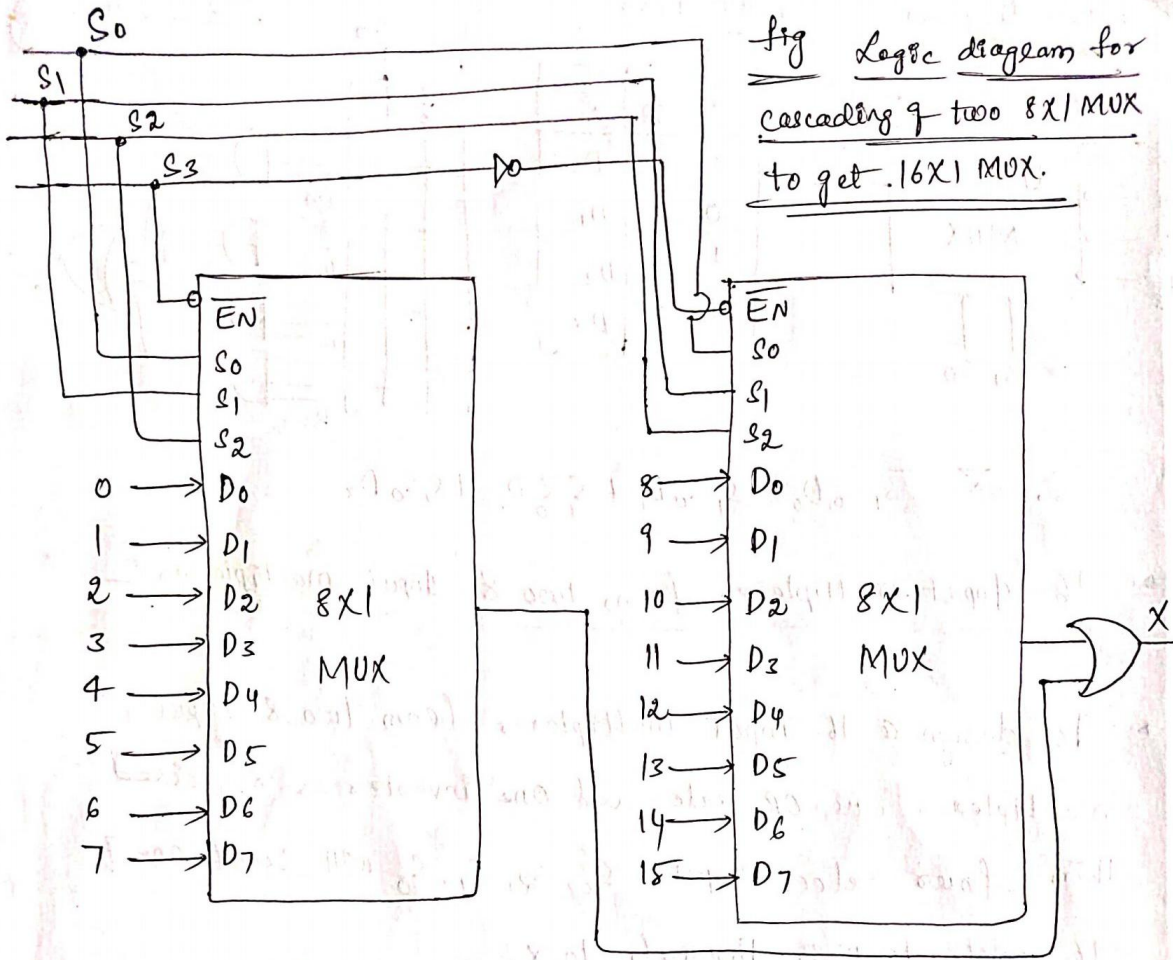


$$\therefore Z = \bar{S}_1 \bar{S}_0 D_0 + \bar{S}_1 S_0 D_1 + S_1 \bar{S}_0 D_2 + S_1 S_0 D_3$$

→ 16-Input multiplexer from two 8-Input multiplexers

→ To design a 16-Input multiplexer from two 8-Input multiplexers one OR gate and one inverter is required. Then four select inputs S_3, S_2, S_1, S_0 will select one of 16 inputs to pass through to X.

→ The S_3 input determines which multiplexer is enabled. When $S_2 = 0$, the left multiplexer is enabled and S_2, S_1 and S_0 inputs determine which of its data input will appear at its output and pass through the OR gate to X. When $S_2 = 1$ the right multiplexer is enabled and S_2, S_1 & S_0 inputs select one of its data inputs for passage to output X.



⇒ Design of a 16:1 MUX using 4:1 MUX :-

To design a 16:1 MUX using 4:1 MUX, five 4:1 MUX is needed. Four inputs are applied successively and 4 select input are required, select input C & D are applied to S₁ & S₀ terminals of the four multiplexers. The outputs of these MUXes are connected as data inputs to the 5th 4x1 MUX & select lines A & B are applied to S₁ & S₀.

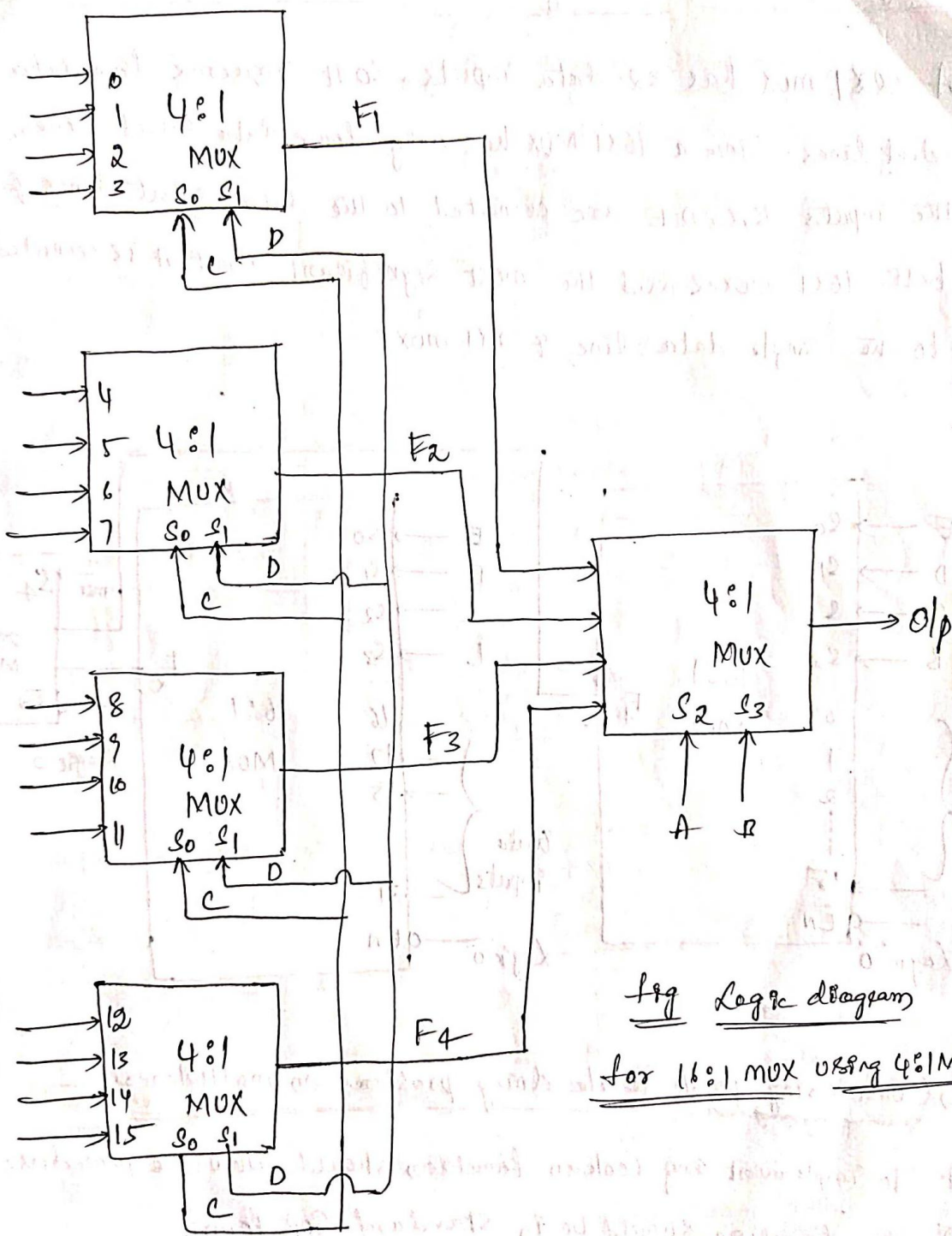
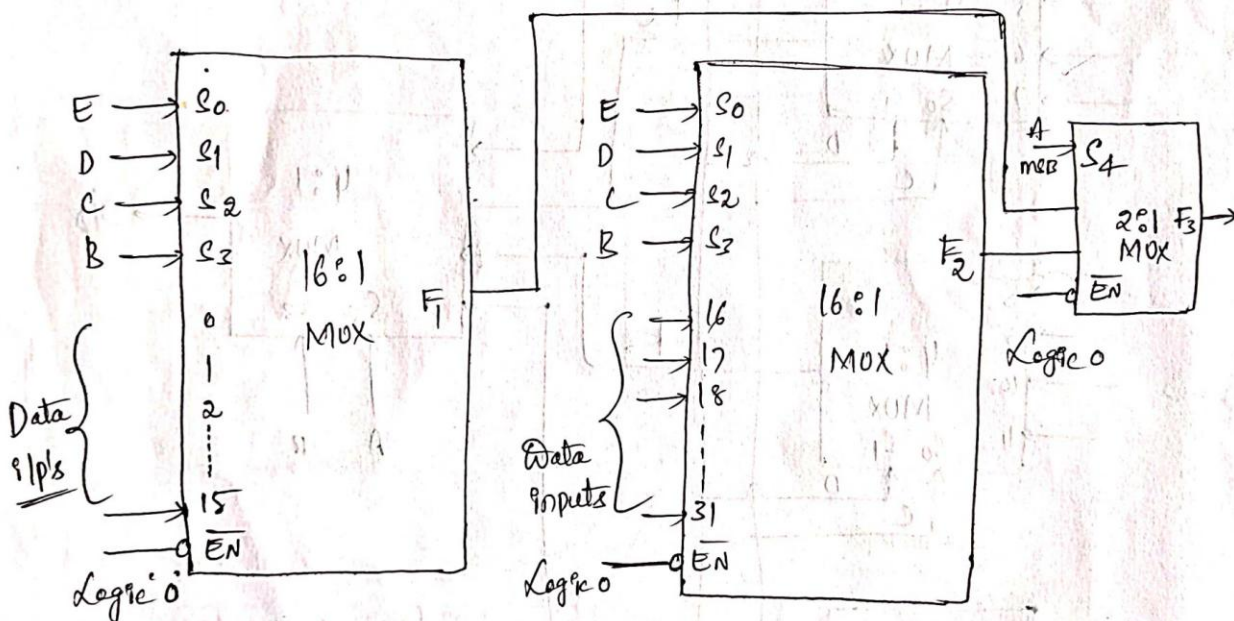


fig Logic diagram
for 16:1 MUX using 4:1 MUX.

⇒ Design of 32x1 MUX using two 16x1 MUX & one 2x1 MUX :-

A 32x1 MUX has 32 data inputs. So it requires five data select lines. Since a 16x1 MUX has only four data select lines, the inputs B, C, D, E are connected to the data select lines of both 16x1 MUXes and the most significant input A is connected to the single data line of 2x1 MUX.



⇒ Considering points while doing problems on multiplexers :-

- * To implement any boolean function should follow the procedure i.e.
- ① The function should be in standard SOP form
- ② Based on number of variables 'n', select multiplexers having (n-1) no. of selection lines.
- ③ Take any two variables as a selection lines.

Q0 $F = \sum m(1, 2, 6, 7)$

Truth Table

s_0 x	s_1 y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

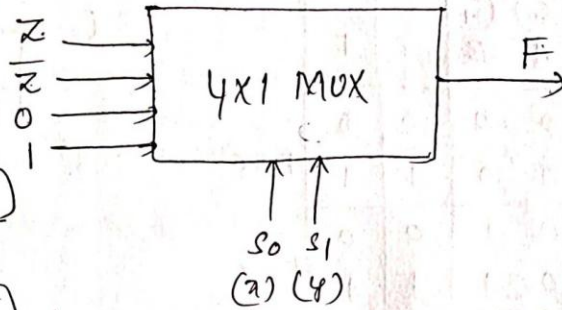
$F=z$

$F=\bar{z}$

$F=0$

$F=1$

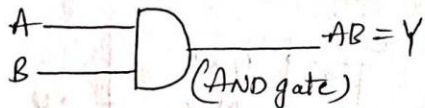
Block Diagram



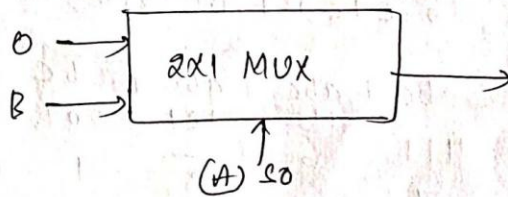
Implemented truth table

s_0	s_1	F
0	0	z
0	1	\bar{z}
1	0	0
1	1	1

Q2 Implementation of AND gate using MUX



Block Diagram



Truth Table

s_0 (A)	B	$Y=AB$
0	0	0
0	1	0
1	0	0
1	1	1

$y=0$

$y=B$

Implementation truth table

s_0	y
0	0
1	B

Q3) Using 4x1 MUX, implement the logic function $F(A, B, C)$

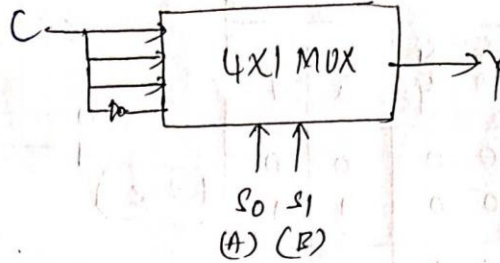
$$= \sum m(1, 3, 5, 6)$$

Sol

(s_0) A	(s_1) B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$F=C$
 $F=C$
 $F=C$
 $F=\bar{C}$

Block Diagram



Implementation table

s_0	s_1	Y
0	0	C
0	1	C
1	0	C
1	1	\bar{C}

Q4) Implement the function $F(a, b, c) = ab + \bar{b}c$ using 4x1 Mux. Convert to its canonical form.

Sol

$$ab(c + \bar{c}) + \bar{b}c(a + \bar{a})$$

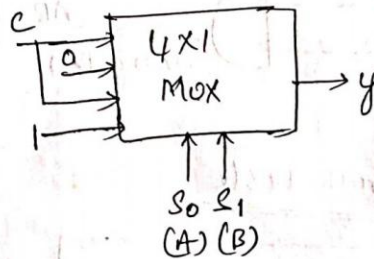
$$abc + ab\bar{c} + a\bar{b}c + \bar{a}\bar{b}c$$

111	110	101	001
7	6	5	1

$$F(a, b, c) = \sum m(1, 5, 6, 7)$$

Implementation Table :-

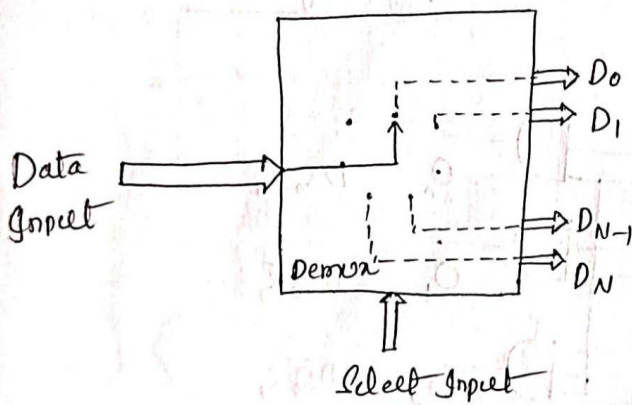
Block Diagram



s_0	s_1	Y
0	0	C
0	1	0
1	0	C
1	1	1

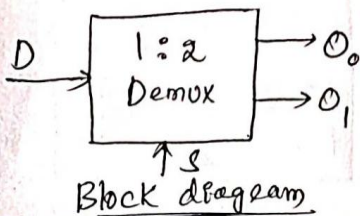
⇒ Demultiplexers :- (Data Distributor)

A Demultiplexer performs the reverse operation of multiplexer. It takes a single input and distributes it over several output. So, Demultiplexer can be called as "data distributor". Since it transmits same data to different destinations, a demultiplexer is 1 to N device.



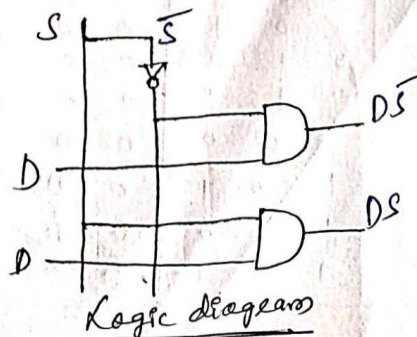
⇒ (1x2) 1 to 2 Demultiplexers :-

The input data line goes to all of the AND gates. The select line enable only one gate at a time, and the data appearing on the input will pass through the selected gate to the associated output line.



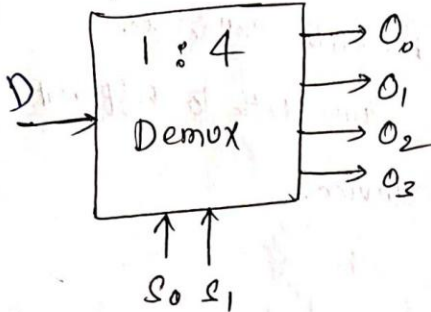
S	0
0	O ₀
1	O ₁

Truth Table



⇒ 1 line to 4 line Demultiplexer :-

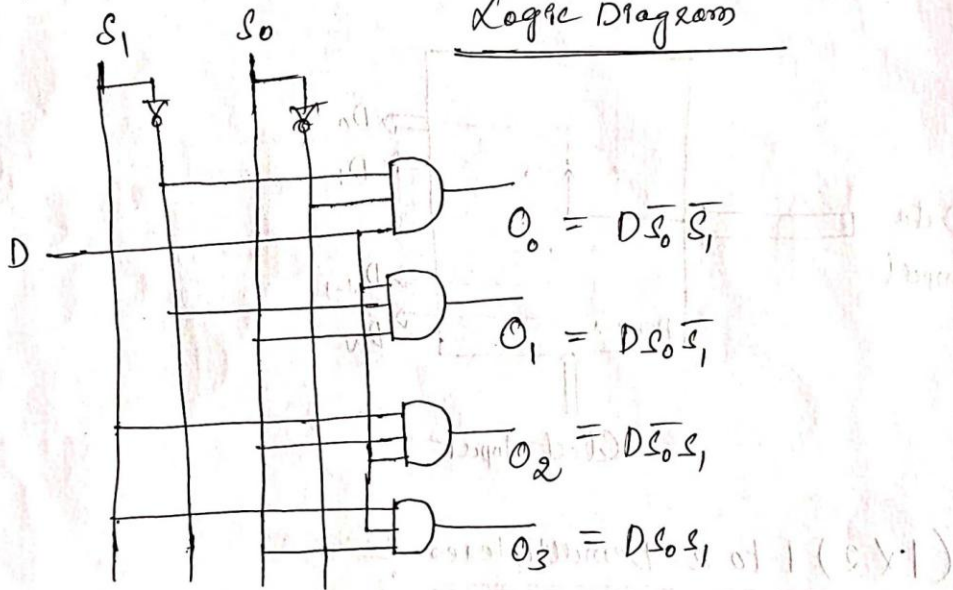
Block diagram



Truth table

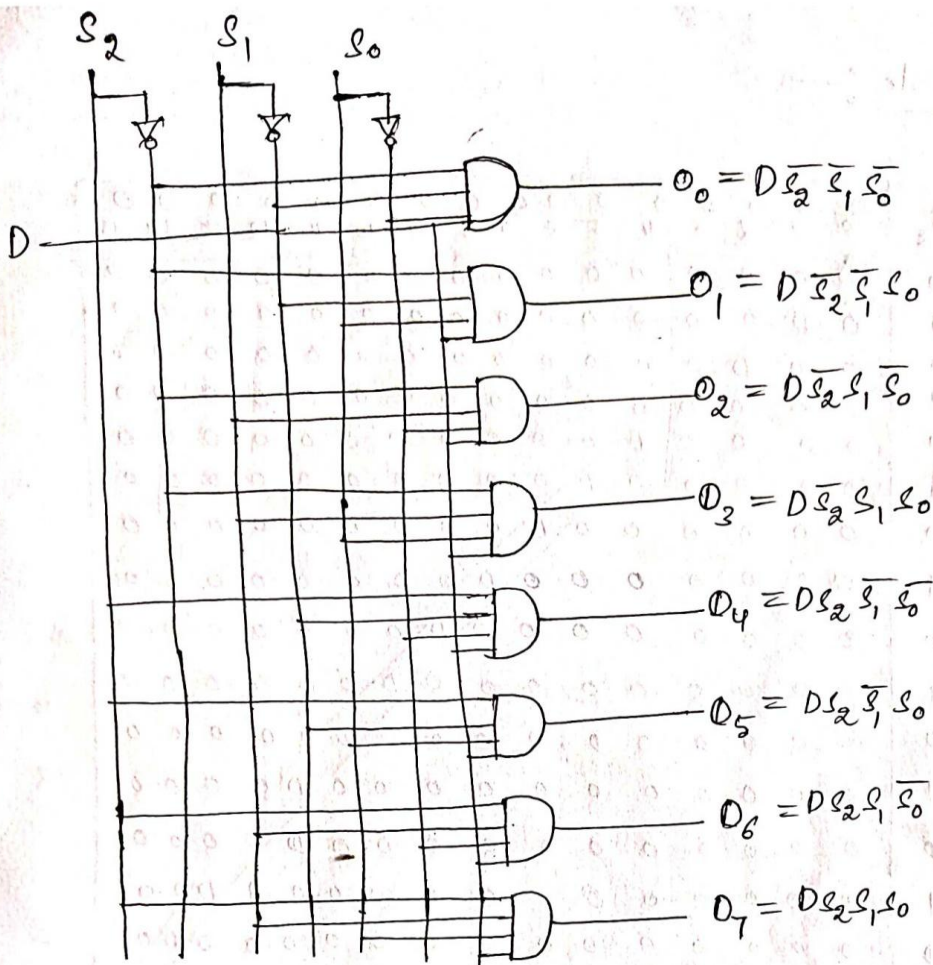
S_1	S_0	O_3	O_2	O_1	O_0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

Logic Diagram



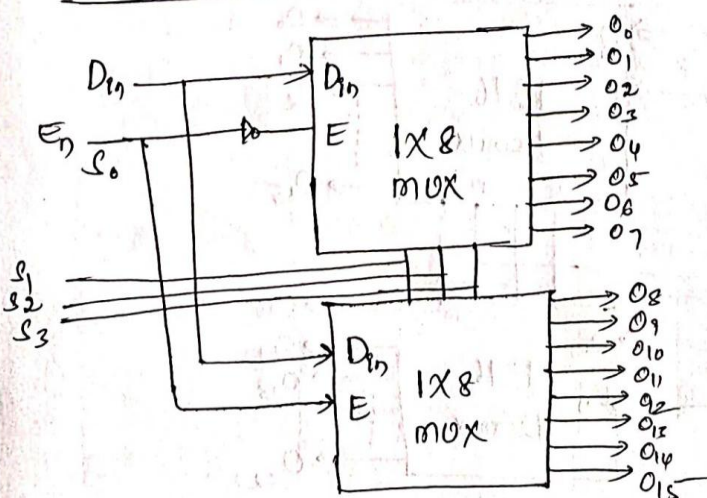
⇒ 1 line to 8 line Demultiplexer :-

S_2	S_1	S_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	D	D
0	0	1	0	0	0	0	0	D	D	0
0	1	0	0	0	0	0	D	D	0	0
0	1	1	0	0	0	D	D	0	0	0
1	0	0	0	0	D	D	0	0	0	0
1	0	1	0	0	D	0	0	0	0	0
1	1	0	0	D	0	0	0	0	0	0
1	1	1	D	0	0	0	0	0	0	0



⇒ Design and Implementation of 1x16 Demultiplexer using

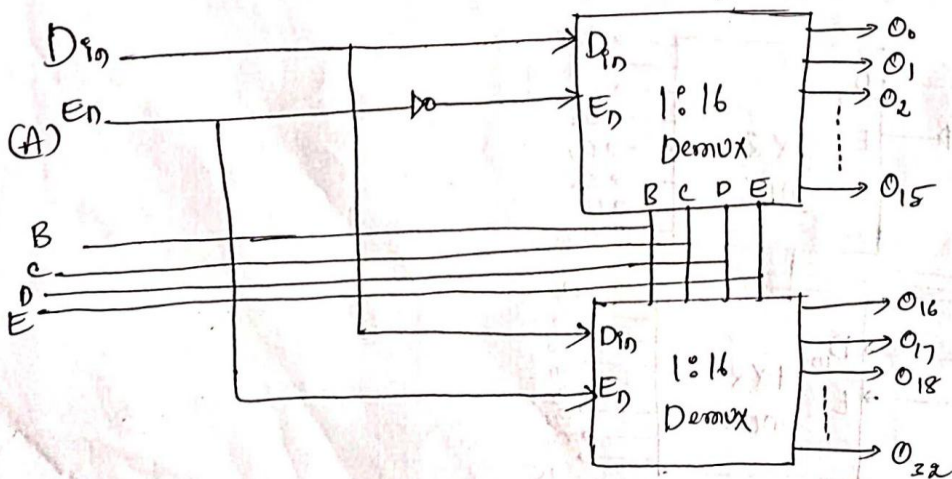
1x8 Demultiplexer :-



⇒ Truth Table :-

s_0	s_1	s_2	s_3	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

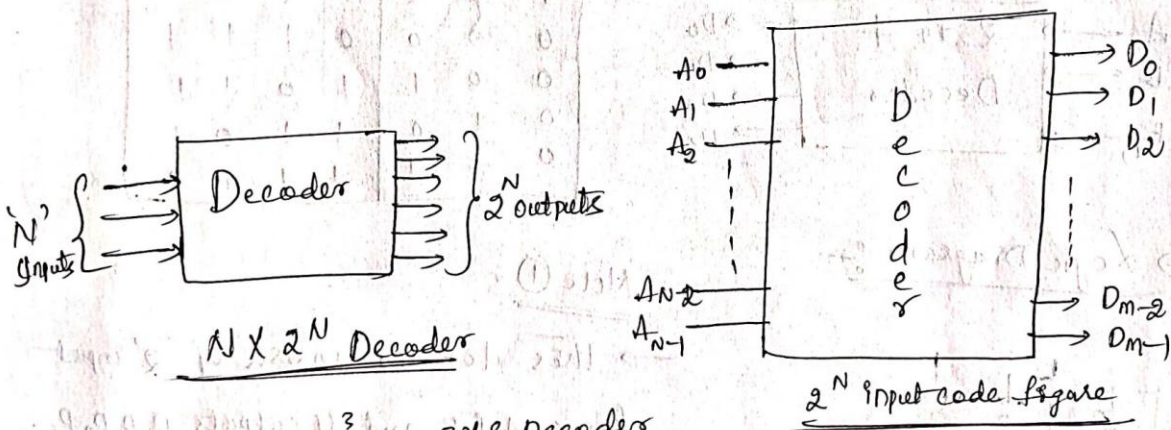
⇒ Design of 1:32 demux using two 1:16 Demux :-



⇒ Decoder:— A Decoder is a logic device that converts an N -bit binary input code into 2^N output lines such that only one output line is activated for each one of the possible combinations of inputs.

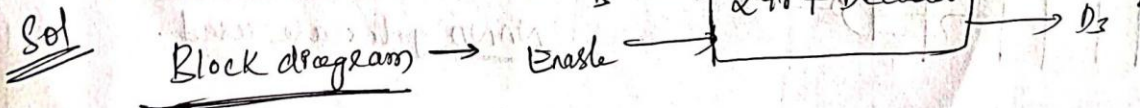
→ Each of N inputs, there are 2^N possible input combinations (or) codes. For each of these input combinations only one of 2^N output lines will be active high, all other outputs will remain inactive.

→ Some decoders are designed to produce active low op, while all the other outputs remains high.



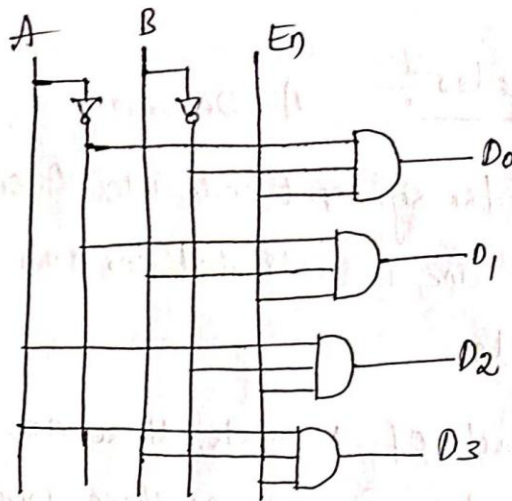
* $n=3 \Rightarrow 3 \times 2^3 \Rightarrow 3 \times 8$ Decoders
 * $n=2 \Rightarrow 2 \times 2^2 \Rightarrow 2 \times 4$ Decoders

Q1 Design and Implementation of 2 to 4 Decoders with active high output (or) using AND gates.



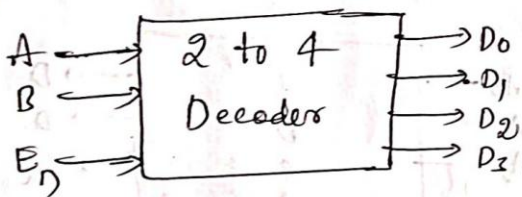
Truth Table

E_n	A	B	D_0	D_1	D_2	D_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1



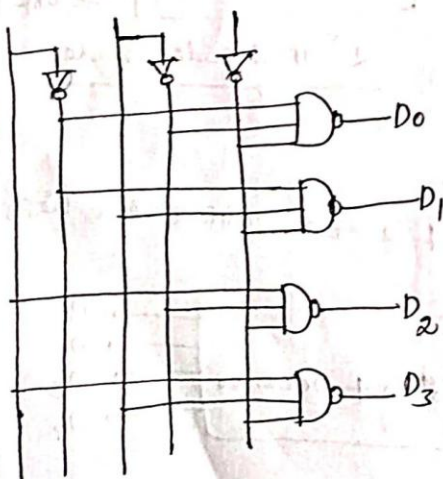
Q2 Design & Implementation of 2 to 4 Decoders with active low output (or) using NAND gates.

Block Diagram



E_n	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Logic Diagram :-



Note ①

→ This decoder consists of '2' input lines A & B and 4 outputs D_0, D_1, D_2, D_3 .

As it uses all AND gates the outputs are Active High.

Note ②

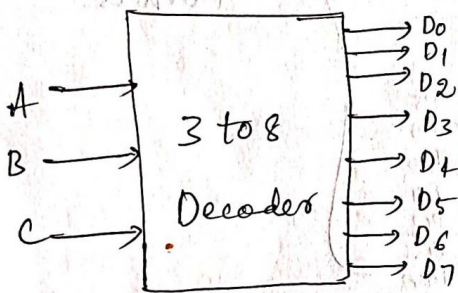
For active Low outputs NAND gates are used.

⇒ 3X 8 Decoder :-

A 3 to 8 Decoder has 3 inputs and 8

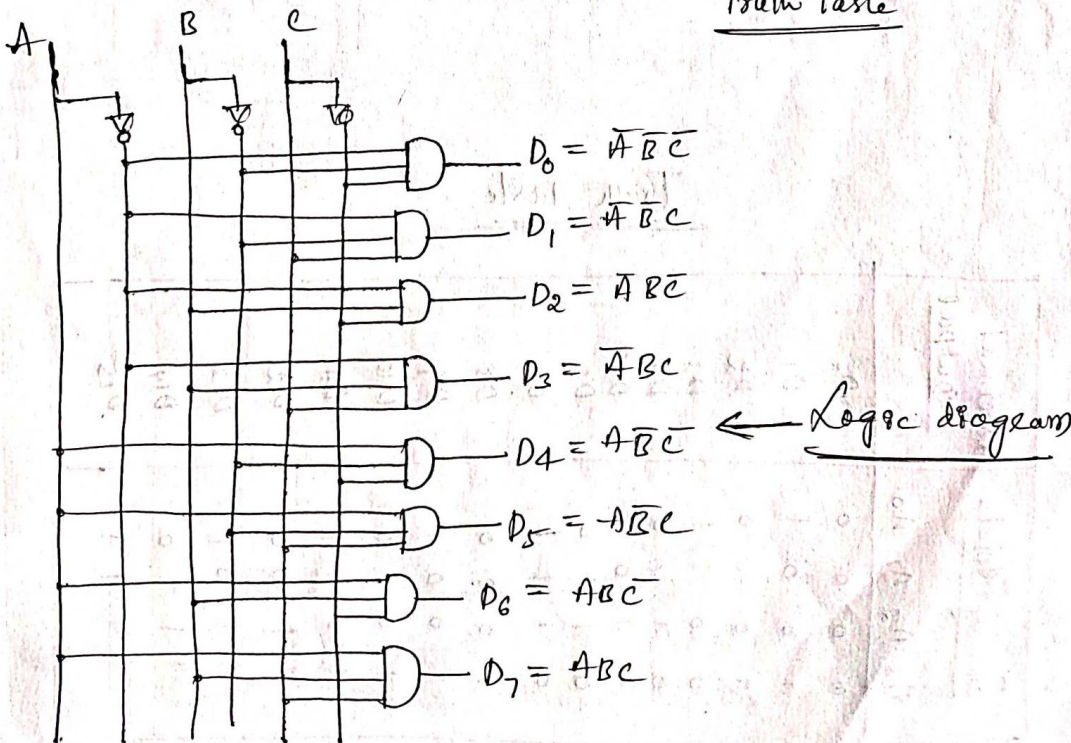
Outputs. It uses all AND gates, therefore the outputs are active high. For active low outputs, NAND gates are used. It can be called a 3 line to 8 line decoder because it has three input lines and eight output lines. It can also be called as a binary to Octal Decoder.

Block Diagram



Inputs			Outputs							
A	B	C	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Truth table



Sequential Circuits - I

- In sequential logic circuits, the output is a function of the present inputs as well as the past inputs and outputs.
- It consists of combinational circuits and memory elements. The past values are provided by feedback from the output back to the input.

Examples of sequential circuits are

- * Counters
- * Shift registers
- * Serial adders
- * Sequence generators

⇒ Block diagram :-

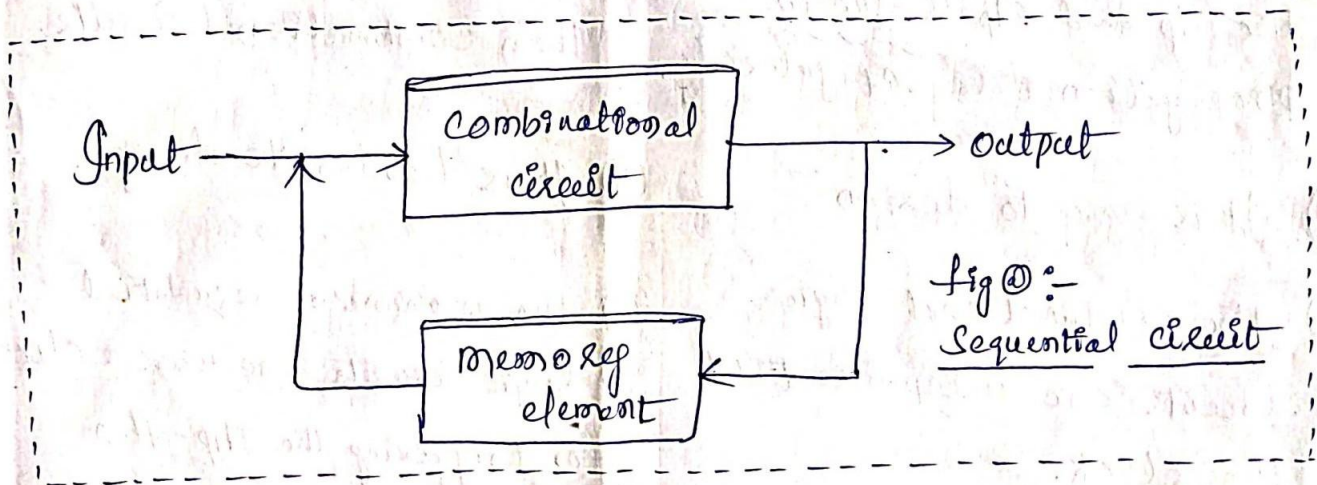


Fig 0 :-
Sequential circuit

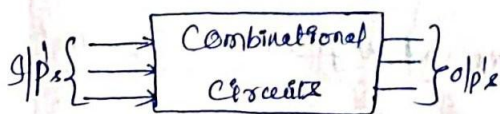
- Sequential circuit includes memory element to store the past data. The information stored in the memory element at any given time defines the present state of the sequential circuits.

Combinational circuit

- ① In combinational circuits, the output variables at any instant of time are dependent only on the present i/p variables.
- ② Memory unit is not required in combinational circuits.
- ③ combinational circuits are faster because the delay b/w the i/p and o/p is due to propagation delay of gates only.
- ④ It is easy to design
- ⑤ the combinational logic circuits are independent of the clock.
- ⑥ the combinational digital circuit don't require any feedback.

⑦ Its behaviour is described by the set of output functions.

⑧ Block diagram :-

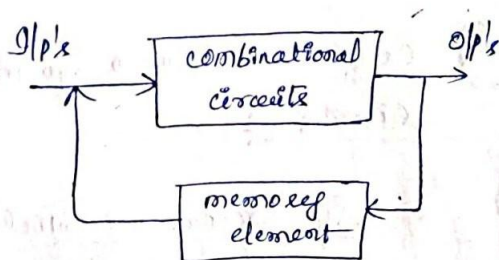


Sequential circuit

- ① In sequential circuits, the output variables at any instant of time are dependent not only on the present i/p variables, but also on the present state, i.e. on past values of the circuit.
- ② Memory unit is required to store the past values of the i/p variables in sequential circuits.
- ③ Sequential circuits are slower than combinational circuits.
- ④ It is harder to design.
- ⑤ the minimum sequential logic circuits use a clock for triggering the flip-flop operation.
- ⑥ the sequential digital logic circuits utilize the feedback from outputs to inputs.

⑦ Its behaviour is described by the set of next state fun's and the set of o/p functions.

⑧ Block diagram :-



Q Comparison between Synchronous and Asynchronous Sequential Circuits :

Synchronous Sequential Circuits

- ① In synchronous circuits, memory elements are clocked Flip-Flops (FF's).
- ② In this, the change in i/p signals can affect memory elements upon activation of clock signal.
- ③ The maximum operating speed of the clock depends on time delays involved.
- ④ They are easier to design.

Asynchronous Sequential Circuits

- ① In asynchronous circuits, memory elements are either unclocked Flip-Flops or time delay elements.
- ② In this, change in i/p signals can affect memory elements at any instant of time.
- ③ Because of the absence of the clock, asynchronous circuits can operate faster than synchronous circuits.
- ④ These are more difficult to design.

⇒ Latch :- The term 'latch' is used for certain flip flops. It refers to non-clocked flip-flops.

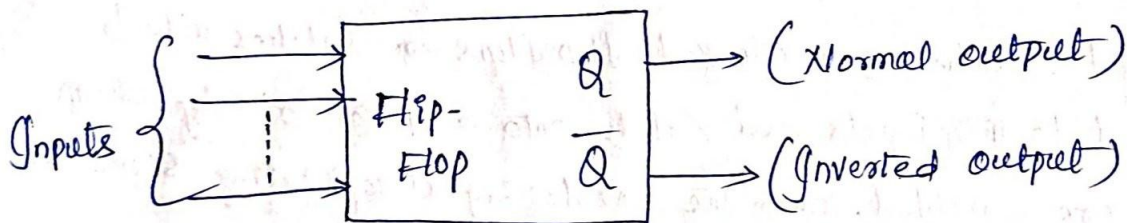
- Latch is a sequential device that checks all its inputs continuously and changes its outputs accordingly at anytime independent of clock signal.
- Gated latches (or) Clocked flip flops are latches which respond to the inputs and latch onto a '1' (or) '0' only when they are enabled, when the enable signal (or) gating signal is high.
- In the absence of Enable (or) gating signal the latch does not respond to the changes in its inputs (here the gating signal may be a clock pulse).
- Latch may be an 'Active high' input latch (or) an active low input latch.
- In Active Latch (High) constructed with NOR gates.
- In Active latch (Low) constructed with NAND gates.

⇒ Flip-Flops :- The most important memory element is the flip-flop, which is made up of an assembly of logic gates.

- There are several different gate arrangements that are used to construct flip-flops in a wide variety of ways.
- Each type of flip-flop has special features (or) characteristics necessary for particular applications.

→ Flip-Flops are the basic building blocks of sequential circuits. Actually flip-flop is an one bit memory device it can store either '1' (or) '0'.

⇒ General flip flop symbol :-



→ The flip-flop can have one (or) more inputs, the flip signals which command the flip-flop to change state are called excitations.

→ The applications of flip-flops are serve as a storage device. It stores a '1' when its output 'Q' is a '1' and it stores a '0' when its output Q is '0'. These are mainly used in Registers and counters.

Q Difference between Latches & Flip-flops :-

0

Latch

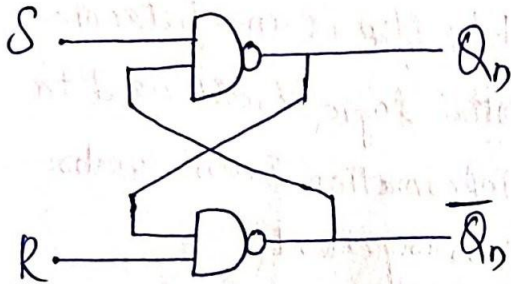
- ① A Latch is an electronic sequential logic circuit used to store information in an asynchronous arrangement.
- ② One Latch can store one bit information, but output state changes only in response to data input.
- ③ Latch is an asynchronous device and it has no clock input.
- ④ Latch holds a bit value and remains constant until new inputs force it to change.
- ⑤ Latches are level sensitive and the o/p level is high. Therefore as long as the level is logic level 1 the o/p can change if the i/p

Flip Flop

- ① A Flip flop is an electronic sequential logic circuit used to store information in a synchronous arrangement.
- ② One flip-flop can store one bit data, but output state changes with clock pulse only.
- ③ Flip-Flop has clock input and its output is synchronised with clock pulse.
- ④ Flip Flops hold a bit value and it remains constant until clock pulse is received.
- ⑤ Flip Flops are edge sensitive they can store the i/p only when there is either a rising (or) falling edge of clock.

⇒ SR latch with NAND Gates / Active-Low latch :-

Circuit diagram :-

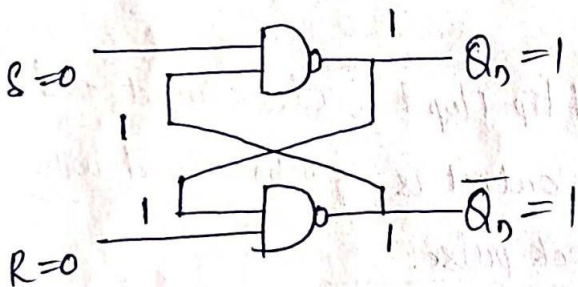


Truth Table

S	R	Q_n	\bar{Q}_n
0	0	Invalid state	
0	1	1	0 (Set)
1	0	0	1 (Reset)
1	1	No Change	

Case i) :-

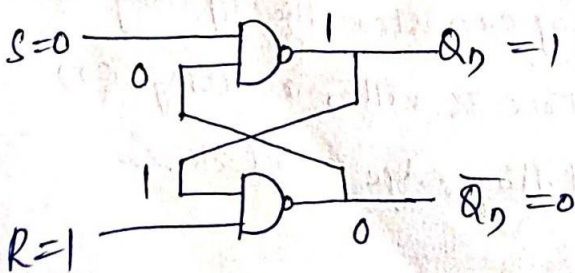
$S=0, R=0$



When $S=0, R=0$ the outputs are $Q_n=1$ & $\bar{Q}_n=1$. So, this is Invalid state / Indetermined.

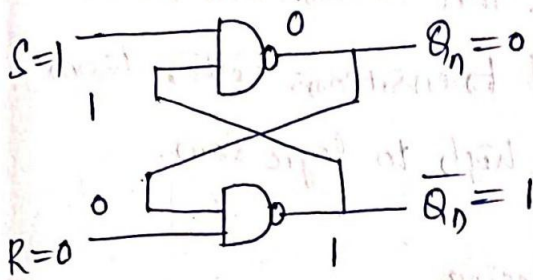
Case ii) :-

When $S=0, R=1$



∴ When $S=0, R=1$ the outputs are $Q_n=1$ & $\bar{Q}_n=0$.

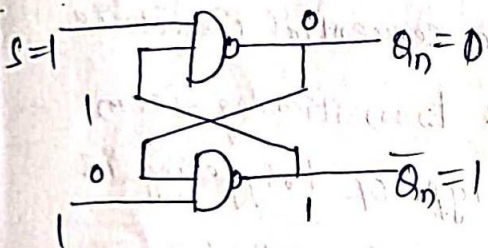
Case iii :- $S=1, R=0$



∴ When $S=1, R=0$ the outputs are $Q_n=0; \bar{Q}_n=1$

Case iv :-

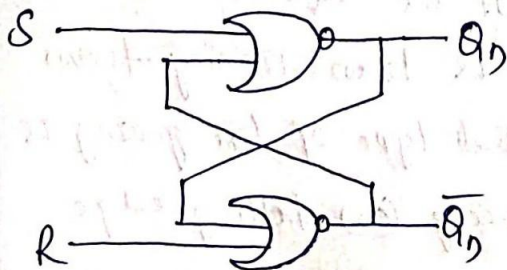
$S=1; R=1$



∴ When $S=1$ & $R=1$ the outputs are $Q_n=0$ is same as previous state. So, the o/p is no change state.

⇒ SR Latch with NOR Gates / Active high Latch :-

⇒ Circuit diagram :-



Truth Table

S	R	Q_n	\bar{Q}_n
0	0	No change	
0	1	1	0
1	0	0	1
1	1	Invalid	

Note :- The above four cases case i, ii, iii, & iv, also present in the above procedure is same here also. Once we do some operation we get above truth table.

→ Flip-Flops :-

It is a memory element, made up of an assembly of logic gates. Flip Flop also known more formally as bistable multivibrator. Flip Flop is a one bit memory element.

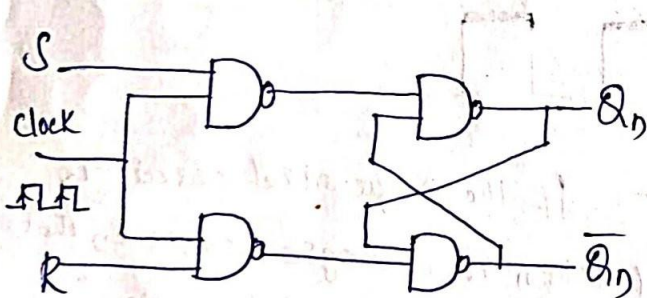
→ Flip-Flops are classified into 4 types

① SR Flip-Flop ② JK Flip-Flop

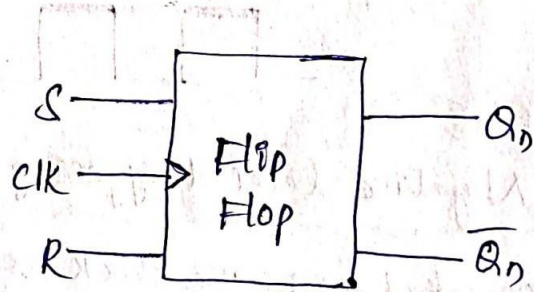
③ D Flip-Flop ④ T Flip-Flop.

→ Clocked SR Flip-Flop :-

→ Block diagram of SR Flip-Flop :-



① Circuit diagram



② Block diagram

③ Truth table

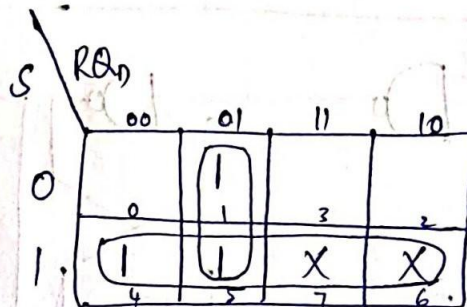
CLK	S	R	Q_n	\bar{Q}_n
1	0	0	No change	
1	0	1	0	1 (Reset)
1	1	0	1	0 (Set)
1	1	1	Invalid state	

(d) Characteristic Table :-

From truth table find characteristic table.

CLK	S	R	Q_n	Q_{n+1}
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	X

(e) Characteristic equation

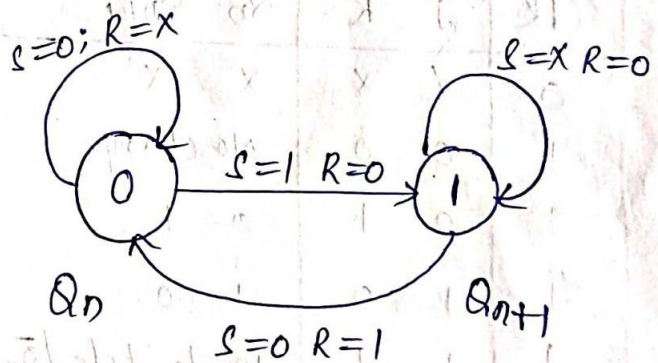


$$Q_{n+1} = S + \bar{R}Q_n$$

(f) Excitation table :-

Q_n	Q_{n+1}	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

(g) State diagram



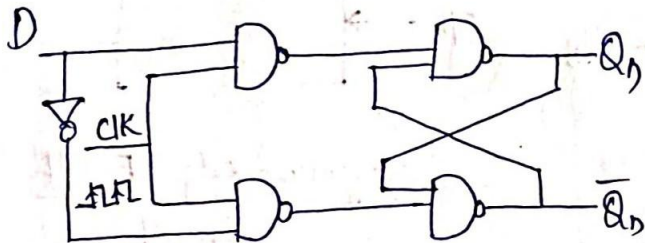
Q_n	Q_{n+1}	S	R
0	0	0	0
0	1	1	0
1	0	0	1
1	1	X	0

① $Q_n Q_{n+1} \rightarrow \begin{matrix} S & R \\ 0 & 0 \\ 0 & 1 \\ \hline 0 & X \end{matrix} \checkmark$
 ② $01 \rightarrow \begin{matrix} 1 & 0 \end{matrix} \checkmark$
 ③ $10 \rightarrow \begin{matrix} 0 & 1 \end{matrix} \checkmark$
 ④ $11 \rightarrow \begin{matrix} 0 & 0 \\ 1 & 0 \\ \hline X & 0 \end{matrix} \checkmark$

* In SR Flip Flop $S=1$ & $R=1$ then this state is indeterminate state. This drawback is avoided in JK Flip Flop.

⇒ D-Flip-Flop :- (Delay flip flop)

(a) Circuit diagram :-



(b) Block diagram



(c) Truth Table :-

CLK	D	Q_n	\bar{Q}_n
1	0	0	1
1	1	1	0

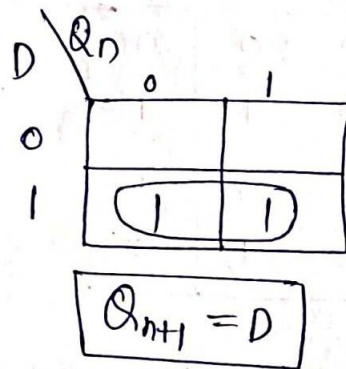
→ A Flip-Flop is also called as Delay (or) Data Flip Flop. It is used to find delay b/w the set and reset.

(d) Characteristic Table :-

It is used to find the next state

D	Q_n	Q_{n+1}
0	0	0
0	1	0
1	0	1
1	1	1

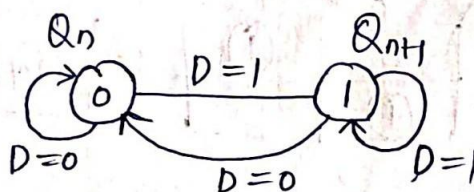
(e) Characteristic equation



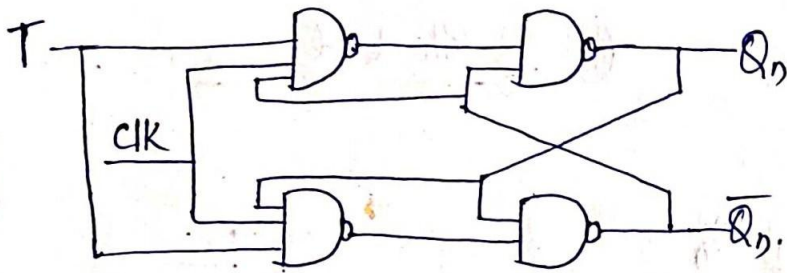
(f) Excitation Table

Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

(g) State diagram :-



⇒ T-Flip Flop :-



Ⓐ Logic diagram

Ⓑ Truth Table

CLK	T	Q_n	\bar{Q}_n
1	0	Q_n	\bar{Q}_n
1	1	Toggle state	

→ no change state

Ⓒ Characteristic Table

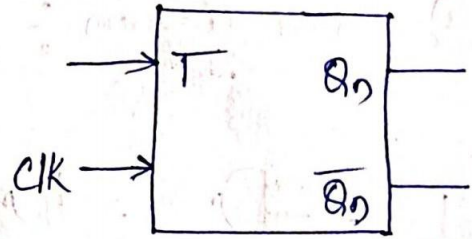
CLK	T	Q_n	Q_{n+1}
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Ⓓ Characteristic equation

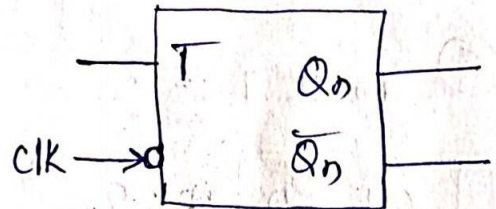
T \ Q_n	0	1
0		Ⓚ
1	Ⓚ	

$$Q_{n+1} = T\bar{Q}_n + \bar{T}Q_n$$

Ⓔ Positive edge T-F/F Symbol



Ⓕ Negative edge T-F/F Symbol



Ⓖ Excitation Table

Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

COUNTERS :-

- A digital counter is a set of flip-flop (FFs) whose state change in response to pulses applied at the input to the counter. ①
- A counter is used to count pulses.
- A counter can also be used as a frequency divider to obtain wave-forms with frequencies that are specific fractions of the clock frequency.
- They are also used to perform the timing function as in digital watches, to create delays, to produce non-sequential binary counts, to generate pulse trains, and to act as frequency counters.
- counters are classified into
 - (i) Asynchronous counters
 - (ii) Synchronous counters.
- Asynchronous counters also called as ripple counters (or) series counters.

comparison of synchronous and asynchronous counters

Asynchronous counters

1. In this type of counter FFs are connected in such a way that the output of first FF drives the clock for the second FF, the output of the second to the clock of the third and so on.
2. All the FFs are not clocked simultaneously
3. Design and implementation is very simple even for more number of states
4. Main drawback of these counters is their low speed as the clock is propagated through a number of FFs before it reaches the last FFs

synchronous counters

1. In this type of counter ^{there} is no connection between the output of first FF and clock input of next FF and so on.
2. All the FFs are clocked simultaneously.
3. Design and implementation becomes tedious and complex as the number of states increases
4. Since clock is applied to all the FFs simultaneously the total propagation delay is equal to the propagation delay of only one FF. Hence they are faster.

Asynchronous Counters :-

②

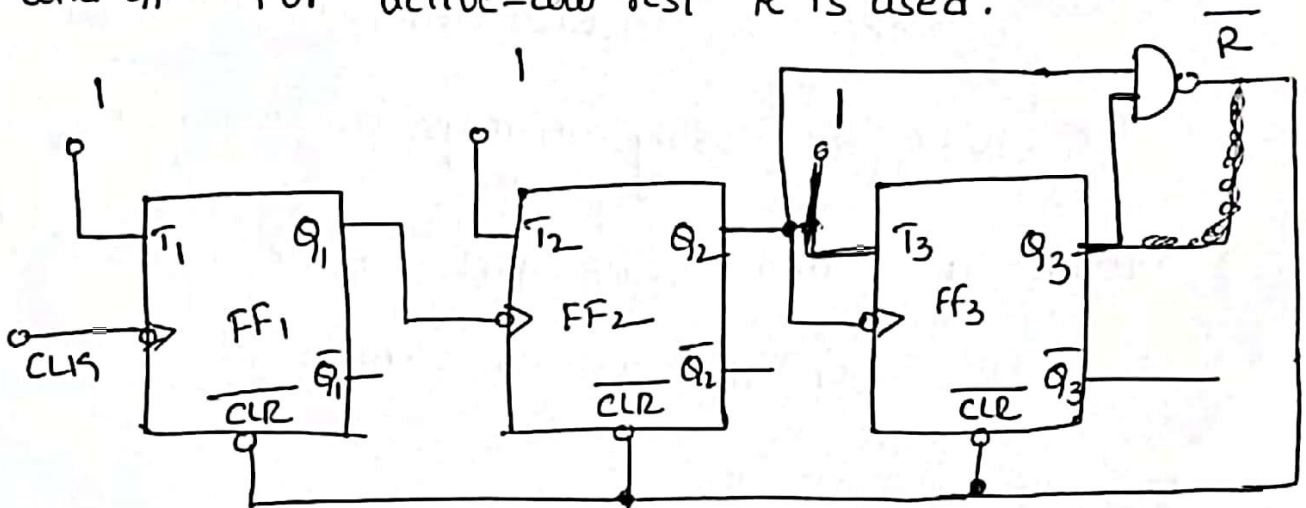
- To design an asynchronous counter, first write the counting sequence, then tabulate the values of reset signal R for various states of the counter and obtain the minimal expression for R or \bar{R} using k-map or any other method.
- Provide a feedback such that R or \bar{R} resets all the FFs after the desired count.

Design of a mod-6 asynchronous counter using TFFs

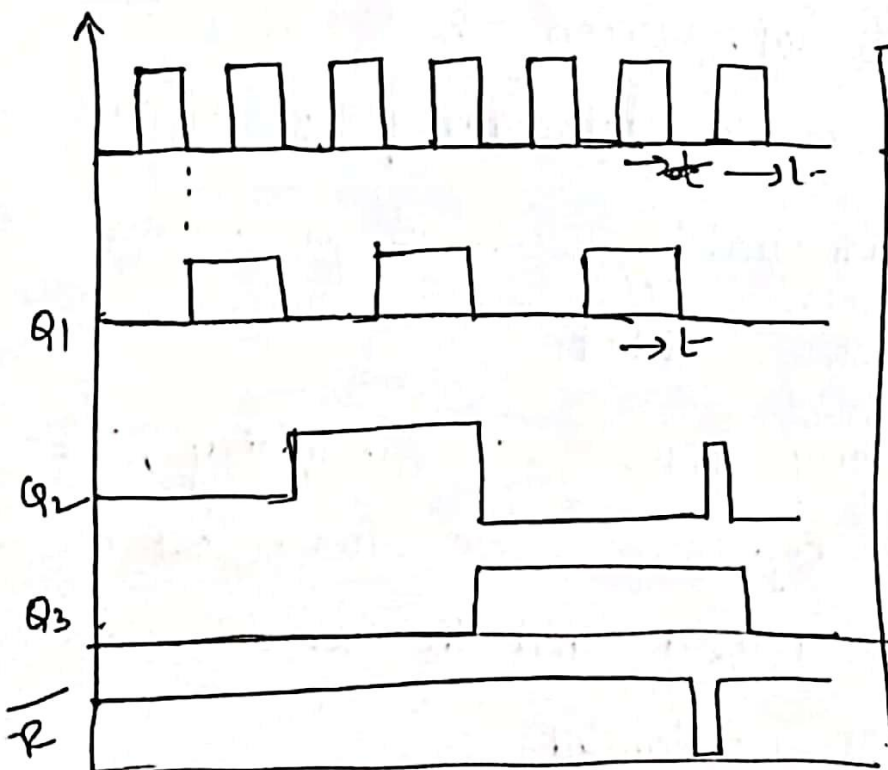
A mod-6 counter has six stable states 000, 001, 010, 011, 100, and 101. When the sixth clock pulse is applied, the counter temporarily goes to 110 state, but immediately resets to 000 because of the feedback provided.

- Here we are using 3FFs for designing, three FFs can have eight possible states, out of which only six utilized and the remaining two states 110 and 111 are invalid.

→ For the design, write a truth table with the present state outputs Q_3, Q_2 and Q_1 , as the variables, and reset R as the output and obtain an Expression for R in terms of Q_3, Q_2 and Q_1 . For active-low reset \bar{R} is used.



Logic diagram.



After Pulses	state			R
	Q_3	Q_2	Q_1	
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
	↓	↓	↓	
	0	0	0	0
	0	0	1	0

Truth table.

→ Design of mod-10 asynchronous counter using 4 T FFs:-

→ A mod-10 counter is a decade counter. It is also called a BCD counter or a divide-by-10 counter. It requires 4 FFs.

→ This counter has ten stable states 0000 through 1001, i.e. it counts from 0 to 9.

→ The initial state is 0000 and after nine clock pulses it goes to 1001. When the tenth clock pulse is applied, the counter goes to state 1010 temporarily, but because of the feedback provided, it resets to initial state 0000.

→ So, there will be a glitch in the waveform of Q_2 . The state 1010 is a temporary state for which the reset signal $R=1$, $R=0$ for 0000 to 1001, and $R=X$ (don't care) for 1011 to 1111.

Shift Register counters :-

→ one of the applications of shift registers is that they can be arranged to form several types of counters.

→ shift register counters are obtained from serial-in, serial-out shift registers by providing feedback from the output of the last FF to the input of the first FF.

These devices are called counters because they exhibit a specified sequence of states.

→ The mostly used shift register counter is the ring counter, as well as the twisted ring counter.

Ring counter :-

→ This is the simplest shift register counter.

the basic ring counter using DFFs is shown below fig (a). The realization of this counter using J-K FFs is shown in fig (b).

→ The FFs are arranged as in a normal shift register, i.e the Q output of each stage is connected to the D input of the next stage, but the Q output of the last FF is connected back to the D input of the first FF such that the array of FFs is arranged in a ring and therefore, the name ring counter.

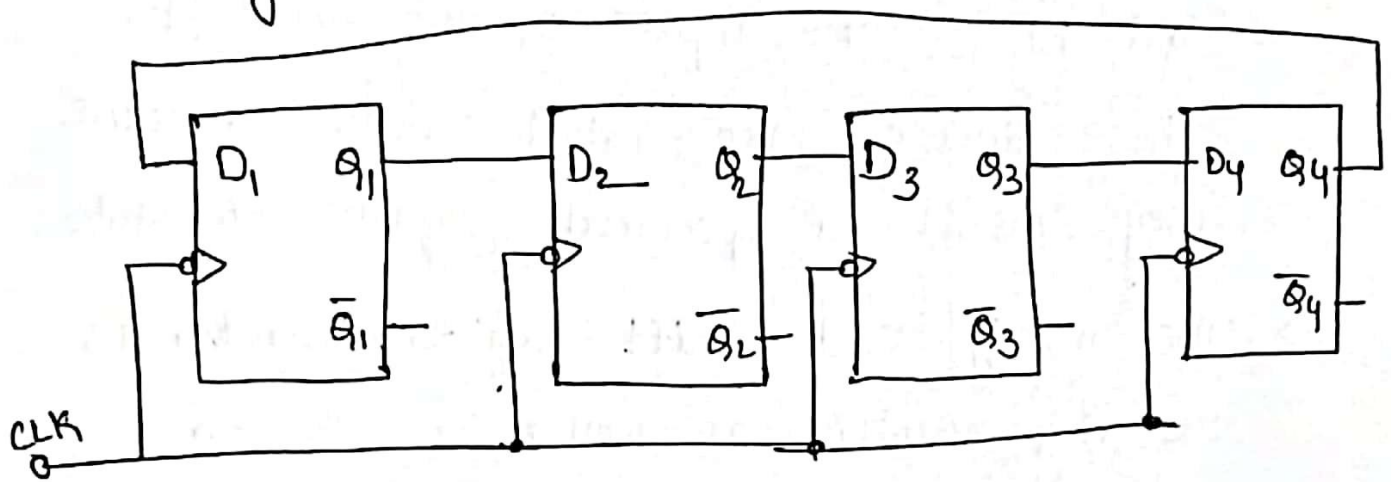


fig (a) :- logic diagram of 4-bit ring counter using D flip-flop

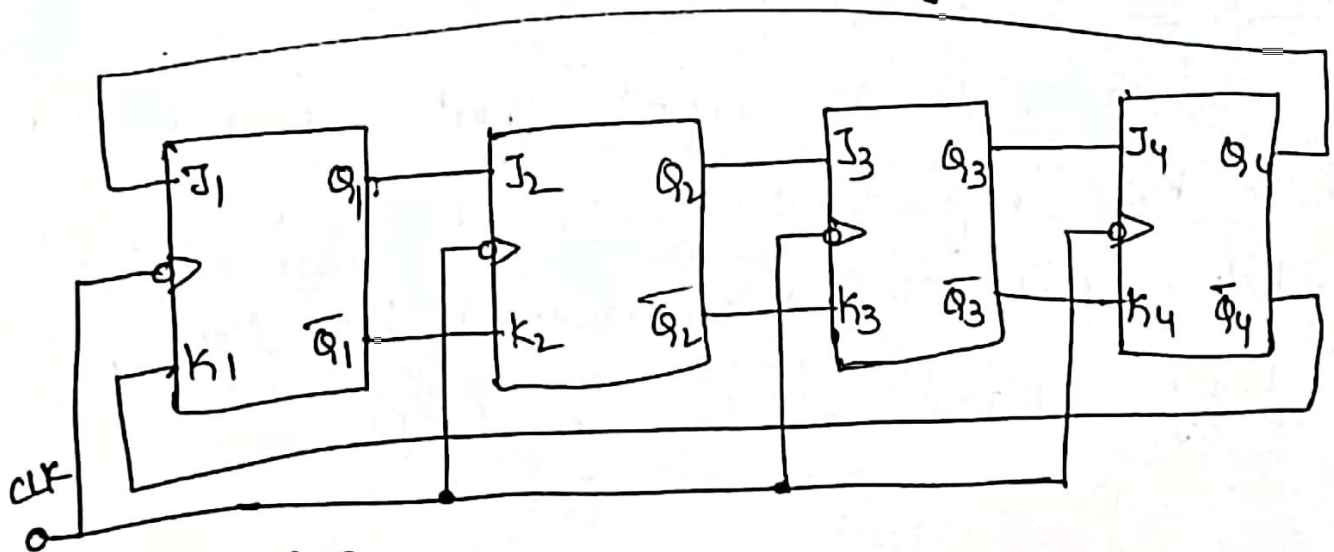


fig (b) : using JK FFs.

- In most instances, only a single 1 is in the register and is made to circulate around the registers as long as clock pulses are applied.
- Initially, the first FF is preset to a 1. So, the initial state is 1000, i.e. $Q_1=1, Q_2=0, Q_3=0$ and $Q_4=0$. After each clock pulse, the contents of the register are shifted to the right by one bit and Q_4 is shifted back to Q_1 .
- The sequence repeats after four clock pulses. The number of distinct states in the ring counter, i.e. the mod of the ring counter is equal to the number of FFs used in the counter.
- An n-bit ring counter can count only n bits, whereas n-bit ripple counter can count 2^n states bits.
- It is entirely a synchronous operation and requires no gates external to FFs, it has the further advantage of being very fast.

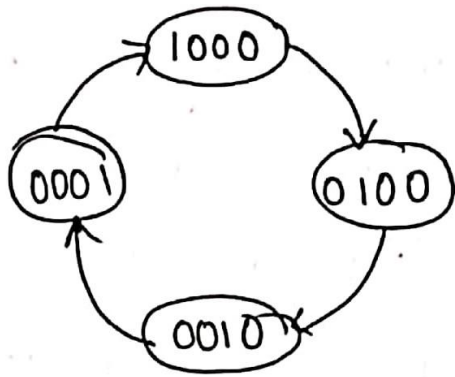
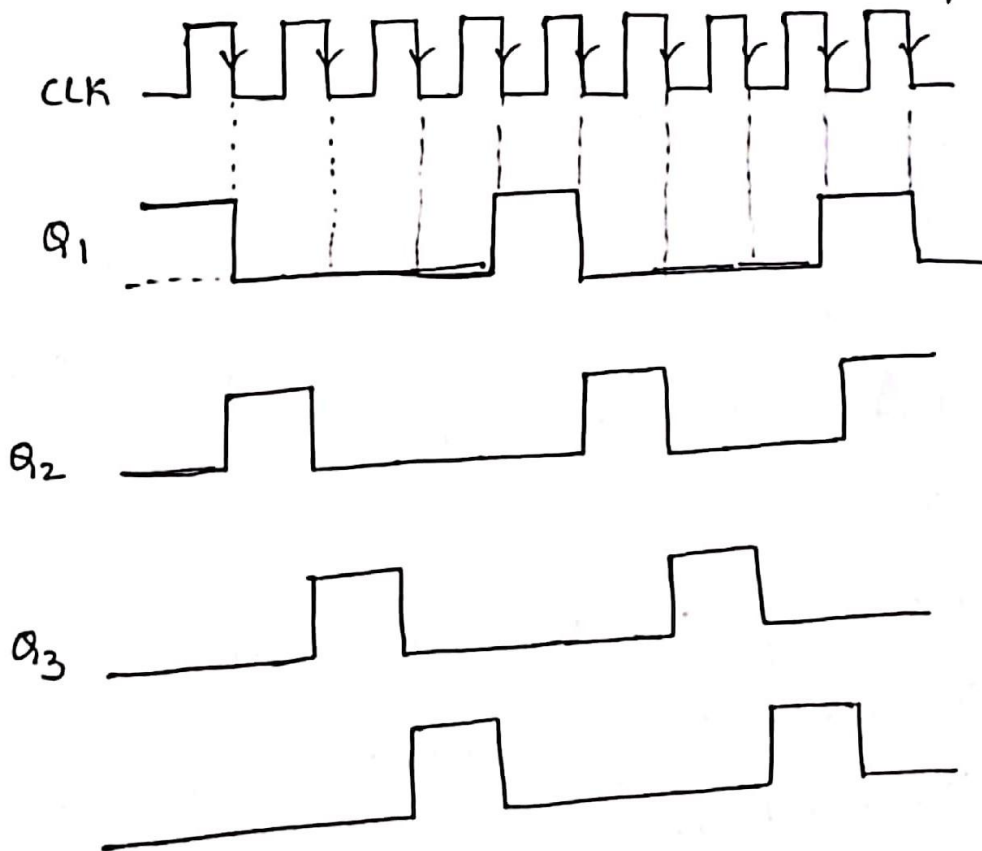


fig :- State diagram.

Q ₁	Q ₂	Q ₃	Q ₄	After clock pulse
1	0	0	0	0
0	1	0	0	1
0	0	1	0	2
0	0	0	1	3
1	0	0	0	4
0	1	0	0	5
0	0	1	0	6
0	0	0	1	7

sequence table



Timing diagram of a 4-bit ring counter

Computer Arithmetic:

Introduction:

- Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems.
- These instructions perform arithmetic calculations and are responsible for the bulk of activity involved in processing data in a computer.

The four basic arithmetic operations are **addition, subtraction, multiplication and division**. From these four basic operations, it is possible to formulate other arithmetic functions and solve scientific problems by means of numerical analysis methods.

- An arithmetic processor is the part of a processor unit that executes arithmetic operations. The data type assumed to reside in **processor registers** during the execution of an arithmetic instruction is specified in the definition of the instruction. An arithmetic instruction may specify binary or decimal data, and in each case the data may be in fixed-point or floating-point form.
- We must be thoroughly familiar with the sequence of steps to be followed in order to carry out the operation and achieve a correct result. The solution to any problem that is stated by a finite number of well-defined procedural steps is called an **algorithm**.
- Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps. A convenient method for presenting algorithms is a **flowchart**.

Addition and Subtraction:

- As we have discussed, there are three ways of representing negative fixed-point binary numbers: signed-magnitude, signed-1's complement, or signed-2's complement. Most computers use the signed-2's complement representation when performing arithmetic operations with integers.

i. Addition and Subtraction with Signed-Magnitude Data:

When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table shown below:

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Algorithm: (Addition with Signed-Magnitude Data)

- When the signs of A and B are identical, add the two magnitudes and attach the sign of A to the result.
- When the signs of A and B are different, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if $A > B$ or the complement of the sign of A if $A < B$.
- If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Algorithm: (Subtraction with Signed-Magnitude Data)

- When the signs of A and B are different, add the two magnitudes and attach the sign of A to the result.

- ii. When the signs of A and B are identical, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if $A > B$ or the complement of the sign of A if $A < B$.
- iii. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

The two algorithms are similar except for the sign comparison. The procedure to be followed for identical signs in the addition algorithm is the same as for different signs in the subtraction algorithm, and vice versa.

Hardware Implementation:

To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers.

- i. Let A and B be two registers that hold the magnitudes of the numbers, and A_s and B_s be two flip-flops that hold the corresponding signs.
- ii. The result of the operation may be transferred to a third register; however, a saving is achieved if the result is transferred into A and A_s . Thus A and A_s together form an accumulator register.

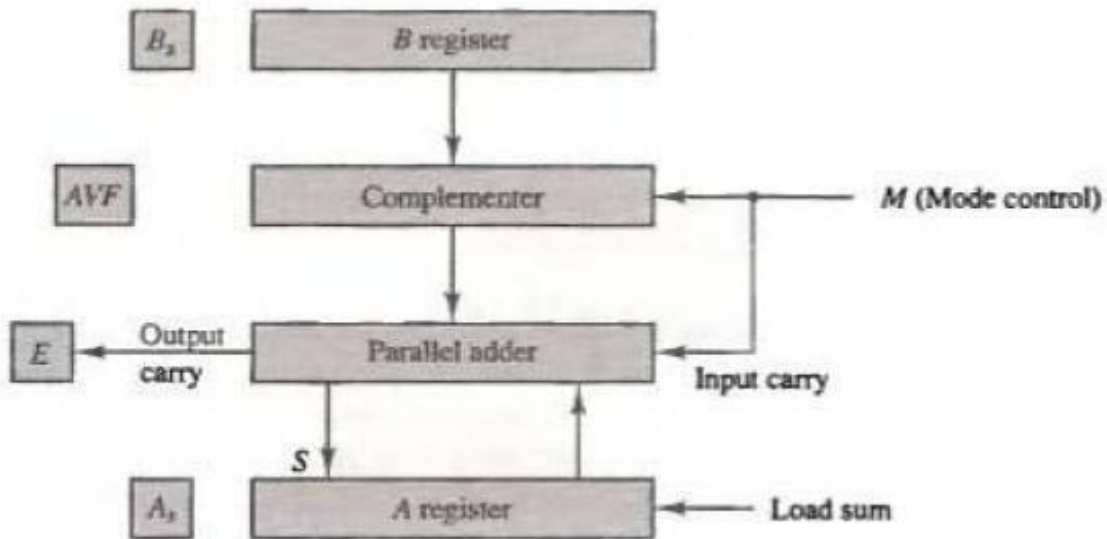
Consider now the hardware implementation of the algorithms above.

- First, a parallel-adder is needed to perform the microoperation $A + B$.
- Second, a comparator circuit is needed to establish if $A > B$, $A = B$, or $A < B$.
- Third, two parallel-subtractor circuits are needed to perform the microoperations $A - B$ and $B - A$. The sign relationship can be determined from an exclusive-OR gate with A_s and B_s as inputs.

The below figure shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops A_s and B_s .

- Subtraction is done by adding A to the 2's complement of B . The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of the two numbers.
- The add-overflow flip-flop AVF holds the overflow bit when A and B are added.

Figure (i): Hardware for addition and subtraction with Signed-Magnitude Data



The complementer provides an output of B or the complement of B depending on the state of the mode control M.

- ❖ When $M = 0$, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum $A + B$.
- ❖ When $M = 1$, the 1's complement of B is applied to the adder, the input carry is 1, and output

$S = A + \bar{B} + 1$. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction $A - B$.

Hardware Algorithm

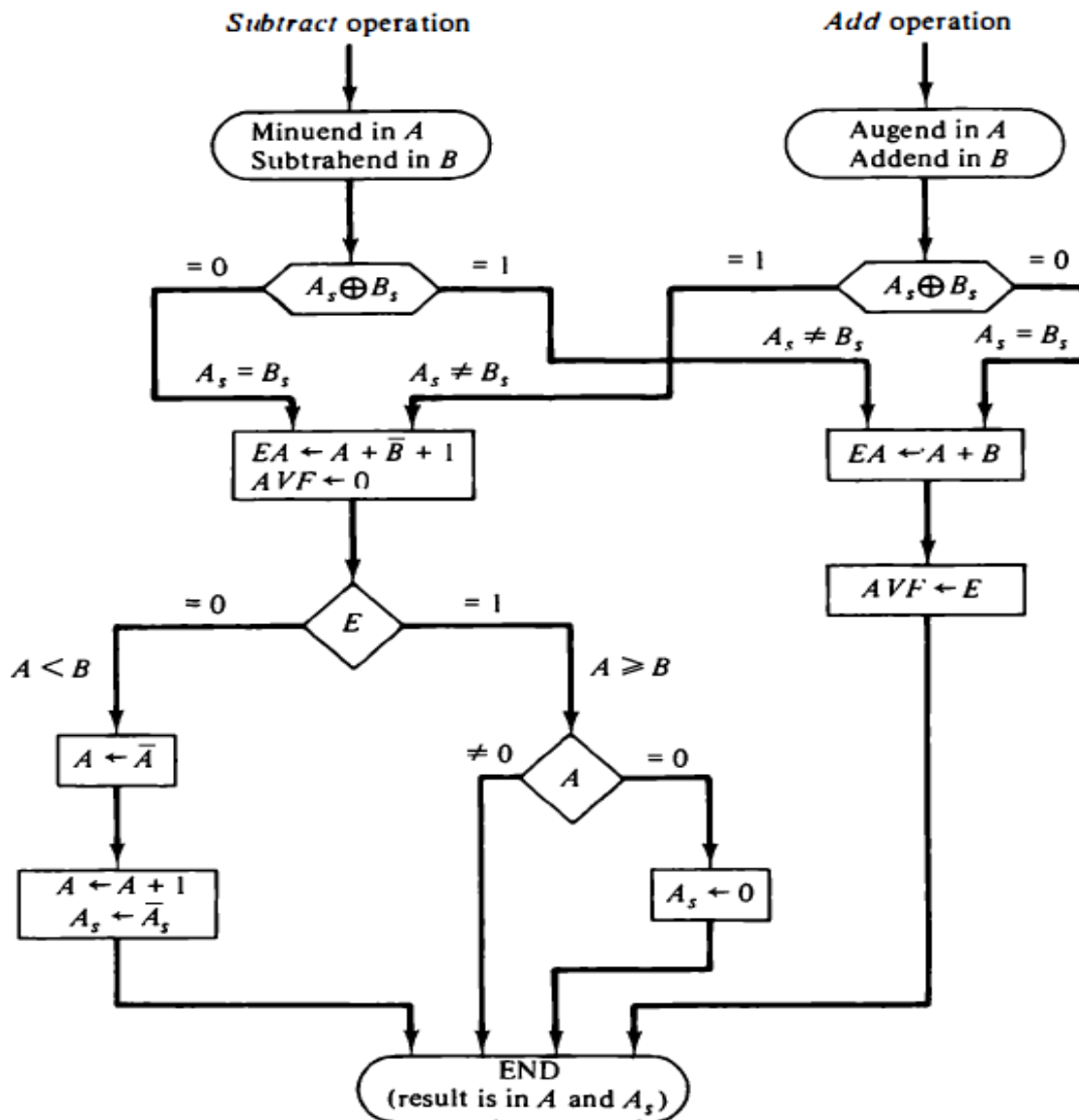


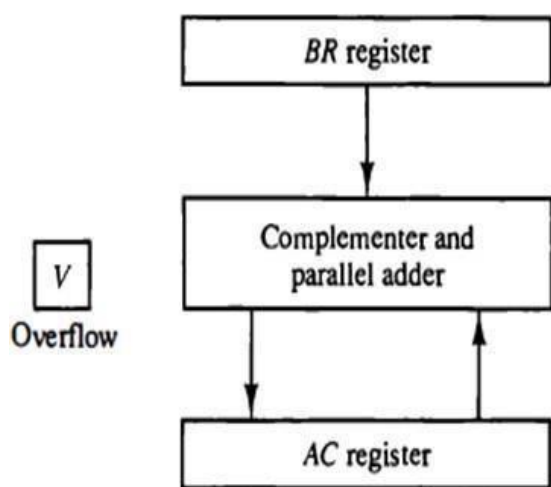
Figure (j): Flowchart for add and subtract operations

ii. Addition and Subtraction with Signed-2's Complement Data

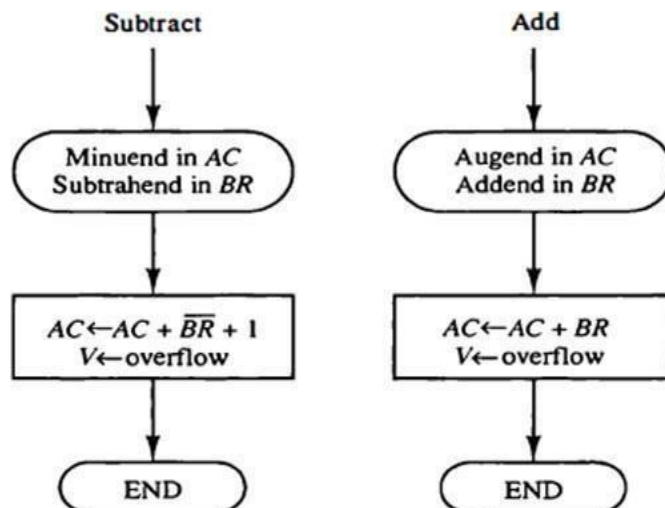
- The register configuration for the hardware implementation is shown in the below Figure(a). We name the A register AC (accumulator) and the B register BR. The leftmost bit in AC and BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded.
- The algorithm for adding and subtracting two binary numbers in signed-2's complement representation is shown in the flowchart of Figure(b). The sum is obtained by adding the contents of AC and

BR (including their sign bits). The overflow bit V is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise. The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR.

- Comparing this algorithm with its signed-magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed-2's complement representation.



Figure(a): Hardware for addition & subtraction of 2's complement numbers



Figure(b): Algorithm for adding & subtracting of 2's complement numbers

Multiplication Algorithms:

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with

paper and pencil by a process of successive shift and adds operations. This process is best illustrated with a numerical example.

23	10111	Multiplicand	
19	\times 10011		Multiplier
	10111		
	10111		
	00000	+	
	00000	}	
	10111	Partial Products	
437	110110101	Product	

The process of multiplication:

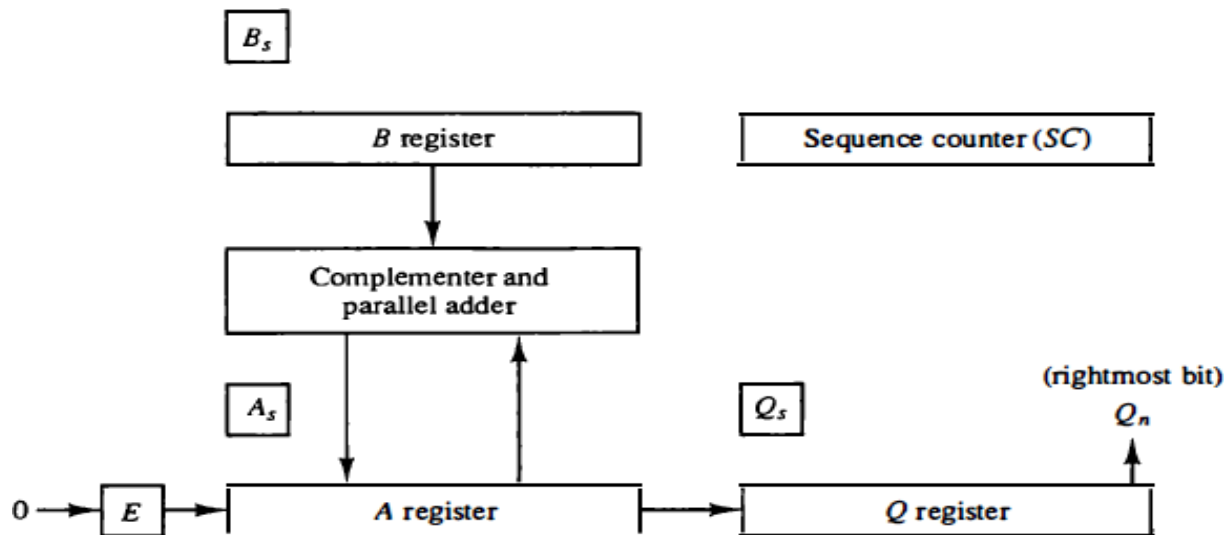
- It consists of looking at successive bits of the multiplier, least significant bit first.
- If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down.
- The numbers copied down in successive lines are shifted one position to the left from the previous number.
- Finally, the numbers are added and their sum forms the product.

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is **positive**. If

they are unlike, the sign of the product is negative.

Hardware Implementation for Signed-Magnitude Data

- The registers A, B and other equipment are shown in Figure (a). The multiplier is stored in the Q register and its sign in Qs. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.



Figure(k): Hardware for multiply operation.

- Initially, the multiplicand is in register B and the multiplier in Q, Their corresponding signs are in B_s and Q_s , respectively
- The sum of A and B forms a partial product which is transferred to the EA register.
- Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift.
- The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right.

In this manner, the rightmost flip-flop in register Q, designated by Q_n , will hold the bit of the multiplier, which must be inspected next.

Hardware Algorithm:

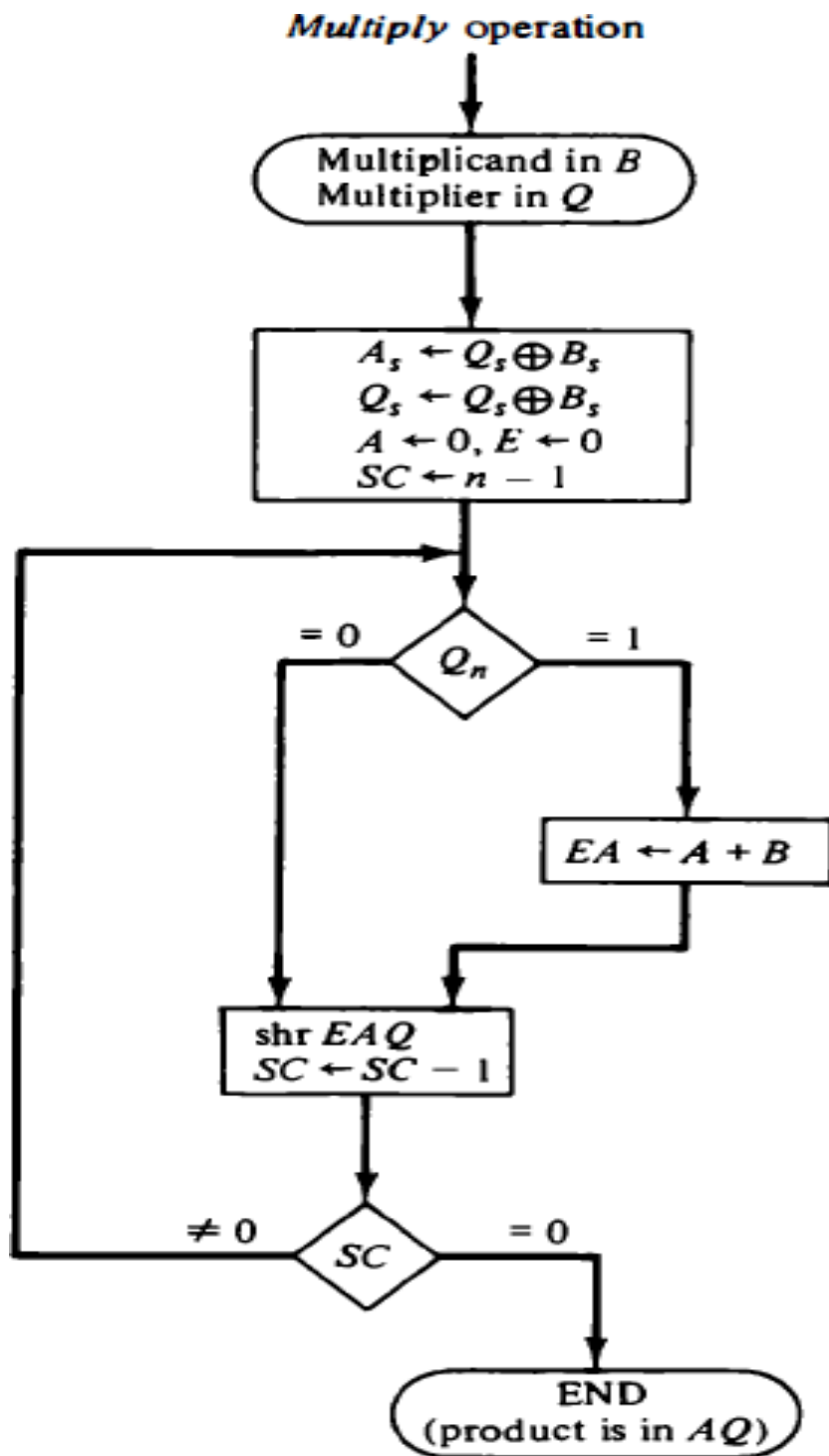
- Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s , respectively. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.
- After the initialization, the low-order bit of the multiplier in Q_n is tested.
 - i. If it is 1, the multiplicand in B is added to the present partial product in A .

- ii. If it is 0, nothing is done. Register EAQ is then shifted once to the right to form the new partial product.

□ The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when $SC = 0$.

□ The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

A flowchart of the hardware multiply algorithm is shown in the below figure (l).



Figure(1): Flowchart for multiply operation.

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		<u>10111</u>		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		<u>10111</u>		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		<u>10111</u>		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				

Figure (m): Numerical Example of multiplication

Booth Multiplication Algorithm:(multiplication of 2's complement data):

□Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation.

□Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

Hardware implementation of Booth algorithm Multiplication:

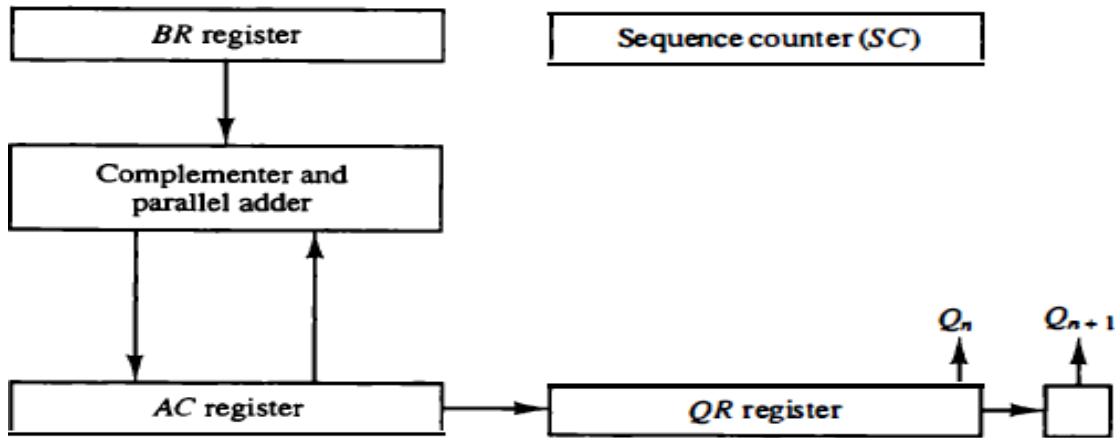


Figure (n): Hardware for Booth Algorithm

The hardware implementation of Booth algorithm requires the register configuration shown in figure (n). This is similar addition and subtraction hardware except that the sign bits are not separated from the rest of the registers. To show this difference, we rename registers A, B, and Q, as AC, BR, and QR, respectively. Q_n designates the least significant bit of the multiplier in register

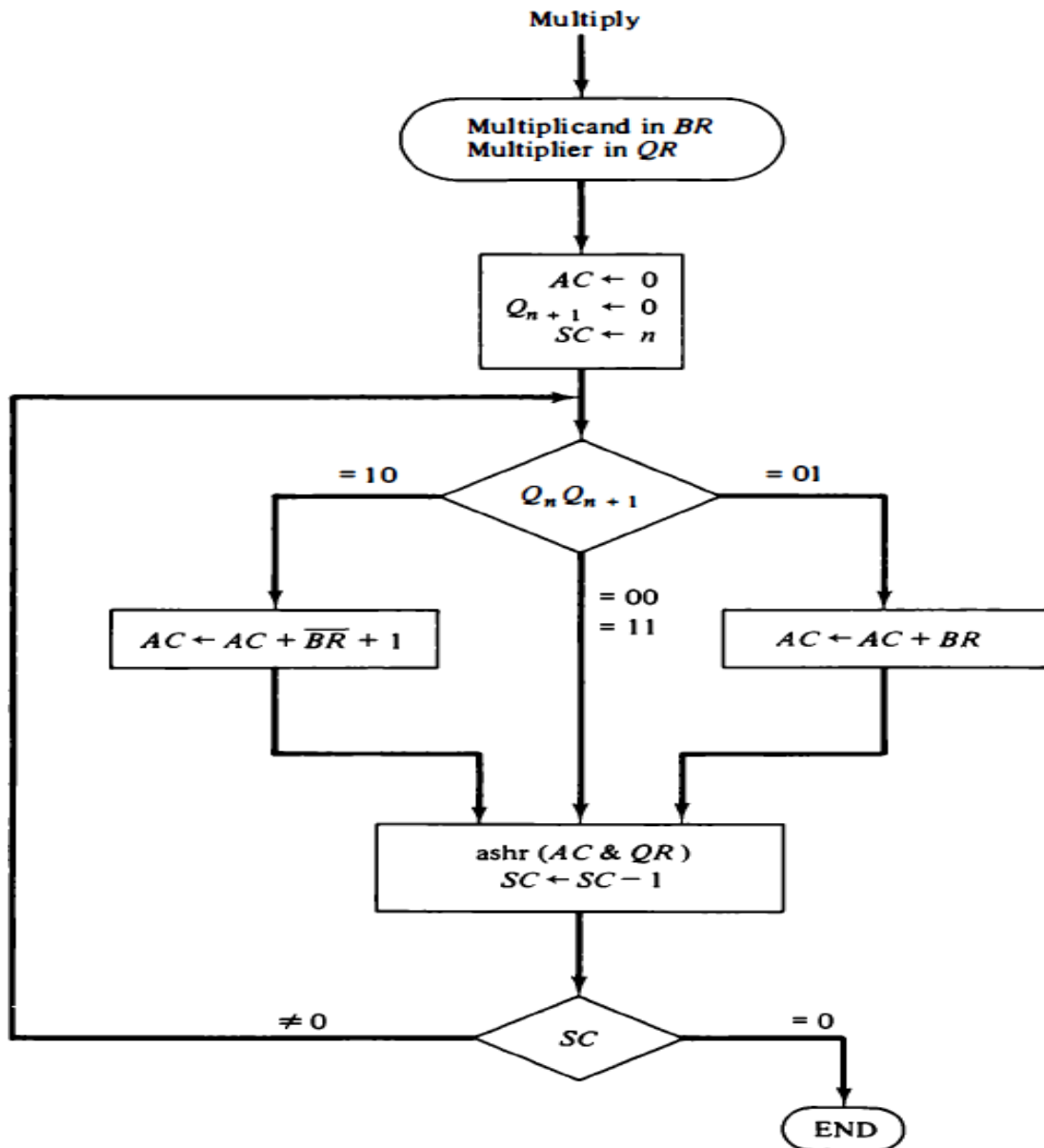
QR. An extra flip-flop Q_{n+1} , is appended to QR to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Figure (o).

Hardware Algorithm for Booth Multiplication:

□ AC and the appended bit Q_{n+1} are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Q_n and Q_{n+1} are inspected.

- i. If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
- ii. If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.
- iii. When the two bits are equal, the partial product does not change.

- iv. The next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the



sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.

Figure (o): Booth Algorithm for multiplication of 2's complement numbers

Example: multiplication of $(-9) \times (-13) = +117$ is shown below. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive.

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	<u>01001</u> 01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	<u>10111</u> 11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	<u>01001</u> 00111			
	ashr	00011	10101	1	000

Figure (p): Example of Multiplication with Booth Algorithm.

Division Algorithms:

- Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations.

The division process is illustrated by a numerical example in the below figure (q).

- The divisor B consists of five bits and the dividend A consists of ten bits. The five most significant bits of the dividend are compared with the divisor. Since the 5-bit number is smaller than B, we try again by taking the sixth most significant bits of A and compare

this number with B . The 6-bit number is greater than B , so we place a 1 for the quotient bit. The divisor is then shifted once to the right and subtracted from the dividend.

- The difference is called a **partial remainder** because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor.
- If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder.
- If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

Divisor:	11010	Quotient = Q
B = 10001)011100000	Dividend = A
	01110	5 bits of A < B, quotient has 5 bits
	011100	6 bits of A > B
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder > B
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder < B; enter 0 in Q; shift right B
	---010100	Remainder > B
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder < B; enter 0 in Q
	-----00110	Final remainder

Figure (q): Example of Binary Division

Hardware Implementation for Signed-Magnitude Data:

The hardware for implementing the division operation is identical to that required for multiplication.

- ✓ The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E.
- ✓ If E = 1, it signifies that A ≥ B. A quotient bit 1 is inserted into Q, and the partial remainder is shifted to the left to repeat the process.
- ✓ If E = 0, it signifies that A < B so the quotient in Q_n remains a 0. The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed.
- ✓ Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A.

The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign of the quotient is plus. If they are unlike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

Divide Overflow

- ❑ The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length.
- ❑ To see this, consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example shown in the above, we note that the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit.
- ❑ This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers.
- ❑ This condition detection must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor,

- i. A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor.
- ii. A division by zero must be avoided. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero. Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

Hardware Algorithm:

1. The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Qs to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient.

2. A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A. If $A \geq B$, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely. If $A < B$, no divide overflow occurs so the value of the dividend is restored by adding B to A.

3. The division of the magnitudes starts by shifting the dividend in AQ to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that $EA > B$ because EA consists of a 1 followed by $n-1$ bits while B consists of only $n-1$ bits. In this case, B must be subtracted from EA and 1 inserted into Q_n for the quotient bit.

4. If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E. If $E = 1$, it signifies that $A \geq B$; therefore, Q_n is set to 1. If $E = 0$, it signifies that $A < B$ and the original number is

restored by adding B to A . In the latter case we leave a 0 in Q_n .

This process is repeated again with registers EAQ . After n times, the quotient is formed in register Q and the remainder is found in register A .

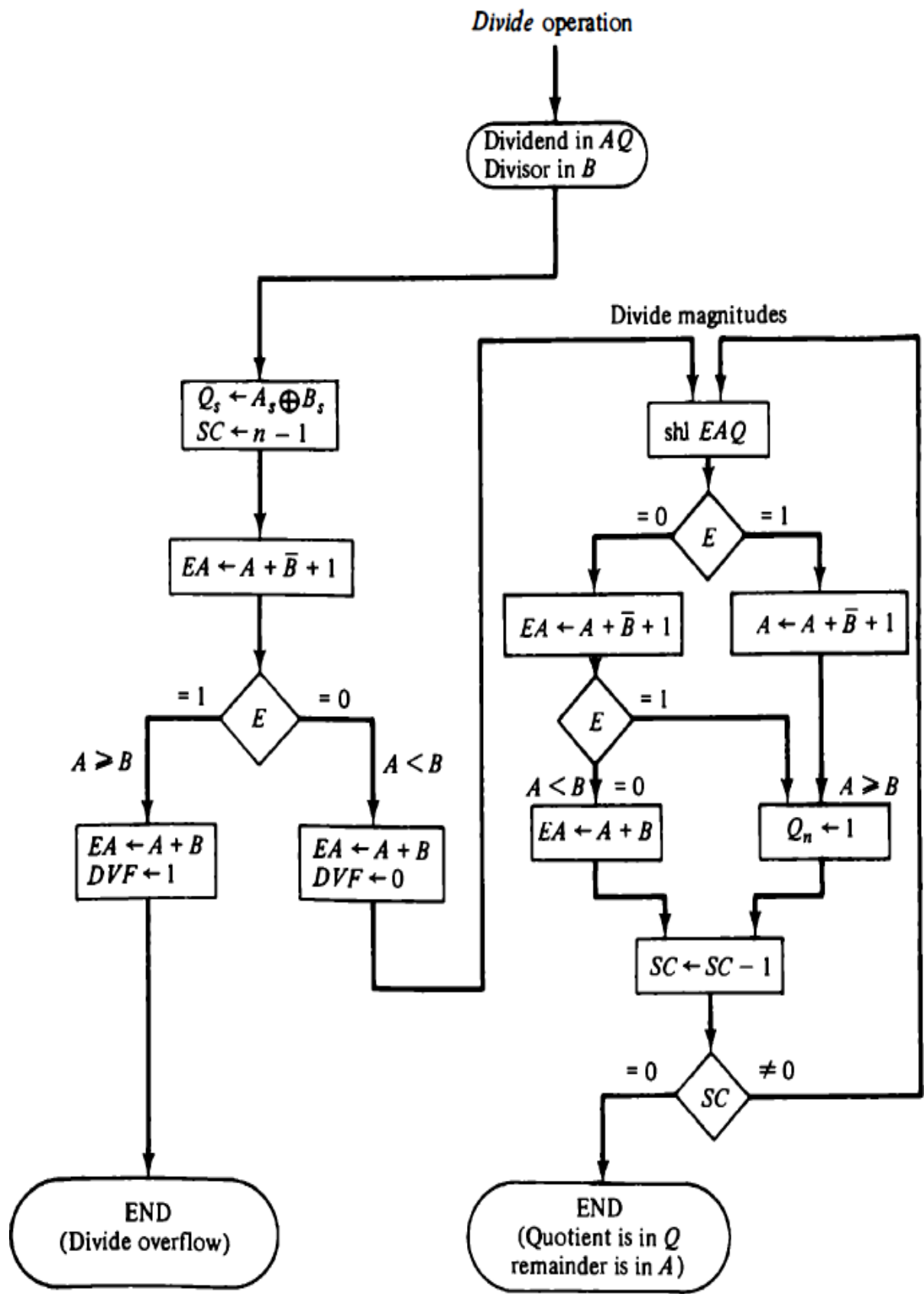


Figure (r): Flowchart for Divide operation

Divisor $B = 10001$,

$\bar{B} + 1 = 01111$

	<u>E</u>	<u>A</u>	<u>Q</u>	<u>SC</u>
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B} + 1$		<u>01111</u>		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\bar{B} + 1$		<u>01111</u>		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\bar{B} + 1$		<u>01111</u>		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		<u>10001</u>		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\bar{B} + 1$		<u>01111</u>		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\bar{B} + 1$		<u>01111</u>		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		<u>10001</u>		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A :		00110		
Quotient in Q :			11010	

Figure (s): Example of Binary Division

Basic Computer Organization and Design

Instruction Codes:

The general purpose digital computer is capable of executing various micro-operations and also can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by using a program.

- ❑ A **program** is a set of instructions that specify the operations, operands, and the sequence by which processing has to occur.
- ❑ A **computer instruction** is a binary code that specifies a sequence of microoperations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microoperations.
- ❑ An **instruction code** is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation.
- ❑ The most basic part of an instruction code is its operation part. The **operation code** of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.
- ❑ The **operation part** of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in processor registers or in memory.
- ❑ An instruction code must therefore specify not only the operation but also the registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored.

Stored Program Organization

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address.

- The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

The below figure shows this type of organization.

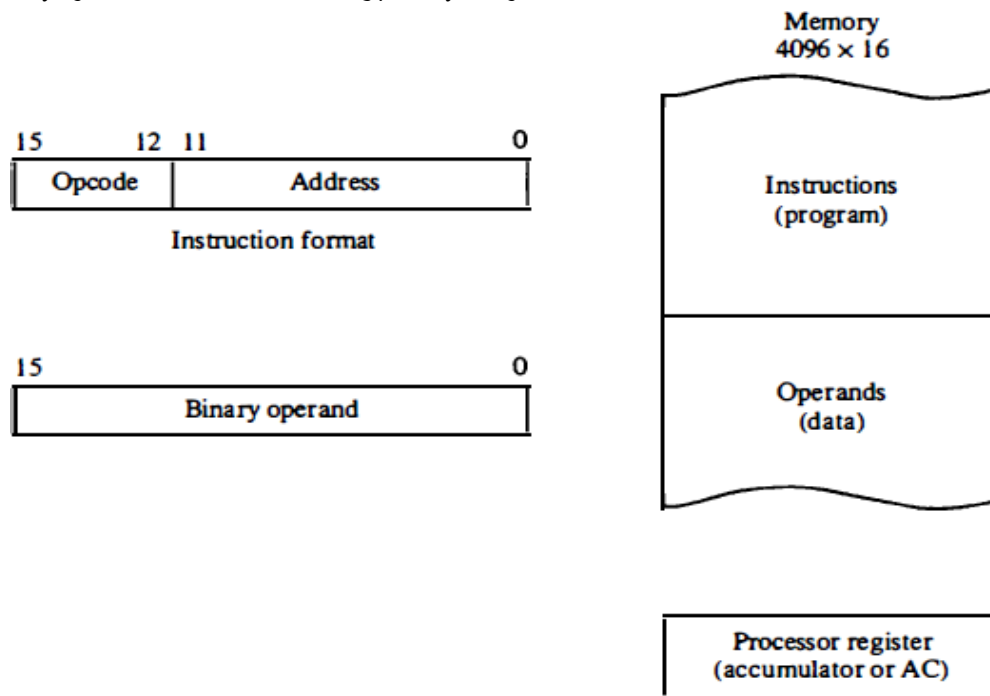


Figure (k): Stored program organization

Instructions are stored in one section of memory and data in another. **EX:** A memory unit with 4096 words, we need 12 bits to specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.

The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

- ❖ Computers that have a single-processor register usually assign to it

the name *accumulator* and label it *AC*. The operation is performed with the memory operand and the content of *AC*.

- ❖ If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as *clear AC*, *complement AC*, and *increment AC* operate on data stored in the *AC* register. They do not need an operand from memory.

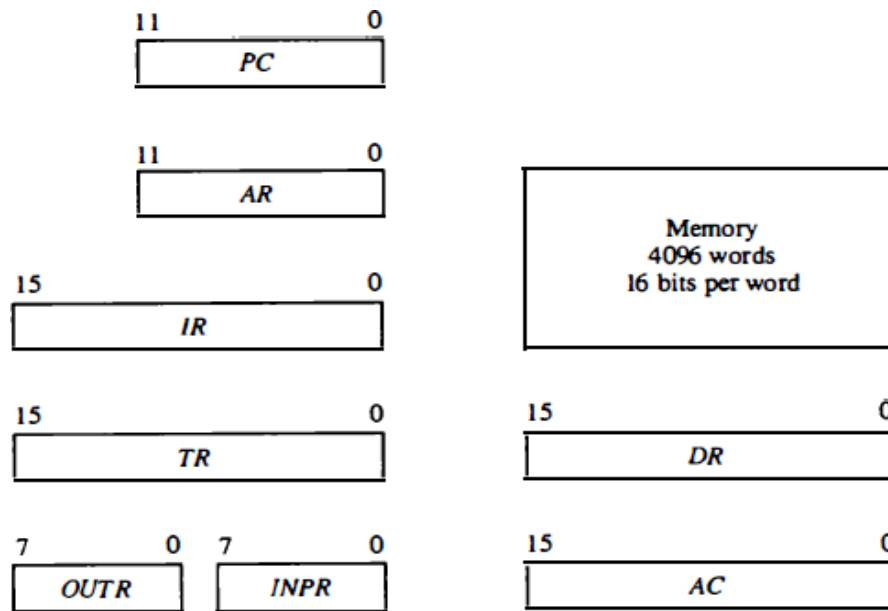
Indirect Address

- -When the second part of an instruction code specifies an operand, the instruction is said to have an *immediate operand*.
- When the second part specifies the address of an operand, the instruction is said to have a *direct address*.

consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on.

This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed. It is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory. The computer needs processor registers for manipulating data and a register for holding a memory address.

The registers available in the computer are shown in the below figure



(m) and table (A), a brief description of their function and the number of bits that they contain also given.

Figure (m): Basic computer registers and memory.

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Table (A): List of Registers for the Basic computer.

Common Bus System:

- The basic computer has eight registers, a memory unit, and a control unit. Paths must be provided to transfer information from one register to another and between memory and registers.
- The number of wires will be excessive if connections are made

between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus.

The connection of the registers and memory of the basic computer to a common bus system is shown in the below figure (n).

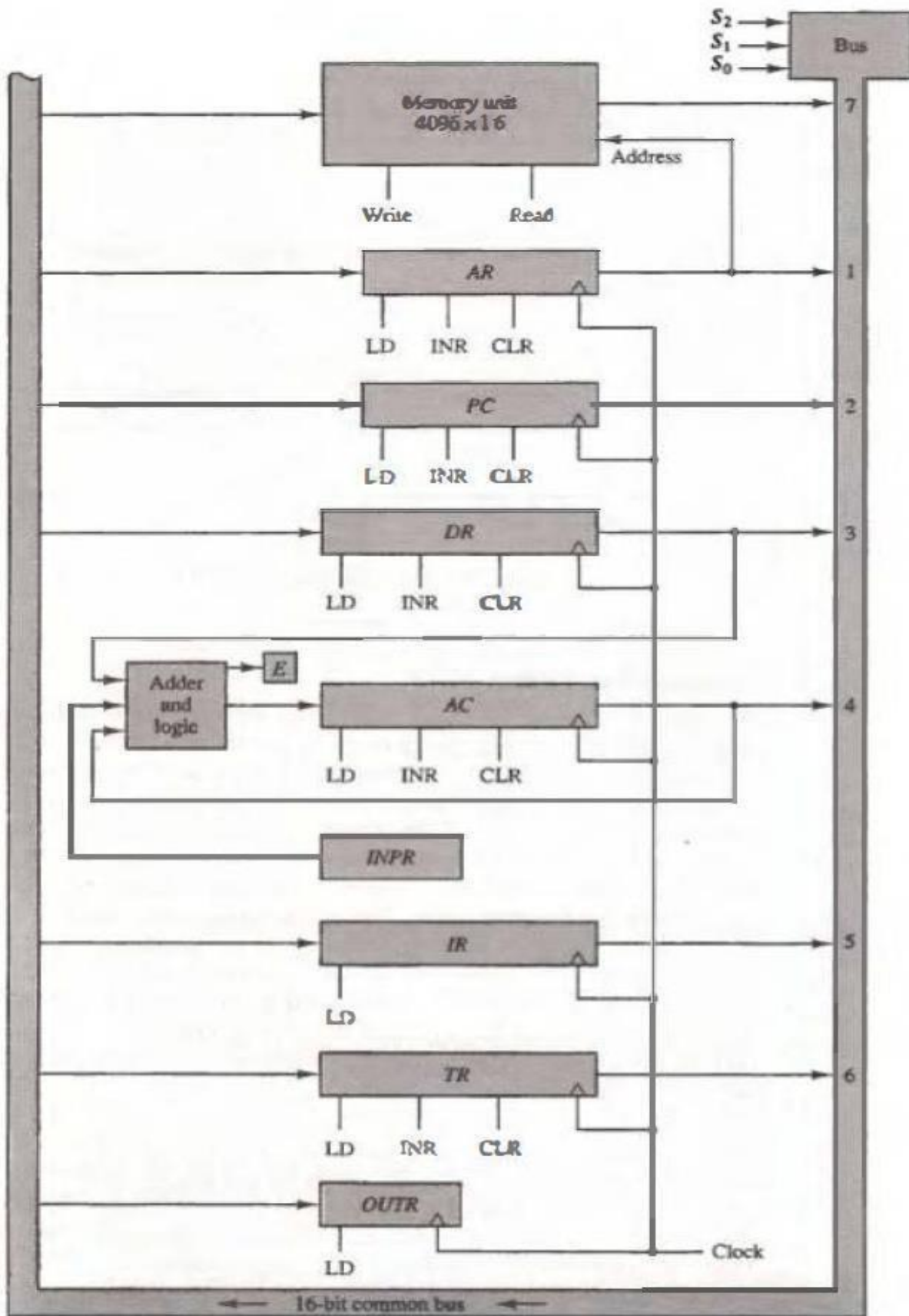


Figure (n): Basic computer registers connected to a common bus.

- The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S_2 , S_1 , and S_0 .

For example 1, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when $S_2S_1S_0 = 011$ since this is the binary value of decimal 3.

For example 2, The memory places its 16-bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

- The content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC.

For example, the two microoperations

$DR \leftarrow AC$ and $AC \leftarrow DR$

can be executed at the same time. This can be done by placing the content of AC on the bus (with $S_2S_1S_0 = 100$), enabling the LD (load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle.

Computer Instructions:

The basic computer has three types of instruction code formats,

1. Memory-reference instruction.
2. Register-reference instruction.
3. An input-output instruction.

Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.

The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction.

- If the three opcode bits in positions 12 to 14 are not equal to 111, the instruction is a *memory-reference type* and the bit in position 15 is taken as the addressing mode I . A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I . $I = 0$ for direct address and $I = 1$ for indirect address.
- If the 3-bit opcode = 111, control then inspects the bit in position 15. If this bit = 0, the instruction is a *register-reference type*. These instructions use 16 bits to specify an operation.
- If the bit $I = 1$, the instruction is an *input-output type*. These instructions also use all 16 bits to specify an operation.

The instructions for the computer are listed in Table (g, h, i).

Symbol	Hexadecimal code		Description
	$I = 0$	$I = 1$	
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero

Table (g): Memory-reference instructions

CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right AC and E
CIL	7040	Circulate left AC and E
INC	7020	Increment AC
SPA	7010	Skip next instruction if AC positive
SNA	7008	Skip next instruction if AC negative
SZA	7004	Skip next instruction if AC zero
SZE	7002	Skip next instruction if E is 0
HLT	7001	Halt computer

Table (h): Register-reference instructions

INP	F800	Input character to <i>AC</i>
OUT	F400	Output character from <i>AC</i>
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag
ION	F080	Interrupt on
IOF	F040	Interrupt off

Table (i): Input-output instructions

The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits.

A) **memory-reference instruction** has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I.

- i. When $I = 0$, the last four bits of an instruction have a hexadecimal digit equivalent from 0 (000) to 6 (110) since the last bit is 0.
- ii. When $I = 1$, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 (1000) to E (1110) since the last bit is 1.

B) **Register-reference instructions** use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12 bits.

C) **The input-output instructions** also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

Instruction Set Completeness

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be **complete** if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions.
2. Instructions for moving information to and from memory and processor registers.
3. Program control instructions together with instructions that check status conditions.
4. Input and output instructions.

Instruction Cycle:

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of subcycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. **Fetch** an instruction from memory.
2. **Decode** the instruction.
3. **Read** the effective address from memory if the instruction has an indirect address.
4. **Execute** the instruction.

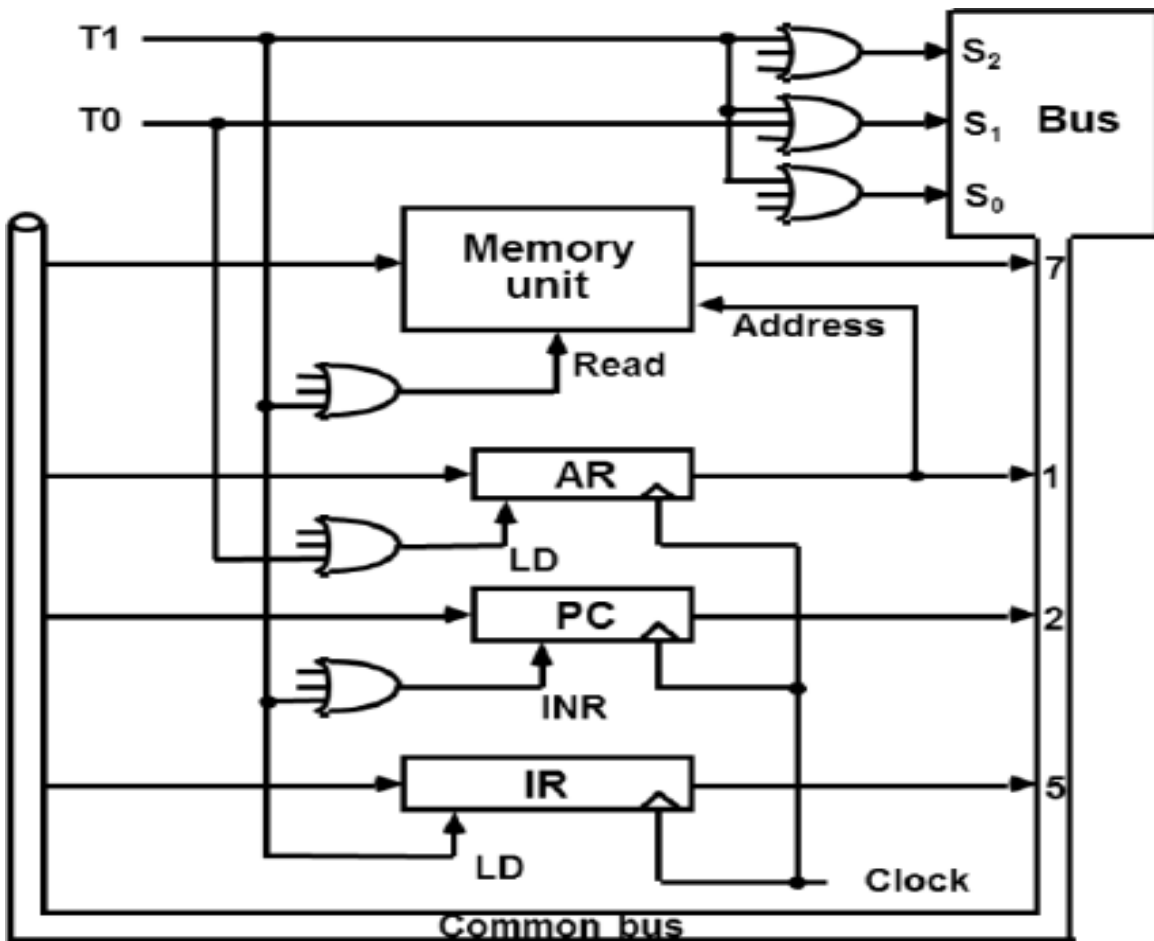
Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch and Decode:

Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0 . After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0 , T_1 , T_2 , and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

$T_0: AR \leftarrow PC \ (S_0S_1S_2=010, T_0=1)$
 $T_1: IR \leftarrow M[AR], PC \leftarrow PC + 1 \ (S_0S_1S_2=111, T_1=1)$
 $T_2: D_0, \dots, D_7 \leftarrow \text{Decode } IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T_0 . The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T_1 . At the same time, PC is



incremented by one to prepare it for the address of the next instruction in the program. At time T_2 , the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. Note that SC is incremented after each clock pulse to produce the sequence T_0 , T_1 , and T_2 .

Figure (o): Register transfers for the fetch phase

The above Figure (o) shows how the first two register transfer statements

are implemented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to achieve the following connection:

1. Place the content of PC onto the bus by making the bus selection inputs $S_2 S_1 S_0$ equal to 010.
2. Transfer the content of the bus to AR by enabling the LD input of AR. The next clock transition initiates the transfer from PC to AR since $T_0 = 1$.

In order to implement the second statement

$$T1: IR \leftarrow M[AR], PC \leftarrow PC + 1$$

It is necessary to use timing signal T_1 to provide the following connections in the bus system.

1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2 S_1 S_0 = 111$.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

Determine the Type of Instruction

The timing signal that is active after the decoding is T_3 . During time T_3 the control unit determines the type of instruction that was just read from memory.

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111.

- If $D_7 = 1$, the instruction must be a register-reference or input-Output type.
- If $D_7 = 0$, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3 . This can be symbolized as follows:

$D_7'IT_3$: $AR \leftarrow M[AR]$
 $D_7'I'T_3$: Nothing
 $D_7I'T_3$: Execute a register-reference instruction
 D_7IT_3 : Execute an input-output instruction

When a memory-reference instruction with $I = 0$ is encountered, it is not necessary to do anything since the effective address is already in AR. However, the sequence counter SC must be incremented when $D_7'T_3 = 1$, so that the execution of the memory-reference instruction can be continued with timing variable T_4 . A register-reference or input-output instruction can be executed with the clock associated with timing signal T_3 . After the instruction is executed, SC is cleared to 0 and control

returns to the fetch phase with $T_0 = 1$.

The flowchart of Figure (p) presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding

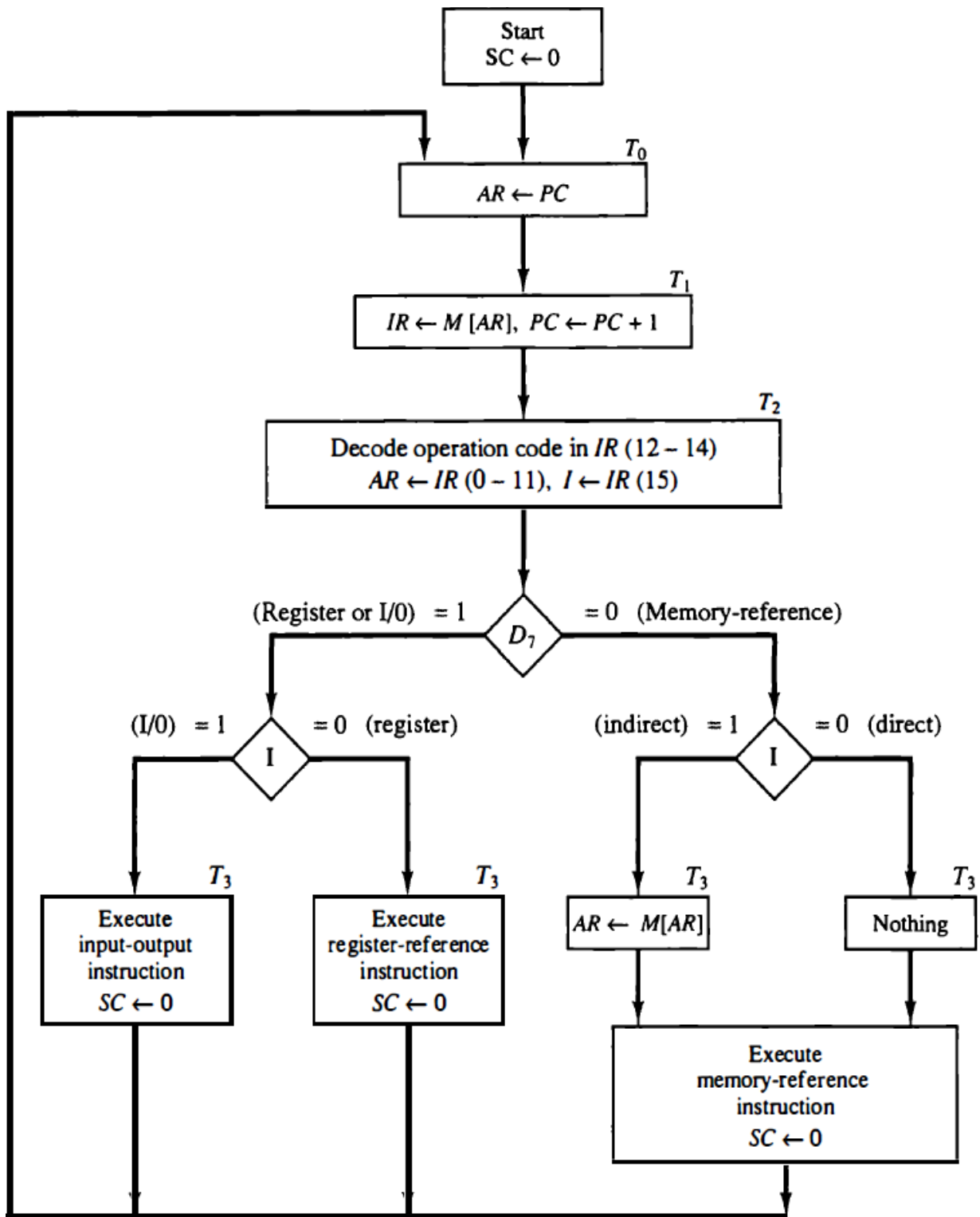


Figure (p): Flowchart for instruction cycle (initial configuration).

Register-Reference Instructions:

Register-reference instructions are recognized by the control when $D_7 =$

1 and $I = 0$. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in $IR(0-11)$. They were also transferred to AR during time T_2 .

- Each control function needs the Boolean relation $D_7 \wedge T_3$, which we designate for convenience by the symbol r . The control function is distinguished by one of the bits in

IR(0-11). By assigning the symbol B_i to bit i of IR, all control functions can be simply denoted by rB_i .

➤ For example, the instruction CLA has the hexadecimal code 7800, which gives the binary equivalent 0111 1000 0000 0000.

- i. The first bit is a zero and is equivalent to 1'.
- ii. The next three bits constitute the operation code and are recognized from decoder output D_7 .
- iii. Bit 11 in IR is 1 and is recognized from B_{11} .

The control function that initiates the microoperation for this instruction is $D_7 I' T_3 B_{11} = r B_{11}$

$D_7 I' T_3 = r$ (common to all register-reference instructions)

$IR(i) = B_i$ [bit in IR(0-11) that specifies the operation]

	r :	$SC \leftarrow 0$	Clear SC
CLA	rB_{11} :	$AC \leftarrow 0$	Clear AC
CLE	rB_{10} :	$E \leftarrow 0$	Clear E
CMA	rB_9 :	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8 :	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7 :	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circulate right
CIL	rB_6 :	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circulate left
INC	rB_5 :	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4 :	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3 :	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2 :	If $(AC = 0)$ then $PC \leftarrow PC + 1$	Skip if AC zero
SZE	rB_1 :	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0 :	$S \leftarrow 0$ (S is a start-stop flip-flop)	Halt computer

Table (j): Execution of Register-Reference Instructions

Memory-Reference Instructions:

- ✓ The below Table (k) lists the seven memory-reference instructions. The decoded output D_i for $i = 0, 1, 2, 3, 4, 5,$ and 6 from the operation decoder that belongs to each instruction is included in the table.
- ✓ The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when $I = 0$, or during timing signal T_3 when $I = 1$. The execution of the memory-reference instructions starts with timing signal T_4 .

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

Table (k): Memory-Reference Instructions

AND : AND to AC

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of the operation is transferred to AC. The microoperations that execute this instruction are:

$D_0T_4: DR \leftarrow M[AR]$

$D_0T_5: AC \leftarrow AC \wedge DR, SC \leftarrow 0$

ADD : ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry C_{out} , is transferred to the E (extended accumulator) flip-flop. The microoperations needed to execute this instruction are:

$D_1T_4: DR \leftarrow M[AR]$

$D_1T_5: AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are:

$D_2T_4: DR \leftarrow M[AR]$

$D_2T_5: AC \leftarrow DR, SC$

$\leftarrow 0$

STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address. Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

$D_3T_4: M[AR] \leftarrow AC, SC \leftarrow 0$

BUN: Branch Unconditionally

- This instruction transfers the program to the instruction specified by the effective address.
- The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one microoperation:

BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available

$M[AR] \leftarrow PC, PC \leftarrow AR + 1$

in PC) into a memory location specified by the effective address. The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine.

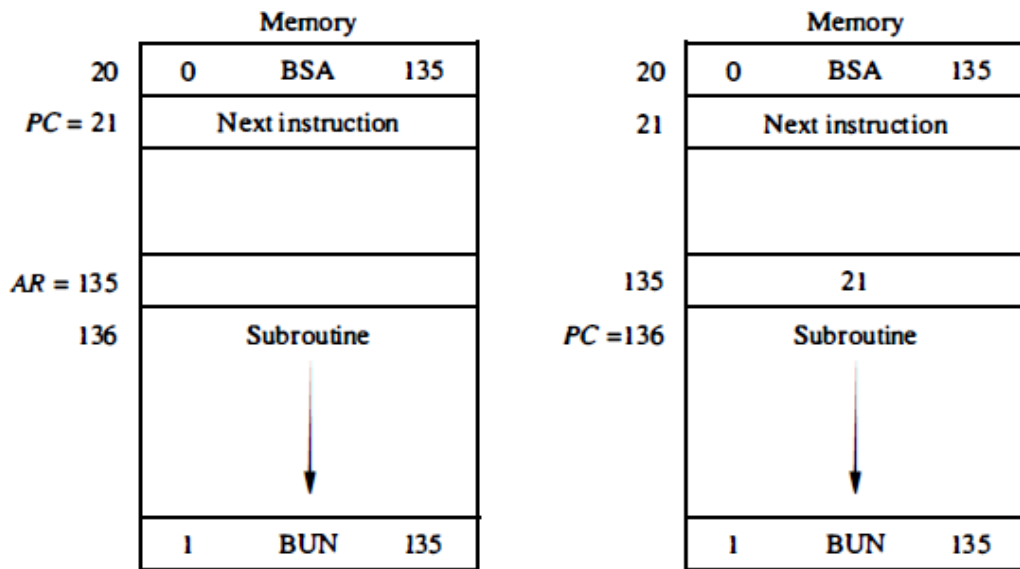
BSA: Branch and Save Return

AddressEX:

The BSA instruction is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 135. After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135. This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

$$M[135] \leftarrow 21, \quad PC \leftarrow 135 + 1 = 136$$

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine. When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to PC. The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.



(a) Memory, PC, and AR at time T_4

(b) Memory and PC after execution

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

$$\begin{aligned}
 D_5T_4: & \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1 \\
 D_5T_5: & \quad PC \leftarrow AR, \quad SC \leftarrow 0
 \end{aligned}$$

Timing signal T_4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR. The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T_5 to transfer the content of AR to PC.

ISZ: Increment and Skip if Zero

This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, PC is incremented by 1. The programmer usually stores a negative number (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program.

$D_6T_4: DR \leftarrow M[AR]$

$D_6T_5: DR \leftarrow DR + 1$

$D_6T_6: M[AR] \leftarrow DR, \text{ if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), SC \leftarrow 0$

A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Figure (q). The control functions are indicated on top of each box. The microoperations that are performed during time T_4 , T_5 , or T_6 , depend on the operation code value. The sequence counter SC is cleared to 0 with the last timing signal in each case. This causes a transfer of control to timing signal T_0 to start the next instruction cycle.

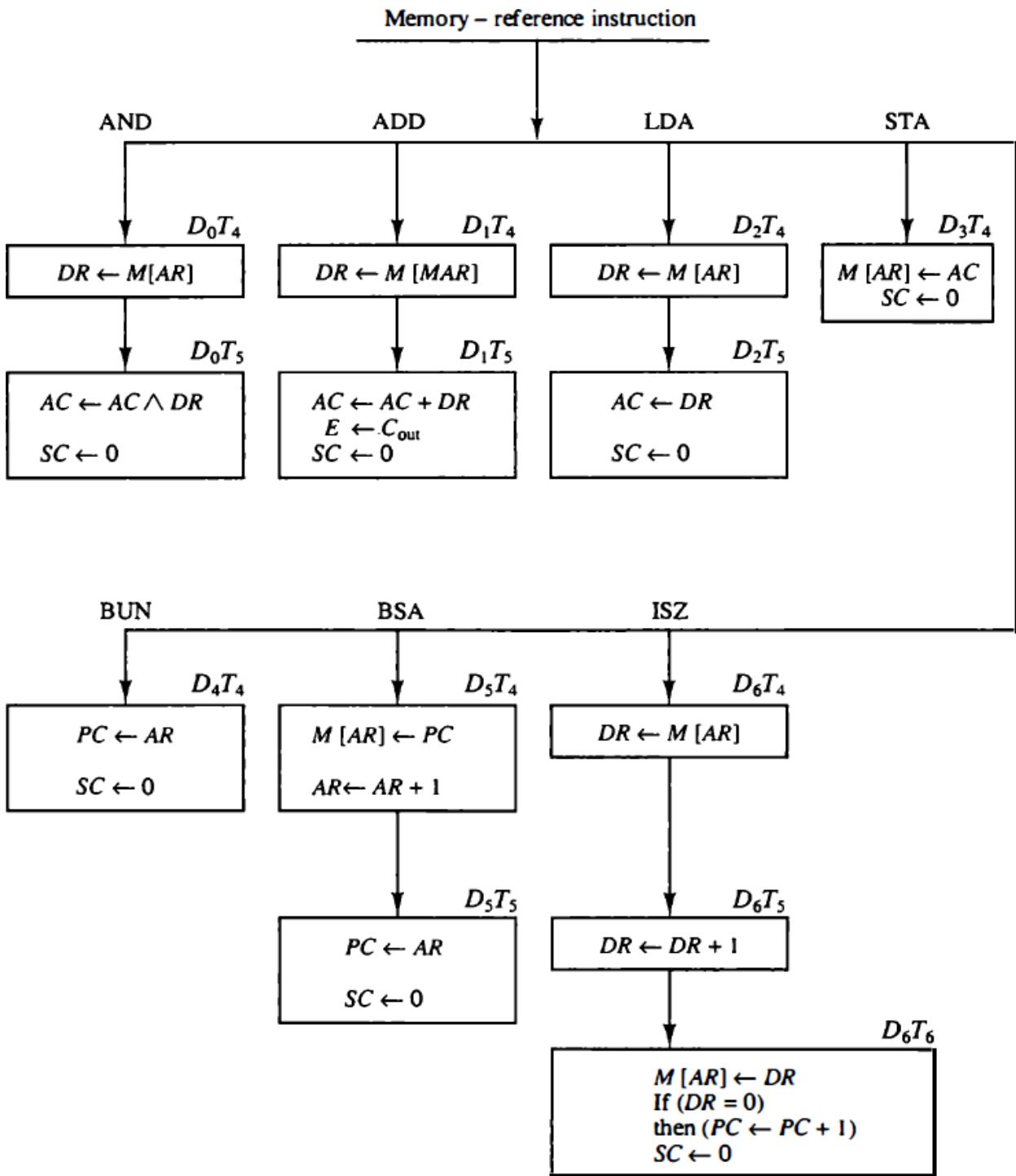


Figure (q): Flowchart for Memory-reference instructions

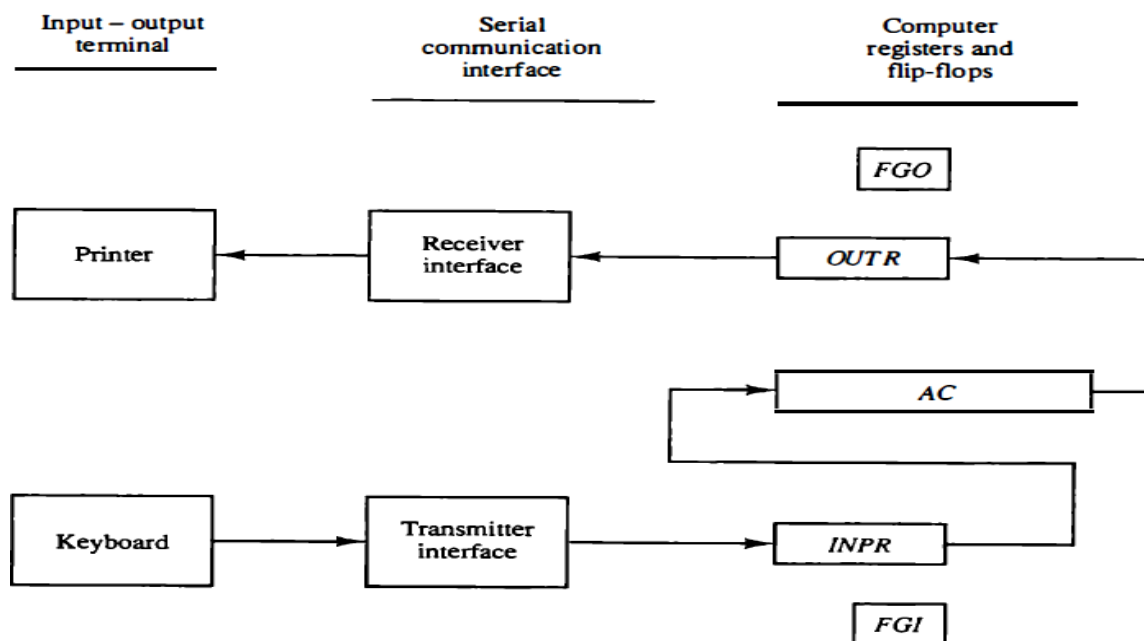
Input-Output and Interrupt.

computer can serve no useful purpose unless it communicates with the external

environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices.

Input-Output Configuration

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR.



The serial information for the printer is stored in the output register OUTR . These two registers communicate with a communication interface serially and with the AC in parallel.

Figure (r): Input-Output configuration

The process of information transfer is as follows: Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTF and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1. The computer does not load a new character into OUTF when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

Input-Output Instructions

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility.

Input-output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and $I = 1$. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed in Table (I).

$D_7IT_3 = p$ (common to all input-output instructions)
 $IR(i) = B_i$ [bit in $IR(6-11)$ that specifies the instruction]

	p :	$SC \leftarrow 0$	Clear SC
INP	pB_{11} :	$AC(0-7) \leftarrow INPR, FGI \leftarrow 0$	Input character
OUT	pB_{10} :	$OUTR \leftarrow AC(0-7), FGO \leftarrow 0$	Output character
SKI	pB_9 :	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8 :	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7 :	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6 :	$IEN \leftarrow 0$	Interrupt enable off

Table (L): Input-Output instructions

Program Interrupt

The process of communication discussed so far is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and the input-output device makes this type of transfer inefficient.

- To see why this is inefficient, consider a computer that can go through an instruction cycle in $1\mu s$. Assume that the input-output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every $100,000\mu s$. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.
- An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility.
- While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is

informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

- The interrupt enable flip-flop IEN can be set and cleared with two instructions (IOF and ION instructions).

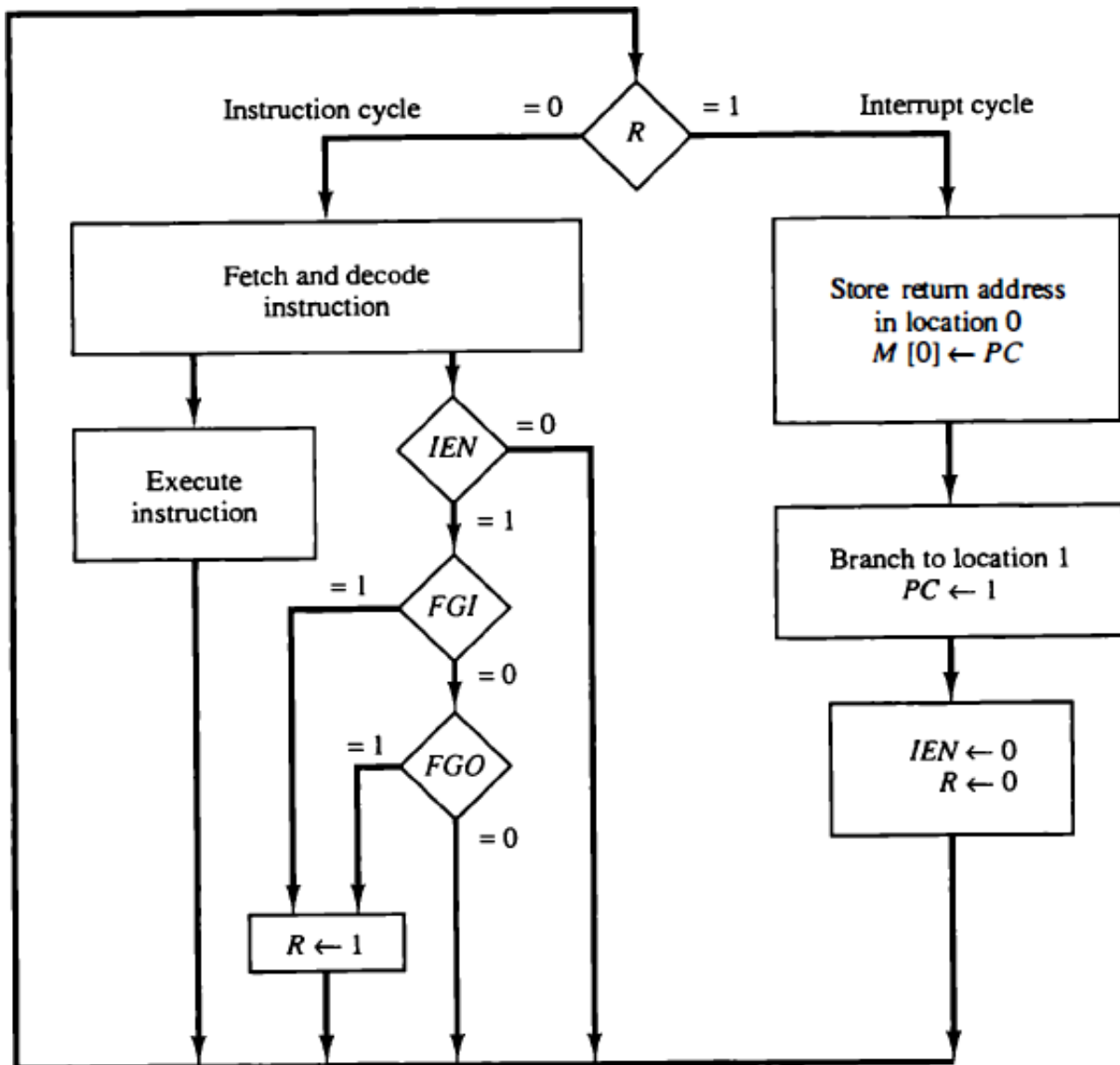


Figure (s): Flowchart for interrupt cycle

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Figure (s).

- ❑ An interrupt flip-flop R is included in the computer. When $R = 0$, the computer goes through an instruction cycle.
- ❑ During the execute phase of the instruction cycle IEN is checked by the control. If it is 0 , it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle.
- ❑ If IEN is 1 , control checks the flag bits. If both flags are 0 , it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while $IEN = 1$, flip-

flop R is set to 1.

- At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

The interrupt cycle is a hardware implementation of a branch and save return address (BSA)

operati

on. EX:

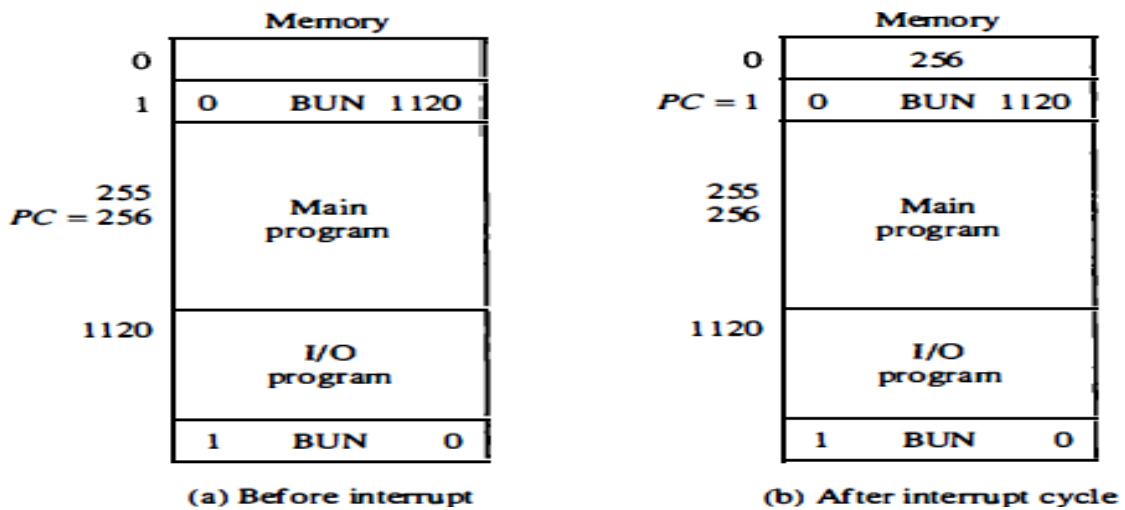


Figure (f): Demonstration of Interrupt Cycle

An example that shows what happens during the interrupt cycle is shown in Figure (f). Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Figure (a).

When control reaches timing signal TO and finds that $R = 1$, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the program returns to the location where it was interrupted. This is shown in Figure (b).

Interrupt Cycle

The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if $LEN = 1$ and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals TO, T1 or T2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

$T_0 T_1 T_2 (IEN)(FGI + FGO): R \leftarrow 1$

During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR. With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increments PC to 1, clears IEN and R, and control goes back to T0 by clearing SC to 0. The beginning of the next instruction cycle has the condition R' T0 and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

$RT_0: AR \leftarrow 0, TR \leftarrow PC$

$RT_1: M[AR] \leftarrow TR, PC \leftarrow 0$

$RT_2: PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$

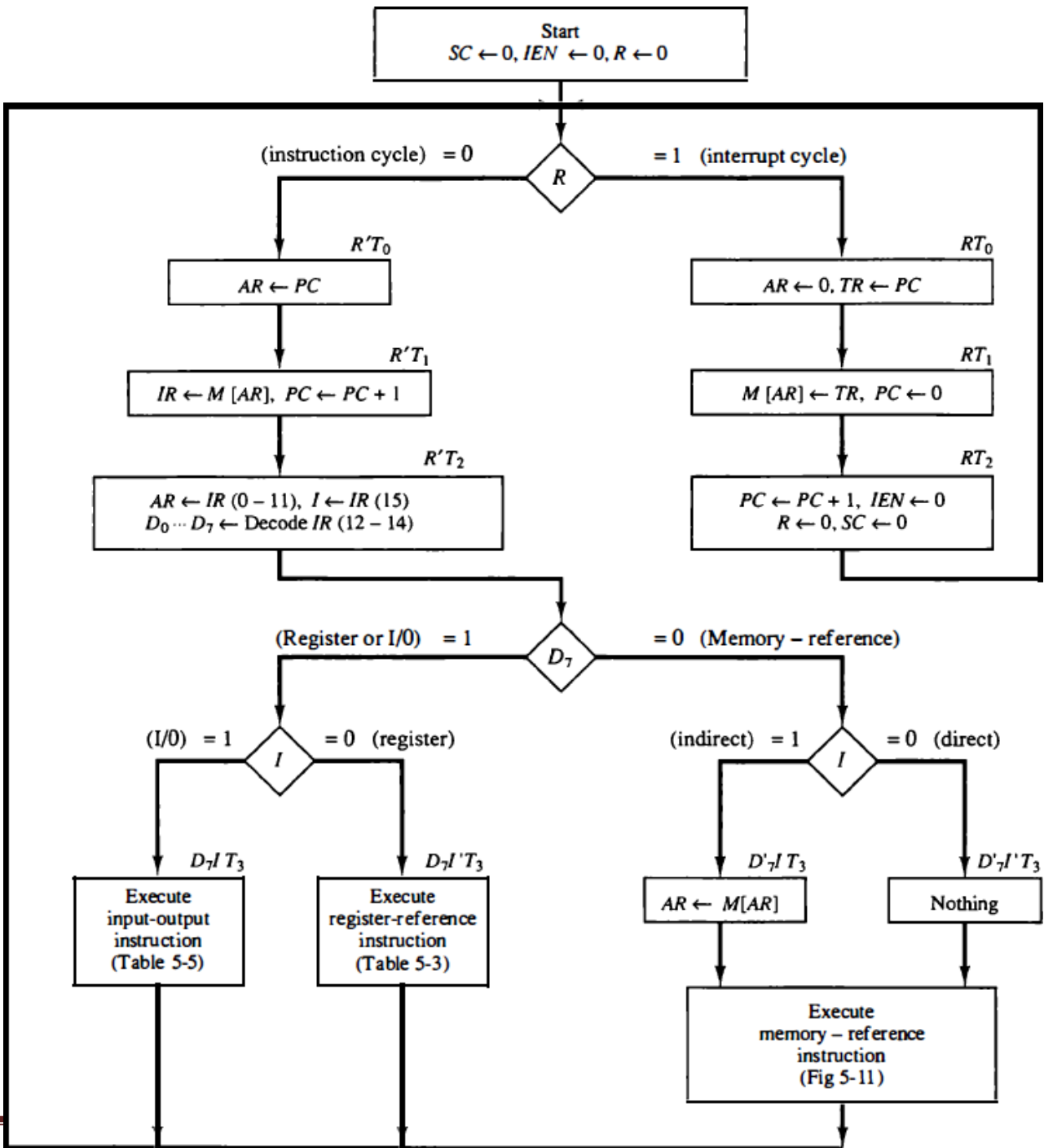
Complete Computer Description:

The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is

shown in the below figure (u). The interrupt flip-flop R may be set at any time during the indirect or execute phases. Control returns to timing signal T_0 after SC is cleared to 0.

- If $R = 1$, the computer goes through an interrupt cycle.
- If $R = 0$, the computer goes through an instruction cycle.

If the instruction is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruction. If the instruction is one of the



register-reference instructions, it will be executed. If it is an input-output instruction, it will be executed.

Figure (u): Flowchart for computer operation

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR], PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \dots, D_7 \leftarrow \text{Decode } IR(12-14),$ $AR \leftarrow IR(0-11), I \leftarrow IR(15)$
Indirect	D_7IT_3 :	$AR \leftarrow M[AR]$
Interrupt	$T_0T_1T_2(IEN)(FGI + FGO)$:	$R \leftarrow 1$
	RT_0 :	$AR \leftarrow 0, TR \leftarrow PC$
	RT_1 :	$M[AR] \leftarrow TR, PC \leftarrow 0$
	RT_2 :	$PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4 :	$DR \leftarrow M[AR]$
	D_0T_5 :	$AC \leftarrow AC \wedge DR, SC \leftarrow 0$
ADD	D_1T_4 :	$DR \leftarrow M[AR]$
	D_1T_5 :	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$
LDA	D_2T_4 :	$DR \leftarrow M[AR]$
	D_2T_5 :	$AC \leftarrow DR, SC \leftarrow 0$
STA	D_3T_4 :	$M[AR] \leftarrow AC, SC \leftarrow 0$
BUN	D_4T_4 :	$PC \leftarrow AR, SC \leftarrow 0$
BSA	D_5T_4 :	$M[AR] \leftarrow PC, AR \leftarrow AR + 1$
	D_5T_5 :	$PC \leftarrow AR, SC \leftarrow 0$
ISZ	D_6T_4 :	$DR \leftarrow M[AR]$
	D_6T_5 :	$DR \leftarrow DR + 1$
	D_6T_6 :	$M[AR] \leftarrow DR, \text{ if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), SC \leftarrow 0$
Register-reference		
	$D_7I'T_3 = r$ (common to all register-reference instructions)	
	$IR(i) = B_i, (i = 0, 1, 2, \dots, 11)$	
	r :	$SC \leftarrow 0$
CLA	rB_{11} :	$AC \leftarrow 0$
CLE	rB_{10} :	$E \leftarrow 0$
CMA	rB_9 :	$AC \leftarrow \overline{AC}$
CME	rB_8 :	$E \leftarrow \overline{E}$
CIR	rB_7 :	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	rB_6 :	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	rB_5 :	$AC \leftarrow AC + 1$
SPA	rB_4 :	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3 :	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2 :	If $(AC = 0)$ then $PC \leftarrow PC + 1$
SZE	rB_1 :	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0 :	$S \leftarrow 0$
Input-output:		
	$D_7IT_3 = p$ (common to all input-output instructions)	
	$IR(i) = B_i, (i = 6, 7, 8, 9, 10, 11)$	
	p :	$SC \leftarrow 0$
INP	pB_{11} :	$AC(0-7) \leftarrow INPR, FGI \leftarrow 0$
OUT	pB_{10} :	$OUTR \leftarrow AC(0-7), FGO \leftarrow 0$
SKI	pB_9 :	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8 :	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7 :	$IEN \leftarrow 1$
IOF	pB_6 :	$IEN \leftarrow 0$

Table (m): Control functions and microoperations for
the Basic computer

Instead of using a flowchart, we can describe the operation of the computer with a list of register transfer statements. This is done by accumulating all the control functions and microoperations in one table, as shown in the below Table (m).

The register transfer statements in this table describe in a concise form the internal organization of the basic computer. They also give all the information necessary for the design of the logic circuits of the computer.

A register transfer language is useful not only for describing the internal organization of a digital system but also for specifying the logic circuits needed for its design.

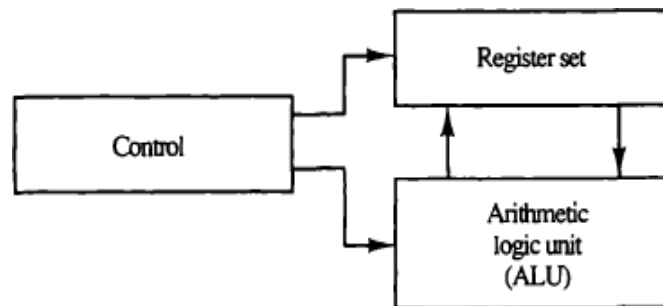
Syllabus:

Central Processing Unit: General Register Organization, STACK Organization. Instruction Formats, Addressing Modes, Data Transfer and Manipulation, Program Control, Reduced Instruction Set Computer.

Microprogrammed Control: Control Memory, Address Sequencing, Micro Program example, Design of Control Unit.

Introduction:

The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU. The CPU is made up of three major parts, as shown in Figure (1). The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of



information among the registers and instructs the ALU as to which operation to perform.

Figure (1): Major components of CPU

One boundary where the computer designer and the computer programmer see the same machine is the part of the CPU associated with the instruction set.

- From the designer's point of view, the computer instruction set provides the specifications for the design of the CPU. The design of a CPU is a task that in large part involves choosing the hardware for implementing the machine instructions.
- The user who programs the computer in machine or assembly language must be aware of the register set, the memory structure, the type of data supported by the instructions, and the function that each instruction performs.

The following sections describe the organization and architecture of the CPU with an emphasis on the user's view of the computer, how the registers communicate with the ALU through buses, explain the operation of the memory stack, the type of instruction formats available, the addressing modes used to retrieve data from memory, and also the concept of reduced instruction set computer (RISC).

General Register Organization:

- We know that the memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplication. Having to

refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer.

- It is more convenient and more efficient to store these intermediate values in processor registers. When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system.
- The registers communicate with each other not only for direct data transfers, but also while performing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

A bus organization for seven CPU registers is shown in the below figure.

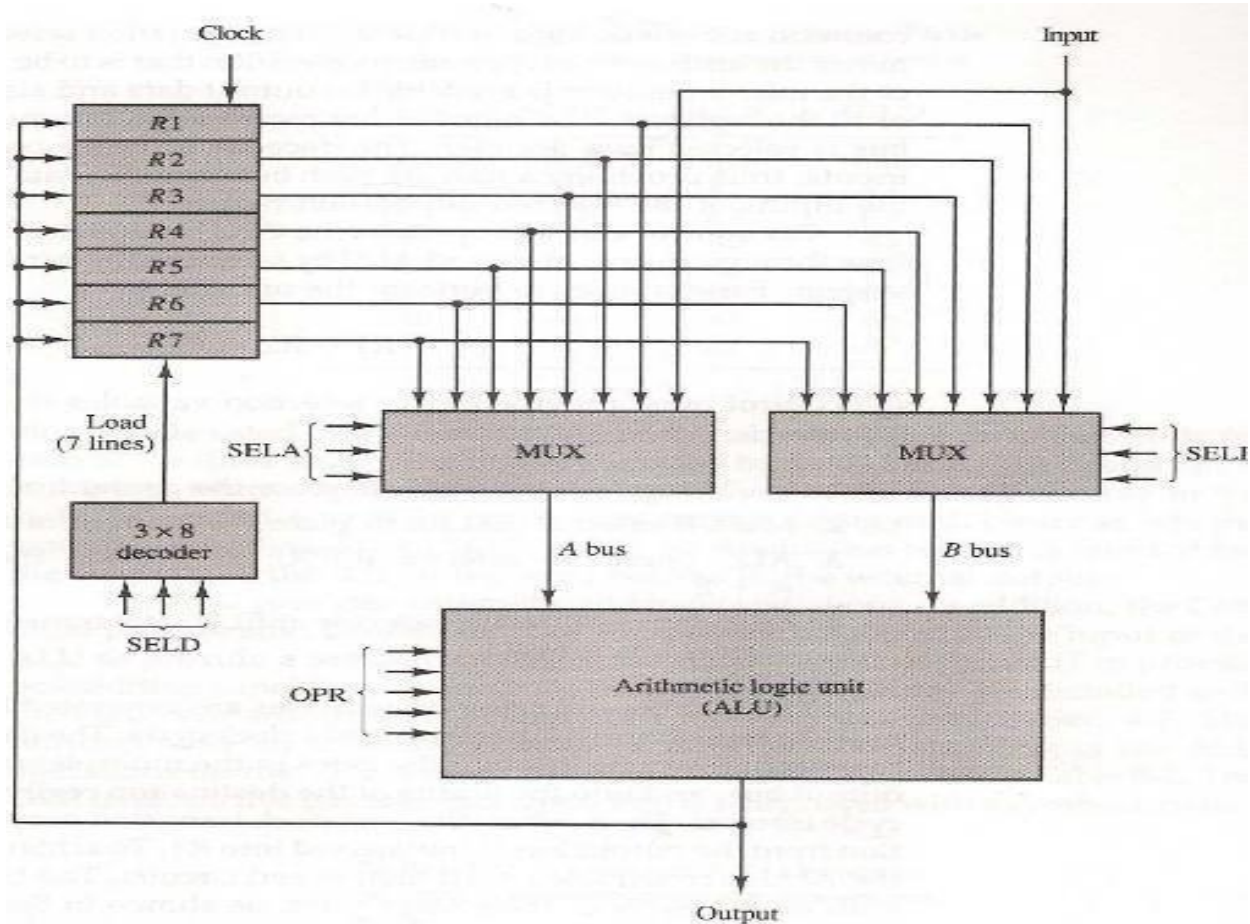


Figure (2): Bus organization for CPU registers

The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus

A. 2. MUX B selector (SELB): to place the content of R3 into bus B.

3. ALU operation selector (OPR): to provide the arithmetic addition $A + B$.

4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.

Control word

There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in Figure (3). It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

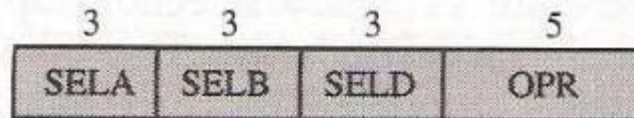


Figure (3): Control word

The encoding of the register selections is specified in the Table (a). The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code.

When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

are available in the external output.

Table (1): Encoding of Register Selection Fields

The ALU provides arithmetic and logic operations. The encoding of the ALU Operations are specified in the Table (b). The OPR field has five bits and each operation is designated with a symbolic name.

OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

Table (2): Encoding of ALU operations

Examples of Microoperations:

A control word of 14 bits is needed to specify a microoperation in the CPU. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement

$$R1 \leftarrow R2 - R3$$

specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding listed in Tables (1) and (2). The binary control word for the subtract microoperation is 010 011 001 00101 and is obtained as follows:

Field:	SELA	SELB	SELD	OPR
Symbol:	R2	R3	R1	SUB
Control word:	010	011	001	00101

The control word for this microoperation and a few others are listed in the below table.

Microoperation	Symbolic Designation				Control Word
	SELA	SELB	SELD	OPR	
$R1 \leftarrow R2 - R3$	R2	R3	R1	SUB	010 011 001 00101
$R4 \leftarrow R4 \vee R5$	R4	R5	R4	OR	100 101 100 01010
$R6 \leftarrow R6 + 1$	R6	—	R6	INCA	110 000 110 00001
$R7 \leftarrow R1$	R1	—	R7	TSFA	001 000 111 00000
$\text{Output} \leftarrow R2$	R2	—	None	TSFA	010 000 000 00000
$\text{Output} \leftarrow \text{Input}$	Input	—	None	TSFA	000 000 000 00000
$R4 \leftarrow \text{shl } R4$	R4	—	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

Table (3): Encoding of ALU operations

STACK Organization:

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

- ❖ The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.

The two operations of a stack are the insertion and deletion of items.

1. Push or push-down (insertion operation)
2. Pop or pop-up (deletion operation)

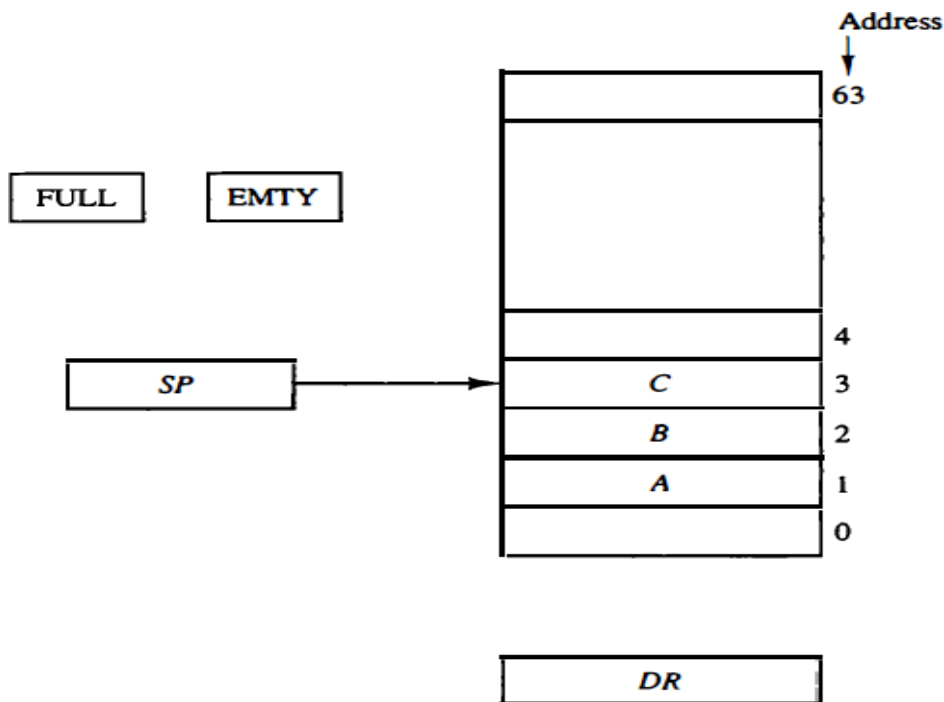


Figure (4): the organization of a 64-word register stack.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure (3) shows the organization of a 64-word register stack. The stack pointer register *SP* contains a binary number whose value is equal to the address of the word that is currently on top of the stack.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. Since *SP* has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since $111111 + 1 = 100000$ in binary, but *SP* can accommodate only the six least significant bits.

Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register *FULL* is set to 1 when the stack is full, and the one-bit register *EMPTY* is set to 1 when the stack is empty of items. *DR* is the data register that holds the binary data to be written into or read out of the stack.

Push:

Initially, *SP* is cleared to 0, *EMPTY* is set to 1, and *FULL* is cleared to 0, so that *SP* points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if *FULL* = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations;

$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack
If ($SP = 0$) then ($FULL \leftarrow 1$)	Check if stack is full
$EMPTY \leftarrow 0$	Mark the stack not empty

Pop:

A new item is deleted from the stack if the stack is not empty (if *EMPTY* = 0). The pop operation consists of the following sequence of microoperations:

$DR \leftarrow M[SP]$	Read item from the top of stack
-----------------------	---------------------------------

$SP \leftarrow SP - 1$ Decrement stack pointer
if $(SP = 0)$ then $(EMPTY \leftarrow 1)$ Check if stack is
empty $FULL \leftarrow 0$ Mark the stack not full

Memory Stack

A stack can exist as a stand-alone unit as in Figure (3) or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure (4) shows a portion of computer memory partitioned into three segments: program, data, and stack.

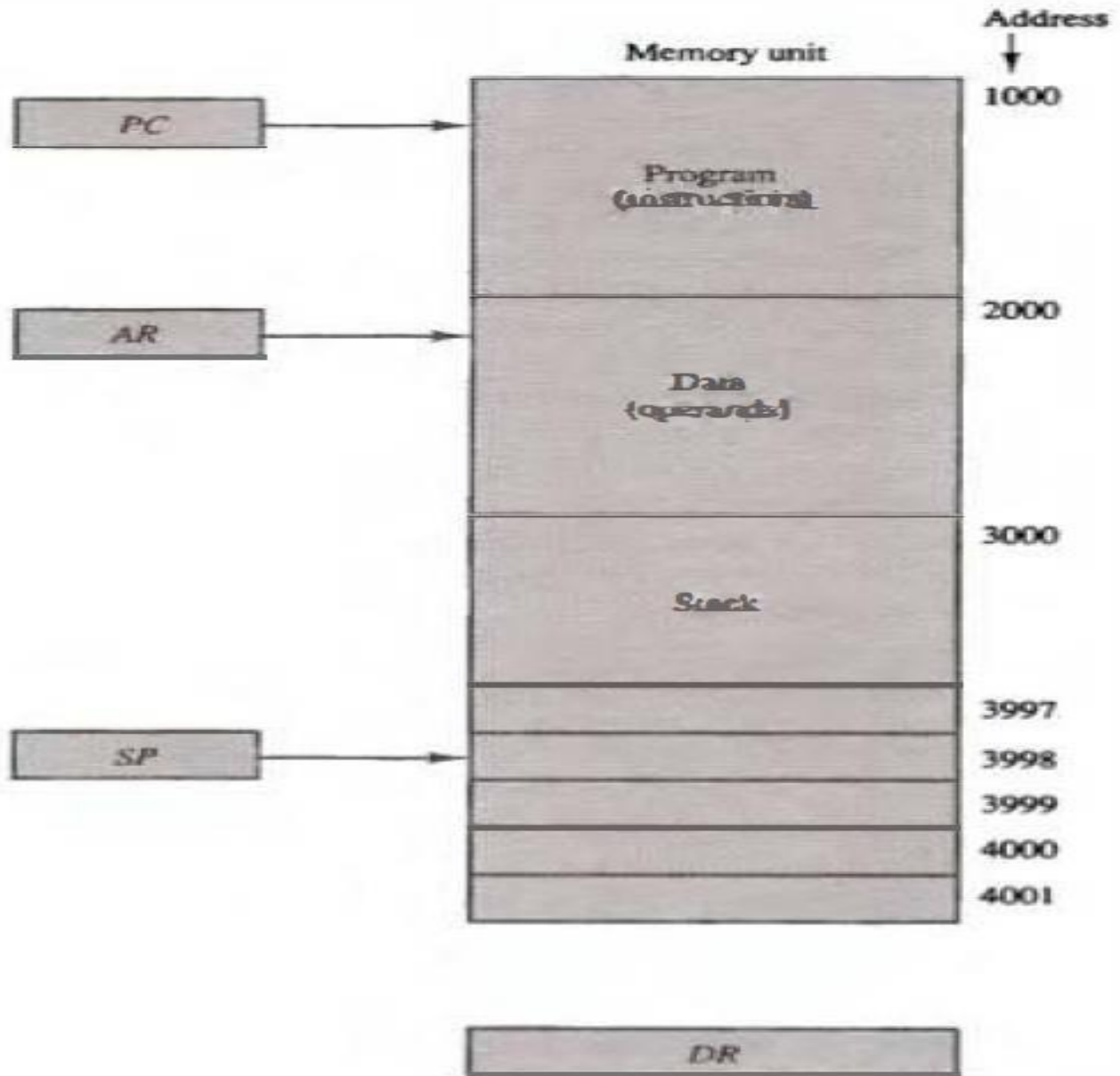


Figure (5): Computer memory with program, data and stack segments

Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expressions imposes difficulties when evaluated by a computer.

$$A * B + C * D \quad \square \text{ infix notation}$$

The Polish mathematician Lukasiewicz showed that arithmetic expressions can be represented in

prefix notation. This representation, often referred to as Polish notation, places the operator

before the operands. The postfix notation, referred to as reverse Polish notation (RPN), places the operator after the operands. The following examples demonstrate the three representations:

$A + B$	Infix notation
$+ AB$	Prefix or Polish notation
$AB +$	Postfix or reverse Polish notation

The reverse Polish notation is in a form suitable for stack manipulation. The expression

$$A * B + C * D$$

is written in reverse Polish notation as

$$AB * CD * +$$

Conversion to Reverse Polish Notation

The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

- This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only $+$, $-$, $*$, $/$, $\%$ operators. The precedence of these operators can be given as follows:

Higher priority $*$, $/$, $\%$

Lower priority $+$, $-$

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression $A + B * C$, then first $B * C$ will be done and the result will be added to A . But the same expression if written as, $(A + B) * C$, will evaluate $A + B$ first and then the result will be multiplied with C . Consider the expression

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

The converted expression is

$$AB + CDE + * F + *$$

Step 1: Add ")" to the end of the infix expression

Step 2: Push "(" on to the stack

Step 3: Repeat until each character in the infix notation is

scanned IF a "(" is encountered, push it onto the stack

IF an operand (whether a digit or a character) is encountered, add it to the postfix expression.

IF a ")" is encountered, then

- Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
- Discard the "(" . That is, remove the (from stack and do not add it to the postfix expression

IF an operator 'O' is encountered, then

- Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than 'O'
- Push the operator 'O' to the stack

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Example 1: $A*B+C*D$, first add "(" to the given expression i.e., $A*B+C*D)$ and also push "(" onto the stack.

Infix Character Scanned	Stack	Postfix Expression
	(
A	(A
*	(*	A
B	(*	AB
+	(+	AB*
C	(+	AB*C
*	(*	AB*C
D	(*	AB*CD
)	(*	AB*CD*+

Example 2: $(A + B) * (C * (D + E) + F)$

□ First add "(" to the given expression i.e., $(A + B) * (C * (D + E) + F)$ and also push "(" onto the stack.

Infix Character Scanned	Stack	Postfix Expression
	(
(((
A	((A
+	((+	A
B	((+	AB
)	(AB+
*	(*	AB+
((*	AB+
C	(*	AB+C
*	(*	AB+C
((*	AB+C
	(*	AB+C
D	(*	AB+CD
	(*	AB+CD

+	(**(+	AB+CD
E	(**(+	AB+CDE

)	(*(*	AB+CDE+
+	(*(+	AB+CDE+*
F	(*(+	AB+CDE+*F
)	(*	AB+CDE+*F+
)		AB+CDE+*F+ *

Evaluation of Arithmetic Expressions

- (1) Push the operands into the stack until an operator is reached
- (2) Pop the top two operands from the stack, compute the result and also push the result back into the stack.
- (3) Continue this process until there are no more operators in the RPN and the final result is in the stack.

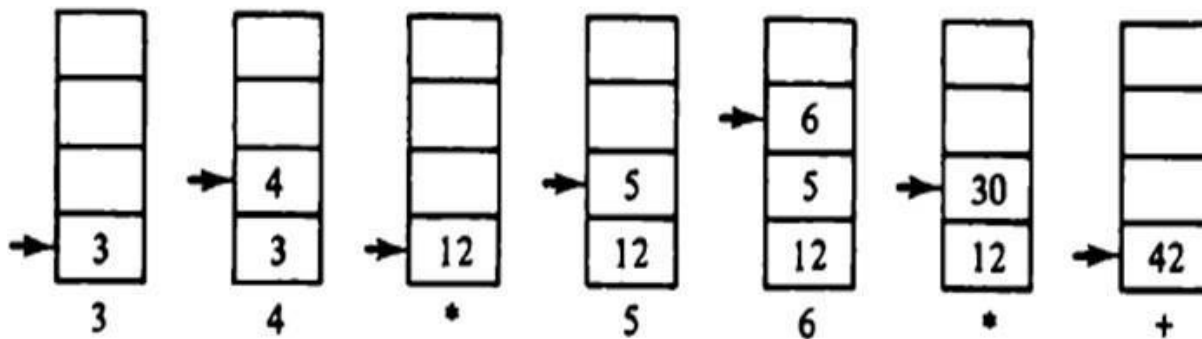
The following numerical example may clarify this procedure. Consider the arithmetic expression

$$(3 * 4) + (5 * 6)$$

In reverse Polish notation, it is expressed as

$$34 * 56 * +$$

Stack operations to evaluate $3 * 4 + 5 * 6$.



Instruction Formats:

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

➤ The operation code field of an instruction is a group of bits that define various

processor operations, such as add, subtract, complement, and shift.

- The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address fields in an instruction.

Operations specified by computer instructions are executed on some data stored in memory or processor registers. Operands residing in memory are specified by their memory address. Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2^k registers in the CPU.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

1. An accumulator-type organization:

All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as:

ADD X

where X is the address of the operand. The ADD instruction in this case results in the operation $AC \leftarrow AC + M[X]$. AC is the accumulator register and $M[X]$ symbolizes the memory word located at address X .

2. A general register type of organization:

The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as

ADD R1, R2, R3

to denote the operation $R1 \leftarrow R2 + R3$. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

ADD R1, R2

would denote the operation $R1 \leftarrow R1 + R2$. Only register addresses for $R1$ and $R2$ need be specified in this instruction.

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

ADD R1, X

would specify the operation $R1 \leftarrow R1 + M[X]$. It has two address fields, one for register $R1$ and the other for the memory address X .

3. A stack organization:

Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction

PUSH X

will push the word at address X to the top of the stack. The stack pointer is updated

automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction

ADD

in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

□ To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

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

using zero, one, two, or three address instructions. We will use the symbols ADD, SUB, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

Three-Address Instructions:

```

ADD   R1, A, B   R1 ← M[A] + M[B]
ADD   R2, C, D   R2 ← M[C] + M[D]
MUL   X, R1, R2  M[X] ← R1 * R2

```

Two-Address Instructions:

```

MOV   R1, A     R1 ← M[A]
ADD   R1, B     R1 ← R1 + M[B]
MOV   R2, C     R2 ← M[C]
ADD   R2, D     R2 ← R2 + M[D]
MUL   R1, R2    R1 ← R1 * R2
MOV   X, R1     M[X] ← R1

```

One-Address Instructions:

```

LOAD   A       AC ← M[A]
ADD    B       AC ← AC + M[B]
STORE  T       M[T] ← AC
LOAD   C       AC ← M[C]
ADD    D       AC ← AC + M[D]
MUL    T       AC ← AC * M[T]
STORE  X       M[X] ← AC

```

Zero-Address Instructions:

```

PUSH   A       TOS ← A
PUSH   B       TOS ← B
ADD            TOS ← (A + B)
PUSH   C       TOS ← C
PUSH   D       TOS ← D
ADD            TOS ← (C + D)
MUL            TOS ← (C + D) * (A + B)
POP    X       M[X] ← TOS

```

RISC Instructions:

```

LOAD   R1, A   R1 ← M[A]
LOAD   R2, B   R2 ← M[B]
LOAD   R3, C   R3 ← M[C]
LOAD   R4, D   R4 ← M[D]
ADD    R1, R1, R2  R1 ← R1 + R2
ADD    R3, R3, R2  R3 ← R3 + R2
MUL    R1, R1, R3  R1 ← R1 * R3
STORE  X, R1     M[X] ← R1

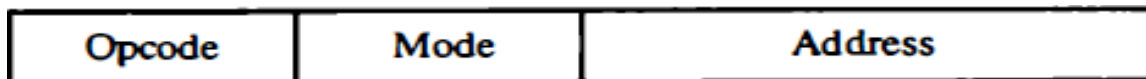
```

Addressing Modes:

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

In simple terms, Addressing mode is the way in which the location of an operand can be specified in an instruction. It generates an effective address (the actual address of the operand).

Instruction format with mode field



Types of Addressing Modes:

1. Implied Mode
2. Immediate Mode
3. Register Mode
4. Register Indirect Mode:
5. Autoincrement or Autodecrement Mode
6. Direct Address Mode
7. Indirect Address Mode
8. Relative Address Mode
9. Indexed Addressing Mode
10. Base Register Addressing Mode

There are two modes that need no address field at all. These are the implied and immediate modes.

1. Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction.

For example, the instruction "complement accumulator (CMA)" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions. Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

2. Immediate Mode: In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the

operation specified in the instruction.

EX: LDAC #34H

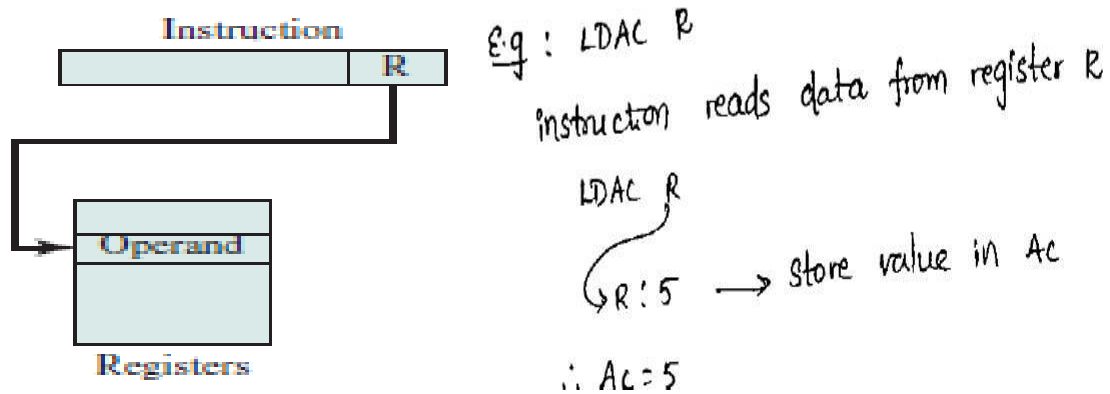
LDAC loads data from memory to



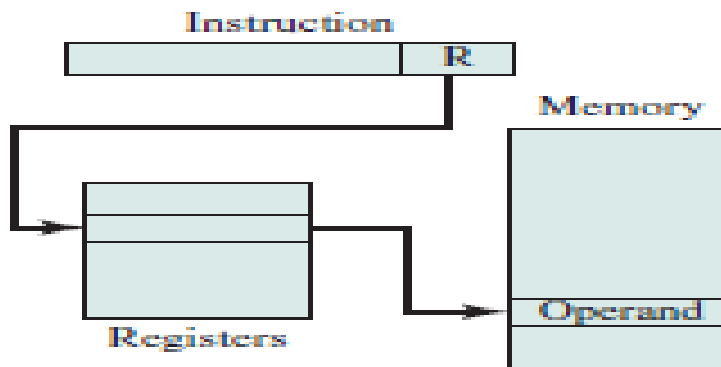
accumulator. Therefore, $AC = 00110100$.

When the address field specifies a processor register, the instruction is said to be in the register mode.

3. Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction.



4. Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself.



EX: LDAC (R1)

If R1 contains the address of an operand in the memory, for example: address of an operand is 2000 which contains a value 350. Result: 350 is stored in the AC.

5. Autoincrement Mode: This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of the register are automatically incremented

Eg: LDAC (R)
 instruction reads address from register R
 instruction reads data from memory location
 R: 5
 5: 10 → store in AC

AC = 10

R: 5 + 1 = 6

to the next value.

6. Autodecrement Mode

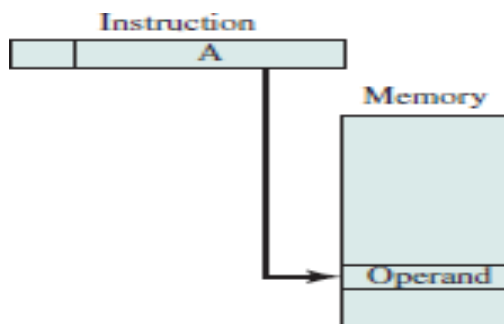
The effective address of the operand is the contents of a register specified in the instruction. Before accessing the operand, the contents of this register are automatically decremented and then the value is accessed.

E.g: LDAC (R)
Instruction reads address from register R
R: 6
R: 6-1=5 decrement value in register R.
Instruction reads data from memory location
5:10

∴ AC = 10

Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.

7. Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies

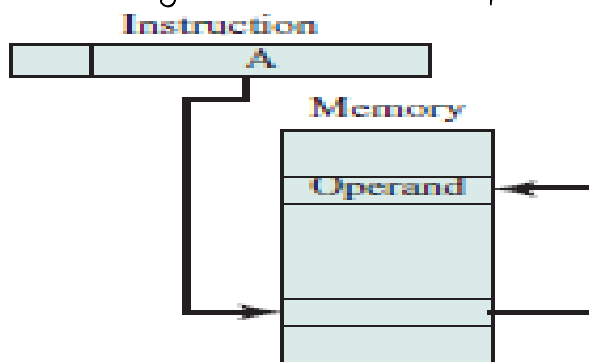


the actual branch address.

Ex: LDAC 5000

This instruction reads the operand from the Memory location 5000. If the memory location 5000 contains a value 250, then it will be stored in AC.

8. Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective



address.

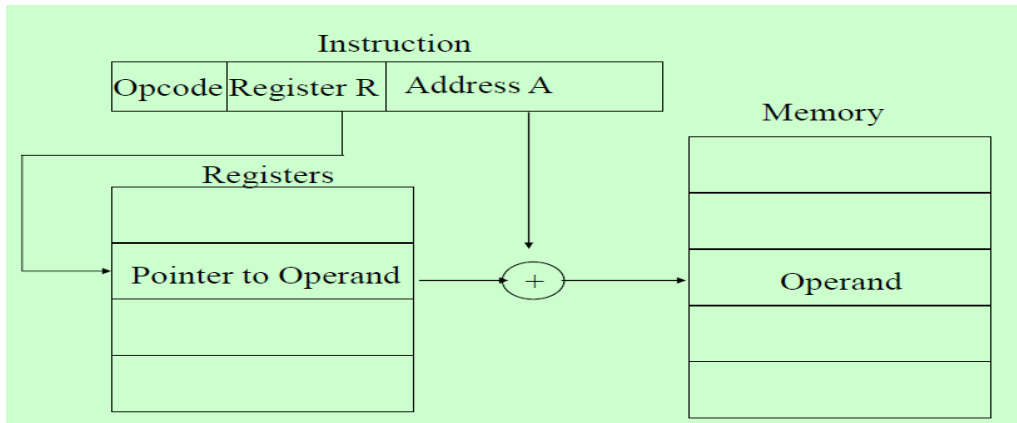
EX: ADD (A), R1

- If A is address of EA. For example: address of A is 1000 which contains 3000, 3000 is an address of an operand (EA).
- This instruction reads an operand from the location address 3000 and adds its contents to R1.

A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation:

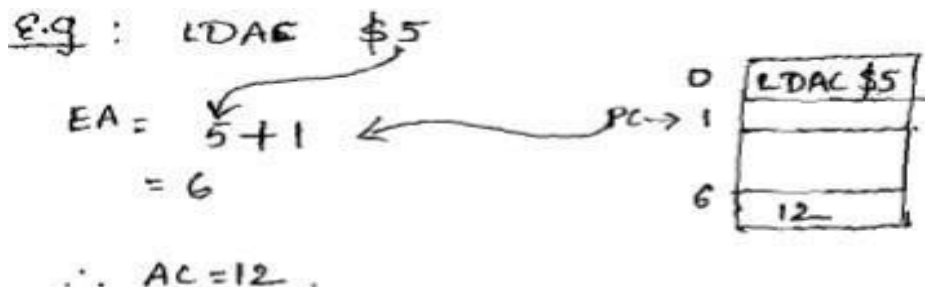
- effective address = address part of instruction + content of CPU register

The CPU register used in the computation may be the program counter, an index register, or a base register. In either case we have a different addressing mode which is used for a different application.



9. Relative Address Mode:

In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

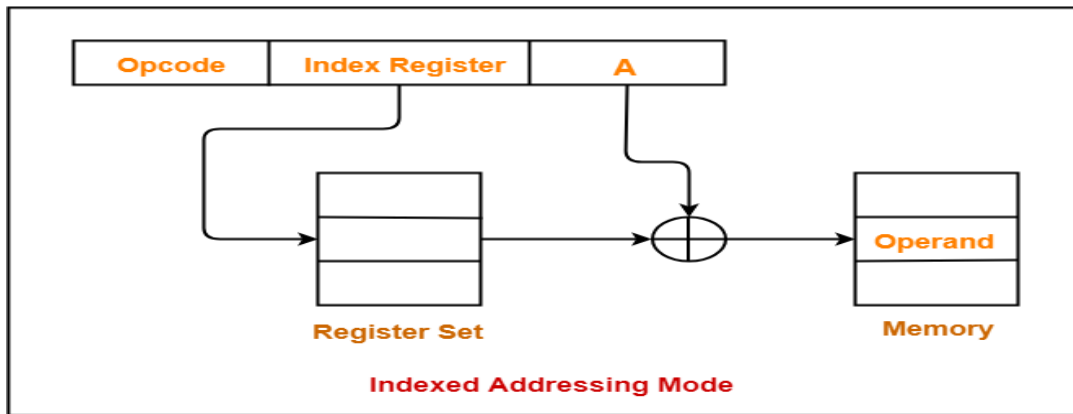


EX:

PC = address of next instruction, i.e., 1. The address given in the instruction is 5. Then EA = 5 + 1 = 6 which contains a value 12. Finally AC contains 12.

10. Indexed Addressing Mode:

In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.



Ex: LDAC A(XR)

Assume XR=100, A=500

This instruction reads the operand from the effective address (600) i.e., EA = XR contents (index register) + 500

$$= 100 + 500 = 600$$

If memory location at 600 contains a value 55 (assume), This 55 will be stored in AC.

11. Base Register Addressing Mode:

In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.

Ex: LDAC A(R)

Assume R=1000,

A=50

This instruction reads the operand from the effective address (1050) i.e.,

EA = R contents (Base register) + 50

$$= 1000 + 50 = 1050$$

If memory location at 1050 contains a value 255 (assume), this 255 will be stored in AC.

Numerical Example:

Address	Memory
200	Load to AC Mode
201	Address = 500
202	Next instruction
399	450
400	700
500	800
600	900
702	325
800	300

Addressing Mode	Effective Address	Content of AC
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

Table (4): Tabular list of some addressing modes of numerical example.

Data Transfer and Manipulation:

Computers provide an extensive set of instructions to give the user the flexibility to carry out various computational tasks. The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields.

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer instructions cause transfer of data from one location to another without changing the binary information content.

Data manipulation instructions are those that perform arithmetic, logic, and shift operations.

Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer.

The instruction set of a particular computer determines the register transfer operations and control decisions that are available to the user.

1. Data transfer instructions

Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. Table (5) gives a list of eight data transfer instructions used in many computers.

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Table (5): Data Transfer Instructions

- ❖ The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- ❖ The store instruction designates a transfer from a processor register into memory.
- ❖ The move instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words.
- ❖ The exchange instruction swaps information between two registers or a register and a memory word.
- ❖ The input and output instructions transfer data among processor registers and input or output terminals.
- ❖ The push and pop instructions transfer data between processor registers and a memory stack.

2. Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

- i. Arithmetic instructions
- ii. Logical and bit manipulation instructions
- iii. Shift instructions

i. Arithmetic instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

Table (6): Arithmetic Instructions

- ❖ A special carry flip-flop is used to store the carry from an operation. The instruction "add with carry" performs the addition on two operands plus the value of the carry from the previous computation.
- ❖ Similarly, the "subtract with borrow" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation.
- ❖ The negate instruction forms the 2's complement of a number.

ii. Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The logical instructions consider each bit of the operand

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

separately and treat it as a Boolean variable.

Table (7): Logic and Bit Manipulation Instructions

iii. Shift Instructions

Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations. In

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

either case the shift may be to the right or to the left.

Table (8): Shift Instructions

The rotate through carry instruction treats a carry bit as an extension of the register

whose word is being rotated. Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry and at the same time, and shifts the entire register to the left.

A possible instruction code format of a shift instruction may include five fields as follows:

OP REG TYPE RL COUNT

OP- operation code field

REG- a register address that specifies the location of the

operand TYPE- a 2-bit field specifying the four different types

of shifts RL- a 1-bit field specifying a shift right or left

COUNT- a k-bit field specifying up to $2^k - 1$ shifts

Program Control:

After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence.

On the other hand, a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data-processing operations.

The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it provides control over the flow of program execution and a capability for branching to different program segments.

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

Table (9): Program Control Instructions

Status Bit Conditions:

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called condition-code bits or flag bits. Figure (6) shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. It is cleared to 0

otherwise. In other words, $Z = 1$ if the output is zero and $Z = 0$ if the output is not zero.

4. Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, $V = 1$ if the output is greater than $+127$ or less than -128 .

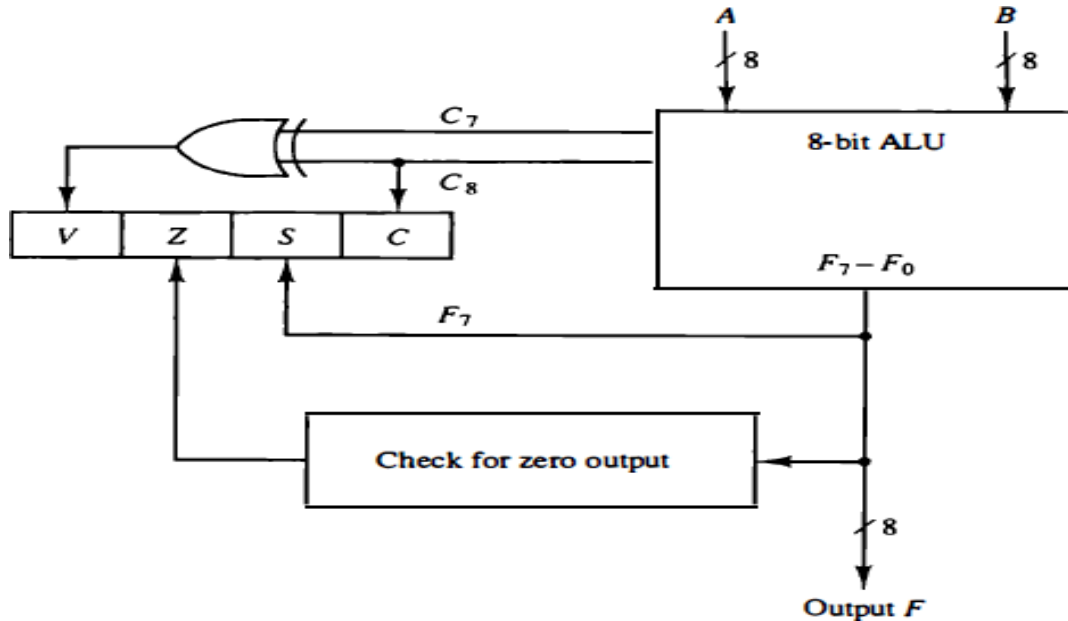


Figure (6): Status register bits

Conditional Branch Instructions:

To change the flow of execution in the program we use some kind of branching instructions which are depending on the some conditions result.

Each mnemonic is constructed with the letter **B** (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter **N** (for no) is inserted to define the 0 state. Thus **BC** is Branch on Carry, and **BNC** is Branch on No Carry.

If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions.

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
<i>Unsigned compare conditions (A - B)</i>		
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
<i>Signed compare conditions (A - B)</i>		
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Table (10): Conditional Branch Instructions

Example: Consider an 8-bit ALU as shown in Figure (6). The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between +127 and -128. Let $A = 11110000$ and $B = 00010100$. To perform $A - B$, the ALU takes the 2's complement of B and adds it to A .

$$\begin{array}{r}
 A: 11110000 \\
 \bar{B} + 1: +11101100 \\
 \hline
 A - B: 11011100 \quad C = 1 \quad S = 1 \quad V = 0 \quad Z = 0
 \end{array}$$

The compare instruction updates the status bits as shown. $C = 1$ because there is a carry out of the last stage. $S = 1$ because the leftmost bit is 1. $V = 0$ because the last two carries are both equal to 1, and $Z = 0$ because the result is not equal to 0.

Subroutine Call and Return:

A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of

instructions. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save address.

A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two operations:

- (1) The address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return, and
- (2) Control is transferred to the beginning of the subroutine. The last instruction of every subroutine, commonly called return from subroutine, transfers the return address from the temporary location into the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

The most efficient way is to store the return address in a memory stack. The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack. The return from subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter. In this way, the return is always to the program that last called a subroutine. A subroutine call is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow \text{effective address}$	Transfer control to the subroutine

If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

By using a subroutine stack, all return addresses are automatically stored by the hardware in one unit. The programmer does not have to be concerned or remember where the return address was stored.

A recursive subroutine is a subroutine that calls itself.

Program interrupt:

Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations:

- (1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later);
- (2) The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and
- (3) An interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter
2. The content of all processor registers
3. The content of certain status conditions

The collection of all status bit conditions in the CPU is sometimes called a program status word or PSW. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU

The last instruction in the service program is a return from interrupt instruction. When this instruction is executed, the stack is popped to retrieve the old PSW and the return address. The PSW is transferred to the status register and the return address to the program counter. Thus the CPU state is restored and the original program can continue executing.

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input-output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are I/O device requesting transfer of data, I/O device finished transfer of data etc.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

□ External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most

common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode.

Reduced Instruction Set Computer (RISC):

- An important aspect of computer architecture is the design of the instruction set for the processor.
- Early computers had small and simple instruction sets, forced mainly by the need to minimize the hardware used to implement them.
- As digital hardware became cheaper with the advent of integrated circuits, computer instructions tended to increase both in number and complexity. Many computers have instruction sets that include more than 100 and sometimes even more than 200 instructions. These computers also employ a variety of data types and a large number of addressing modes.

A computer with a large number of instructions is classified as a *complex instruction set computer*, abbreviated CISC.

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a *reduced instruction set computer* or RISC.

CISC Characteristics

The major characteristics of CISC architecture are:

1. A large number of instructions-typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently
3. A large variety of addressing modes-typically from 5 to 20 different modes
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of RISC architecture are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution
7. Hardwired rather than microprogrammed

control. Other characteristics attributed to RISC architecture

are:

1. A relatively large number of registers in the processor unit
2. Use of overlapped register windows to speed-up procedure call and return
3. Efficient instruction pipeline
4. Compiler support for efficient translation of high-level language programs into machine language .

Overlapped Register Windows

Procedure call and return occurs quite often in high-level programming languages. When translated into machine language, a procedure call produces a sequence of instructions that save register values, pass parameters needed for the procedure, and then calls a subroutine to execute the body of the procedure. After a procedure return, the program restores the old register values, passes results to the calling program, and returns from the subroutine. Saving and restoring registers and passing of parameters and results involve time consuming operations.

A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values.

Berkeley RISC I

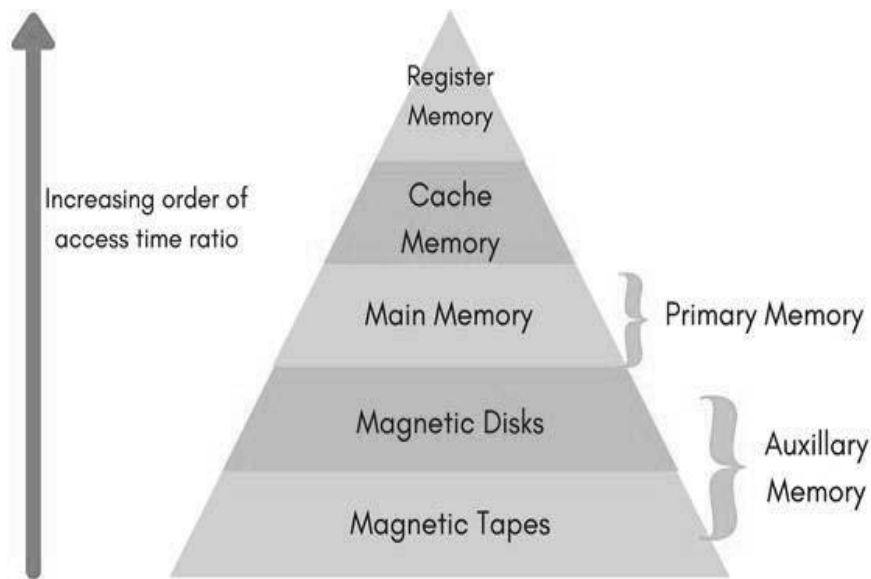
- One of the first projects intended to show the advantages of RISC architecture was conducted at the University of California, Berkeley.
- The Berkeley RISC I is a 32-bit integrated circuit CPU. It supports 32-bit addresses and either 8-, 16-, or 32-bit data. It has a 32-bit instruction format and a total of 31 instructions.
- There are three basic addressing modes: register addressing, immediate operand, and relative to PC addressing for branch instructions. It has a register file of 138 registers arranged into 10 global registers and 8 windows of 32 registers in each.

UNIT - 5

Memory Organization: Memory Hierarchy, Main Memory -RAM And ROM Chips, Memory Address map, Auxiliary memory-magnetic Disks, Magnetic tapes, Associate Memory,-Hardware Organization, Match Logic, Cache Memory -Associative Mapping , Direct Mapping, Set associative mapping ,Writing in to cache and cache Initialization , Cache Coherence ,Virtual memory-Address Space and memory Space ,Address mapping using pages, Associative memory

page table ,page Replacement .

Memory Hierarchy



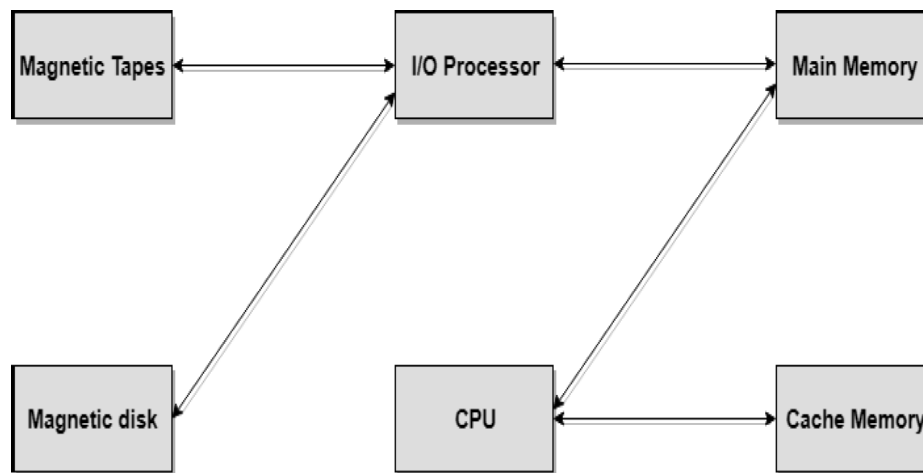
The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.

Auxillary memory access time is generally 1000 times that of the main memory, hence it is at the bottom of the hierarchy.

The main memory occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through input/output processor (I/O).

When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use.

The cache memory is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about 1 to 7~10



Memory Access Methods

Each memory type, is a collection of numerous memory locations. To access data from any memory, first it must be located and then the data is read from the memory location. Following are the methods to access information from memory locations:

1. **Random Access:** Main memories are random access memories, in which each memory location has a unique address. Using this unique address any memory location can be reached in the same amount of time in any order.
2. **Sequential Access:** This method allows memory access in a sequence or in order.
3. **Direct Access:** In this mode, information is stored in tracks, with each track having a separate read/write head.

Main Memory

The memory unit that communicates directly within the CPU, Auxiliary memory and Cache memory, is called main memory. It is the central storage unit of the computer system. It is a large and fast memory used to store data during computer operations. Main memory is made up of RAM and ROM, with RAM integrated circuit chips holding the major share.

- **RAM: Random Access Memory**
 - **DRAM: Dynamic RAM,** is made of capacitors and transistors, and must be refreshed every 10~100 ms. It is slower and cheaper than SRAM.
 - **SRAM: Static RAM,** has a six transistor circuit in each cell and retains data,

until powered off.

- NVRAM: Non-Volatile RAM, retains its data, even when turned off. Example: Flashmemory.
- ROM: Read Only Memory, is non-volatile and is more like a permanent storage for information. It also stores the bootstrap loader program, to load and start the operating system when computer is turned on. PROM (Programmable ROM), EPROM (Erasable PROM) and EEPROM (Electrically Erasable PROM) are some commonly used ROMs.

Memory Address map:

- The addressing of memory can establish by means of a table that specifies the memory address assigned to each chip.
- The table, called a memory address map, is a pictorial representation of assigned address space for each chip in the system, shown in the table.
- To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM.
 - The RAM and ROM chips to be used specified in figures.

Component	Hexa address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000 - 007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080 - 00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100 - 017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180 - 01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200 - 03FF	1	x	x	x	x	x	x	x	x	x

- The component column specifies whether a RAM or a ROM chip used.
- Moreover, The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip.
- The address bus lines listed in the third column.
- Although there 16 lines in the address bus, the table shows only 10 lines because the other 6 not used in this example and assumed to be zero.
- The small x's under the address bus lines designate those lines that must connect to the address inputs in each chip.
- Moreover, The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines.
- The x's always assigned to the low-order bus lines: lines 1 through 7 for the RAM. And lines 1 through 9 for the ROM.
- It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example, we choose bus lines 8 and 9 to represent four distinct binary combinations.
- Also, The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to $2^9 = 512$ bytes.

- The distinction between a RAM and ROM address done with another bus line. Here we choose line 10 for this purpose.
- When line 10 0, the CPU selects a RAM, and when this line equal to 1, it selects the ROM.

Auxiliary Memory

Devices that provide backup storage are called auxiliary memory. For example: Magnetic disks and tapes are commonly used auxiliary devices. Other devices used as auxiliary memory are magnetic drums, magnetic bubble memory and optical disks.

It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

Cache Memory

The data or contents of the main memory that are used again and again by CPU, are stored in the cache memory so that we can easily access that data in shorter time.

Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory then the CPU moves onto the main memory. It also transfers block of recent data into the cache and keeps on deleting the old data in cache to accommodate the new one.

Hit Ratio

The performance of cache memory is measured in terms of a quantity called hit ratio. When the CPU refers to memory and finds the word in cache it is said to produce a hit. If the word is not found in cache, it is in main memory then it counts as a miss.

The ratio of the number of hits to the total CPU references to memory is called hit

ratio. $\text{Hit Ratio} = \text{Hit} / (\text{Hit} + \text{Miss})$

Associative Memory

It is also known as content addressable memory (CAM). It is a memory chip in which each bit position can be compared. In this the content is compared in each bit cell which allows very fast table lookup. Since the entire chip can be compared, contents are randomly stored without considering addressing scheme. These chips have less storage capacity than regular memory chips.

Memory Mapping and Concept of Virtual Memory

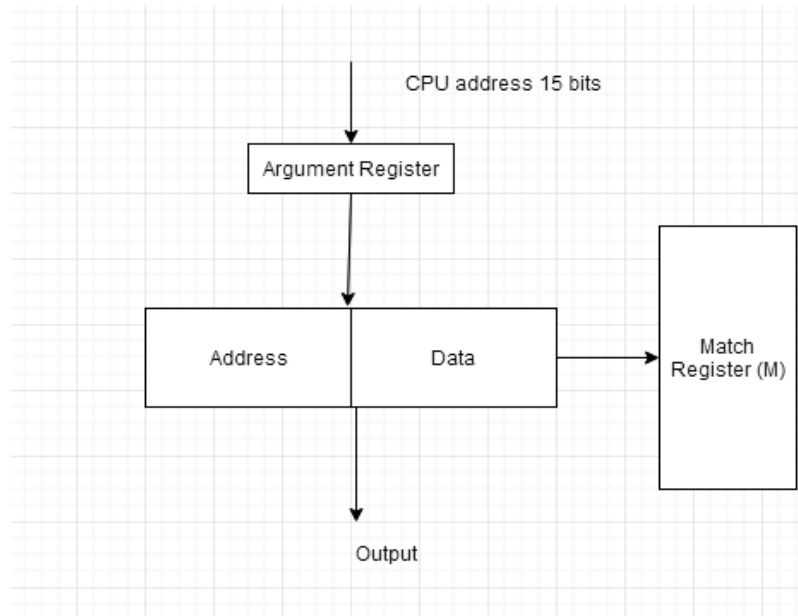
The transformation of data from main memory to cache memory is called mapping. There are 3 main types of mapping:

- Associative Mapping
- Direct Mapping
- Set Associative Mapping

Associative Mapping

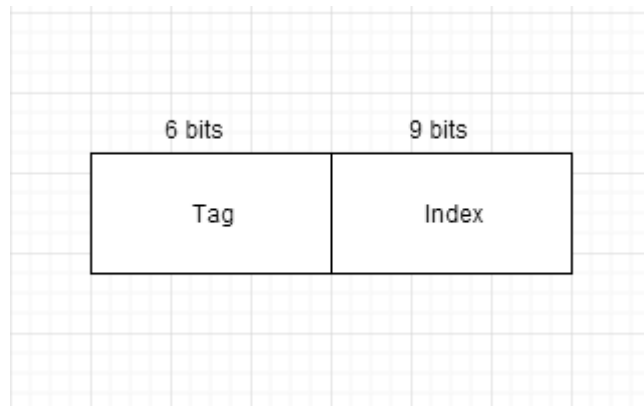
The associative memory stores both address and data. The address value of 15 bits is 5

digit octal numbers and data is of 12 bits word in 4 digit octal number. A CPU address of 15 bits is placed in argument register and the associative memory is searched for matching address.



Direct Mapping

The CPU address of 15 bits is divided into 2 fields. In this the 9 least significant bits constitute the index field and the remaining 6 bits constitute the tag field. The number of bits in index field is equal to the number of address bits required to access cache memory.



Set Associative Mapping

The disadvantage of direct mapping is that two words with same index address can't reside in cache memory at the same time. This problem can be overcome by set associative mapping.

In this we can store two or more words of memory under the same index address. Each data word is stored together with its tag and this forms a set.

Tag	Data	Address

Replacement Algorithms

Data is continuously replaced with new data in the cache memory using replacement algorithms.

Following are the 2 replacement algorithms used:

- FIFO - First in First out. Oldest item is replaced with the latest item.
- LRU - Least Recently Used. Item which is least recently used by CPU is removed.

Writing in to cache and cache Initialization:

The benefit of write-through to main memory is that it simplifies the design of the computersystem. With write-through, the main memory always has an up-to-date copy of the line. So when aread is done, main memory can always reply with the requested data.

If write-back is used, sometimes the up-to-date data is in a processor cache, and sometimes it is in main memory. If the data is in a processor cache, then that processor must stop main memory from replying to the read request, because the main memory might have a stale copy of the data. This is more complicated than write-through.

Also, write-through can simplify the cache coherency protocol because it doesn't need the *Modify* state. The *Modify* state records that the cache must write back the cache line before it invalidates or evicts the line. In write-through a cache line can always be invalidated without writingback since memory already has an up-to-date copy of the line.

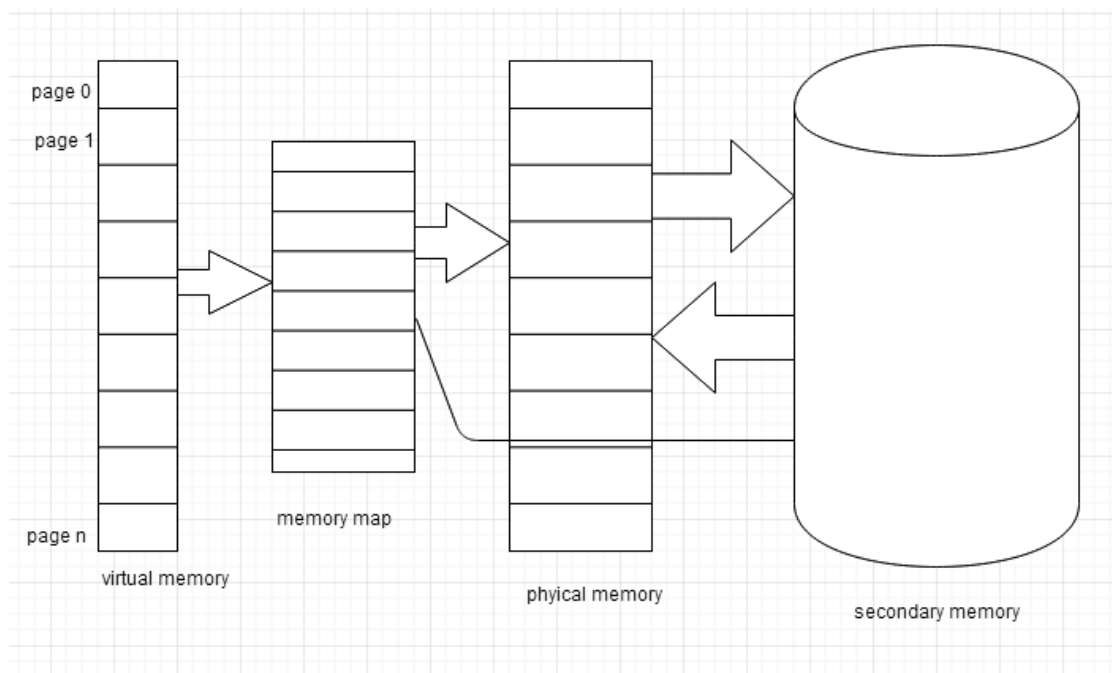
Cache Coherence:

In a shared memory multiprocessor with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion.

Virtual Memory

Virtual memory is the separation of logical memory from physical memory. This separation provides large virtual memory for programmers when only small physical memory is available.

Virtual memory is used to give programmers the illusion that they have a very large memory even though the computer has a small main memory. It makes the task of programming easier because the programmer no longer needs to worry about the amount of physical memory available.



Address mapping using pages:

- The table implementation of the address mapping is simplified if the information in the address space. And the memory space is each divided into groups of fixed size.
- Moreover, The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each.
- The term page refers to groups of address space of the same size.
- Also, Consider a computer with an address space of 8K and a memory space of 4K.
- If we split each into groups of 1K words we obtain eight pages and four blocks as shown in the figure.
- At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

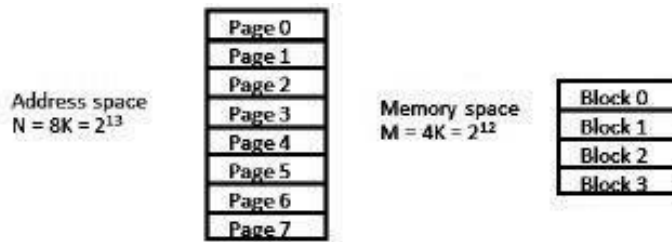
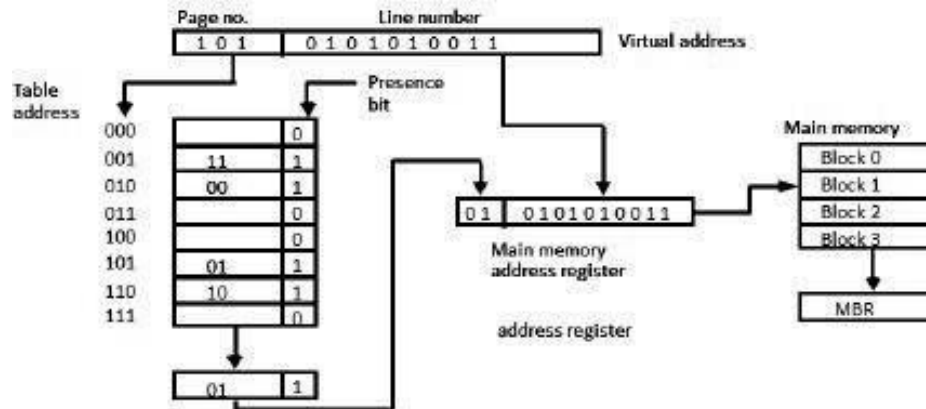


Figure Address and Memory space split into group of 1K words



Associative memory page table:

The implementation of the page table is vital to the efficiency of the virtual memory technique, for each memory reference must also include a reference to the page table. The fastest solution is a set of dedicated registers to hold the page table but this method is impractical for large page tables because of the expense. But keeping the page table in main memory could cause intolerable delays because even only one memory access for the page table involves a slowdown of 100 percent and large page tables can require more than one memory access. The solution is to augment the page table with special high-speed memory made up of associative registers or translation look aside buffers (TLBs) which are called ASSOCIATIVE MEMORY.

Page replacement

The advantage of virtual memory is that processes can be using more memory than exists in the machine; when memory is accessed that is not present (a page fault), it must be paged in (sometimes referred to as being "swapped in", although some people reserve "swapped in" to refer to bringing in an entire address space).

Swapping in pages is very expensive (it requires using the disk), so we'd like to avoid page faults as much as possible. The algorithm that we use to choose which pages to evict to make space for the new page can have a large impact on the number of page faults that occur.

UNIT - 5

Input-Output Organization: Peripheral Devices, Input-Output Interface, Asynchronous data transfer Modes of Transfer, Priority Interrupt Direct memory Access, Input -Output Processor (IOP) Pipeline And Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, Dependencies, Vector Processing.

Introduction:

The I/O subsystem of a computer provides an efficient mode of communication between the central system and the outside environment. It handles all the input-output operations of the computer system.

Peripheral Devices

Input or output devices that are connected to computer are called peripheral devices. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be the part of computer system. These devices are also called peripherals.

For example: *Keyboards, display units and printers* are common peripheral devices. There are three types of peripherals:

1. Input peripherals : Allows user input, from the outside world to the computer. Example: Keyboard, Mouse etc.
2. Output peripherals: Allows information output, from the computer to the outside world. Example: Printer, Monitor etc
3. Input-Output peripherals: Allows both input (from outside world to computer) as well as, output (from computer to the outside world). Example: Touch screen etc.

Interfaces

Interface is a shared boundary between two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.

There are two types of interface:

1. CPU Interface
2. I/O Interface

Let's understand the I/O interface in details,

Input-Output Interface

Peripherals connected to a computer need special communication links for interfacing with CPU. In computer system, there are special hardware components between the CPU and peripherals to control and manage the input-output transfers. These components are called input-output interface units because they provide communication links between processor bus and peripherals. They provide a method for transferring information between internal system and input-output devices.

Asynchronous Data Transfer

We know that, the internal operations in individual unit of digital system are synchronized by means of clock pulse, means clock pulse is given to all registers within a unit, and all data transfer among internal registers occur simultaneously during occurrence of clock pulse. Now, suppose any two units of digital system are designed independently such as CPU and I/O interface.

And if the registers in the interface (I/O interface) share a common clock with CPU registers, then transfer between the two units is said to be synchronous. But in most cases, the internal timing in each unit is independent from each other in such a way that each uses its own private clock for its internal registers. In that case, the two units are said to be asynchronous to each other, and if data transfer occur between them this data transfer is said to be Asynchronous Data Transfer.

But, the Asynchronous Data Transfer between two independent units requires that control signals be transmitted between the communicating units so that the time can be indicated at which they send data.

This asynchronous way of data transfer can be achieved by two methods:

1. One way is by means of strobe pulse which is supplied by one of the units to other unit. When transfer has to occur. This method is known as "Strobe Control".
2. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another signal to acknowledge receipt of the data. This method of data transfer between two independent units is said to be "Handshaking".

The strobe pulse and handshaking method of asynchronous data transfer are not restricted to I/O transfer. In fact, they are used extensively on numerous occasions requiring transfer of data between two independent units. So, here we consider the transmitting unit as source and receiving unit as destination.

As an example: The CPU, is the source during an output or write transfer and is the destination unit during input or read transfer.

And thus, the sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination.

So, while discussing each way of data transfer asynchronously we see the sequence of control in

both terms when it is initiated by source or when it is initiated by destination. In this way, each way of data transfer, can be further divided into parts, source initiated and destination initiated.

We can also specify, asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that exists between the control and the data buses.

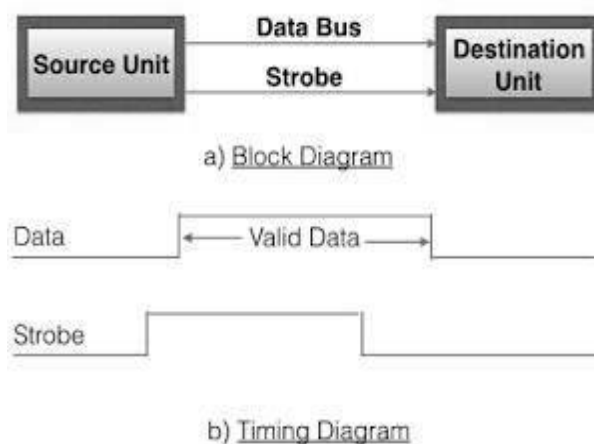
Now, we will discuss each method of asynchronous data transfer in detail one by one.

1. Strobe Control:

The Strobe Control method of asynchronous data transfer employs a single control line to time each transfer. This control line is also known as strobe and it may be achieved either by source or destination, depending on which initiates transfer.

Source initiated strobe for data transfer:

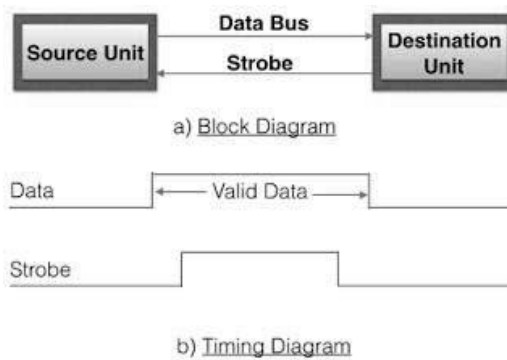
The block diagram and timing diagram of strobe initiated by source unit is shown in figure below:



In block diagram we see that strobe is initiated by source, and as shown in timing diagram, the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates a strobe pulse. The information on data bus and strobe control signal remain in the active state for a sufficient period of time to allow the destination unit to receive the data. Actually, the destination unit, uses a falling edge of strobe control to transfer the contents of data bus to one of its internal registers. The source removes the data from the data bus after it disables its strobe pulse. New valid data will be available only after the strobe is enabled again.

Destination-initiated strobe for data transfer:

The block diagram and timing diagram of strobe initiated by destination is shown in figure below:



In block diagram, we see that, the strobe initiated by destination, and as shown in timing diagram, the destination unit first activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe. And source removes the data from data bus after a predetermined time interval.

Now, actually in computer, in the first case means in strobe initiated by source - the strobe may be a memory-write control signal from the CPU to a memory unit. The source, CPU, places the word on the data bus and informs the memory unit, which is the destination, that this is a write operation.

And in the second case i.e., in the strobe initiated by destination - the strobe may be a memory read control from the CPU to a memory unit. The destination, the CPU, initiates the read operation to inform the memory, which is a source unit, to place selected word into the data bus.

2. Handshaking:

The disadvantage of strobe method is that source unit that initiates the transfer has no way of knowing whether the destination has actually received the data that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit, has actually placed data on the bus.

This problem can be solved by handshaking method.

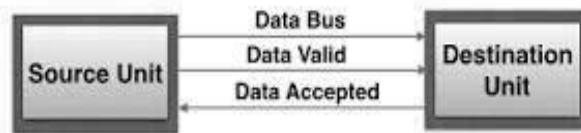
Hand shaking method introduces a second control signal line that provides a replay to the unit that initiates the transfer.

In it, one control line is in the same direction as the data flow in the bus from the source to destination. It is used by source unit to inform the destination unit whether there are valid data in the bus. The other control line is in the other direction from destination to the source. It is

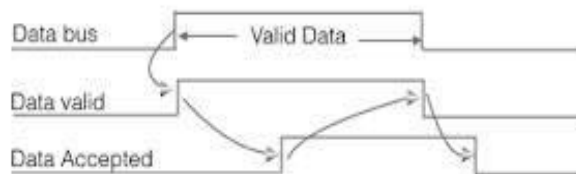
used by the destination unit to inform the source whether it can accept data. And in it also, sequence of control depends on unit that initiate transfer. Means sequence of control depends whether transfer is initiated by source and destination. Sequence of control in both of them are described below:

Source initiated Handshaking:

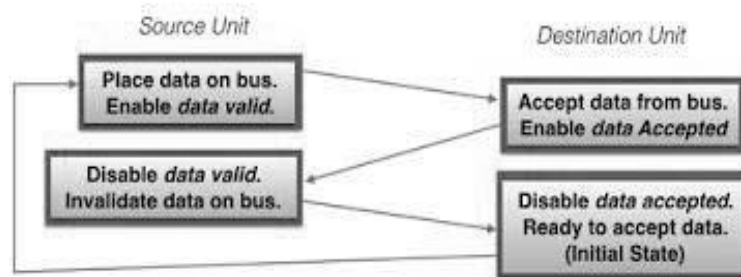
The source initiated transfer using handshaking lines is shown in figure below:



a) Block Diagram



b) Timing Diagram



c) Sequence Diagram(Sequence of events)

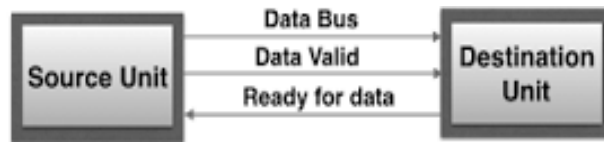
In its block diagram, we see that two handshaking lines are "data valid", which is generated by the source unit, and "data accepted", generated by the destination unit.

The timing diagram shows the timing relationship of exchange of signals between the two units. Means as shown in its timing diagram, the source initiates a transfer by placing data on the bus and enabling its data valid signal. The data accepted signal is then activated by destination unit after it accepts the data from the bus. The source unit then disables its data valid signal which invalidates the data on the bus. After this, the destination unit disables its data accepted signal and the system goes into initial state. The source unit does not send the next data item until after the destination unit shows its readiness to accept new data by disabling the data accepted signal.

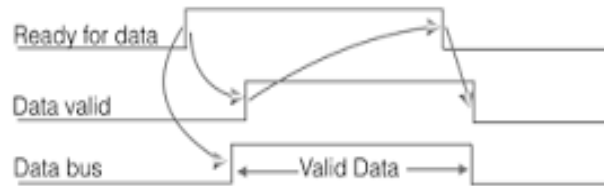
This sequence of events described in its sequence diagram, which shows the above sequence in which the system is present, at any given time.

Destination initiated handshaking:

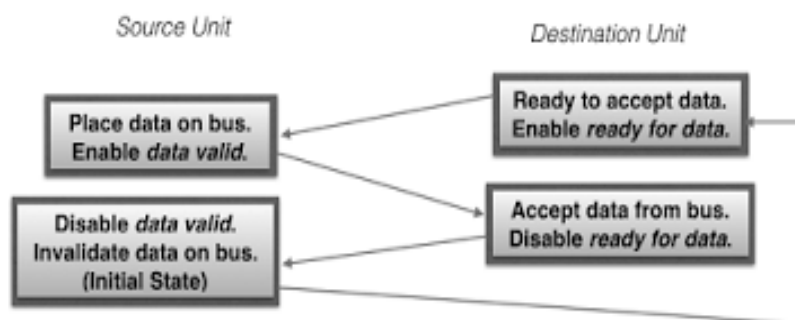
The destination initiated transfer using handshaking lines is shown in figure below:



a) Block Diagram



b) Timing Diagram



c) Sequence Diagram(sequence of events)

In its block diagram, we see that the two handshaking lines are "data valid", generated by the source unit, and "ready for data" generated by destination unit. Note that the name of signal data accepted generated by destination unit has been changed to ready for data to reflect its new meaning.

In it, transfer is initiated by destination, so source unit does not place data on data bus until it receives ready for data signal from destination unit. After that, hand shaking process is same as that of source initiated.

The sequence of event in it are shown in its sequence diagram and timing relationship between signals is shown in its timing diagram.

Thus, here we can say that, sequence of events in both cases would be identical. If we consider ready for data signal as the complement of data accept. Means, the only difference between source and destination initiated transfer is in their choice of initial state.

Modes of I/O Data Transfer

Data transfer between the central unit and I/O devices can be handled in generally three types of modes which are given below:

1. Programmed I/O
2. Interrupt Initiated I/O
3. Direct Memory Access

Programmed I/O

Programmed I/O instructions are the result of I/O instructions written in computer program. Each data item transfer is initiated by the instruction in the program.

Usually the program controls data transfer to and from CPU and peripheral. Transferring data under programmed I/O requires constant monitoring of the peripherals by the CPU.

Interrupt Initiated I/O

In the programmed I/O method the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is time consuming process because it keeps the processor busy needlessly.

This problem can be overcome by using interrupt initiated I/O. In this when the interface determines that the peripheral is ready for data transfer, it generates an interrupt. After receiving the interrupt signal, the CPU stops the task which it is processing and service the I/O transfer and then returns back to its previous processing task.

Direct Memory Access

Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This technique is known as DMA.

In this, the interface transfer data to and from the memory through memory bus. A DMA controller manages to transfer data between peripherals and memory unit.

Many hardware systems use DMA such as disk drive controllers, graphic cards, network cards and soundcards etc. It is also used for intra chip data transfer in multicore processors. In DMA, CPU would initiate the transfer, do other operations while the transfer is in progress and receive an interrupt from the DMA controller when the transfer has been completed.

Priority Interrupt

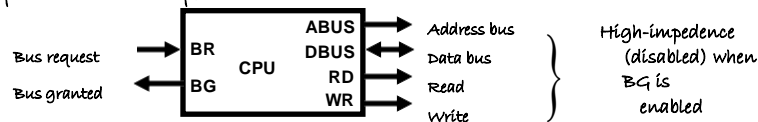
A priority interrupt is a system which decides the priority at which various devices, which generates the interrupt signal at the same time, will be serviced by the CPU. The system has authority to decide which conditions are allowed to interrupt the CPU, while some other interrupt is being serviced. Generally, devices with high speed transfer such as *magnetic disks* are given high priority and slow devices such as *keyboards* are given low priority.

When two or more devices interrupt the computer simultaneously, the computer services the device with the higher priority first.

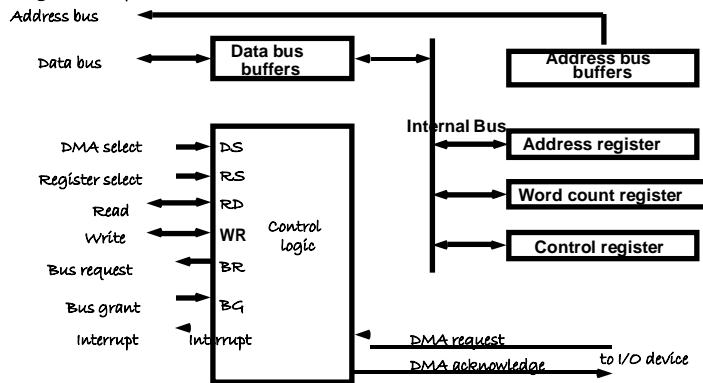
DIRECT MEMORY ACCESS

Block of data transfer from high speed devices, Drum, Disk, Tape

CPU bus signals for DMA transfer

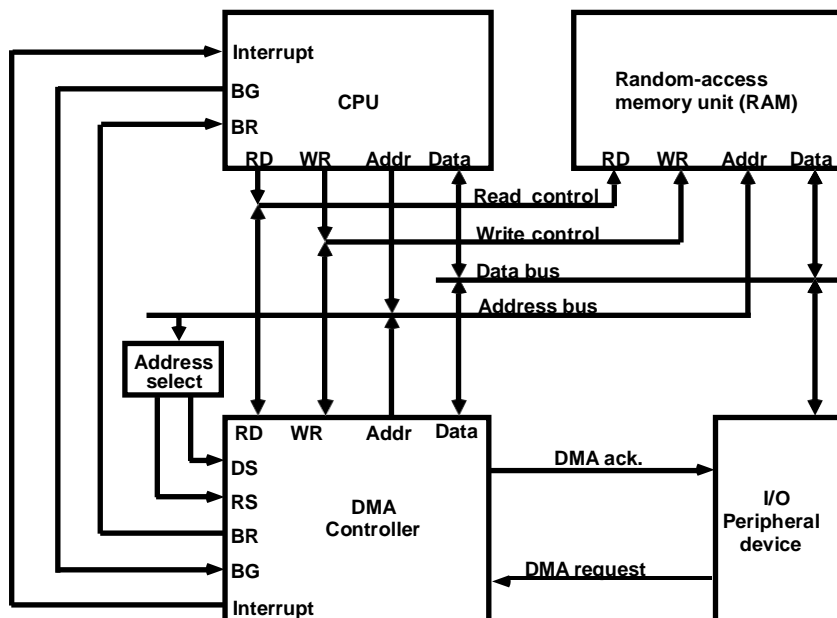


Block diagram of DMA controller



- * DMA controller - Interface which allows I/O transfer directly between Memory and Device, freeing CPU for other tasks
- * CPU initializes DMA Controller by sending memory address and the block size (number of words)

DMA TRANSFER



Input/output Processor

An input-output processor (IOP) is a processor with direct memory access capability. In this, the computer system is divided into a memory unit and number of processors.

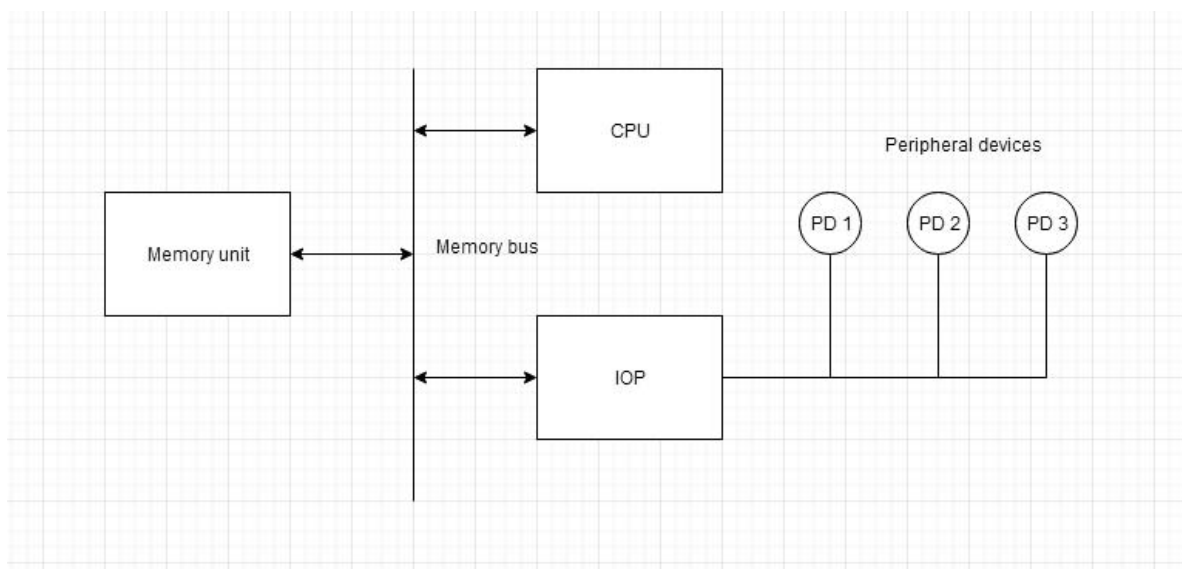
Each IOP controls and manage the input-output tasks. The IOP is similar to CPU except that it handles only the details of I/O processing. The IOP can fetch and execute its own instructions. These IOP instructions are designed to manage I/O transfers only.

Block Diagram Of I/O Processor:

Below is a block diagram of a computer along with various I/O Processors. The memory unit occupies the central position and can communicate with each processor.

The CPU processes the data required for solving the computational tasks. The IOP provides a path for transfer of data between peripherals and memory. The CPU assigns the task of initiating the I/O program.

The IOP operates independent from CPU and transfer data between peripherals and memory.



The communication between the IOP and the devices is similar to the program control method of transfer. And the communication with the memory is similar to the direct memory access method.

In large scale computers, each processor is independent of other processors and any processor can initiate the operation.

The CPU can act as master and the IOP act as slave processor. The CPU assigns the task of initiating operations but it is the IOP, who executes the instructions, and not the CPU. CPU instructions provide operations to start an I/O transfer. The IOP asks for CPU through interrupt.

Instructions that are read from memory by an IOP are also called *commands* to distinguish them from instructions that are read by CPU. Commands are prepared by programmers and are stored in memory. Command words make the program for IOP. CPU informs the IOP where to find the commands in memory.

II B. Tech I Semester Regular Examinations, March - 2021
COMPUTER ORGANIZATION
 (Com to CSE, IT)

Time: 3 hours

Max. Marks: 75

Answer any **FIVE** Questions each Question from each unit
 All Questions carry **Equal** Marks
 ~~~~~

- 1 a) Draw and explain Von Neumann Architecture. Explain Moore's Law. [8M]  
 b) Give the major characteristics of RISC and CISC architectures. [7M]
- Or
- 2 a) Explain IEEE-754 model for floating point representation. [8M]  
 b) Explain about Booth's multiplication algorithm and solve Multiply 7 and 3. [7M]
- 3 a) Explain the I/O instructions and type of I/O instructions. [8M]  
 b) Write a program to evaluate the arithmetic statement  $A=X-Y+C/P+Q$  using a stack organized computer with zero address instructions. [7M]
- Or
- 4 a) Explain about computer registers set in detailed. [8M]  
 b) Explain indirect address mode and how the effective address is calculated in this case. [7M]
- 5 a) Write the procedure to mitigate number of bits in micro instructions. [8M]  
 b) Explain arithmetic micro operations with examples. [7M]
- Or
- 6 a) What is a micro-operation of list and explain the four categories of the most common micro-operations? [8M]  
 b) Differentiate the relative addressing and index addressing modes. [7M]
- 7 a) Discuss about Cache-mapping functions. [8M]  
 b) What is associative memory? Explain with the help of block diagram. Also mention the situation in which associative memory can be effectively utilized. [7M]
- Or
- 8 a) Explain the Direct mapping in cache memory with an example. [8M]  
 b) Explain about Direct Memory Access (DMA). [7M]
- 9 a) Explain instruction pipeline with neat timing diagram. [8M]  
 b) Discuss Flynn's classification of computer. [7M]
- Or
- 10 a) Draw and explain arithmetic pipeline for floating point multiplication. [8M]  
 b) Explain about Interconnection network. [7M]

**II B. Tech I Semester Regular Examinations, March - 2021**  
**COMPUTER ORGANIZATION**  
 (Com to CSE, IT)

Time: 3 hours

Max. Marks: 75

Answer any **FIVE** Questions each Question from each unit  
 All Questions carry **Equal** Marks

- ~~~~~
- 1 a) Discuss Arithmetic addition and subtraction with signed-2's complement representation. [8M]  
 b) Is there any alternate of Von-Neumann architecture? If exists than give the basic idea of them. [7M]
- Or
- 2 a) Discuss modified Booth algorithm with suitable example. [8M]  
 b) Discuss the advantages, disadvantages, and applications of i) Excess – 3 code ii) Gray Code(Illustrate with one example each) [7M]
- 3 a) Explain the significance of the shift micro operations. [8M]  
 b) Explain about Arithmetic Micro operations in detailed. [7M]
- Or
- 4 a) What is the difference between a direct and an indirect address instruction? How many references to memory are needed for each type of instruction to bring an operand into a processor register? [8M]  
 b) How an interrupt is recognized? Explain the interrupt cycle. [7M]
- 5 a) What are addressing modes? Give an overview of the addressing modes. [8M]  
 b) Justify the statement “Stack computer consist of an operation code only with no address field”. [7M]
- Or
- 6 a) Discuss the different addressing modes of an instruction. [8M]  
 b) How stack is implemented in a general microprocessor system. [7M]
- 7 a) What is virtual memory? With the help of neat sketch explain the method of virtual to physical address translation. [8M]  
 b) Explain the READ and WRITE operations in Associative Memory. [7M]
- Or
- 8 a) What is cache memory? Explain different types of mapping from main memory to cache memory. [8M]  
 b) Give the hardware organization of associative memory. Why associative memory is faster than other memories. Deduce the logic equation used to find the match in the associative memory. [7M]
- 9 a) Explain about pipeline multiplexer. [8M]  
 b) Write short note on i) Magnetic Disks ii) Magnetic tapes [7M]
- Or
- 10 a) Draw and explain arithmetic pipeline for floating point addition. [8M]  
 b) Explain about Hypercube and Mesh network. [7M]



**II B. Tech I Semester Regular Examinations, March - 2021**  
**COMPUTER ORGANIZATION**  
 (Com to CSE, IT)

Time: 3 hours

Max. Marks: 75

Answer any **FIVE** Questions each Question from each unit  
 All Questions carry **Equal** Marks

- ~~~~~
- 1 a) Explain with the help of an example, the use of hamming code as error detection and correction code. [8M]  
 b) State the condition in which overflow occurs in case of addition & subtraction of two signed 2's complement number. How is it detected? [7M]
- Or
- 2 a) Convert hexadecimal number F2A7C2 to binary and octal numbers. [8M]  
 b) Explain the computer hierarchy of computer systems. [7M]
- 3 a) Design a hardware circuit to implement logical shift, arithmetic shift and circular shift operations. State your design specifications. [8M]  
 b) Explain how logic micro operation is work with suitable example? [7M]
- Or
- 4 a) A computer uses a memory unit with 256K words of 32 bits each. A binary Instruction code is stored in one word of memory. The instruction has four parts: an indirect bit, an operation code, a register code part to specify one of 64 registers, and an address part.  
 (a) How many bits are there in the operation code, the register code part, and the address part?  
 (b) How many bits are there in the data and address inputs of the memory?  
 b) Explain the following with respect to logic micro operations [7M]  
 i) Selective Set ii) Selective Complement iii) Selective Clear iv) Mask
- 5 a) What do you mean by addressing mode? Explain the following addressing modes with examples. [8M]  
 i) Index addressing mode ii) Relative addressing mode  
 b) What are different instruction formats we are using? [7M]
- Or
- 6 a) Explain various types of interrupts in detail. [8M]  
 b) Explain the difference between hardwired control and micro programmed control. Is it possible to have a hardwired control associated with a control memory? [7M]
- 7 a) Explain in detail the different mappings used for cache memory. [8M]  
 b) Discuss the main features of associative memory Page Table. How does it work in mapping the virtual address into Physical memory address? [7M]
- Or
- 8 a) Draw the block diagram of a DMA controller and explain its functioning? [8M]  
 b) Explain about the direct mapping. [7M]

- 9 a) Formulate a four segment instruction pipeline for a computer. Specify the operation to be performed in each segment. [8M]  
b) Draw and explain arithmetic pipeline for floating point multiplication. [7M]
- Or
- 10 a) What is pipelining? Name the two pipeline organizations. Explain about the arithmetic pipeline with the help of an example. [8M]  
b) Explain the characteristics of multiprocessor system. [7M]





**II B. Tech I Semester Regular Examinations, March - 2021**  
**COMPUTER ORGANIZATION**  
 (Com to CSE, IT)

Time: 3 hours

Max. Marks: 75

Answer any **FIVE** Questions each Question from each unit  
 All Questions carry **Equal** Marks

- ~~~~~
- 1 a) Draw and explain Von Neumann Architecture. Explain Moore's Law. [8M]  
 b) Discuss Arithmetic addition and subtraction with signed-2's complement representation. [7M]
- Or
- 2 a) Difference between RISC and CISC architectures. [8M]  
 b) Explain about Booth's multiplication algorithm and solve Multiply 9 and 7. [7M]
- 3 a) Explain the I/O instructions and type of I/O instructions. [8M]  
 b) Explain the significance of the shift micro operations. [7M]
- Or
- 4 a) Design a hardware circuit to implement logical shift, arithmetic shift and circular shift operations. State your design specifications. [8M]  
 b) Explain indirect address mode and how the effective address is calculated in this case. [7M]
- 5 a) Write the procedure to mitigate number of bits in micro instructions. [8M]  
 b) Justify the statement "Stack computer consist of an operation code only with no address field". [7M]
- Or
- 6 a) What is a micro-operation of list and explain the four categories of the most common micro-operations? [8M]  
 b) What do you mean by addressing mode? Explain the following addressing modes with examples. [7M]  
 i) Index addressing mode ii) Relative addressing mode.
- 7 a) What is cache memory? Explain different types of mapping from main memory to cache memory. [8M]  
 b) What is associative memory? Explain with the help of block diagram. Also mention the situation in which associative memory can be effectively utilized. [7M]
- Or
- 8 a) Discuss the main features of associative memory Page Table. How does it work in mapping the virtual address into Physical memory address? [8M]  
 b) Explain about Direct Memory Access (DMA). [7M]
- 9 a) Explain instruction pipeline with neat timing diagram. [8M]  
 b) Explain about Hypercube and Mesh network. [7M]
- Or
- 10 a) Draw and explain arithmetic pipeline for floating point multiplication. [8M]  
 b) Explain about Interconnection network. [7M]

**II B. Tech I Semester Supplementary Examinations, September - 2021****COMPUTER ORGANIZATION**

(Com to CSE, IT)

Time: 3 hours

Max. Marks: 75

Answer any **FIVE** Questions each Question from each unitAll Questions carry **Equal** Marks

- ~~~~~
- 1 a) Explain how a binary number can be converted to an octal and a hexadecimal number. [8M]  
b) Explain in detail about fixed-point representation. [7M]
- Or
- 2 a) Design a 4-bit odd parity generator and checker and explain it. [8M]  
b) Draw a flowchart which explains multiplication of two signed magnitude fixed point number. [7M]
- 3 a) With the help of block diagram, explain the 4-bit binary subtractor. [8M]  
b) With the help of a diagram explain one stage of arithmetic logic shift unit. [7M]
- Or
- 4 a) Discuss in brief about the flowchart for basic computer operations. [8M]  
b) What is interrupt? Explain different types of interrupt. [7M]
- 5 a) Illustrate the use of various addressing mode with examples. [8M]  
b) List and explain the functions of control unit. [7M]
- Or
- 6 a) Writ about symbolic micro program and binary micro program. [8M]  
b) Discuss the two techniques to design the control unit. [7M]
- 7 a) Explain the various available formats and storage capability of DVD. [8M]  
b) What do you mean by the associative memory? Give the Hardware organization of associative memory. [7M]
- Or
- 8 a) Discuss the various cache block replacement algorithms. [8M]  
b) What are the different kinds of DMA? Explain. [7M]
- 9 a) Explain multiprocessor system and its characteristics. [8M]  
b) Discuss the functioning of crossbar switching network. [7M]
- Or
- 10 a) Explain the concept of speedup ratio with an example. [8M]  
b) Mention the pipeline conflicts that cause the instruction pipeline to deviate from its normal operation. [7M]

