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

$$\underbrace{d_{n-1}\ d_{n-2} \cdots d_1\ d_0}_{\text{Integer part}}\ .\ \underbrace{d_{-1}\ d_{-2} \cdots d_{-m}}_{\text{Fractional Part}}$$

Radix point

Number systems are classified as

ⓐ Binary Number System
ⓑ Decimal Number System
ⓒ Octal Number system
ⓓ Hexa decimal Number System.

ⓐ The Binary number system is a radix-2. Thes is represented interms of 0 and 1. The radix point is known as the binary point and 0 and 1 is a binary bits.

ⓑ 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.

ⓒ 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.

ⓓ 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.

Ex:- 1 0 1 1 0 1 0
↓ ↓
MSB (Most Significant bit) LSB (Least Significant bit)

Decimal number system to Binary number system conversion :-

① $(25)_{10} = (?)_2$

2	25
2	12 — 1
2	6 — 0
2	3 — 0
	1 — 1

From Bottom to top

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

(Successive Division method)

② $(72)_{10} = (?)_2$

2	72
2	36 — 0
2	18 — 0
2	9 — 0
2	4 — 1
2	2 — 0
	1 — 0

From Bottom to Top

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

⇒ Fractional part :-

③ $(0.25)_{10} = (?)_2$ Top to Bottom

$0.25 \times 2 = 0.5 \rightarrow 0$
$0.5 \times 2 = 1.0 \rightarrow 1$
$0.0 \times 2 = 0.0 \rightarrow 0$
$0.0 \times 2 = 0 \rightarrow$ ignore it

Whenever we get repeated numbers just ignore it

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

④ $(0.8125)_{10} = (?)_2$

$0.8125 \times 2 = 1.6250 \rightarrow 1$
$0.6250 \times 2 = 1.250 \rightarrow 1$
$0.250 \times 2 = 0.50 \rightarrow 0$
$0.50 \times 2 = 1.0 \rightarrow 1$
$0.0 \times 2 = 0.0 \rightarrow 0$

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)

$0.625 \times 2 = 1.250 \rightarrow 1$

$0.250 \times 2 = 0.50 \rightarrow 0$

$0.50 \times 2 = 1.0 \rightarrow 1$

$0.0 \times 2 = 0.0 \rightarrow 0$

(1010)

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

⑥ $(25.125)_{10} = (?)_2$

2	25	
2	12 – 1	
2	6 – 0	
2	3 – 0	
	1 – 1	

(11001)

$0.125 \times 2 = 0.250 \rightarrow 0$

$0.250 \times 2 = 0.50 \rightarrow 0$

$0.5 \times 2 = 1.0 \rightarrow 1$

$0.0 \times 2 = 0.0 \rightarrow 0$

(0010)

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

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

$1\times2^5+0\times2^4+1\times2^3+0+2^2+1\times2^1+1\times2^0 = 32+0+8+0+2+1$

$= 43$

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

③ $(101.10)_2 = (?)_{10}$

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

$1\times2^2+0\times2^1+1\times2^0.+1\times2^{-1}+0\times2^{-2}$

$= 4+0+1.+0.5+0$

$= 5+0.5 = 5.5$

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

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}

$=1\times2^4+1\times2^3+0\times2^2+1\times2^1+0\times2^0.+0\times2^{-1}+1\times2^{-2}+0\times2^{-3}$

$= 16+8+0+2+0.+0+0.25+0$

$= 26+0.25 = 26.25$

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

⑤ $(10010)_2 = (?)_{10}$

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

$1\times2^4+0\times2^3+0\times2^2+1\times2^1+0\times2^0$

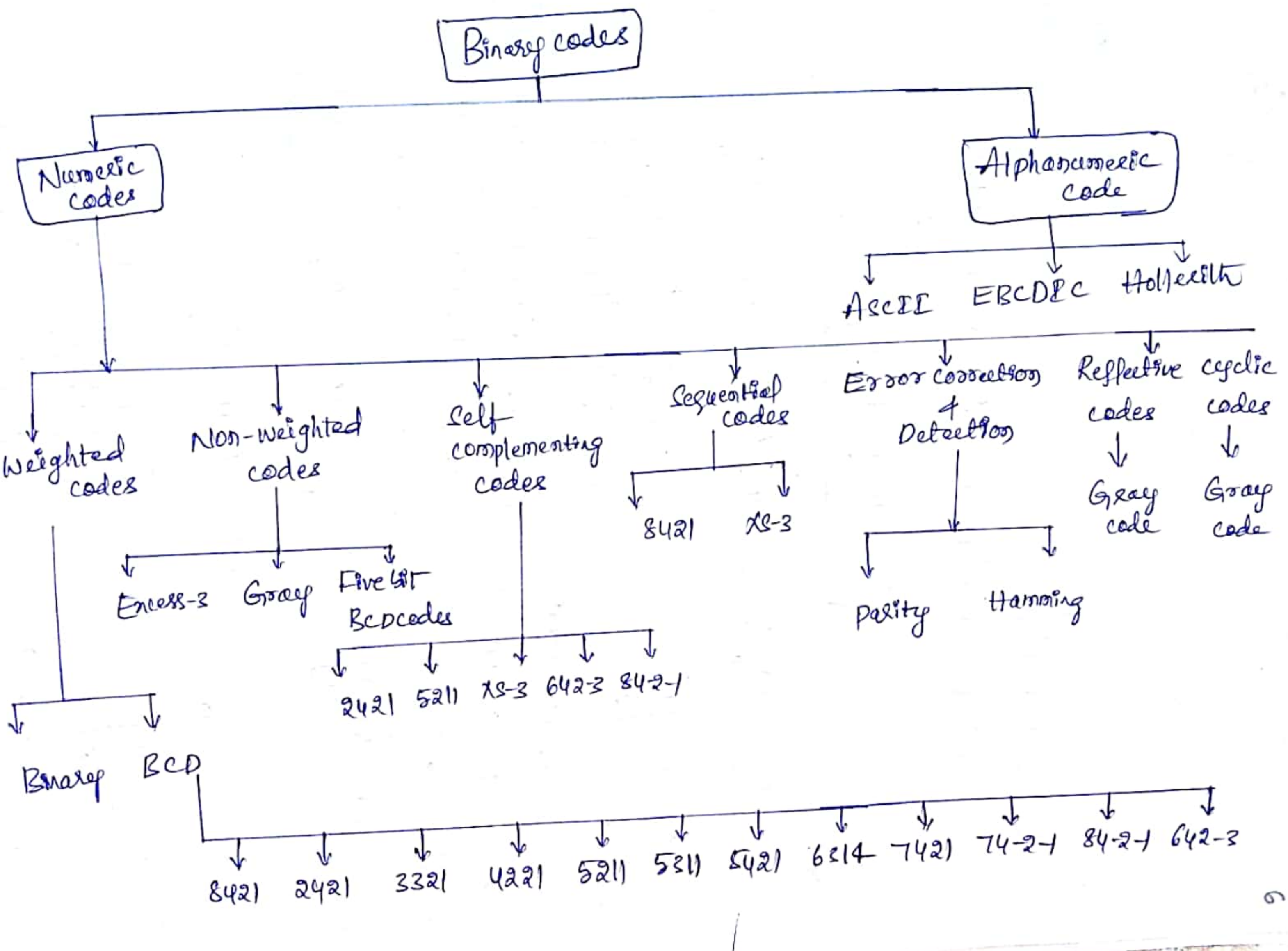
$= 16+0+0+2+0 = (18)_{10}$

⑥ $(011.01)_2 = (?)_{10}$

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

$0\times2^2+1\times2^1+1\times2^0.+0\times2^{-1}+1\times2^{-2}$

$= 0+2+1.+0+0.25 = (3.25)_{10}$



→ Binary coded decimal Numbers (BCD) :-

→ BCD code uses four bits to represents 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
8-2-1, 642-3.

→ Representation of BCD code

Ex:- ① 12 (Each digit is represented by four bits)
↙ ↘
0001 0010

Ex:- ② 14 (Each digit is indicated by group of 4-bits)
↓ ↓
0001 0100

There are 6 unused states in BCD (1010, 1011, 1100, 1101, 1110, 1111)
↓ ↓ ↓ ↓ ↓ ↓
10 11 12 13 14 15.

Decimal number		BCD
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:-
3 5 6
↓ ↓ ↓
0011 0101 0110.

⇒ 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

14
0001 0100
8-bits required

12
0001 0010
8-bits required.

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

Binary → 8 4 2 1
1 1 0 0
only 4-bits

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

BCD → 1 2
0001 0010
8-bits.

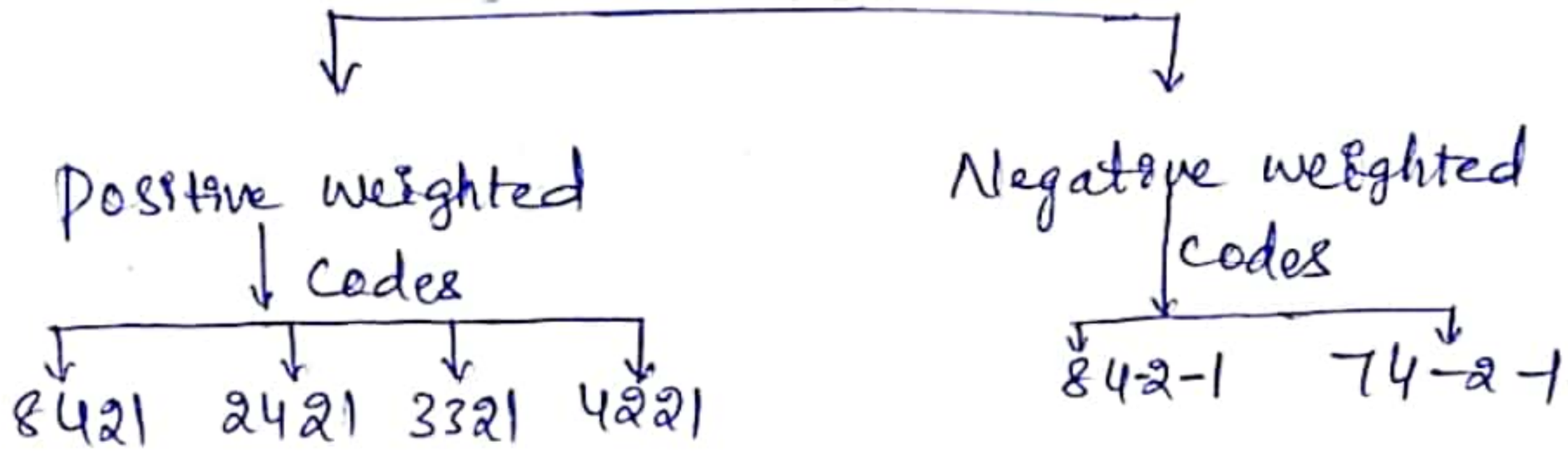
Note :- From the above concept, we can conclude one thing that BCD is simple to represent decimal numbers but some times it takes more no. of bits. So, it occupies memory. Arithmetic operations are more complex than the Binary.

Ex: 92 in Binary representation

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

92 in BCD
9 2
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 / 0101	1010 / 0110	1001 / 0111
7	0111	1101 / 0111	1101	1101 / 1011

All these are positive weighted codes

Ex^o:-

Decimal	8421	7421
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.

2+4+2+1	5+2+1+1	6+4+2-3	8+4-2-1
=9	=9	=9	=9

Decimal digital	8421
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Ex:① 5 in Excess-3 is $5+3=8=1000$ (8421)

It can 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 $(9-5=4)$

∴ It is a selfcomplementing code

Ex:② 4 in 2421 = 0100

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

5 is the 9's complement of 4.

∴ 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.

∴ BCD is not a self complementing code

Ex:④ Binary (8421) code for 7 is 0111

$0111 \xrightarrow{\text{1's complement}} 1000 = 8$ (Binary code 8) of decimal

8 is not the 9's complement of 7

∴ Binary code (8421) is not a self complementing code.

⇒ Cyclic codes :- cyclic codes are those in which each successive code word differes 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 is why we can't do arithmetic operation by using this code.

Ex:-

Decimal No.	Binary	Gray
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

(a) convert $(1010)_2$ to gray code

Sol 1010
↓
ans 1111

(b) convert $(0110)_2$ to gray code

0 1 1 0
↓
ans 0101

X-OR Truth Table

A	B	A ⊕ 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.
$\therefore 2^7 = 128$ symbols are used to represent Infor's.

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

32 → Space
33 → !
34 → "
35 → #
36 → $
37 → %
38 → &
39 → €
40 →)

42 → *
43 → +
44 → ,
45 → -
46 → .
47 → /
48 → 57 → (0-9)
58 → :
59 → ;
60 → <

61 → =
62 → >
63 → ?
64 → @
65-90 (A-Z) capital letters
91 → [
92 → \
93 →]
94 → ^
95 → _ (underscore)

96 → `

97–122 (a–z) small Letters

123 → {

124 → | (vertical bar)

125 → }

126 → ~

0–31 → Character control

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

Ex°:-

1001001	1000001	1001101
↓	↓	↓
73	65	77
↓	↓	↓
I	A	M

Ex°:-

1001010	1001100
↓	↓
74	76
↓	↓
J	L

→ ASCII codes are used in micro computer (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.

Ex°:-

1010111	space	1000010	space	1000111	space	1000011
↓	↓	↓	↓	↓	↓	↓
87	32	66	32	71	32	67
↓		↓		↓		↓
W		B		G		C

→ 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 bits 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

$$2^p \geq m+p+1$$
$$2p \geq 4+p+1$$
$$2^3 \geq 4+3+1$$
$$8 \geq 8$$

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

7	6	5	4	3	2	1
			2^2		2^1	2^0
m_4	m_3	m_2	P_3	m_1	P_2	P_1
0	0	1	(P3)	1	(P2)	(P1)

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

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

$\therefore P_1 = 0$.

$P_2 = 2, 3, 6, 7 = P_2\,1\,0\,0$ To become the even parity ($\because P_2 = 1$)

$= 1100$

$\therefore P_2 = 0$

$P_3 = 4, 5, 6, 7 = P_3\,1\,0\,0$ to become the even parity ($\because P_3 = 1$)

$= 1100$

$\therefore P_3 = 1$

Error position = By combining the parity bits

$P_3P_2P_1 = P_3P_2P_1 = 0110 = (6)_{10}$

Error is located at 2nd position

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

After correcting = 0111110.

Ex:

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

$2^3 \geq 3 + 5 \;\therefore\; 8 \geq 8$ (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)

$P_1 \Rightarrow 1, 3, 5, 7 \rightarrow P_1110 \rightarrow 0110$ ($P_1 = 0$)

$P_2 \rightarrow 2, 3, 6, 7 \rightarrow P_2110 \rightarrow 0110$ ($P_2 = 0$)

$P_2 \rightarrow 4, 5, 6, 7 \rightarrow P_3110 \rightarrow 0110$ ($P_3 = 0$)

Total message bits

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

odd parity :-

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

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

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

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 recieved 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

(even parity)

$E_1 \rightarrow 1, 3, 5, 7 \rightarrow 1100 \rightarrow$ to make it even parity

$E_1 = 0.$

$E_2 \rightarrow 2, 3, 6, 7 \rightarrow 1100 \Rightarrow E_2 = 0$

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

Error $= E_3 E_2 E_1 = 000$ (0th position)

odd parity

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

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

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

Error $= E_3 E_2 E_1 = 111$ ^ 7th position error is there

So

Corrected code may be 1110001

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

Sol Given message bit

$$m = 10111$$

By using trail and error method we should find parity bits

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

$2^1 \geq 5+1+1$ 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

1 2 3 4 5 6 7 8 9

P_1 P_2 m_1 P_2 m_2 m_3 m_4 P_4 m_5

P_1 P_2 1 P_3 1 1 0 P_4 1

$\therefore m = 1\ 0\ 1\ 1\ 1$

$m_5\ m_4\ m_3\ m_2\ m_1$

Step ①

Bit destination	m_5	P_8	m_4	m_3	m_2	P_4	m_1	P_2	P_1
Bit Location	9	2^3 8	7	6	5	2^2 4	3	2^1 2	2^0 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 $p=0$; so $P_1 = 0$

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

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

$P_4 \rightarrow P_4 m_2 m_3 m_4 = P_4 110$ 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

The 9 bit hamming code

9 8 7 6 5 4 3 2 1

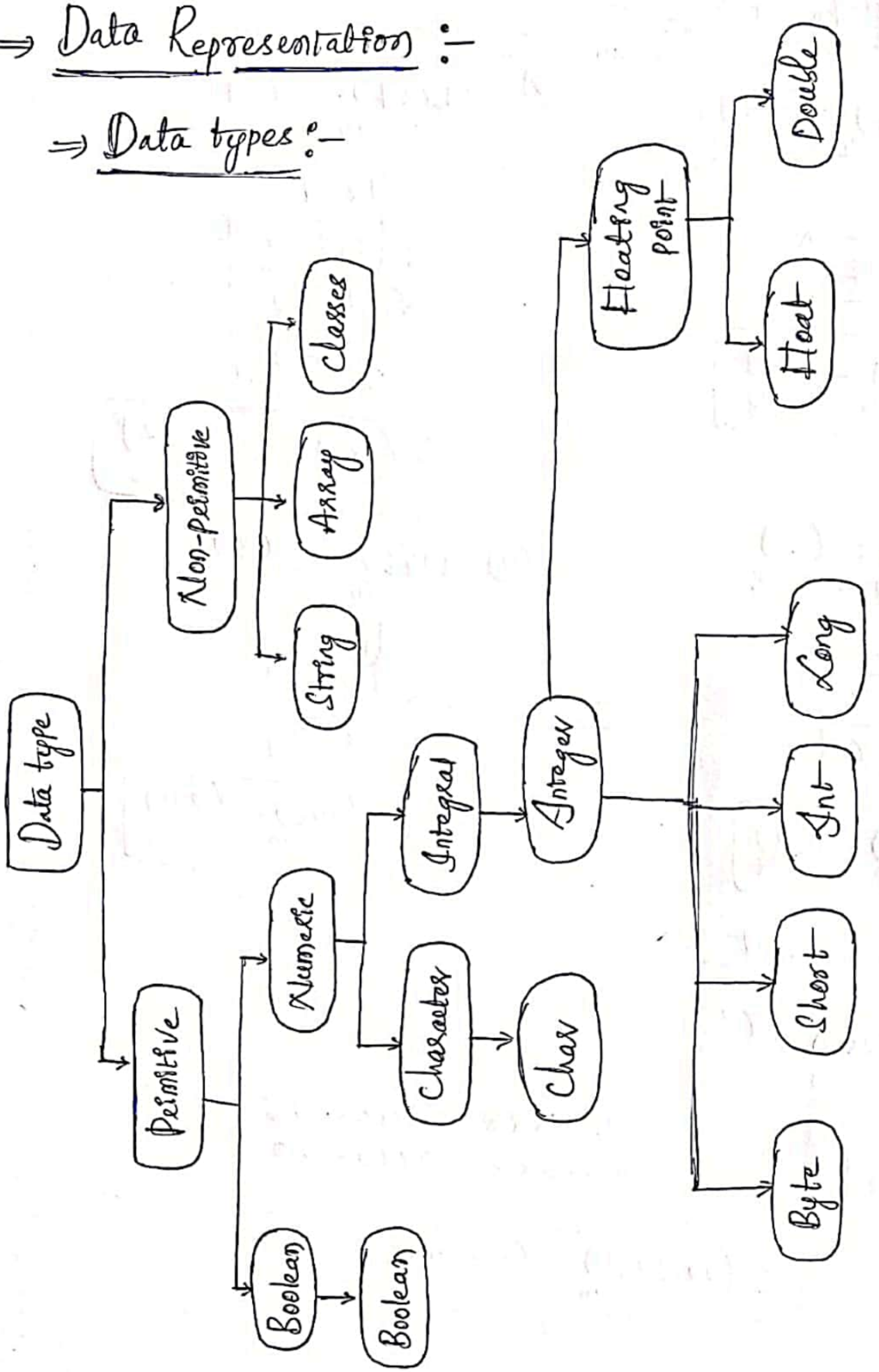
1 0 0 1 1 1 1 1 0

Error in 6^{th} position

✓ 100011110

⇒ Data Representation :-

⇒ Data types :-



⇒ Decimal to octal :-

① $(20)_{10} = (?)_8$

8 | 20
8 | 2 – 4 ↑

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

② $(1234)_{10} = (?)_8$

8 | 1234
8 | 154 – 2
8 | 19 – 2
2 – 3 ↑

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

③ $(183)_{10} = (?)_8$

8 | 183
8 | 22 – 7
2 – 6 ↑

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

④ $(145)_{10} = (?)_8$

8 | 145
8 | 18 – 1
2 – 2 ↑

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

⇒ Fractional part :-

① $(27.625)_{10} = (?)_8$

8 | 27
3 – 3 ↑

$0.625 \times 8 = 5.000 \longrightarrow 5$

$0.000 \times 8 = 0.000 \longrightarrow 0$

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

② $(3287.5100098)_{10} = (?)_8$

Sol

Integer part

8 | 3287
8 | 410 — 7
8 | 51 — 2
6 — 3

$0.5100098 \times 8 = 4.0800 \rightarrow 4$
$0.0800 \times 8 = 0.640 \rightarrow 0$
$0.640 \times 8 = 5.125 \rightarrow 5$
$0.125 \times 8 = 1.000 \rightarrow 1$

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

③ $(20.5)_{10} = (?)_8$

Fractional part

8 | 20
2 — 4

$0.5 \times 8 = 4.0 \rightarrow 4$
$0.0 \times 8 = 0.0 \rightarrow 0$

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

④ $(60.7)_{10} = (?)_8$

8 | 60
7 — 4

$0.7 \times 8 = 5.6 \rightarrow 5$
$0.6 \times 8 = 4.8 \rightarrow 4$
$0.8 \times 8 = 6.4 \rightarrow 6$
$0.4 \times 8 = 3.2 \rightarrow 3$
$0.2 \times 8 = 1.6 \rightarrow 1$

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

⇒ Decimal to Hexadecimal :- (H = 16)

① $(20)_{10} = (?)_{H}$

16 | 20
1 – 4 ↑

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

② $(1234)_{10} = (?)_{H}$

16 | 1234
16 | 77 – 2
4 – 13 (or) D ↑

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

③ $(20.5)_{10} = (?)_{H}$

16 | 20
1 – 4 ↑

$0.5 \times 16 = 8.0 \longrightarrow 8$ ↓
$0.0 \times 16 = 0.0 \longrightarrow 0$

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

④ $(675.625)_{10} = (?)_{16}$

16 | 675
16 | 42 – 3
2 – 10 (or) A ↑

$0.625 \times 16 = 10.000 \longrightarrow 10$ (or) A ↓
$0.000 \times 16 = 0.000 \longrightarrow 0$

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

⇒ Binary to octal :-

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

① $(101)_2 = (?)_8$

4 2 1
1 0 1 → 5

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

② $(1101)_2 = (\ )_8$

0 0 1 | 1 0 1 = 15

1 5

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

③ $(10.11001)_2 = (?)_8$

10 . 11001

010 . 110 010

2 6 2

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

④ $(011010110.11)_2 = (?)_8$

011010110 . 11

011 010 110 . 110

3 2 6 6

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

⑤ $(1101101.01101)_2 = (\ )_8$

001 101 101 . 011 010 → append zero

append zero.

1 5 5 3 2

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

⇒ Binary to Hexadecimal :-

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

① $(1101011011)_2 = (?)_{16}$

0011 0101 1011

3 5 11 (or) B

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

② $(1101011011.110101)_2 = (\ )_4$

0011 0101 1011 . 1101 0100

3 5 11 (or) B 13 (or) D 4

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

③ $(1010011010111)_2 = (?)_{16}$

0010 1001 1010 1111

2 9 10 (or) A 15 (or) F

$\therefore (1010011010111)_2 = (29AF)_{16}$

④ $(100101011.101110)_2 = (\ )_{16}$

0001 0010 1011 . 1011 1000

1 2 11 (or) B 11 (or) B 8

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

⇒ 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	.	$\bar{8}^1$

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

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

$= 16 + 4 + 0.5 = 20.5$

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

③ $(6327.4051)_8 = (?)_{10}$

6	3	2	7	.	4	0	5	1
8^3	8^2	8^1	8^0	.	$\bar{8}^1$	$\bar{8}^2$	$\bar{8}^3$	$\bar{8}^4$

$= 6 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times \frac{1}{8} + 0 \times \frac{1}{8^2} + 5 \times \frac{1}{8^3} + 1 \times \frac{1}{8^4}$

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

$= (3367.5100098)_{10}$

④ $(1234.342)_8 = (\quad)_{10}$

1	2	3	4	.	3	4	2
8^3	8^2	8^1	8^0	.	$\bar{8}^1$	$\bar{8}^2$	$\bar{8}^3$

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

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

$= (668.440)_{10}$.

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

① $(15)_8 = (?)_2$

1 → 001
5 → 101

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

② $(736)_8 = (?)_2$

7 → 111
3 → 011
6 → 110

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

③ $(563)_8 = (?)_2$

5 → 101
6 → 110
3 → 011

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

④ $(725)_8 = (\ )_2$

7 → 111
2 → 010
5 → 101

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

⑤ $(326)_8 = (?)_2$

3 → 011
2 → 010
6 → 110

$(326)_8 = (011010110)_2$

⑥ $(452)_8 = (\ )_2$

4 → 100
5 → 101
2 → 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 @r) octal to decimal then decimal to hexadecimal.

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

@r) $(\ )_8 \rightarrow (\ )_{10} \rightarrow (\ )_{16}$

① $(356.63)_8 = (\ )_{16}$

Step① Octal to binary

Step② Binary to Hexadecimal

3 5 6 . 6 3
↓ ↓ ↓ ↓ ↓
011 101 110 . 110 011

0000 0110 1110 . 1100 1100
0 E E C C
=14 =14 =12 =12

∴ $(356.63)_8 = (0EE.CC)_{16}$

② $(247.52)_8 = (\ )_{16}$

Step① Octal to binary

Step② Binary to Hexadecimal

2 4 7 . 5 2
↓ ↓ ↓ ↓ ↓
010 100 111 . 101 010

0000 1010 0111 . 1010 1000
0 10 7 10 8
(or) (or)
A A

∴ $(247.52)_8 = (0A7.A8)_{16}$

⇒ Hexadecimal to Other Number system :-

⇒ Hexadecimal to binary :-

① $(2F9A)_{16} = (\ )_2$

2 → 0010
F → 1111
9 → 1001
A → 1010

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

② $(6A3)_{16} = (\quad)_2$

6 → 0110
A → 1010
3 → 0011

$\therefore (6A3)_{16} = (0110\ 1010\ 0011)_2$

③ $(58C)_{16} = (\quad)_2$

5 → 0101
8 → 1000
C → 1100

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

④ $(7DE3)_{16} = (\quad)_2$

7 → 0111
D → 1101
E → 1110
3 → 0011

$\therefore (7DE3)_{16} = (0111\ 1101\ 1110\ 0011)_2$

⇒ Hexadecimal to Decimal :-

① $(3A.2F)_{16} = (\quad)_{10}$

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

$3\times16^1 + 10\times16^0 + 2\times16^{-1} + 15\times16^{-2}$

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

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

② $(5E.7A)_{16} = (\quad)_{10}$

5	E	.	7	4
16^1	16^0	.	16^{-1}	16^{-2}

$5\times16^1 + 14\times16^0 + \frac{7}{16^1} + \frac{10}{16^2}$

$= 90 + 14 + 0.43 + 0.03$

$= (104.46)_{10}$

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

⟹ 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 :-

$$(\quad)_{16} \rightarrow (\quad)_{2} \rightarrow (\quad)_{8}$$

(or)

$$(\quad)_{16} \rightarrow (\quad)_{10} \rightarrow (\quad)_{8}$$

① $(B9F.AE)_{16} = (\quad)_8$

✓ Hexadecimal to binary

✓ Binary to Octal

B	9	F	.	A	E
1011	1001	1111	.	1010	1110

$= 101\ 110\ 011\ 111 . 101\ 011\ 100$

$5\ \ 6\ \ 3\ \ 7\ .\ 5\ \ 3\ \ 4$

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

② $(A8C.BC7)_{16} = (\quad)_8$

✓ Hexadecimal to binary

✓ Binary to octal

A	8	C	.	B	C	7
1010	1000	1100	.	1011	1100	0111

$= 001\ 010\ 100\ 011\ 100 . 101\ 111\ 000\ 111$

$1\ \ 2\ \ 4\ \ 3\ \ 4\ .\ 5\ \ 7\ \ 0\ \ 7$

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

⇒ 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 $\boxed{(r^n - N) - 1}$

⇒ 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 $\boxed{r^n - N}$

Binary (2) → 1's complement
Binary (2) → 2's complement

Octal (8) → 7's complement
Octal (8) → 8's complement

Decimal (10) → 9's complement
Decimal (10) → 10's complement

Hexadecimal (16) → 15's complement
Hexadecimal (16) → 16's complement

⇒ 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.

(1)

Ex:- 0101101000 → (Given binary number)

1010010111 → (1's complement form)

1111 ← (adding 1)

1010010000 → 2's complement

⇒ Binary addition

0+0	= 0	
0+1	= 1	
1+0	= 1	
1+1	= 1	0 → sum

↓ carry

1+1+1 = 1 1

↓ ↓

carry sum

(2) 0110101010 → Given binary number

1001010101 → 1's complement form

+1 → adding '1'

1001010110 → 2's complement form.

⇒ 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, being 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~~ Method :

Sol Normally

$25 \longrightarrow 00011001$

$-14 \longrightarrow 11110001$

$+11$

$\boxed{1}\ 00001010$

End around carry

00001011

$14 \rightarrow 00001110$

1's complement $\rightarrow 11110001$

1 → (adding end around carry)

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

+14 → 00001110

−25 → 11100110

−11 11111010 → No carry

25 → 00011001

14 → 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 11110100 is −00001011. The result is −11₁₀

(3) Add −25 to −14 using 8-bit 1's complement method.

25 → 00011001 (normal form)

14 → 00001011 (normal form)

−25 → 11100110 (1's complement)

−14 → 11110100 (1's complement)

−39 ← 1 11010011

End around carry

11010011

1111 (adding 1 end around carry)

11010100

MSB ↓

00101011 → 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. (↓ -ve sign)

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

+25 → 0 0 0 1 1 0 0 1
+14 → 0 0 0 0 1 1 1 0
+39 → 0 0 1 0 0 1 1 1

There is no carry. The MSB is '0'. So, the result is positive and is in pure binary. Therefore, the result is $+39_{10}$.

Ex:②

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

36 → 0 0 1 0 0 1 0 0 — 36 → 0 0 1 0 0 1 0 0
−20 → 1 1 1 0 1 0 1 1 (1st complement) — 20 → 0 0 0 1 0 1 0 0
+16

1 0 0 0 0 1 1 1 1 (adding end around carry)
End around carry

0 0 0 1 0 0 0 0

MSB is 0

⇒ 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

+36 → 0 0 1 0 0 1 0 0 — +36 → 0 0 1 0 0 1 0 0
+20 → 0 0 0 1 0 1 0 0 — +20 → 0 0 0 1 0 1 0 0

0 0 1 1 1 0 0 0

MSB is 0

, 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.

−36 → 11011011 → 1's complement

36 → 00100100

20 → 00010100

−20 → 11101011

−56 ← 1 11001110 → adding of end around carry

end around carry

11000111

MSB = 1

⇒ 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 —1's complement→ −00111000 = −(56)10

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

−36 —1's comple→ 11011011

+20 —Normal form→ 00010100

−16 ← 11101111

MSB = 1

→ 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 → −00010000 = (−16)10.

⇒ 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 'zero' 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

+14 = 0000 1110 (normal format)
1111 0001 (1's complement)
1 (2's complement)
1111 0010 → 2's complement form

25 → 0001 1001
−14 → 1111 0010
1 0000 1011

Ignore carry MSB

→ 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 +00001011 = $+11_{10}$

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

+25 → 00011001

−25 → 11100110 → (1's complement form)

$\quad\quad\quad\quad\quad$ 1

11100111 → 2's complement form

+14 → 00001110

−25 → 11100111

11110101 (No carry)

↑ MSB = 1

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 11110101, that is 00001010

$\quad\quad\quad\quad\quad$ 1

−00001011 = $(-11)_{10}$

(3) Imp ** Given the two binary numbers X = 1010100 and Y = 1000011, perform subtraction (a) X−Y (b) Y−X using 2's complements

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

0111100 → 1's complement

$\quad\quad\quad\quad\quad$ 1

0111101 → 2's complement

① Find 2's complement of subtrahend

② Add subtrahend to the minuend.

$X = 1010100$

$Y = 0111101$ → 2's complement of Y

carry: 1111

[1] 0 0 0 0 0 0 1

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 → 1's complement

  1

0101100 → 2's complement

$Y = 1000011$

0101100 → 2's complement of X

1101111

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 → 1's complement

  1

0010001 → correct answer 2's complement form.

Imp: 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 → 1's complement of Y

 1111

[1] 0010000

End around carry ↓ 1

0010001

↑ MSB = 0 so, it is in true binary form

(b) Y − X

Y = 1000011
X = 0101011 → 1's complement of Y

 11

1101110

↑ MSB = 1

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

1's complement of 1101110 is −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. 2's complement of subtrahend to minuend.

② If there is a carry it indicates that the answer is +ve 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 9999
3465
6534

(9's complement of 3465)

By using formula

$(r^n - N) + 1$
$(10^4 - 3465) + 1 = 6534$

② 782.54

Sol 999.99
782.54
217.45

③ 4526.075

Sol 9999.999
4526.075
5473.924

9's complement Method of Subtraction :-

Q Subtract the following numbers using 9's complement method.

(1) 745.81 − 436.62

Step (1)

```
  999.99
  436.62
  ------
  563.37  → 9's complement of 436.62
```

Step (2)

```
   745.81
   563.37
  -------
 [1] 309.18
        1   → (adding of end around carry)
  -------
   309.19  → final answer.
```

Carry indicates the answer is +ve

(2) 436.62 − 745.81

Step (1)

```
  999.99
  745.81
  ------
  254.18  → 9's complement of 745.81
```

Step (2)

```
  436.62
  254.18
  ------
  690.80
```

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

```
  999.99
  690.80
 -------
 -309.19   → Therefore the answer is -309.18.
```

⇒ 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
```
  9999
  3465
 ------
  6534   → 9's complement
     1
 ------
  6535   → 10's complement
 ------
```

② 782.54

Sol
```
  999.99
  782.54
 -------
  217.45   → 9's complement
      1
 -------
  217.46   → 10's complement
 -------
```

③ 4526.075

```
  9999.999
  4526.075
 ---------
  5473.924   → 9's complement
        1
 ---------
  5473.925   → 10's complement
 ---------
```

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 is its 10's complement form and put negative sign infront of the answer.

(1) 745.81 − 436.62

step (1)

```
  999.99
  436.62
---------
  563.37
       1
---------
  563.38  → 10's complement form
```

Step (2)

```
  745.81
  562.38
---------
1 309.19
```

ignore the carry

carry indicates result is positive

(2) 436.62 − 745.81

step (1)

```
  999.99
  745.81
---------
  254.18
       1
---------
  254.19
```

Step (2)

```
  436.62
  254.19
---------
  690.81  → no carry answer is -ve
```

Step (3)

```
  999.99
  690.81
---------
  309.18
       1
---------
  309.19
```

15's complement Method :-

Q Find the 15's complement of the following numbers

(a) 6A36

Sol FFFF
(-) 6A36
95C9

(or)

15 15 15 15
6 A 3 6
9 5 C 9

(b) 9AD.3A

15 15 15 . 15 15
9 A D . 3 A
65 2 . C 5 → (15's complement)

⇒ 15's complement method of subtraction

(a) 69B − C14

Step① 15's complement of (−C14)

15 15 15
C 1 4
3 E B → (15's complement of (−C14)

Step② 69B − C14 = 69B + (15's complement of (−C14)

1 1
6 9 B
(+) 3 E B
A 8 6

$(22)_{10} = (16)_H$
$(24)_{10} = (18)_H$

There is no carry, result is (-) ve

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

```
  15  15  15
   A   8   6
-------------
   5   7   9
-------------
```

∴ Final result is $-(579)_{16}$

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

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

```
  15  15  15
 - 6   9   B
-------------
   9   6   4  → 15's complement
-------------
```

Step (2)

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

```
   C14
+  964
-------
 (1) 578
-------
```

$(21)_{10} = (15)_{16}$

carry

There is a carry, so the result is true

```
  578
+   1   (End around carry)
-------
  579
-------
```

∴ 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

15 15 15

A 8 C

5 7 3 → 15's complement

16's complement → 573
⊕1
574

⇒ 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)

15 15 15

C 1 4

3 E B → 15's complement

16's complement → 3EB
1
3EC

Step ② :-

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

C 9 B
3 E C
1 0 8 7

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's complement of $(-3B2.3C)$

15 15 15. 15 15
− 3 B 2 3 C

C 4 D . C 3 → 15's complement

16's complement is given by,

C 4 D . C 3
1

C 4 D . C 4 → 16's complement

step ②

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

2A4.2D
C4D.C4

EF1.FT → intermediate result

there is no carry. So the result is (+)ve

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

```
  15  15  15 . 15  15
   E   F   1 .  F   1
-------------------------
   1   0   E . 0   E  → 15's complement
               ⊕   1
-------------------------
   1   0   E . 0   F  → 16's complement
```

∴ 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

```
  777.77
  135.74
-----------
  642.03  → 7's complement
```

∴ 234.65 + (7's complement of (−135.74))

```
  234.65
  642.03
-----------
(1) 076.70
```

carry

$(8)_{10} = (10)_8$

(carry ⇒ result is positive)

076.70
(+) 1
76.71 (End around carry)

∴ Result is +76.71

(B) 135.74 − 236.65

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

777.77
236.65
541.12 → 7's complement

∴ 135.74 − 236.65 = 135.74 + (7's complement of (−236.65))

⇒ 135.74
541.12 → (7's complement)
677.06 → Intermediate result

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

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

777.77
⊖ 677.06
100.71

∴ Final result is — 100.71

Subtract the following using 8's complement method:

(a) 246.31 − 162.45

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

777.77
162.45
615.32 → (7's complement)
⊕ 1
615.33 → (8's complement)

∴ 246.31 − 162.45 = 246.31 + (8's complement of (−162.45))

⇒ 246.31
615.33
(1) 063.64
carry

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

∴ Final result is +63.64

(b) 162.45 − 246.31

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

777.77
⟷ 246.31
531.46 → (7's complement)
⊕ 1
531.47 → (8's complement)

∴ 162.45 − 246.31 = 162.45 + (8's complement of (−246.31))

⇒ 162.45
531.47
714.14 → (Intermediate result)

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

777.77
714.14
063.63 → 7's complement
(+) 1
063.64 → 8's complement

∴ Final result is − 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)

Mantissa = 123.154

Exponent = 5

Base = 10 (decimal)

Eg:-

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

② mass of electron $= 9.1 \times 10^{-28}$ gm.

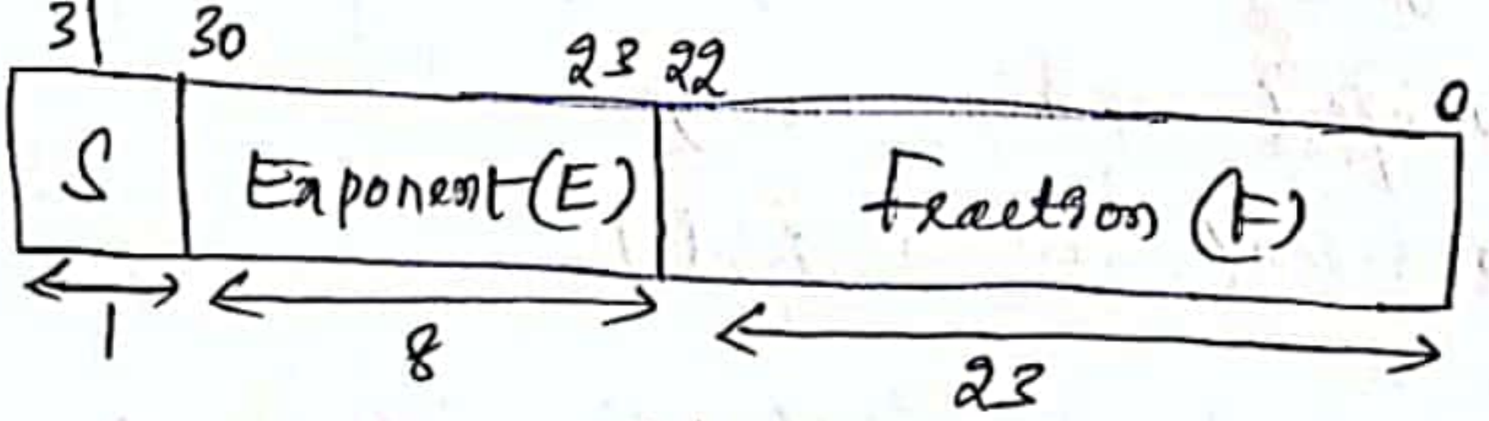
Ex:- Given the number 732.136×10^7

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.

Ex°

$(11.8)_{10} = (1011.11001\ldots)_2$

$= (1.01111001)_2 * 2^3$

$= (1.01111001)_2 * 2^{(11)_2}$

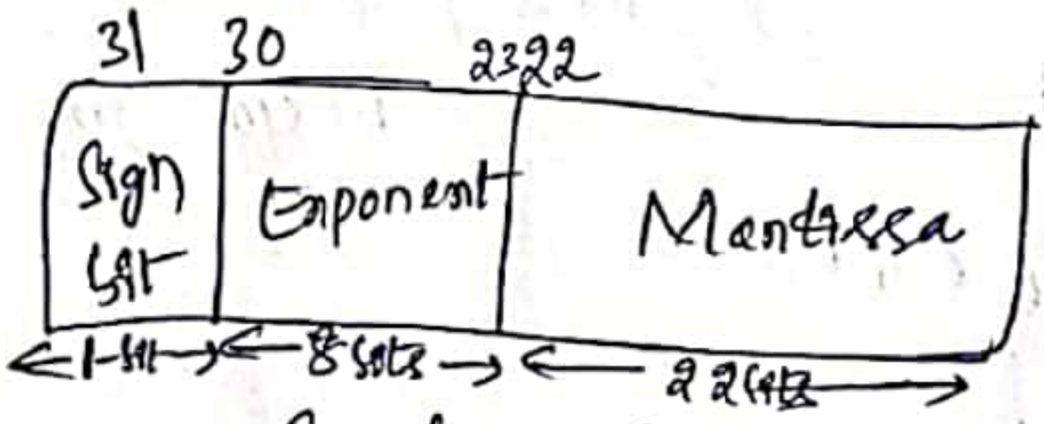
Decimal representation

$12345 = 1.2345 \times 10^4$ → Exponent

1.2345 ↓ mantissa (or) Significant

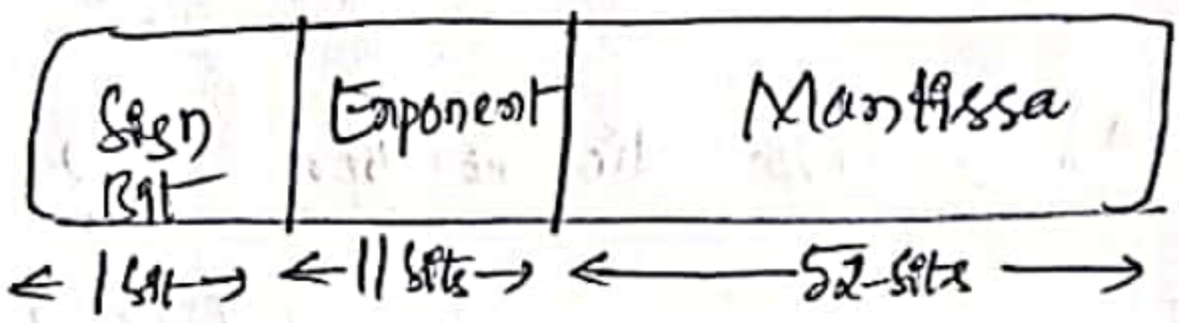
10 ↓ Base

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



Single precision

32-bits



Double precision

64-bits

(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)

$$\begin{cases} N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, & 1 \le \text{exponent} \le 254 \\ N = (-1)^S \times 0.\text{fraction} \times 2^{\text{exponent}-126}, & \text{exponent} = 0. \end{cases}$$

* Value $= (-1)^S \times (1+F) \times 2^{E-127}$. (or)

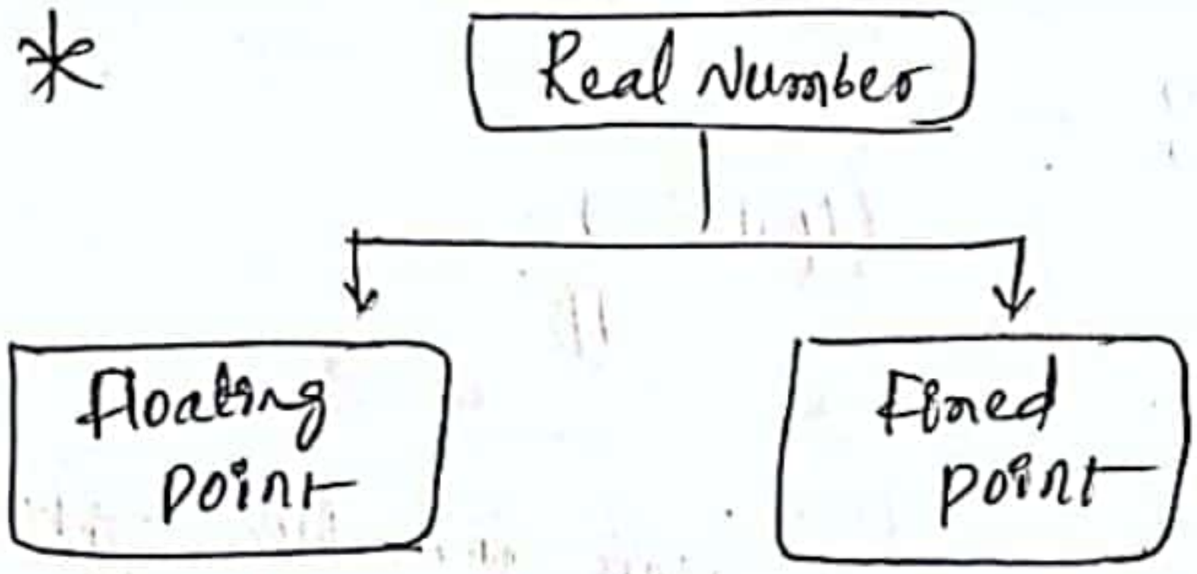
Single precision ↑

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

↑ double precision

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

Fixed point Representation	Floating point Representation
① A representation of real data type for a number that has a fixed number of digits after the radix point.	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 limited range of values.	② Used to represent a wide range of values.
③ High performance	③ High performance
④ Less flexible	④ More flexible



* Limitation of floating points
✓ Size of mantissa is fixed.

Sol Use a floating point format with a larger mantissa.
double (53 bits) long double (64 bits)

Problems:-

Q What is the 111.11 in decimal
(a) 7.75
(b) 21
(c) 7.375
(d) 15.25

Q What is 8.5 in binary
(a) 11111111.11111
(b) 1000.01
(c) 0.100011
(d) 1000.10.

Ex:- −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

133 in binary

10000101

= 1.1100 10101 × 2^6

127
6
133

∴ 1 10000101 11001010 1.

Right shift

Ex:-

Sign	Exponent	Mantissa
0	1001010	11101000

0100 1010 1110 1000

4 10 14 8

A E

= (4AE8)H

Ex:- 0.0001100 1100 1100 1100 1100 1100 2 representation floating point in 32-bits.

Sol 1.100 1100 1100 1100 1100 1100 × 2^-4

128 64 32 16 8 4 2 1

0 1 1 1 1 0 1 1

= 132

Exponent = −4 + 127 = 123

Sign bit = 0

Mantissa = 100 1100 1100 1100 1100 1100

S (bit)	Exponent 8 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 number may be positive (or) negative. Different formats are used for representation signed binary numbers. They are

① Signed magnitude representation

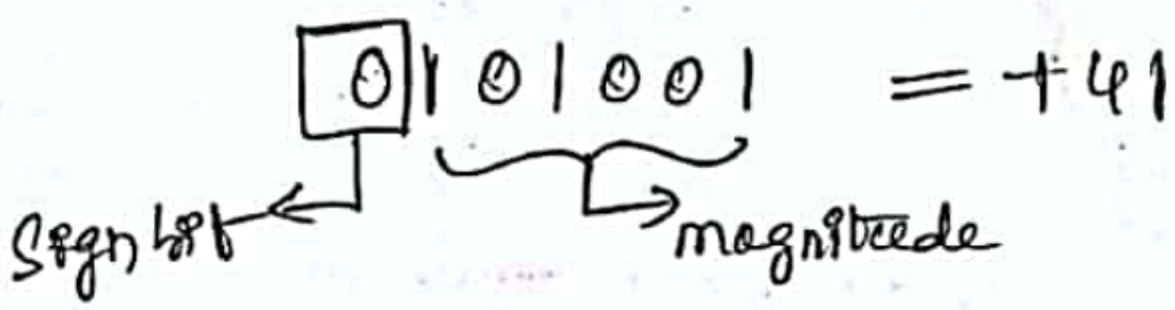
② 1's complement representation

③ 2's complement representation

① Sign magnitude representation :-

In signed magnitude form, an additional bit called the sign bit is placed infront 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 :-



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

② 1's complement representation :-

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

0 1100 11 = +51; MSB = 0 for +ve

↑ Sign bit

1 0011 00 = −51; MSB = 1 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 number 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 the 1's complement number.

0 1 1 0 0 1 1 = +51

↑ Sign bit — magnitude

1 0 0 1 1 0 1 = −51 [In sign 2's complement form]

↑ Sign bit — magnitude

Number system	+9	−9
Unsigned	+1001	−1001
Sign magnitude	Sign bit 0 1001	Sign bit 1 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 sign magnitude, sign 1's complement and sign 2's complement representation.

Sol

	+51	−51
Sign magnitude	0 110011	1 110011
	Sign bit	Sign bit
Sign 1's complement	0 110011	1 001100
	Sign bit	Sign bit
Sign 2's complement	0 110011	1 001101
	Sign bit	Sign bit

Q Represent +43 and −43 in sign magnitude, sign 1's complement and 2's complement representation

Sol

	+43	−43
Sign magnitude	0 101011	1 101011
	Sign bit	Sign bit
Sign 1's complement	0 101011	1 010100
	Sign bit	Sign bit
Sign 2's complement	0 101011	1 010101
	Sign bit	Sign 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.

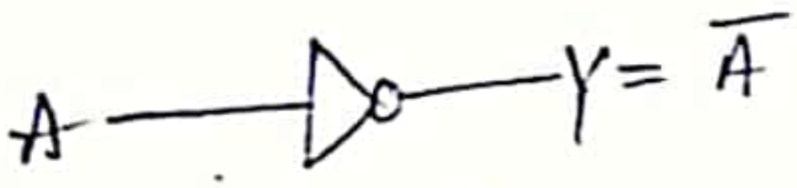
→ Minimization of logic expressions can be done by using boolean theorems and laws.

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

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

⇒ Logic Gates :-

① NOT gate (or) Inverter :-

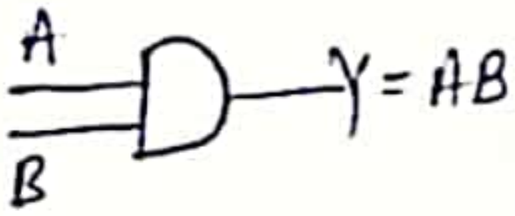


Symbol

Truth table

A	Y = Ā
0	1
1	0

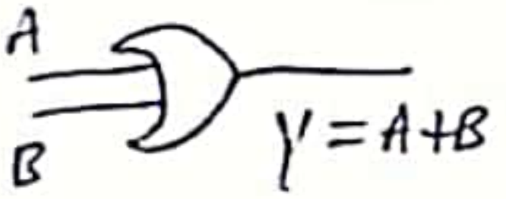
② AND Gate :-



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

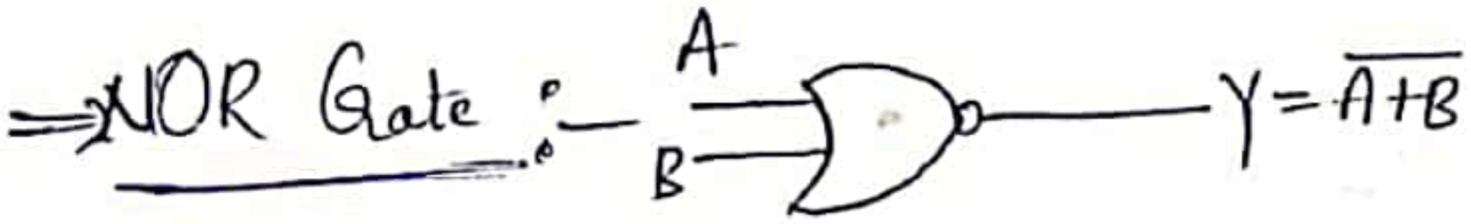
∴ these 3 gates are primary gates

③ OR Gate :-



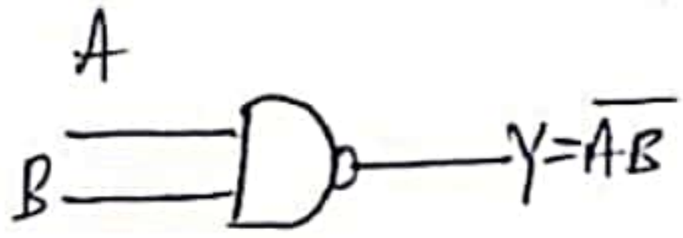
A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

⇒ NOR Gate :-



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

⇒ NAND Gate :-

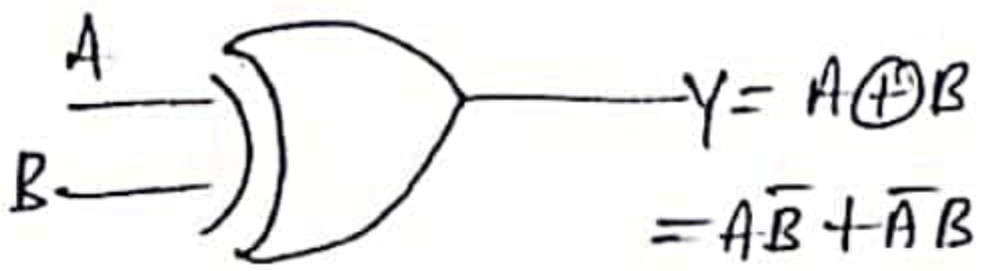


A	B	$Y = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

∴ The above two gates are universal gates

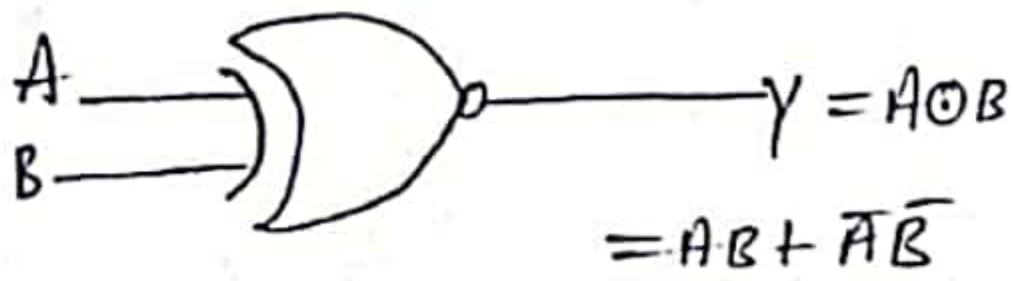
Special Gates :-

⇒ EX-OR gate :-



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

⇒ EX-NOR gate :-



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

⇒ Laws of Boolean Algebra :-

AND Operation	OR Operation	NOT Operation
1) $0 \cdot 0 = 0$	5) $0 + 0 = 0$	9) $\bar{0} = 1$
2) $0 \cdot 1 = 0$	6) $0 + 1 = 1$	10) $\bar{1} = 0$
3) $1 \cdot 0 = 0$	7) $1 + 0 = 1$	
4) $1 \cdot 1 = 1$	8) $1 + 1 = 1$	

⇒ Complement Law

$\overline{0} = 1$

$\overline{1} = 0$

If $A = 0$ then $\overline{A} = 1$

If $A = 1$ then $\overline{A} = 0$

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

AND Law

$A \cdot 0 = 0$

$A \cdot 1 = A$

$A \cdot A = A$

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

proof

$= A \cdot A$

$= A \cdot A + 0$

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

$= A(A + \overline{A}) = A(1) = A$

⇒ OR Law :-

$A + 0 = A$

$A + 1 = 1$

$A + \overline{A} = 1$

$A + A = A$

proof

$A + A = A$

$(A + A)\,1$

$A + A\,(A + \overline{A})$

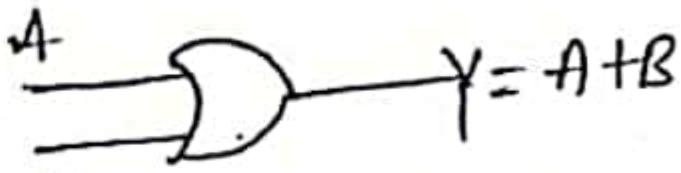
$A + A \cdot A + A \cdot \overline{A}$

$= A + A \cdot A + 0$

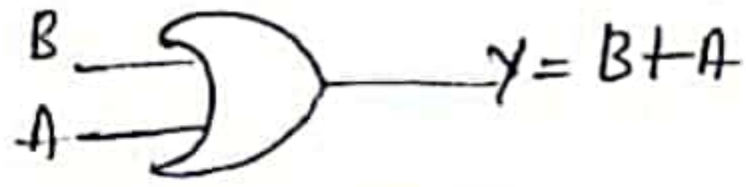
$= A(1 + A) = A \quad (1 + A = 1)$

⇒ Commutative Law :-

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

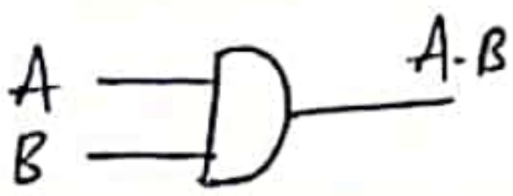


A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

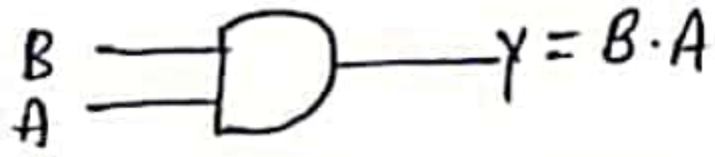


B	A	B+A
0	0	0
0	1	1
1	0	1
1	1	1

Law ② :- $A \cdot B = B \cdot A$



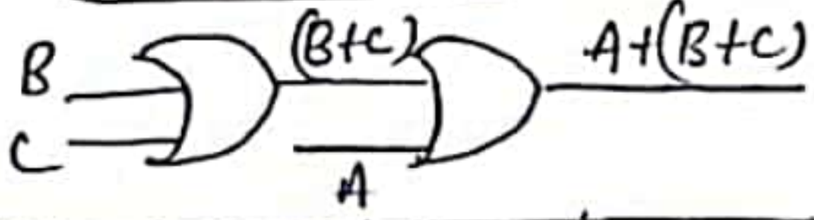
A	B	A·B
0	0	0
0	1	0
1	0	0
1	1	1



B	A	B·A
0	0	0
0	1	0
1	0	0
1	1	1

⇒ Associative Law :-

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

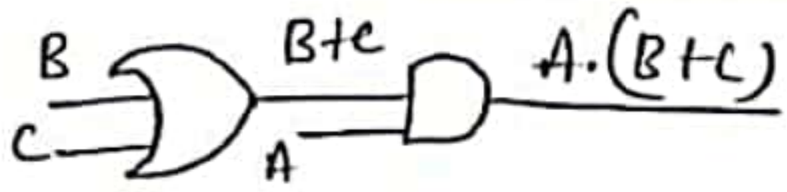


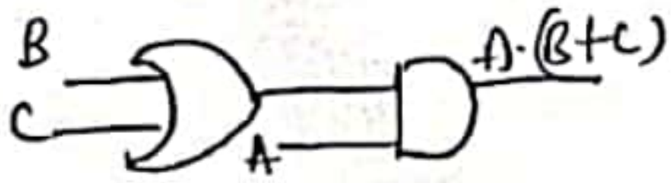
A	B	C	(B+C)	A+(B+C)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

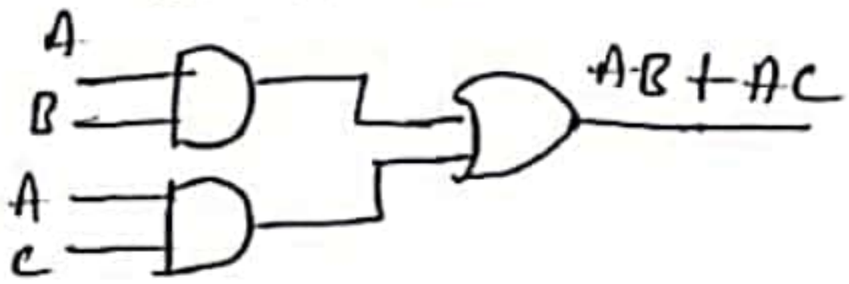
A	B	C	(A+B)	(A+B)+C
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

⇒ Distributive Law :-

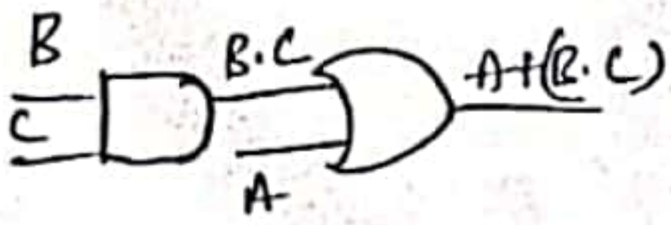
① $A \cdot (B+C) = AB + AC$

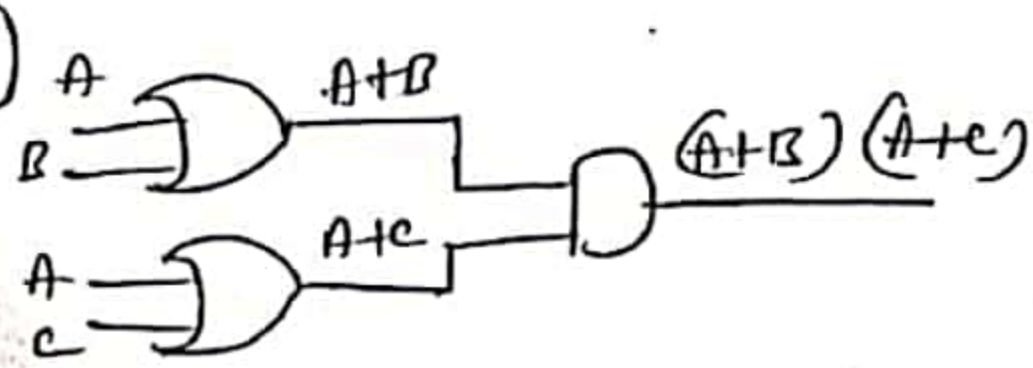






② $A+(B \cdot C) = (A+B)(A+C)$





⇒ Consensus Theorem :-

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

$$L.H.S = AB + \bar{A}C + BC$$

$$= AB + \bar{A}C + BC(A + \bar{A}) \qquad \because A + \bar{A} = 1$$

$$= AB + \bar{A}C + BCA + BC\bar{A} \qquad 1 + C = 1$$

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

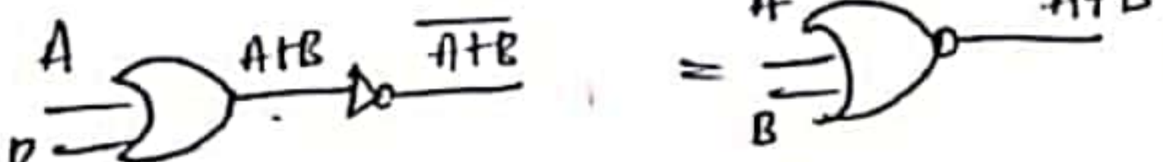
$$= AB(1) + \bar{A}C(1)$$

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

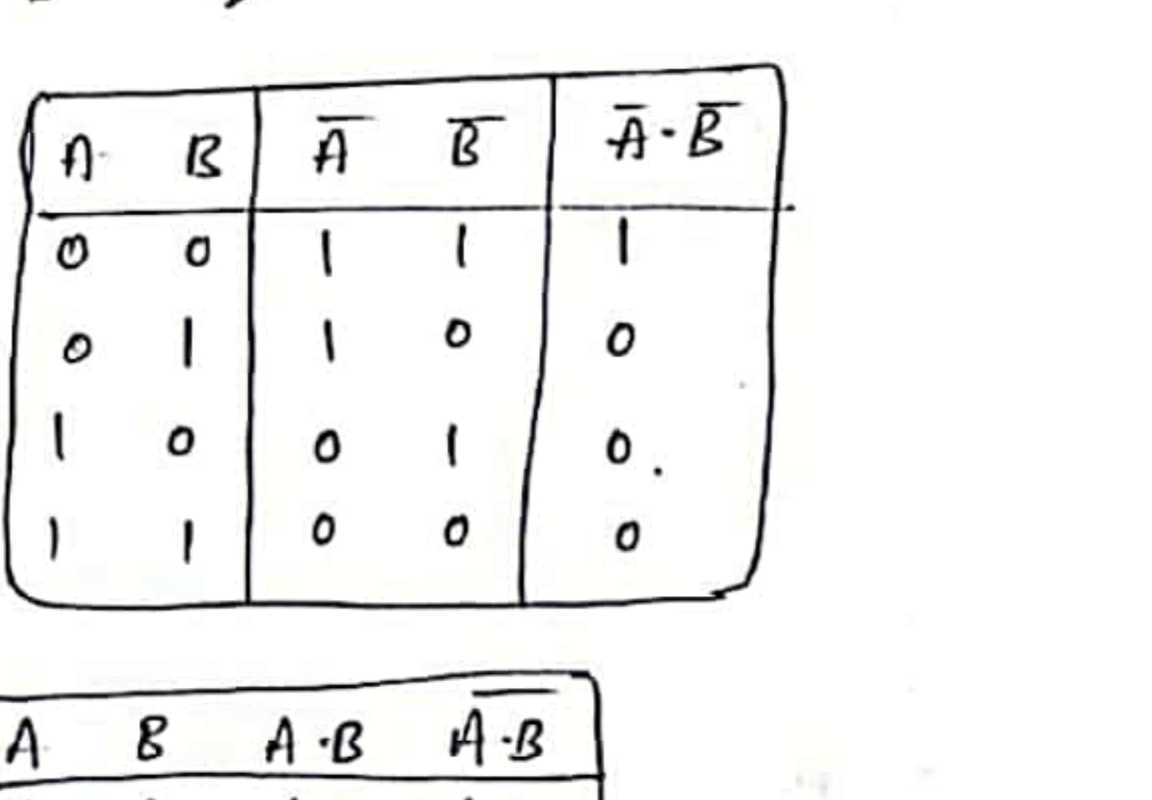
$$= R.H.S \qquad \therefore \boxed{L.H.S = R.H.S}$$

⇒ Demorgan's Theorem :-

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

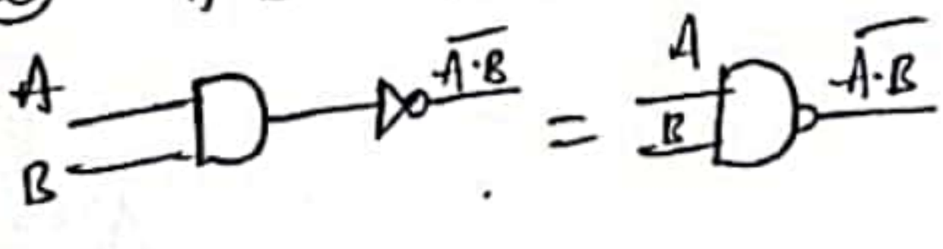


A	B	A+B	$\overline{A+B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

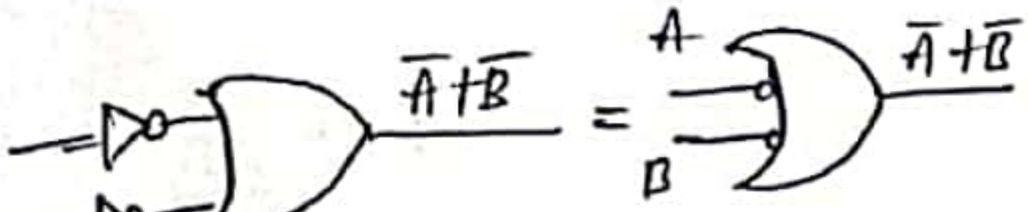


A	B	$\bar{A}$	$\bar{B}$	$\bar{A} \cdot \bar{B}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

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



A	B	A·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 '.' sign and '.' sign to '+' sign. Variables (A, B) also complemented.

Note: ① $\underbrace{AB+AB+AB+AB}_{\text{parenthesis}} = AB$ (Same no of variables are repeated we may considered single term)

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

③ $AB\bar{C} + AB\bar{C} + AB\bar{C}D$

$AB\bar{C}(1+D) = AB\bar{C}(1) = AB\bar{C}$.

④ $AB\bar{C}0 + AB\bar{C}\bar{D}$

$= AB\bar{C}(D+\bar{D})$

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

① $f = A+B\left[A\bar{C} + (B+\bar{C})D\right]$

$= A+B\left[A\bar{C} + BD + \bar{C}D\right]$

$= A + \left[ABC + \underbrace{BBD}_{\text{repeated terms; we can consider as single.}} + B\bar{C}D\right]$ $\left| \because BBD = BD \right.$

$= A(1+BC) + BD(1+\bar{C})$

$\boxed{f = A+BD}$ $(\because 1+BC = 1 \; ; \; 1+\bar{C} = 1)$

② $f = \left(\overline{A+B\bar{C}}\right) \cdot (A\bar{B} + ABC)$

$= \bar{A} \cdot \overline{B\bar{C}} \; (A\bar{B} + ABC)$

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

$= \bar{A}BC A\bar{B} + \bar{A}ABCBC$

$= \bar{A}A B\bar{B}C + \bar{A}A BBCC$

$= 0 \; 0 + 0$

$\boxed{f = 0}$

$\left| \because A\bar{A} = 0 \; ; \; B\bar{B} = 0 \right.$

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

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

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

Q Find the complements of the following expressions

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

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

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

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

→ Karnaugh Map (K-map) Representation :-

0

1

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

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

(1) Sum of product (SOP) :- This is also called as disjunctive Normal form (DNF). Variables present in this variables are called 'minterms' $(m_0, m_1, m_2 \cdots)$

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

(

→ Standard SOP form :- (SSOP) · It is also called as Disjunctive Canonical form (DCF)

(2) product of sum (POS) :- This is also called as conjunctive Normal Form (CNF). Variables present in this form is called 'Maxterm' $(M_1, M_2, M_3 \cdots)$

Ex:- $F(A,B,C) = \Pi(M_1, M_2, M_6, M_7)$

$= \Pi M(1,2,6,7)$

⇒ Standard POS form :- This form is also called as conjunctive Canonical form (CCF)

Ex:- $f(A,B,C) = (\bar{A}+\bar{B})(A+B)$

$= (\bar{A}+\bar{B}+C\cdot\bar{C})(A+B+C\cdot\bar{C})$ $\because C\cdot\bar{C}=0$.

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

Ex:- Convert SOP to Standard SOP form

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

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

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

Repeated ABC product is there. we should write only one

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

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)

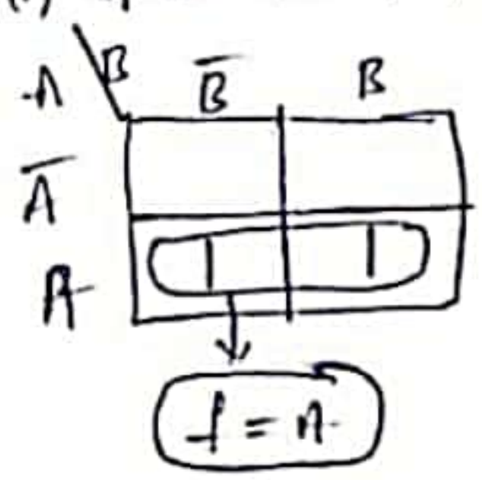
Rules for K-map minimization :-

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.

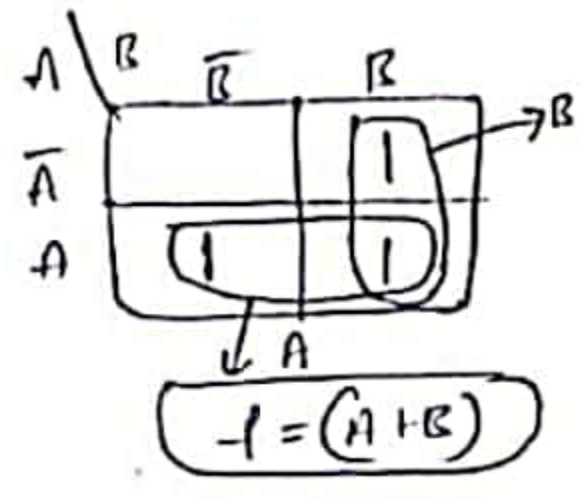
Q Problems on K-map representation

2-variable K-map (SOP)

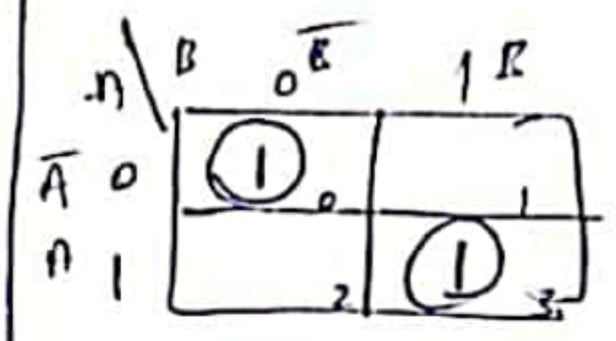
i, $f = A\bar{B} + \bar{A}B$



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



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

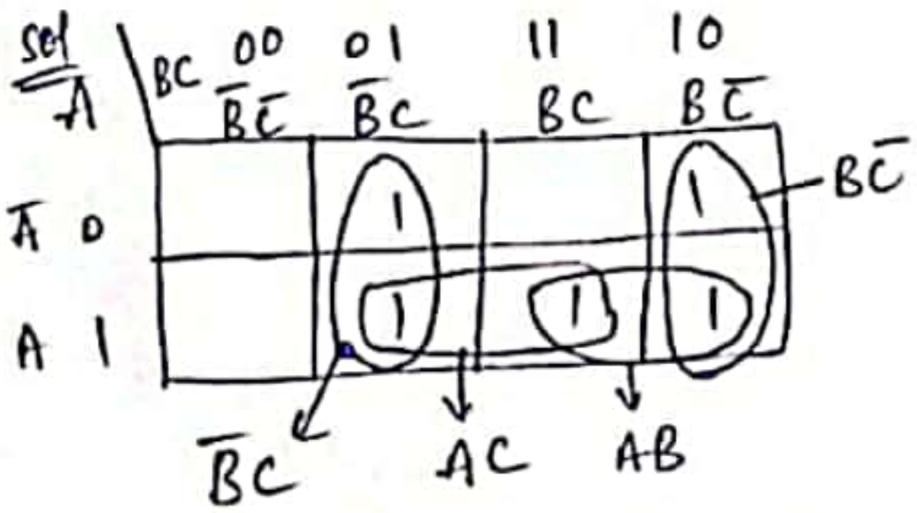


$= \bar{A}\bar{B} + AB.$

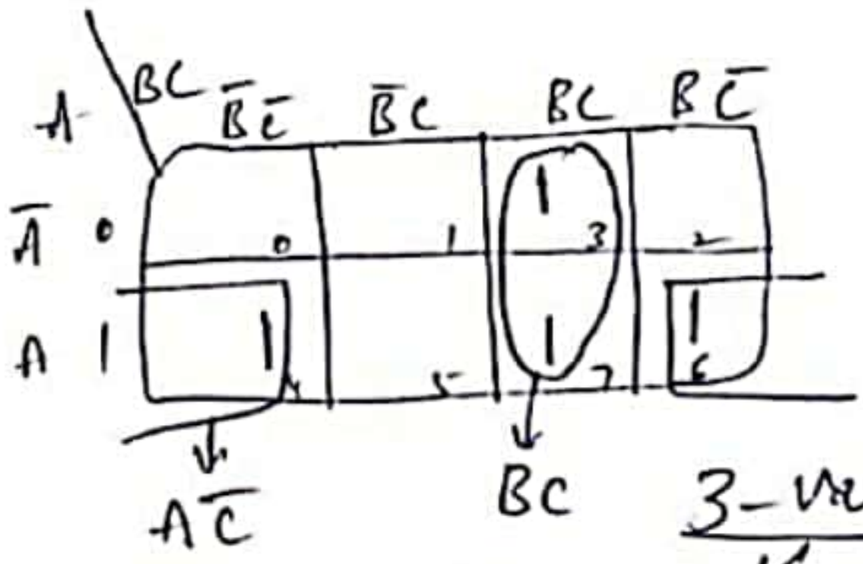
Q $f = \bar{A}\bar{B}\bar{C} + A\bar{B}C + \bar{A}BC + ABC + ABC$

$f(A,B,C) = Sm(3,4,6,7)$

Sol



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

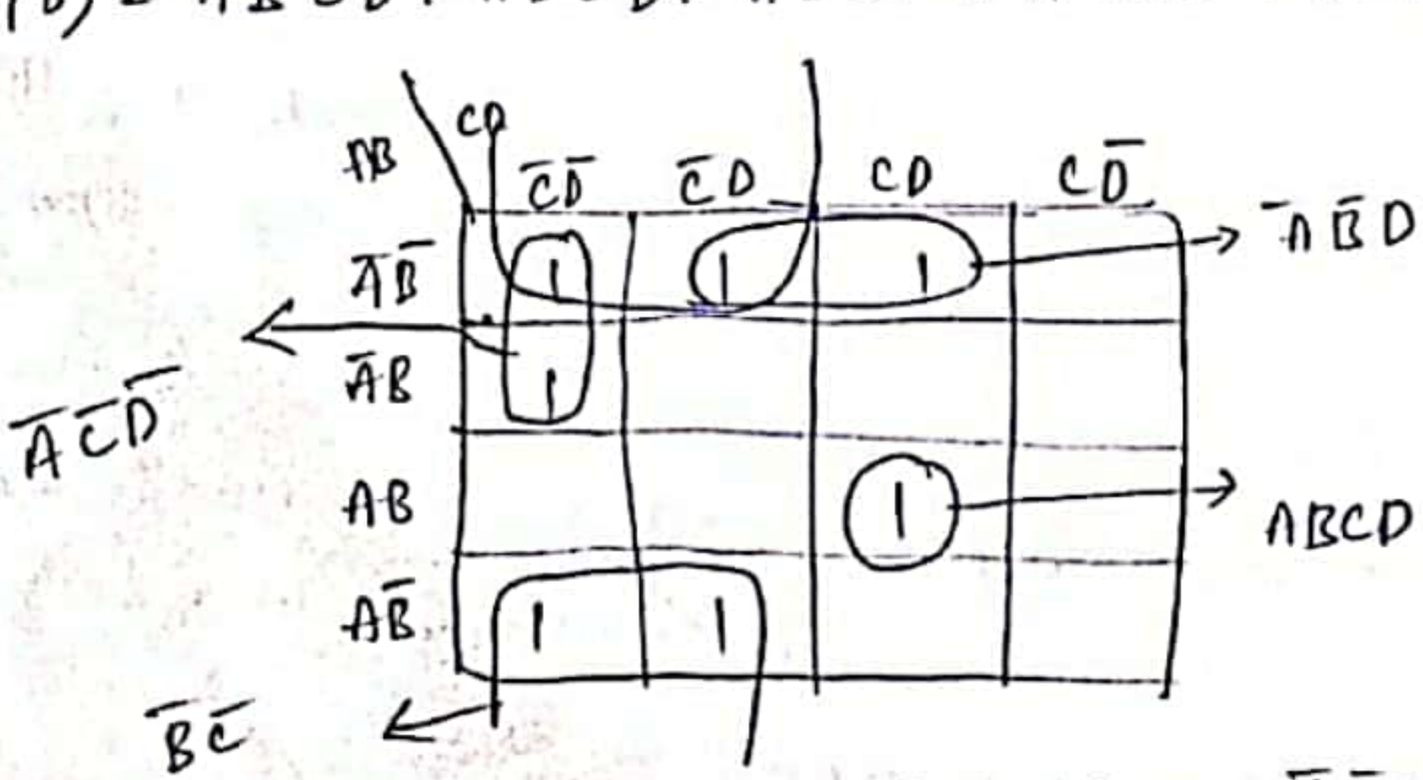


$f = BC + A\bar{C}$

3-variable K-map

Q Reduce the SOP expression using 4 variable K-map

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



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

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

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

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

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

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

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

$= \bar{A}B + AB$
$\downarrow\downarrow \quad \downarrow\downarrow$
$0\,1 \quad 1\,1$

$= m_1 + m_3$

$= \Sigma m(1,3)$

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

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

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

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

$= ABC + AB\bar{C} + ABC + AB\bar{C}$ $\because C+\bar{C}=1$
$\downarrow\downarrow\downarrow \quad \downarrow\downarrow\downarrow \quad \downarrow\downarrow\downarrow \quad \downarrow\downarrow\downarrow$
$1\,1\,0 \quad 1\,0\,1 \quad 1\,1\,1 \quad 1\,1\,0$
$6 \quad 5 \quad 7 \quad 3$

$= m_6 + m_5 + m_7 + m_3$

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 maxterms.

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

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

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

$= (A+B)(A+\bar{B})(\bar{A}+\bar{B})$
$\downarrow\downarrow \quad \downarrow\downarrow \quad \downarrow\downarrow$
$0\,0 \quad 0\,1 \quad 1\,1$

$= M_0\ M_1\ M_2$

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

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

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

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

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

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

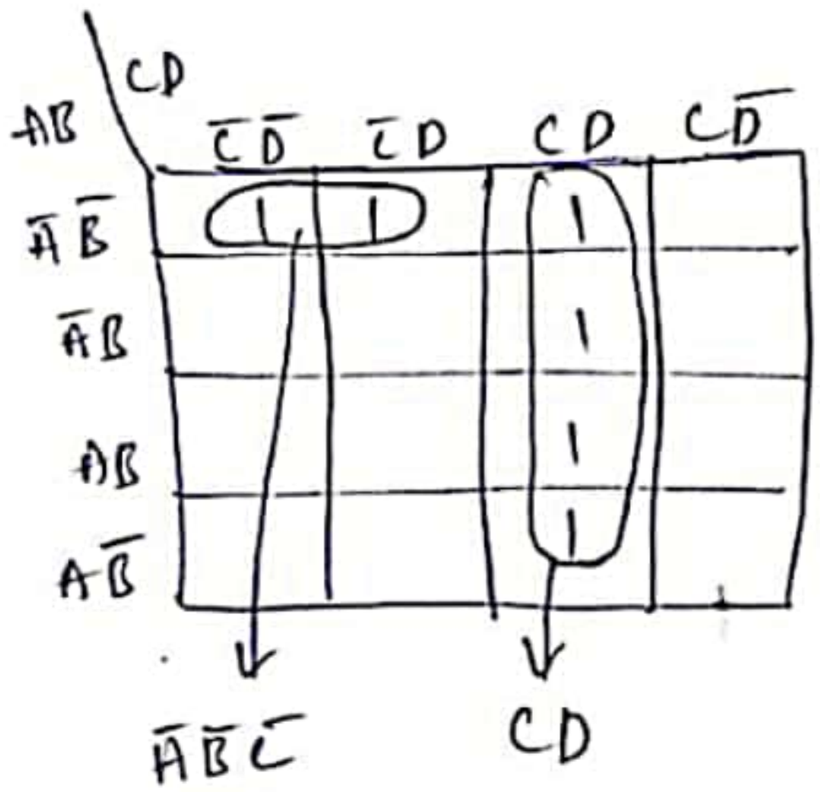
$= (A+B)(A+\bar{B})(\bar{A}+B)$ $\because (A+B)(A+B) = (A+B)$
$\downarrow\downarrow \quad \downarrow\downarrow \quad \downarrow\downarrow$ $(A+\bar{B})(A+\bar{B}) = (A+\bar{B})$
$0\,0 \quad 0\,1 \quad 1\,0$ $(\bar{A}+B)(\bar{A}+B) = (\bar{A}+B)$

$M_0 \cdot M_1 \cdot M_2 = \Pi M(0,1,2)$

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.

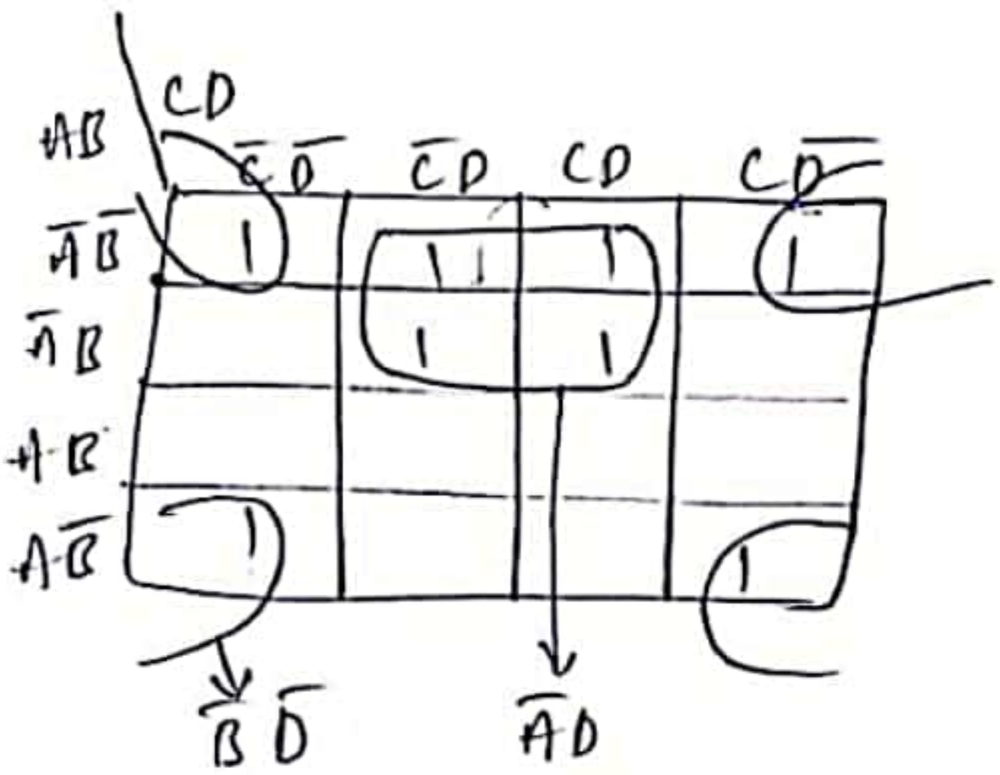
This is extremely useful and extensively used in the minimization of function of 2, variable K-map, 3-variable K-map, 4-variable K-map and so on.

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



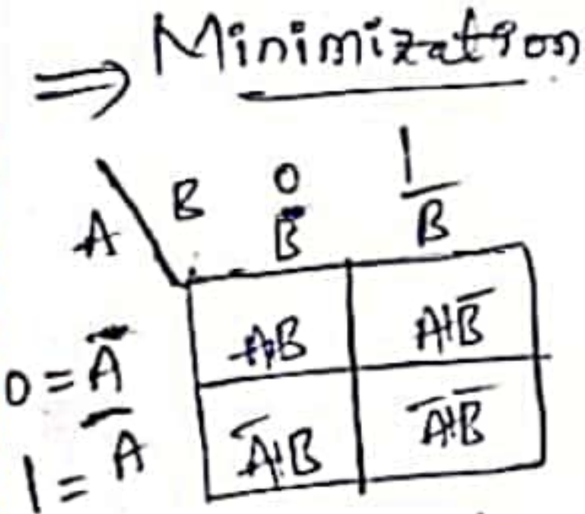
$\therefore f = \bar{A}\bar{B}\bar{C} + CD$

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



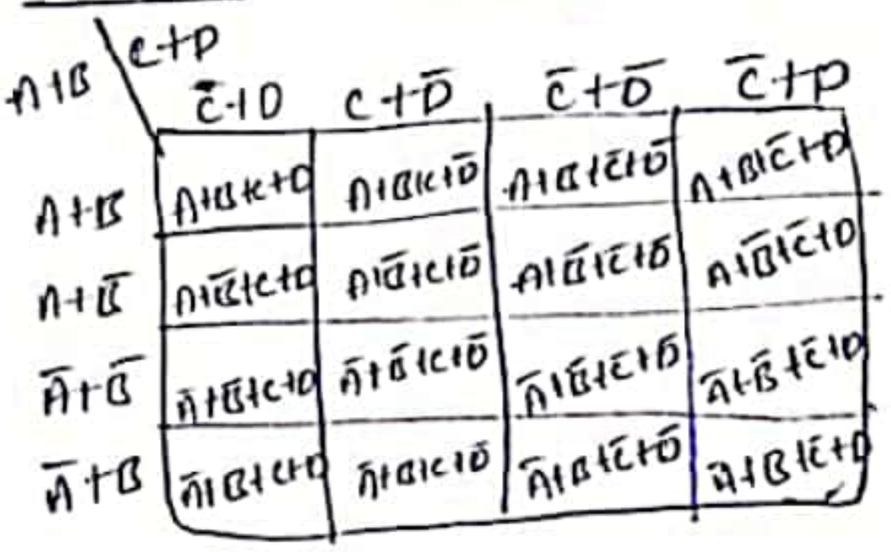
$\therefore f = B\bar{D} + \bar{A}D.$

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

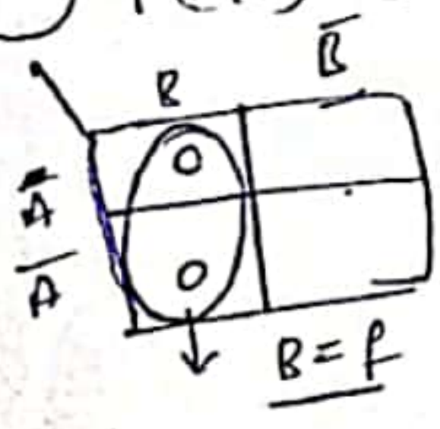


2-variable

3-variable

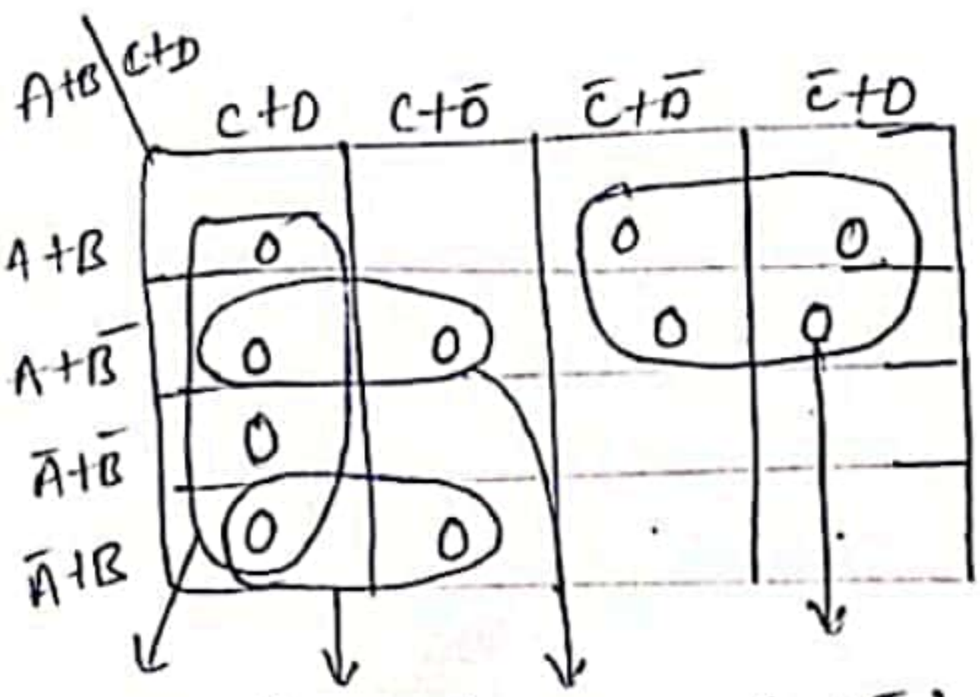


Q $f(A,B) = (A+B)(\bar{A}+B)$



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

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



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

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

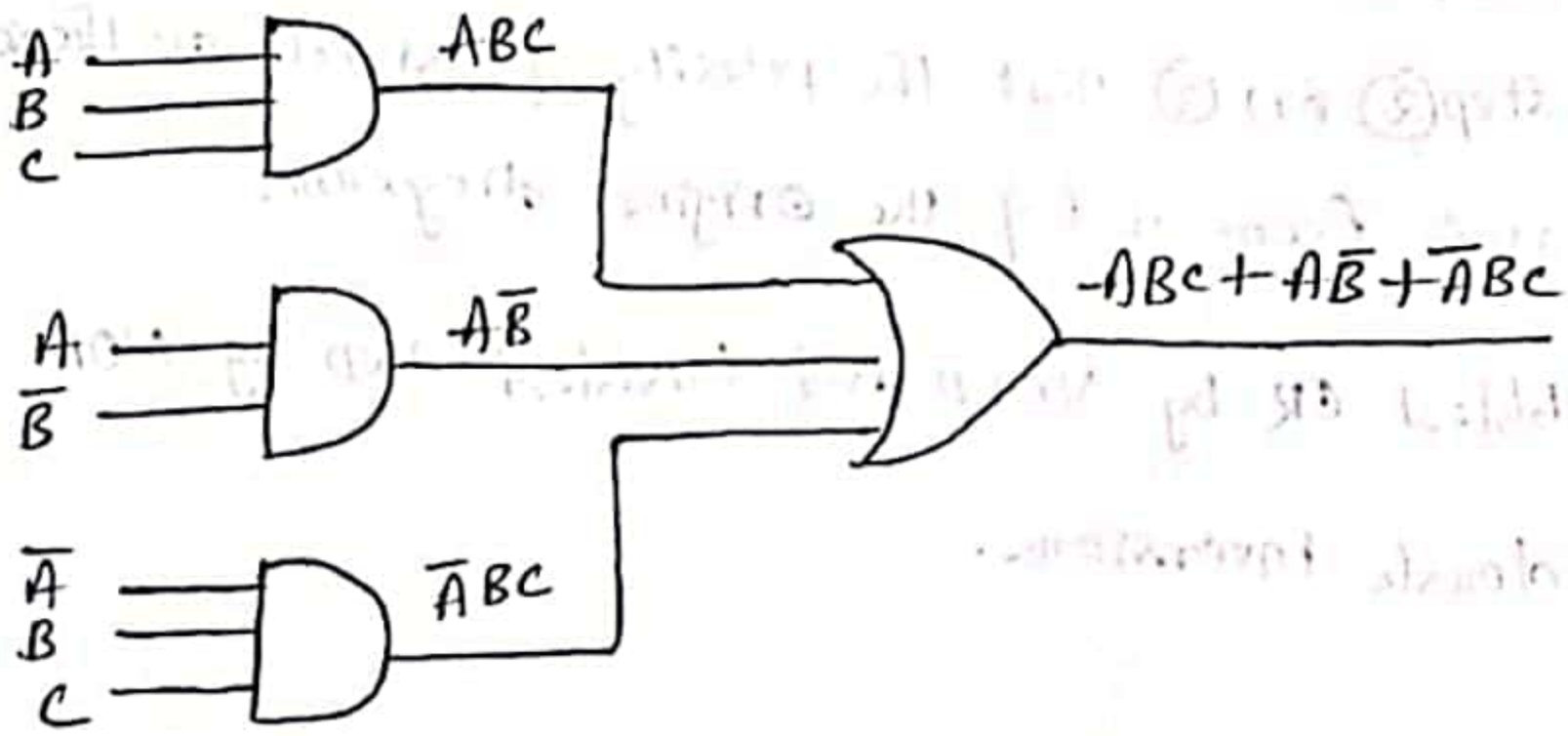
126

⇒ XIAND and XIOR Implementation :-

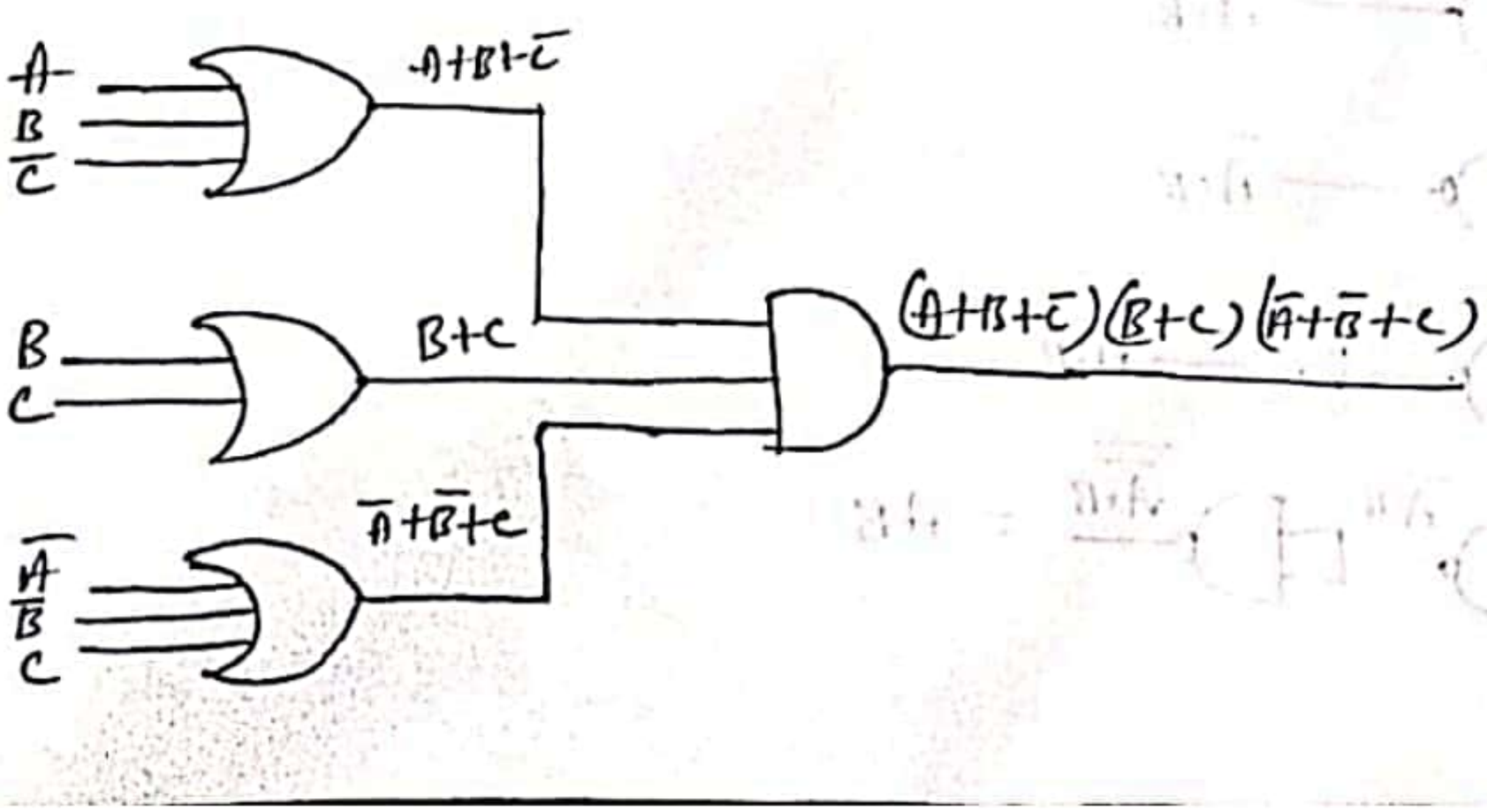
In the design of digital circuits, the minimal boolean expressions are usually obtained in SOP form (or) POS form. Sometimes the minimal expressions may also be expressed in hybrid form.

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 choosen, add a circle at the output of each AND gate and at the inputs to all the OR gates.

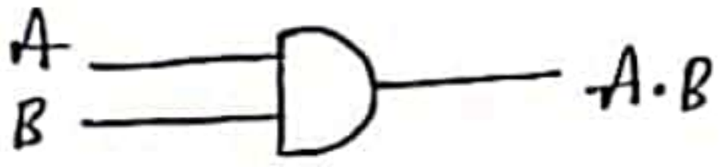
(3) If NOR hardware is choosen, add a circle at the output of each OR gate and at the inputs to all the AND gates.

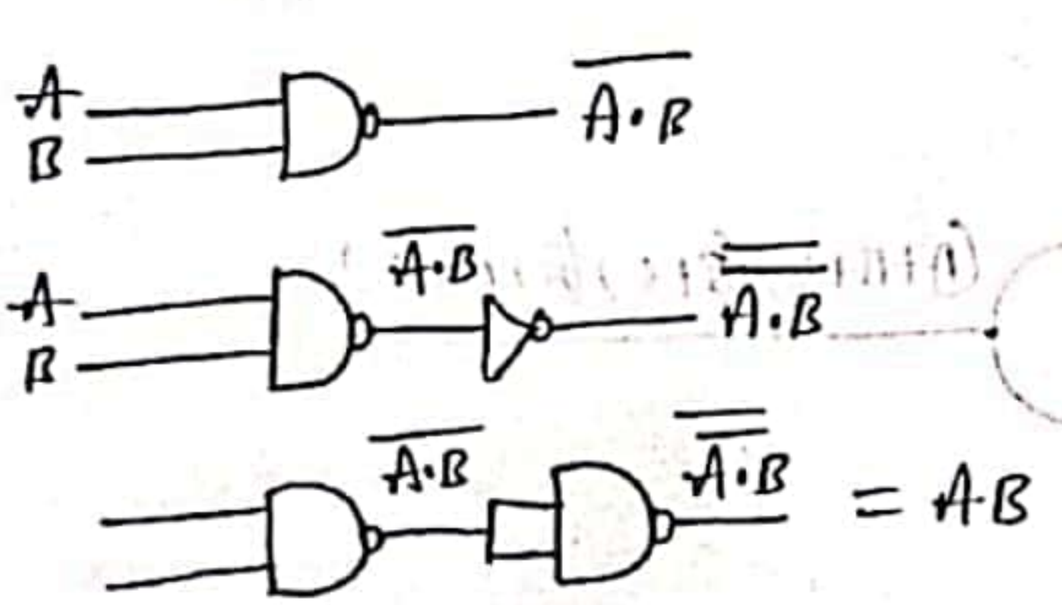
(4) Add (or) Subtract an inverter on each line that received a circle in Step (2) (or) (3) that the polarity of signals on those lines remains from that of the original diagram.

(5) Replace bubbled OR by NAND and bubbled AND by NOR.

(6) Eliminate double inversions.

(1) Implementation of AND gate using NAND Gate :—

(1) 

(2) 

④ OR gate using NOR

① A, B → A+B

② A, B → $\overline{A+B}$

③ A, B → $\overline{\overline{A+B}}$

④ A, B → $\overline{\overline{A+B}} = A+B$

⑤ Realization of NOT gate using NAND & NOR

① A → $\overline{A}$

② A → $\overline{A}$

③ A → $\overline{A \cdot A} = \overline{A}$

Single i/p AND gate.

⇒ Using NOR

A → $\overline{A}$

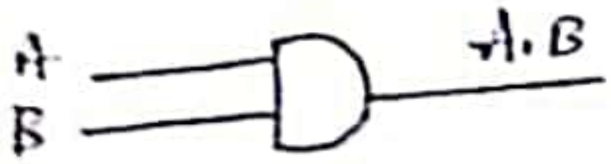
A → $\overline{A}$

A → $\overline{A+A} = \overline{A}$

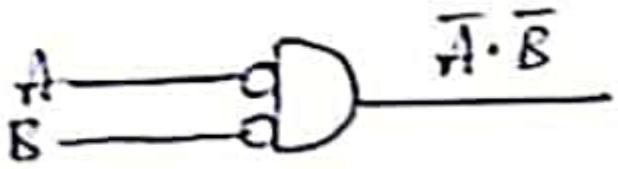
Single i/p NOR gate.

(2) Realization of AND gate Using NOR Gate :-

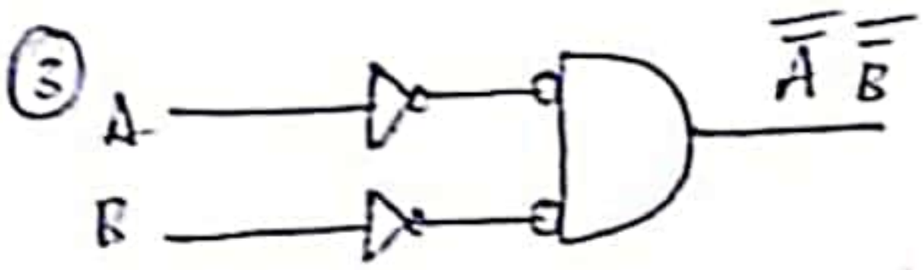
(1)



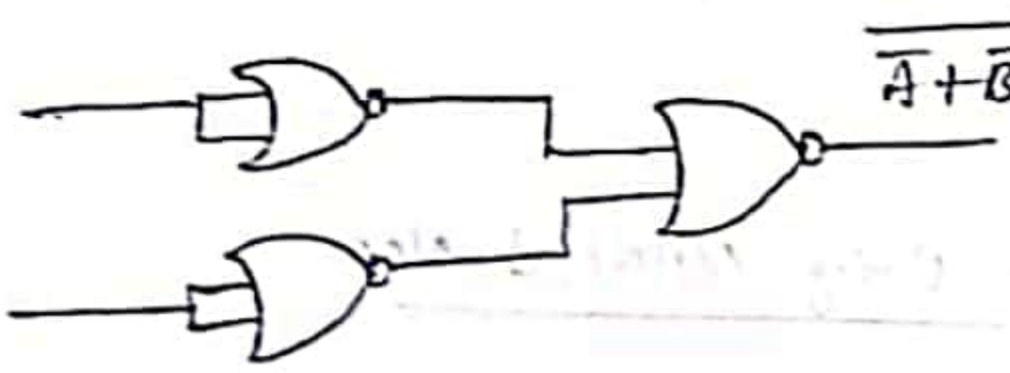
(2)



(3)



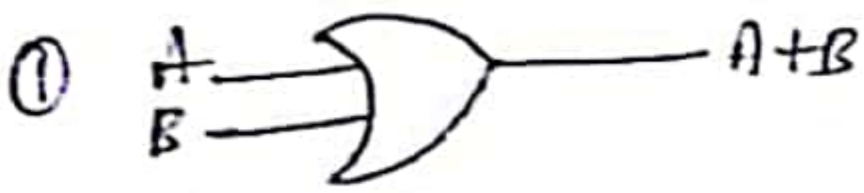
(4)



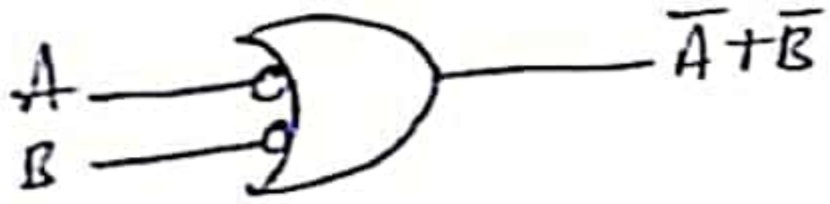
$= \overline{\overline{A}} \cdot \overline{\overline{B}} = A \cdot B$

(3) Realization of OR gate Using NAND & NOR gates :

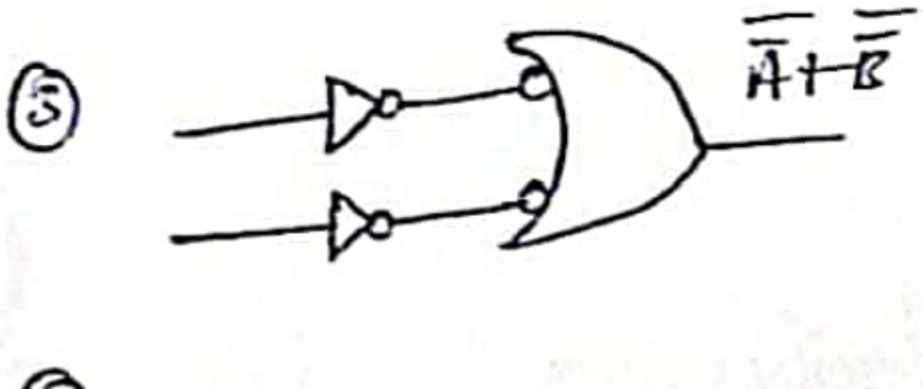
(1)



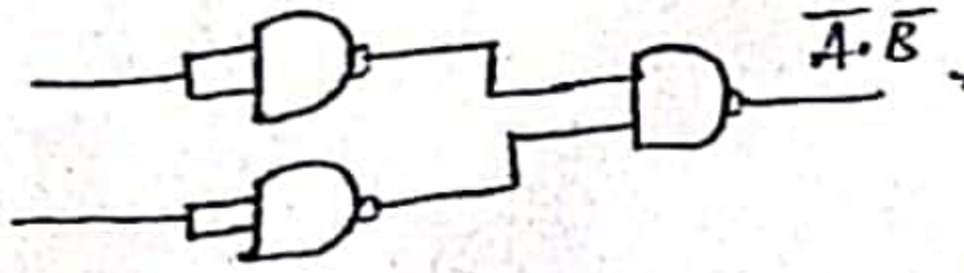
(2)



(3)



(4)



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

Realization of EX-OR gate using NAND & NOR using NAND gate :-

(130)

(1) A, B → $Y = \overline{A}B + A\overline{B} = A \oplus B$

(2) A, B → $\overline{A}$, B → $\overline{A} \cdot B$; A, B → $A \cdot \overline{B}$; $\overline{B}$

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

(3) A, B → $\overline{A \cdot B}$; A, B → $\overline{A \cdot B}$

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

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

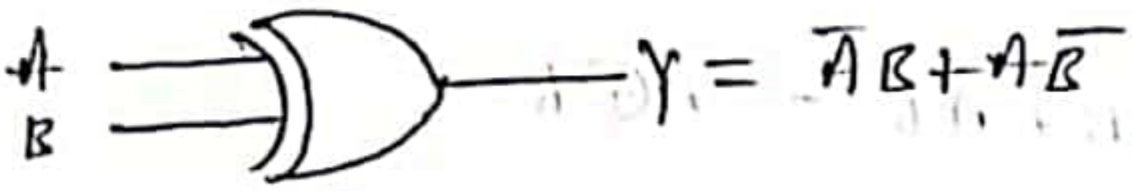
(4)

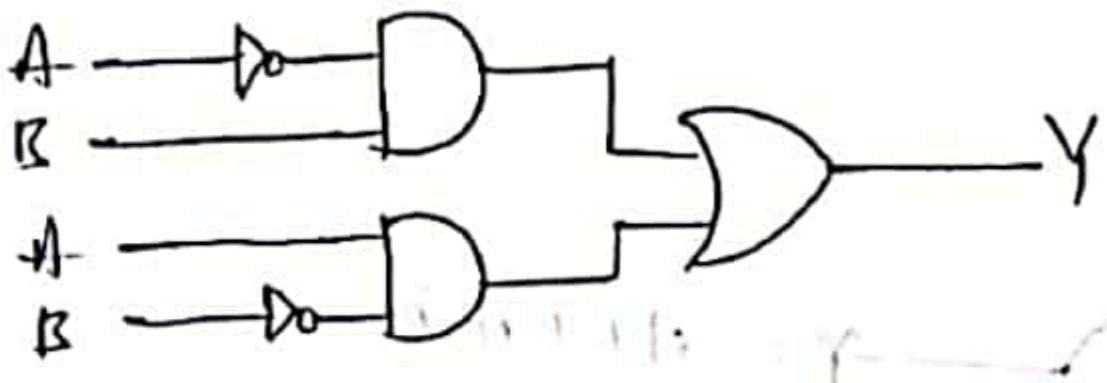
(5) Instead of not gate place single I/p NAND gate and, Instead of bubbled OR gate place NAND gate

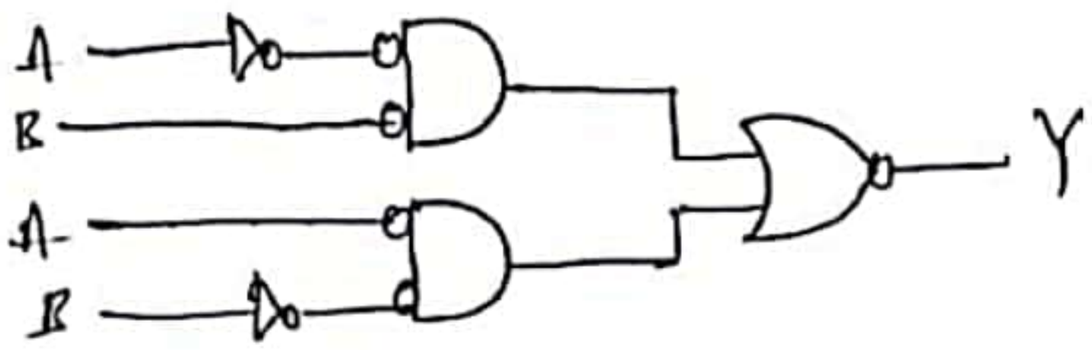
A, B → A, B → Y

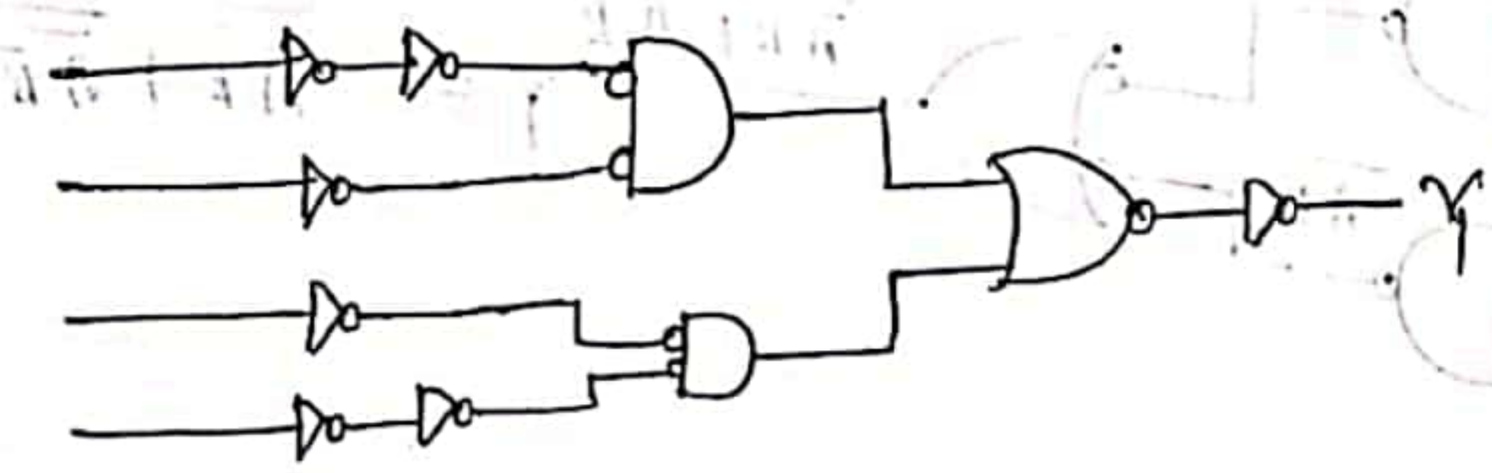
⟹ Using NOR Gate :-

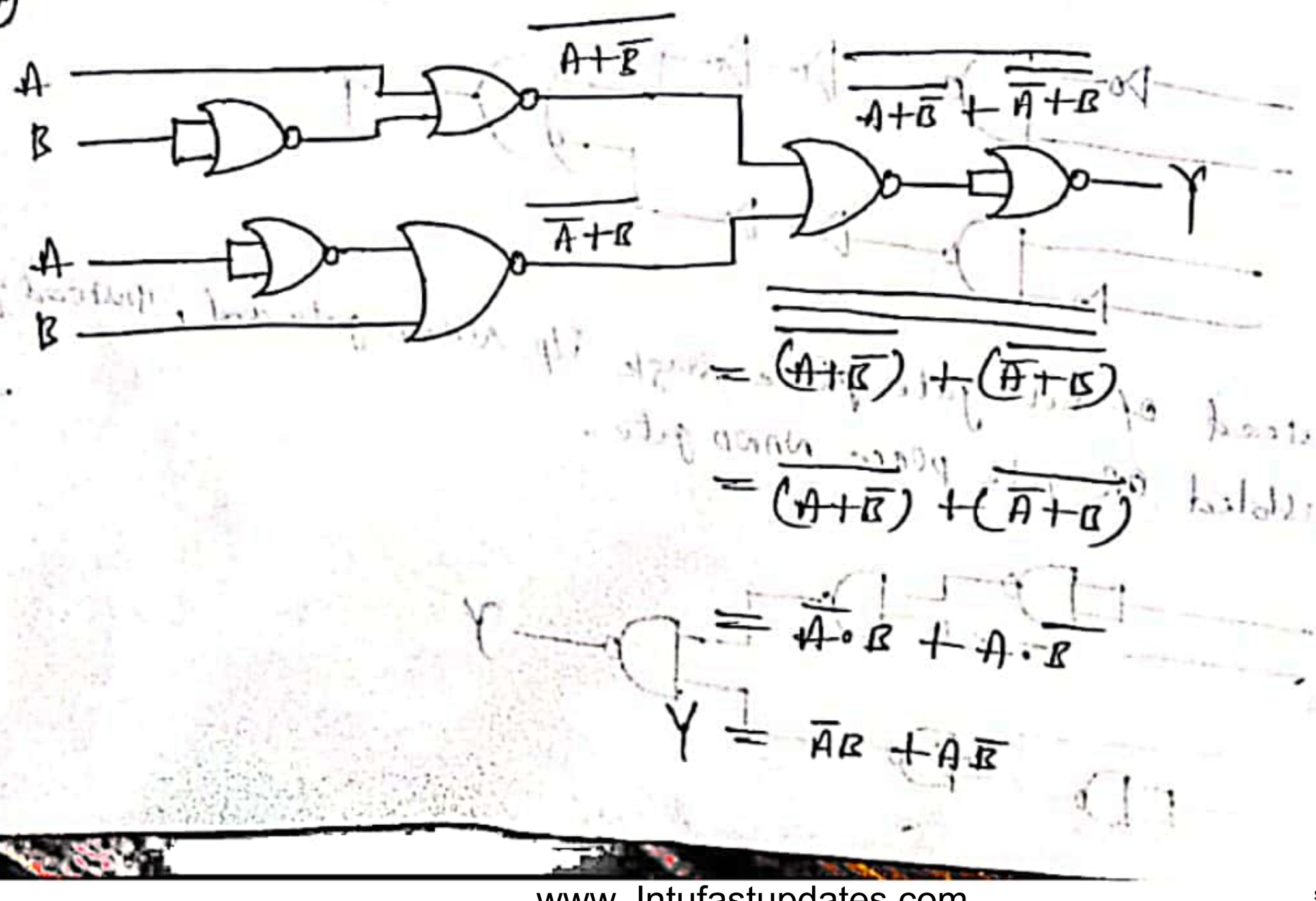
(131)

① 

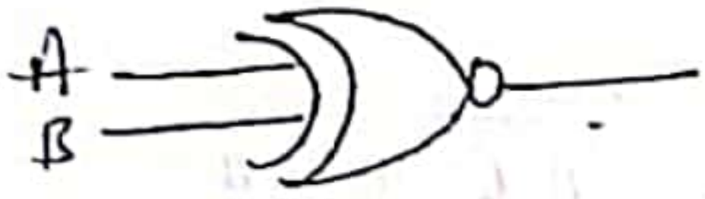
② 

③ 

④ 

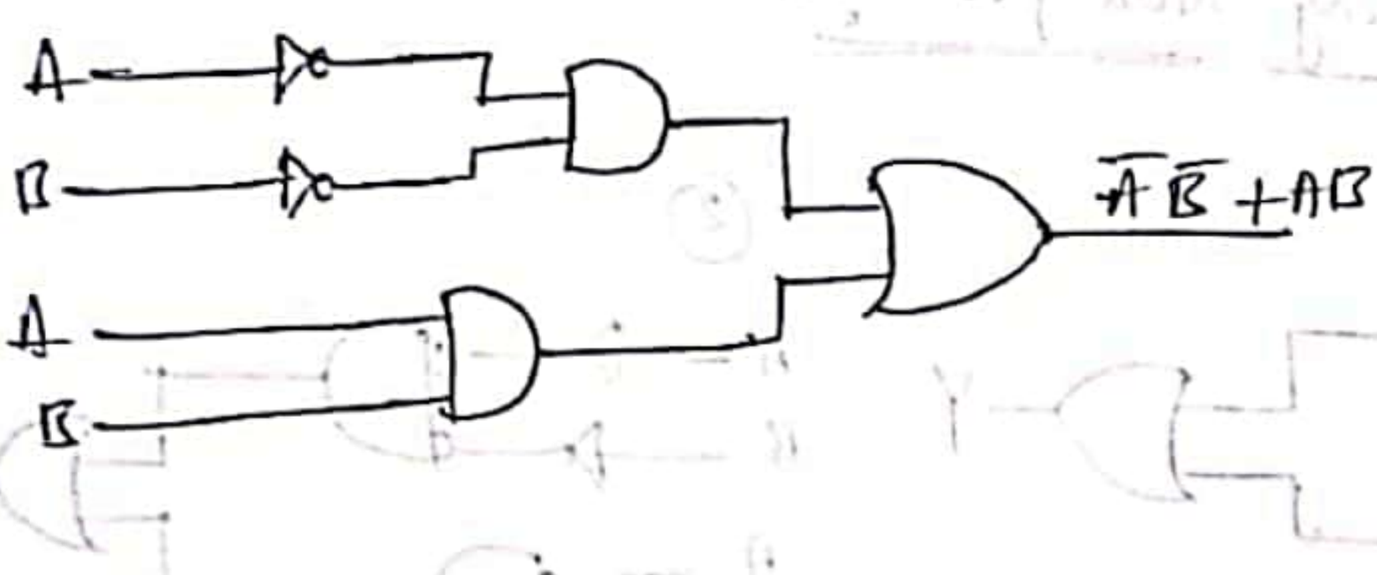
⑤ 

⟹ Realization Of X-NOR gate Using NAND & NOR :—

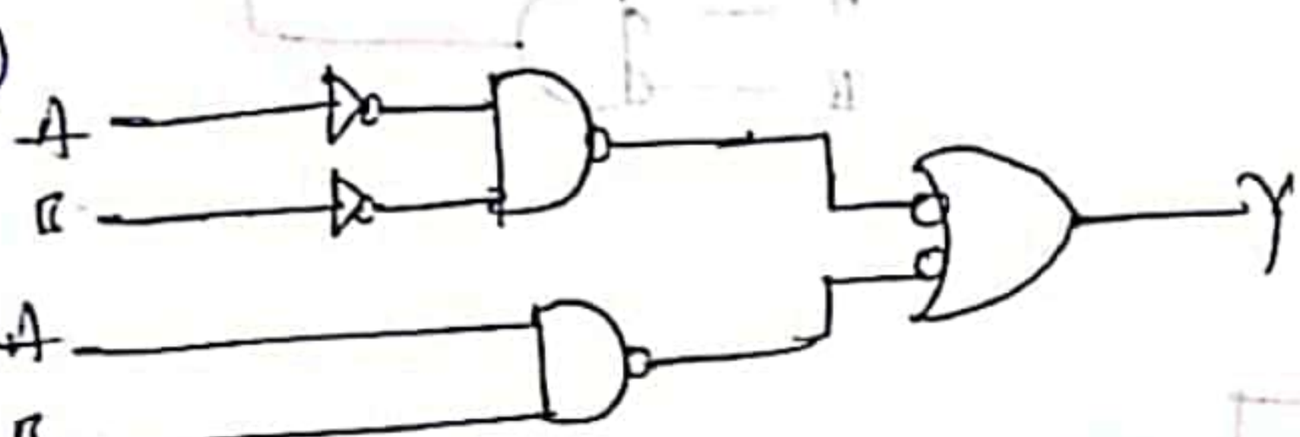


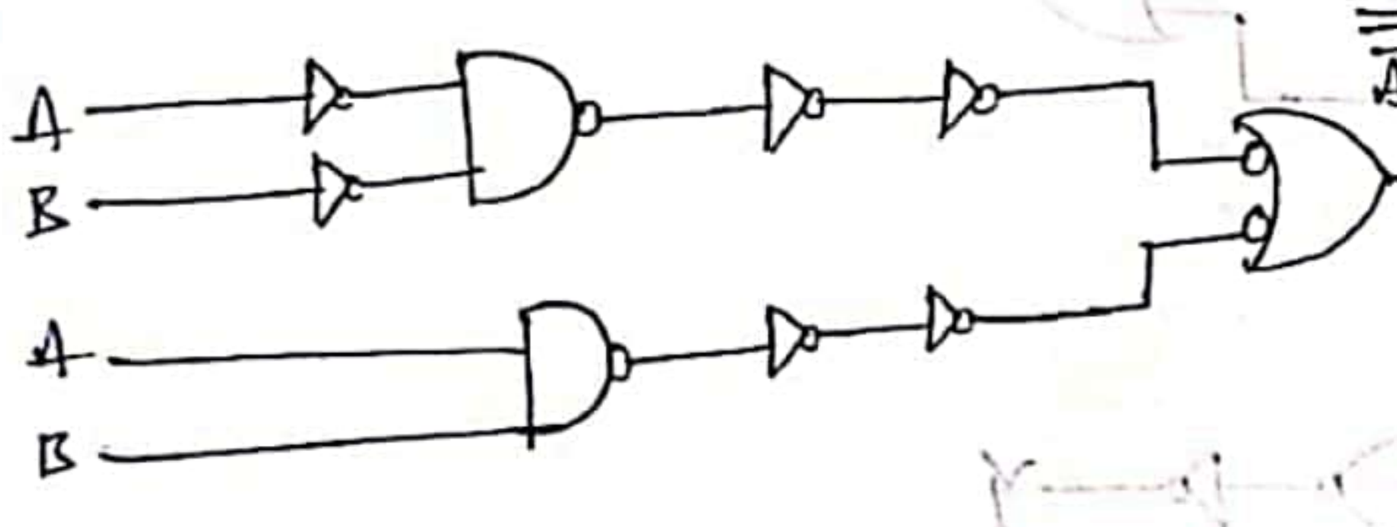
$\bar{A}\cdot\bar{B}+AB = A \odot B$

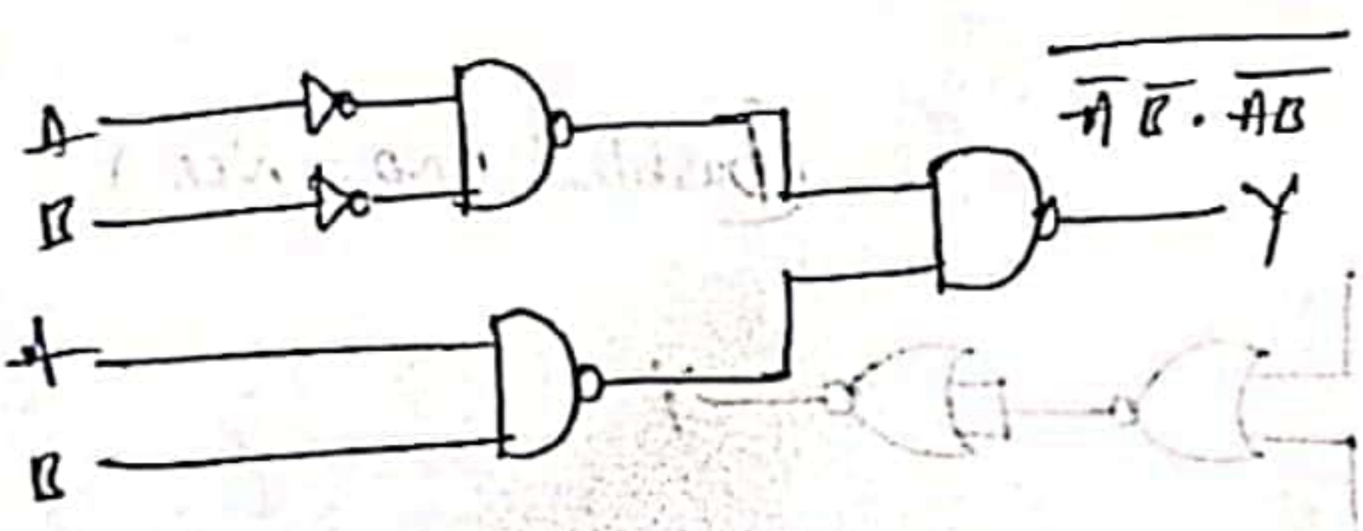
⟹ Using NAND Implementation :—

① 

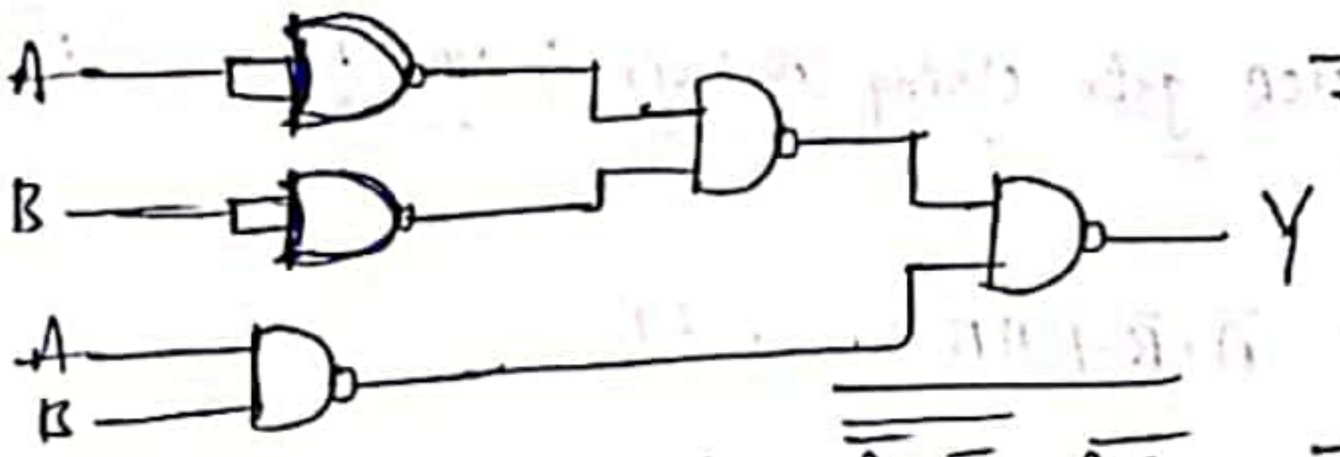
$Y = \bar{A}\bar{B}+AB$

② 

③ 

④ 

(Bubbled OR gate
= NAND gate)



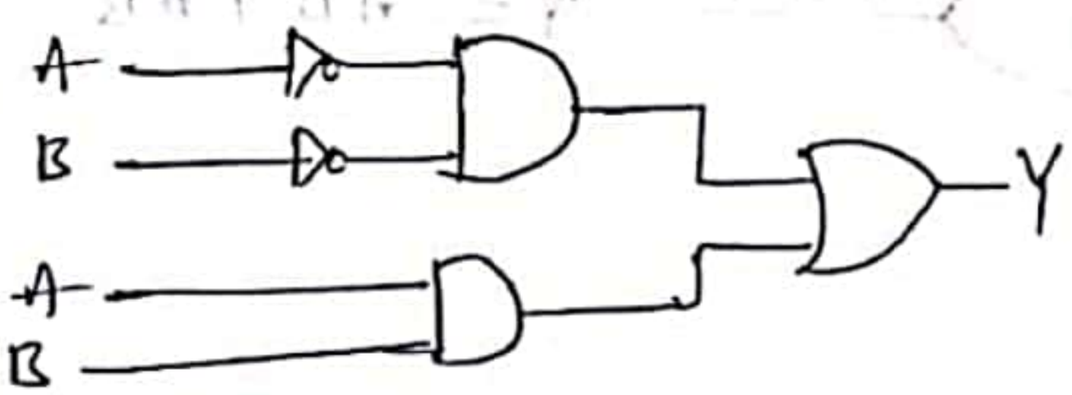
$$\overline{\overline{A \cdot \overline{B}} \cdot \overline{\overline{A} \cdot B}}$$

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

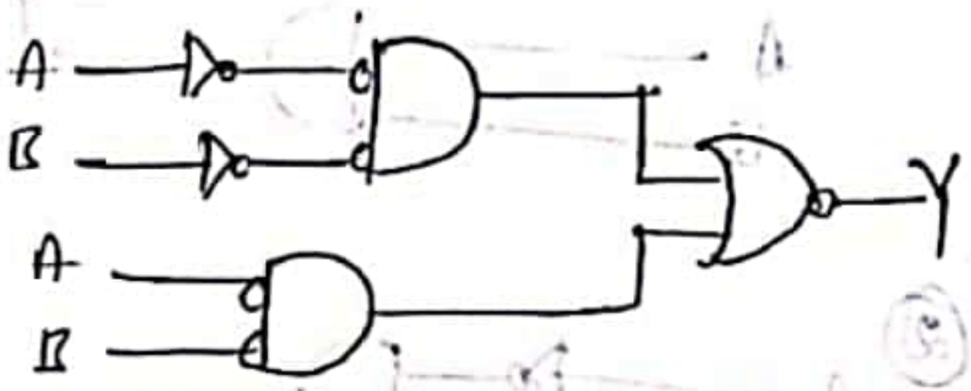
$$= \overline{A \cdot \overline{B}} + A \cdot B$$

⇒ X-NOR Gate using NOR gate :-

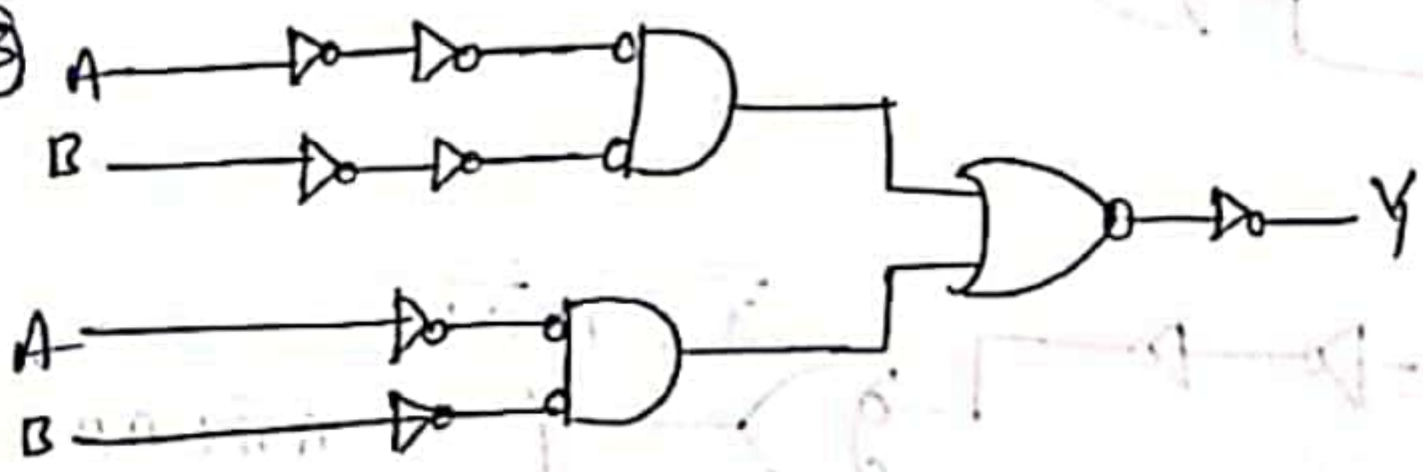
①



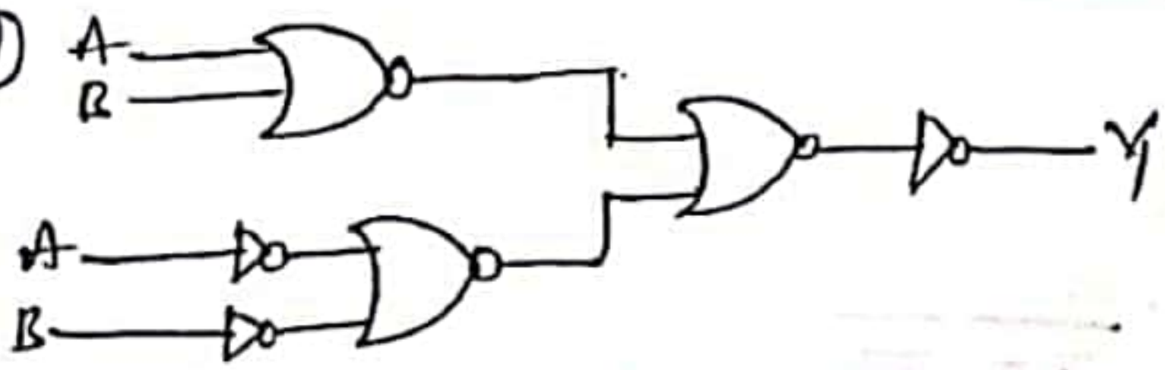
②



③

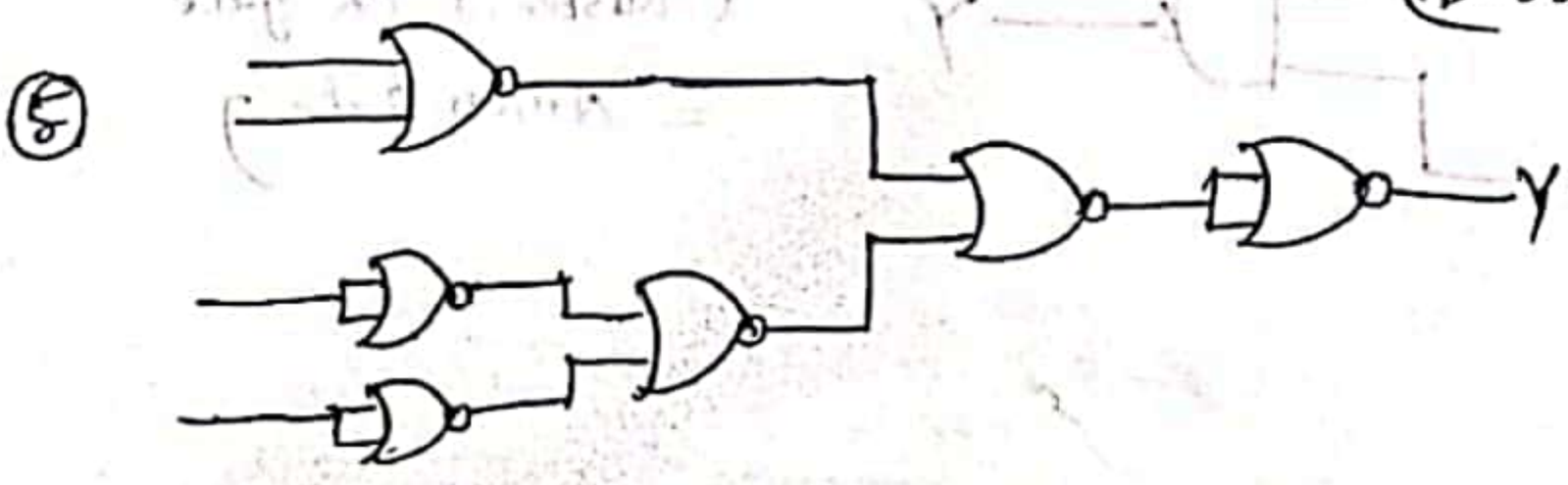


④



(Bubbled AND = NOR)

⑤



⇒ Implement the following functions using NAND Gates :— (134)

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

(b) $F_2 = w\bar{x}+\bar{x}y(\bar{z}+\bar{w})$

Sol

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

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

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

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

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

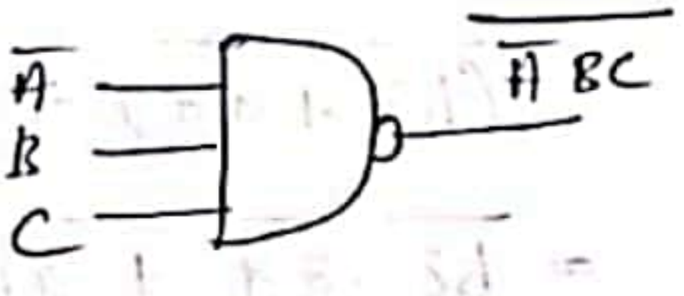
$= A(1+D)+\overline{B}+\overline{C}$

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

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

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

∴



$\left(\because \overline{B}+AB = \overline{B}+A\right.$

$\left.\overline{C}+ACD = \overline{C}+AD\right.$

Redundant Law) R.L.R

$\therefore 1+D = 1$

(b) $F_2 = w\bar{x}+\bar{x}y(\bar{z}+\bar{w})$

$= w\bar{x}+\bar{x}yz+\bar{w}\bar{x}y$

$= \bar{x}(w+\bar{w}y)+\bar{x}yz$

$= \bar{x}(w+y)+\bar{x}yz$

$= \bar{x}w+\bar{x}y+\bar{x}yz$

$= \bar{x}w+\bar{x}y(1+z)$

$= \bar{x}w+\bar{x}y$

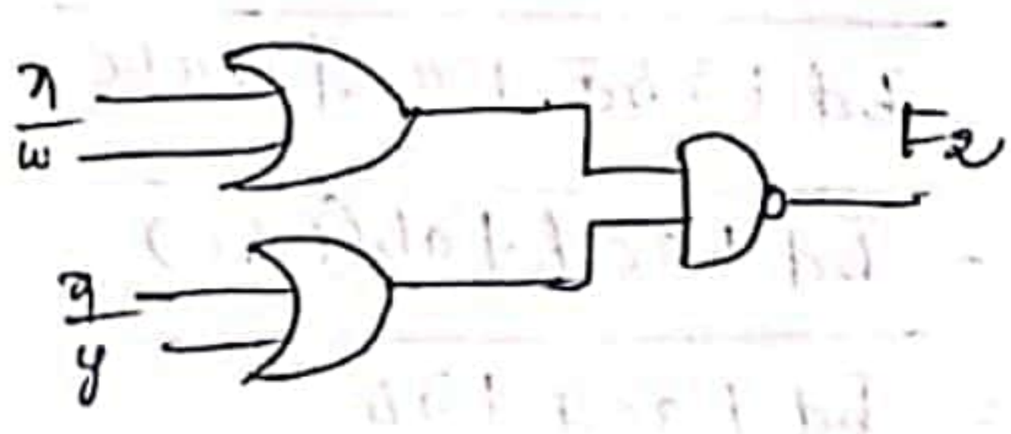
$= \overline{\overline{\bar{x}w+\bar{x}y}}$

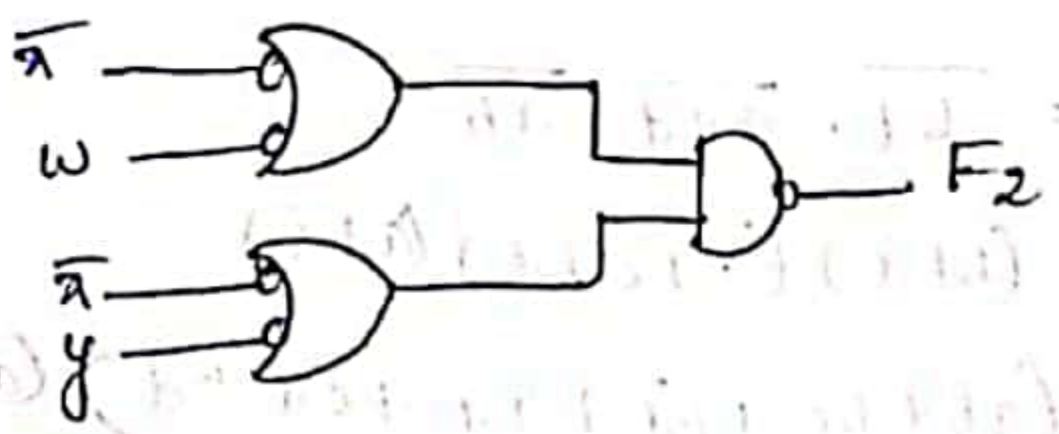
$= \overline{\bar{x}(w+y)}$

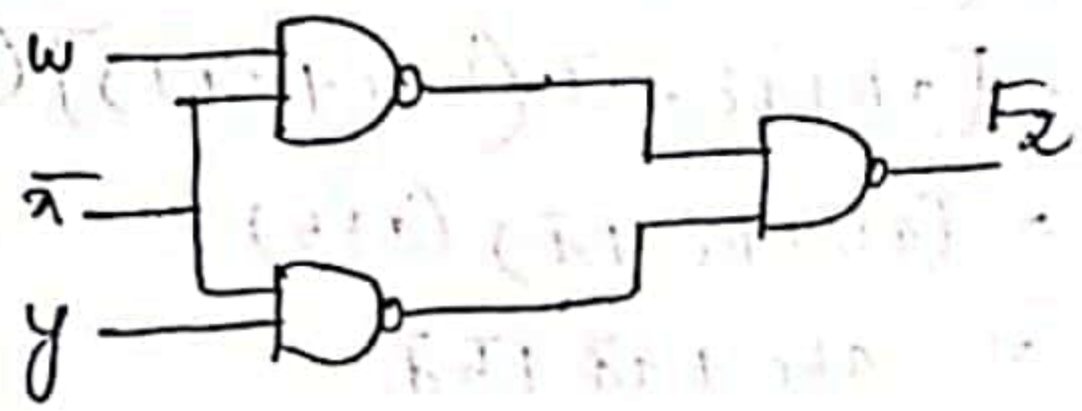
$= \overline{\overline{\bar{x}w}\cdot\overline{\bar{x}y}}$

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

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







⇒ Find the complement of the following boolean function & Reduce them to minimum no. of literals. (125)

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

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

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

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

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

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

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

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

$= 1$

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

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

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

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

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

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

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

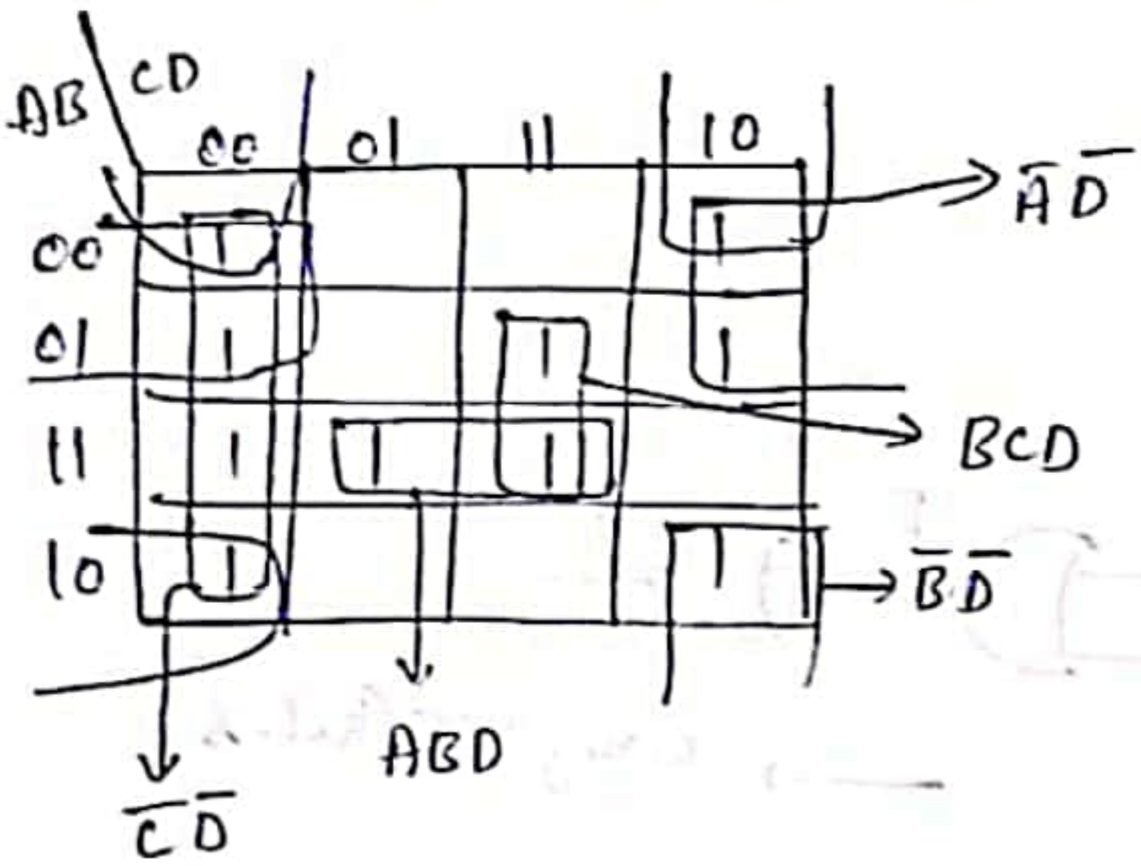
$= (\bar{a}b+b\bar{c}+\bar{d})(a+\bar{b})$

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

Reduce using mapping the following expression and implement the real minimal expression in universal logic (136)

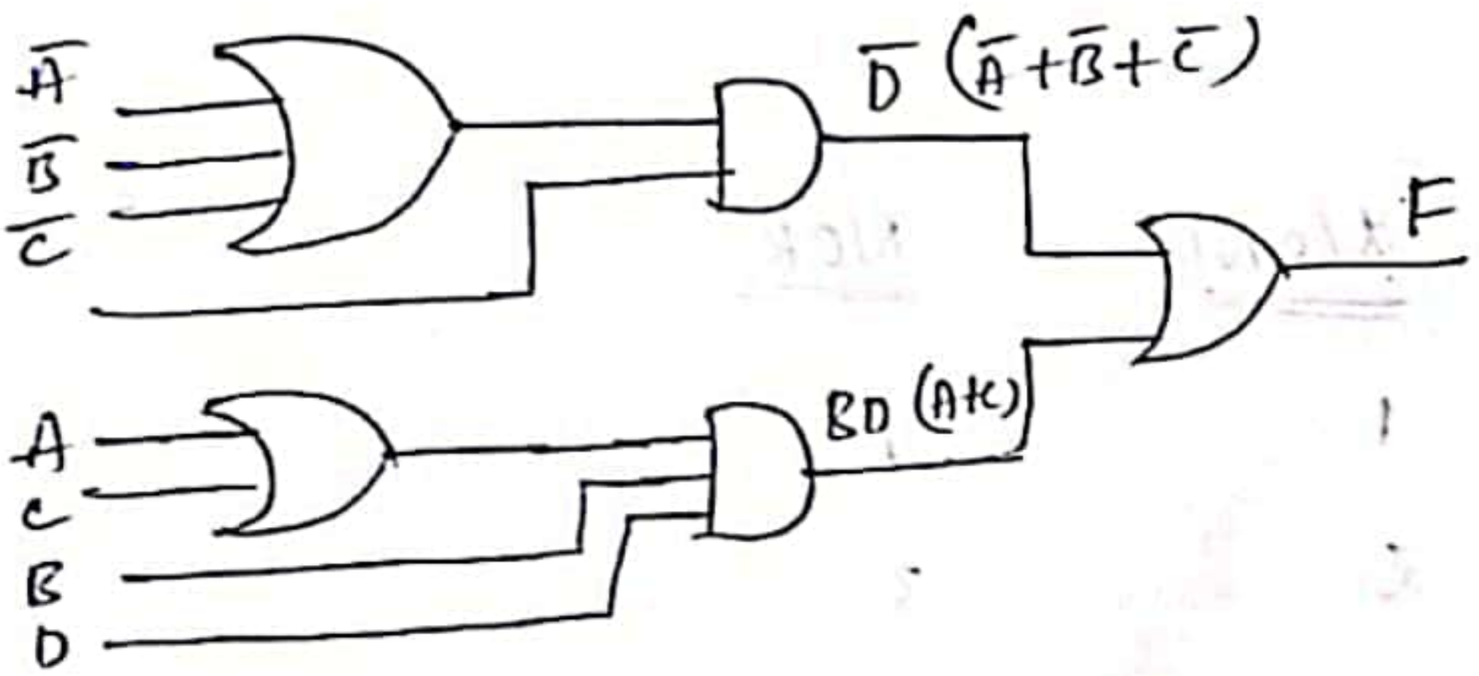
$F = \Sigma m(0, 2, 4, 6, 7, 8, 10, 12, 13, 15)$

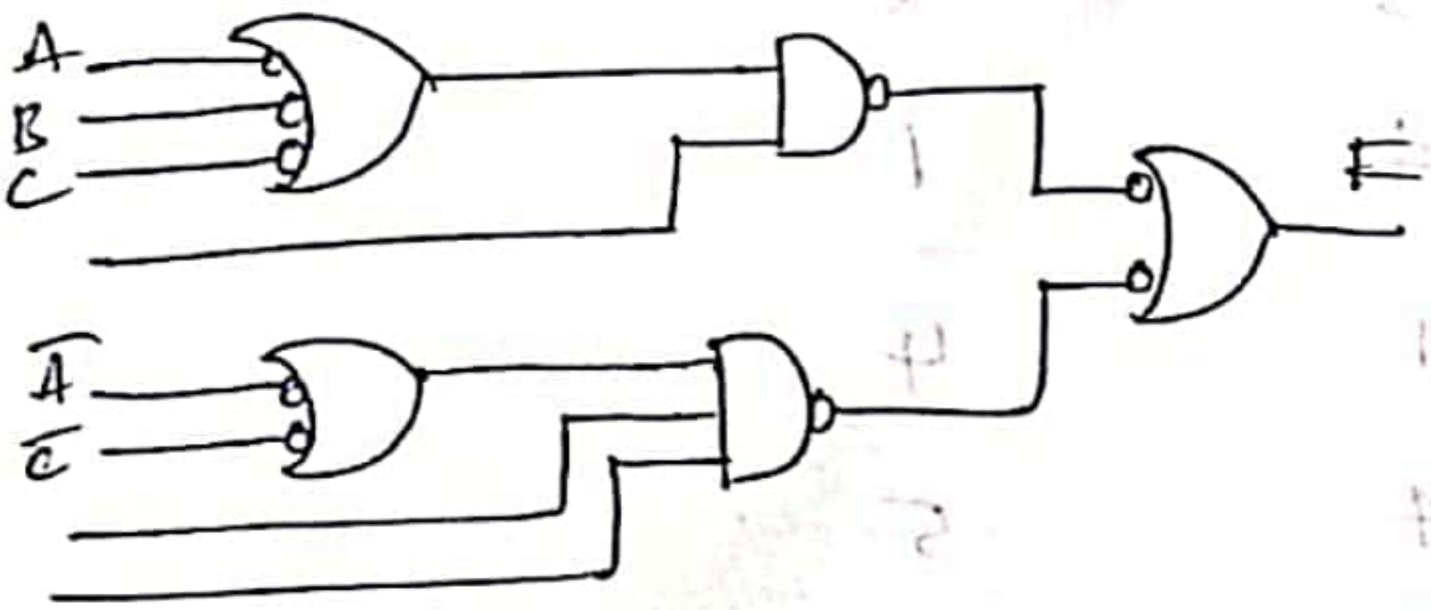
Sol

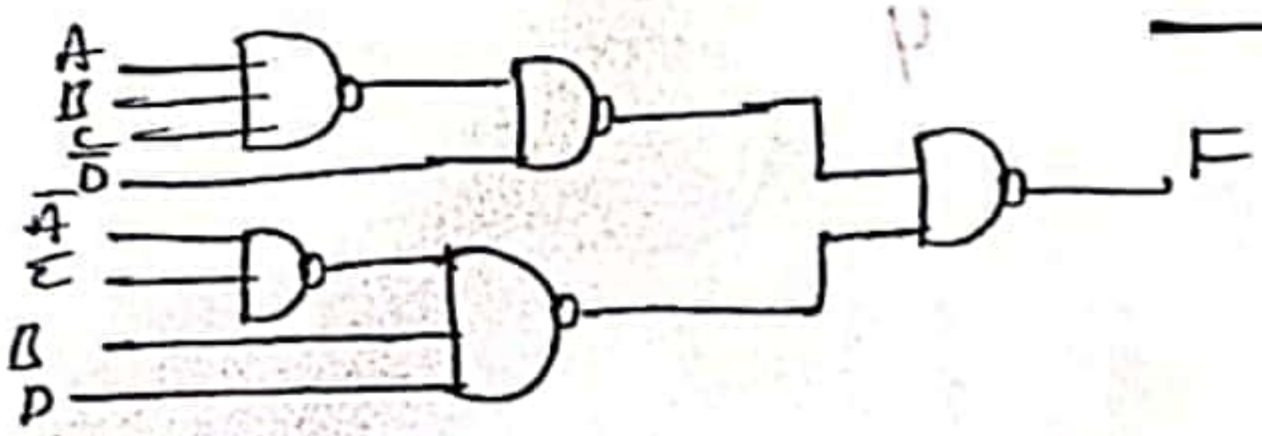


$F = \bar{C}\bar{D} + \bar{A}\bar{D} + \bar{B}\bar{D} + A\bar{B}D + B\bar{C}D$

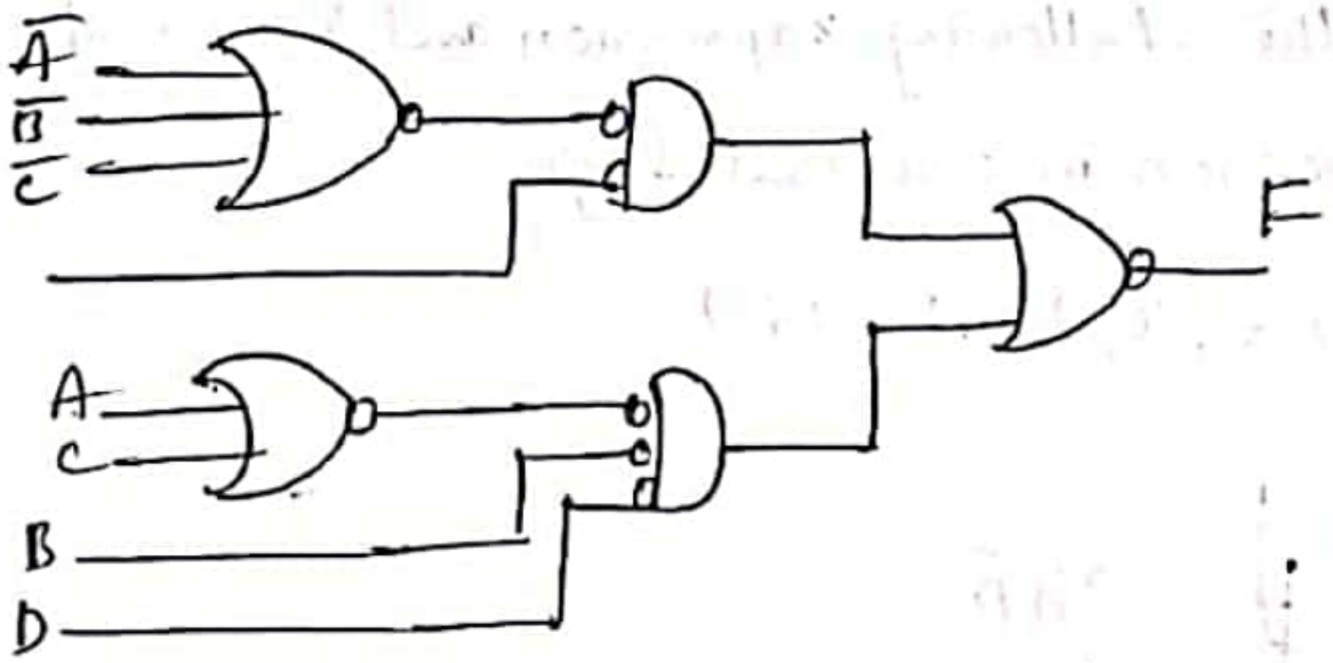
$= \bar{D}(\bar{A}+\bar{B}+\bar{C}) + BD(A+C)$

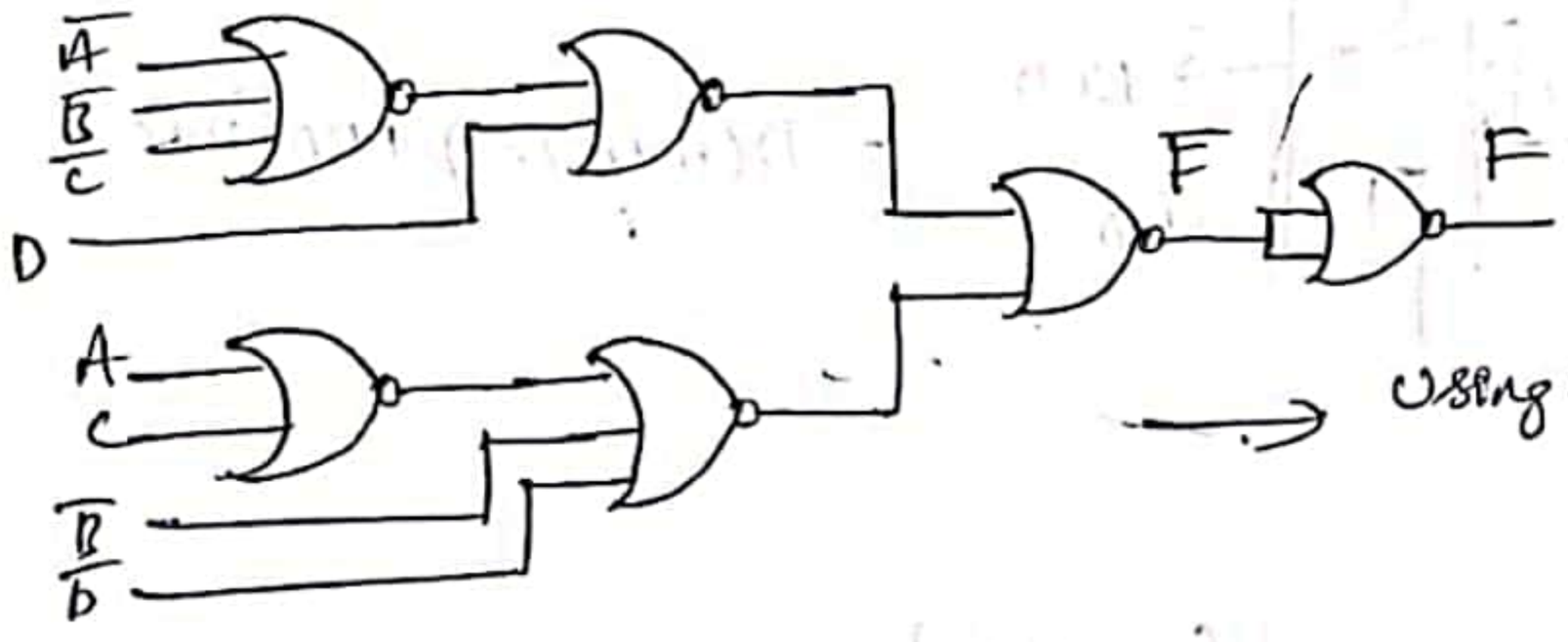






→ (Using NAND gates)





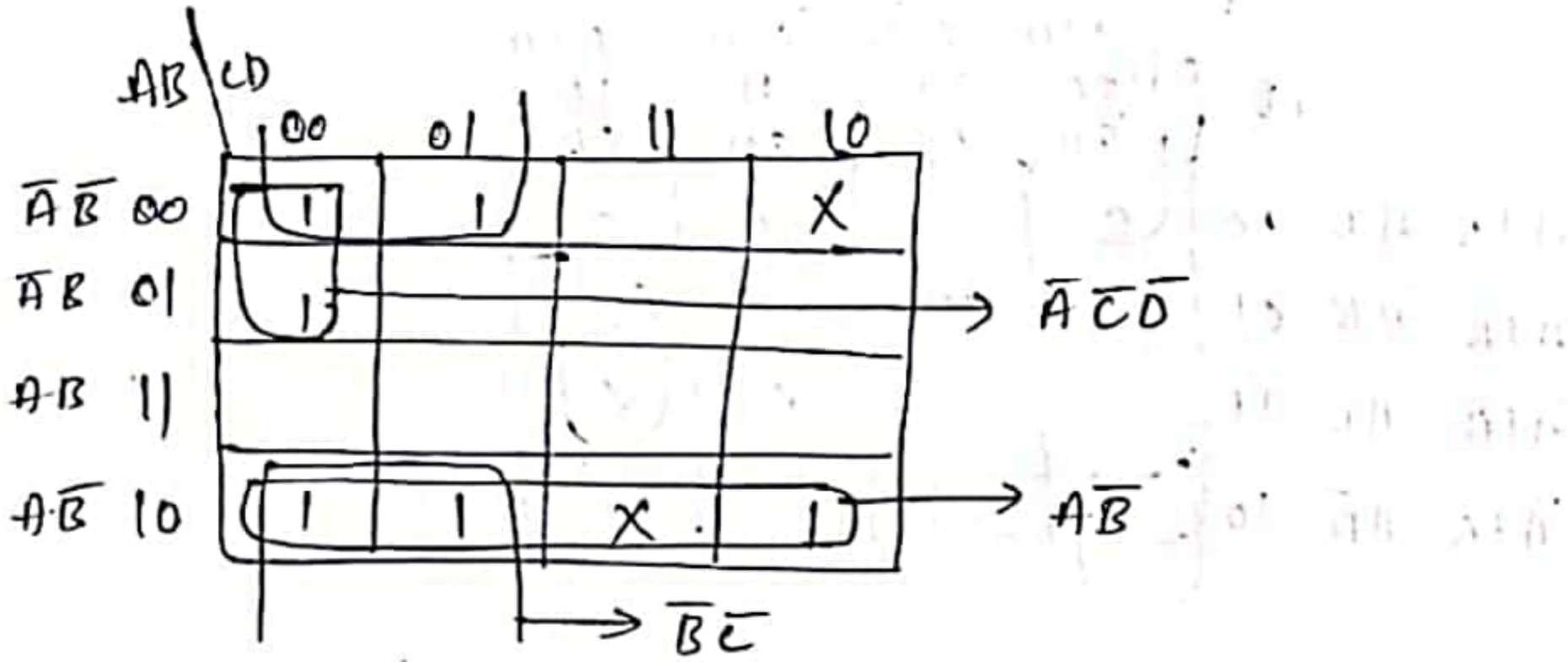
	NAND	NOR
NOT	1	1
AND	2	3
OR	3	2
NOR	4	1
NAND	1	4
X-OR	4	5
X-NOR	5	4

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

(138)

$\therefore$ TM $(3,5,6,7,12,13,14,15) + d(2,11)$

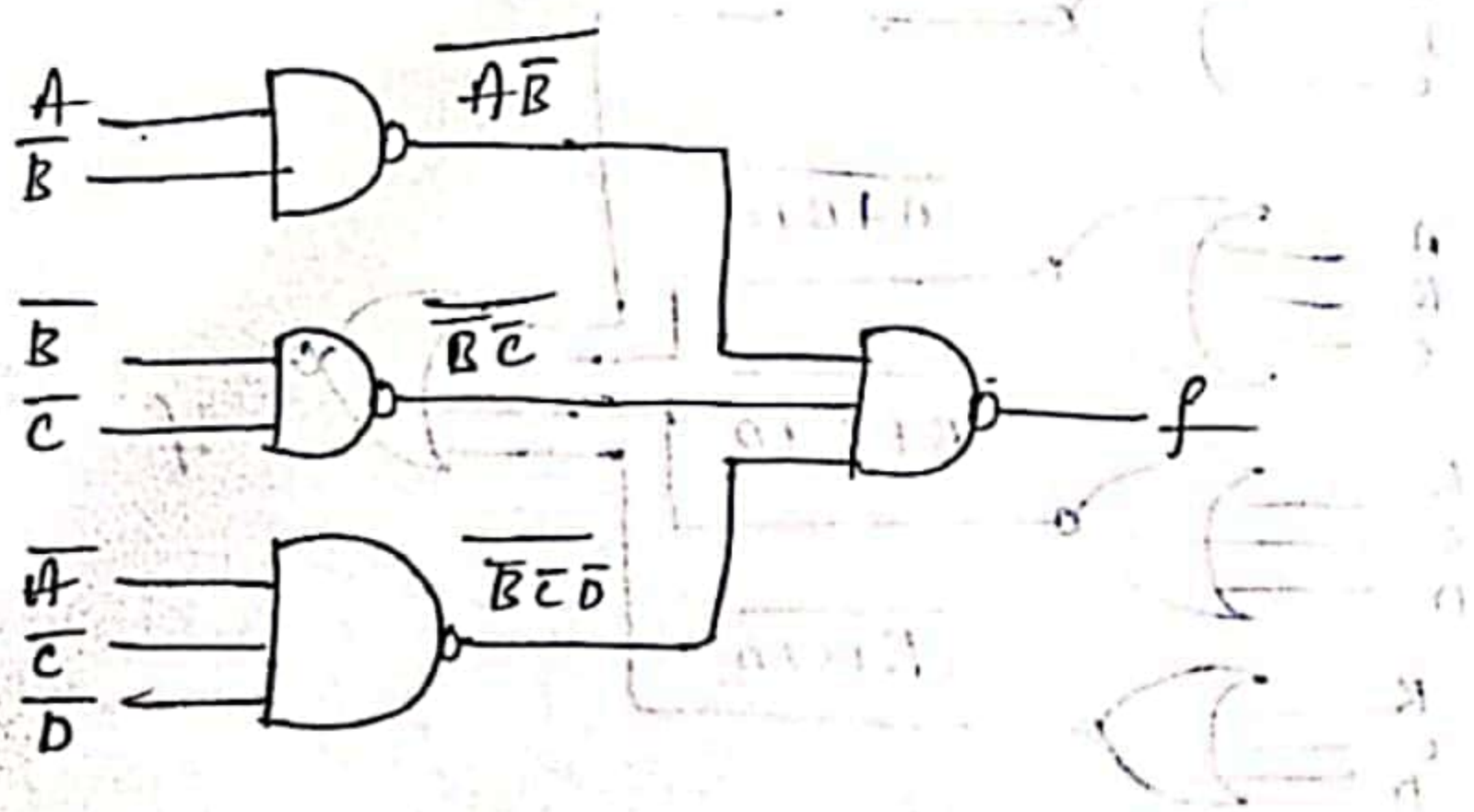
$\Rightarrow \Sigma m(0,1,4,8,9,10) + d(2,11)$



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

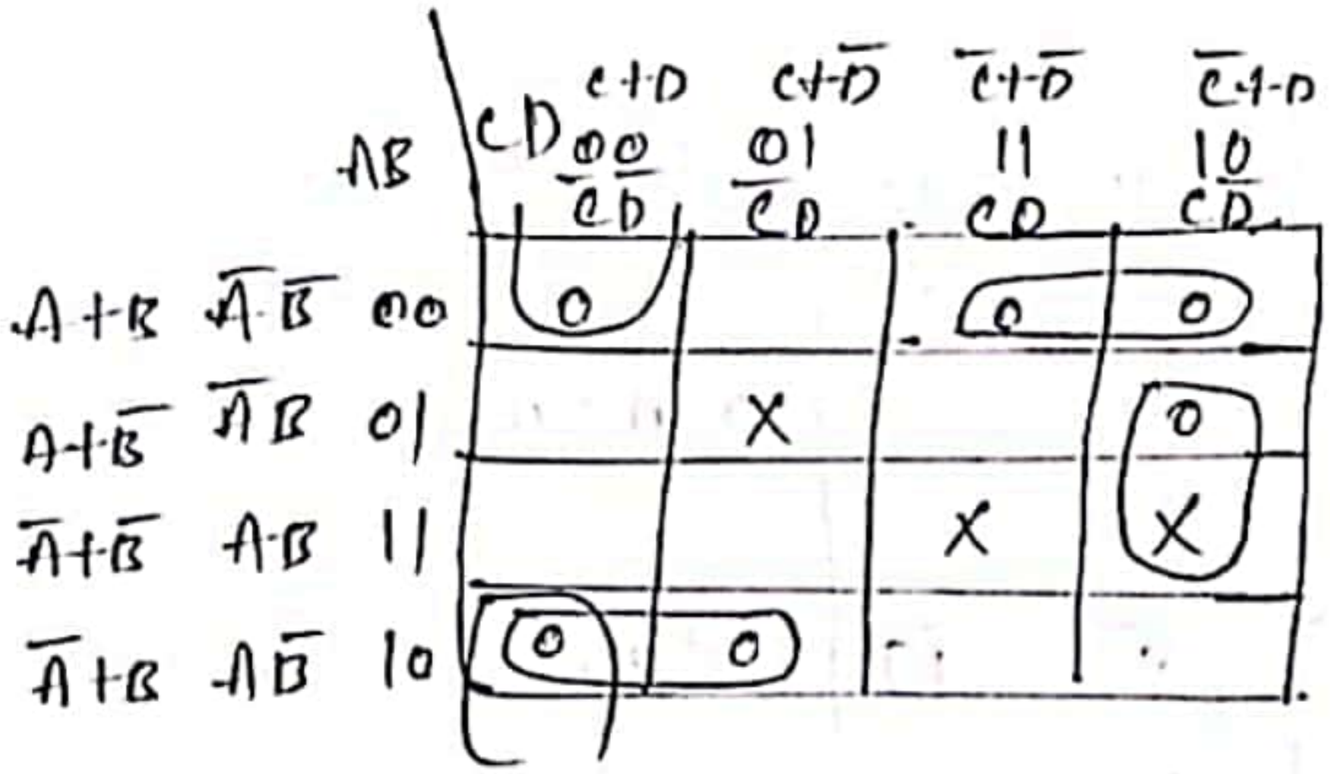
W.K.T $f = \overline{\overline{A\bar{B} + \bar{B}\bar{C} + \bar{A}\bar{C}\bar{D}}}$

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



Q Minimize the expression using K-map and Implement it Using NOR gate

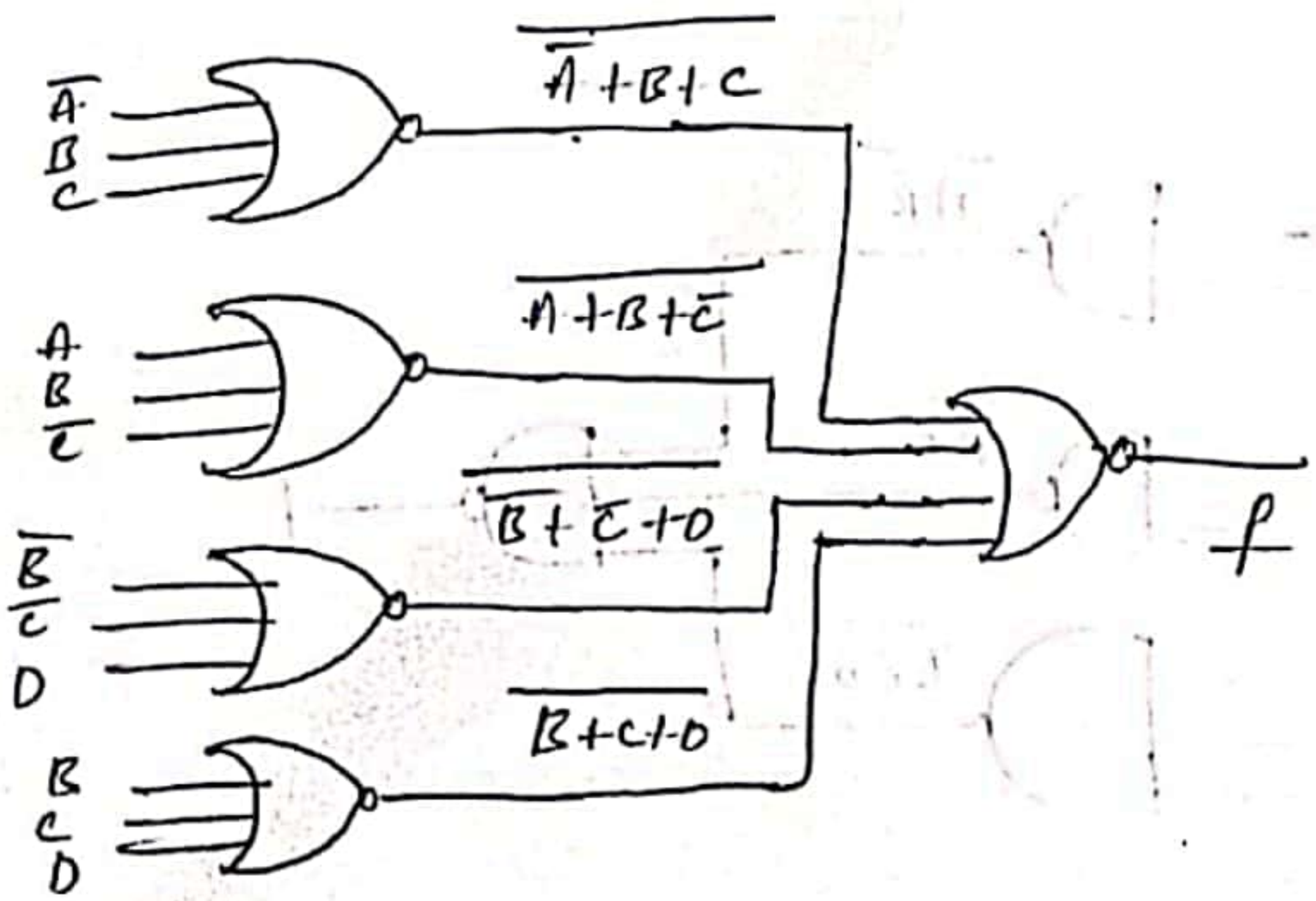
$$f(A,B,C,D) = \Sigma m(1,4,7,10,11,12,13) + \Sigma d(5,14,15)$$
$$= \Pi M(0,2,3,6,8,9) + \Sigma d(5,14,15)$$



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

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

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



UNIT-II

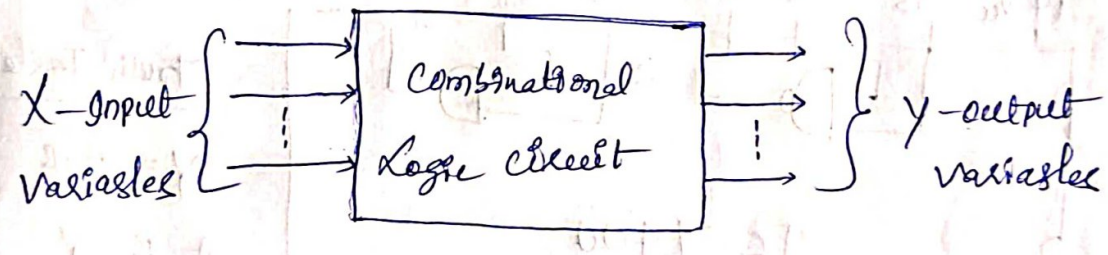
⇒ Combinational circuits :- Digital Logic circuits

→ 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.

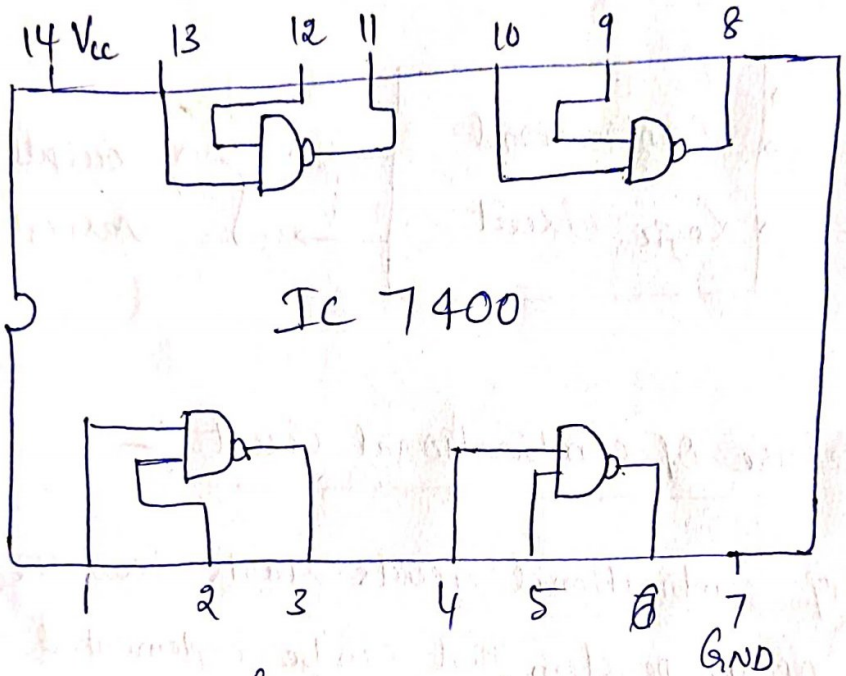
(1) From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each

(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.



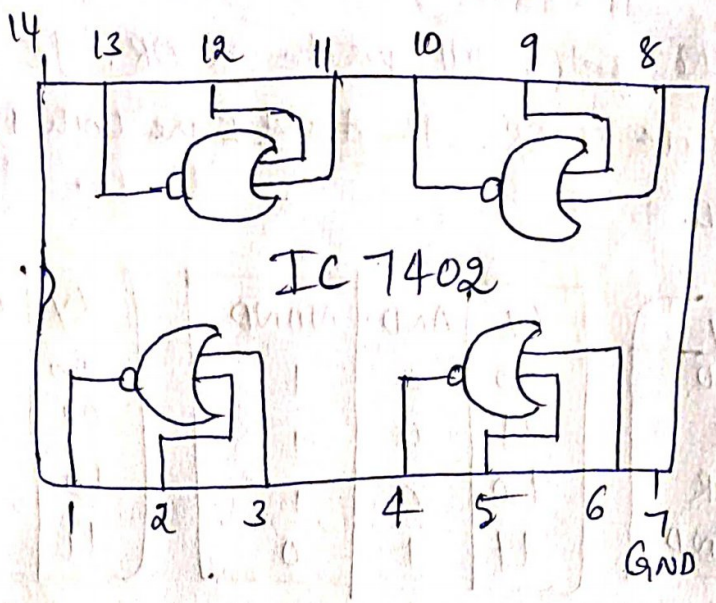
Truth Table

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

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

fig IC 7400

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)~~ or all inputs are 1, otherwise output is 1.

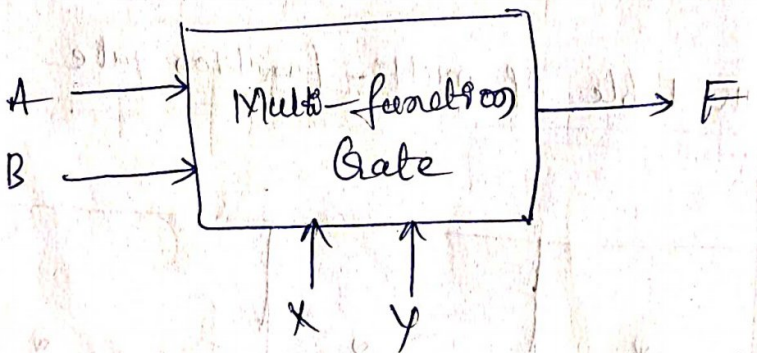


Truth Table

$Y = \overline{A+B}$

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 multi-function 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 operform. 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

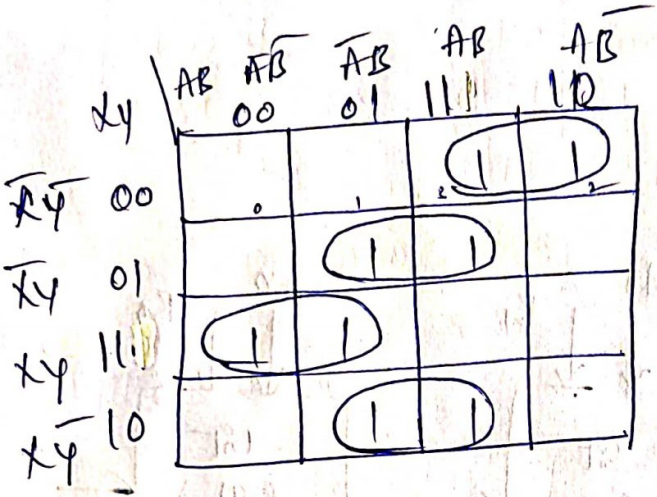
XY	OR	NOR
00	0	1
01	1	0
10	1	0
11	1	0

X and Y depend upon what A and B are doing. The truth tables for AND, NAND, OR and NOR can be seen for above 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	2OR gate	2AND 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	0 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 1 1 1 1

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.



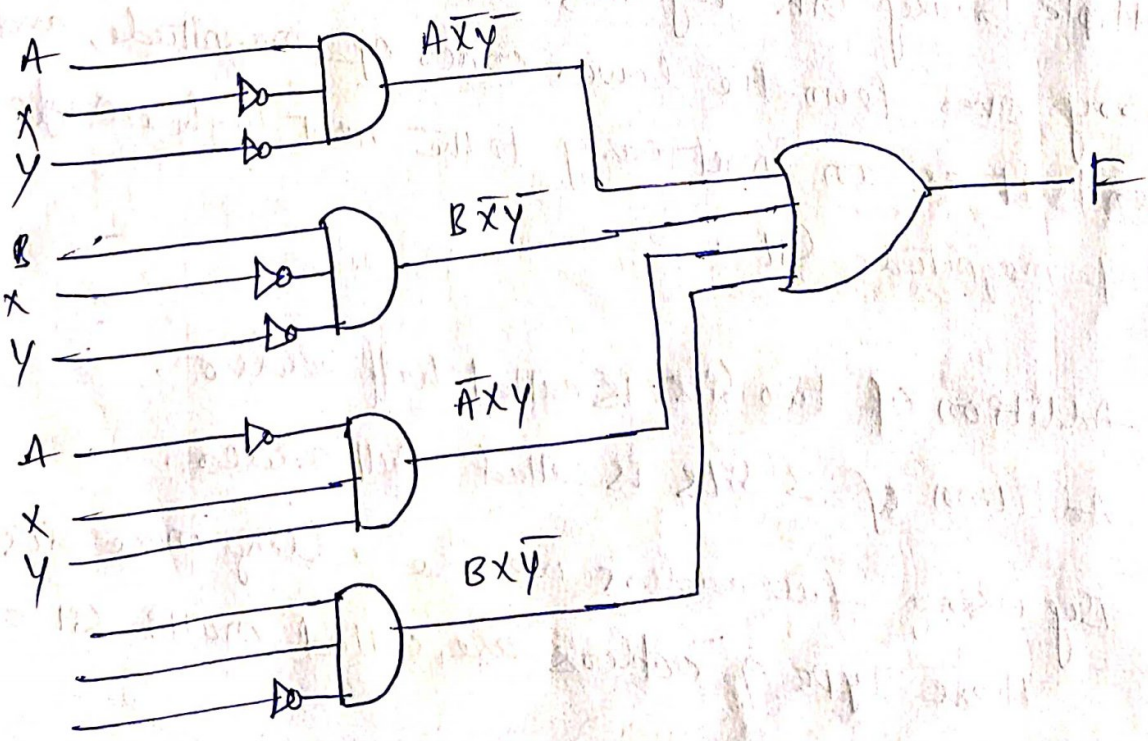
$\Sigma m(2,3,5,7,9,11,12,13)$

By using the ones on the table, the function can be written as a sum of products (SOP).

To do this you need ABXY to multiply out to equal 1. Therefore, the unsimplified equation for this:

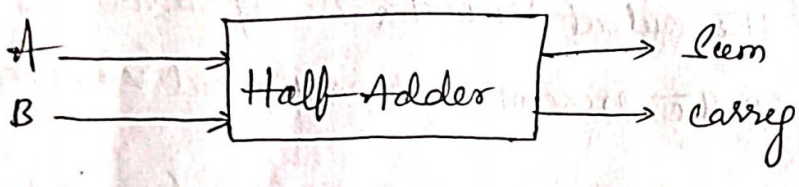
$$F = A\bar{X}\bar{Y} + B\bar{X}\bar{Y} + \bar{A}XY + BX\bar{Y}$$

⇒ 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.

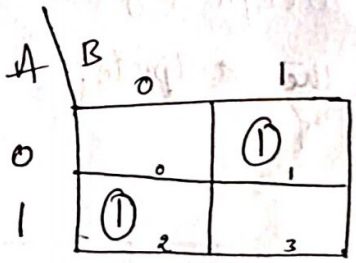
⇒ **Block diagram :-**



⇒ **Truth table :-**

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

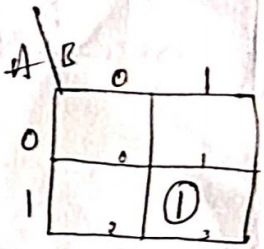
K-map for (S)



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

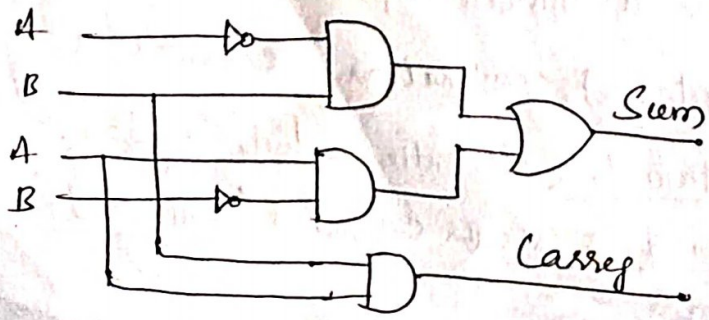
$S = A \oplus B$

K-map for (C)

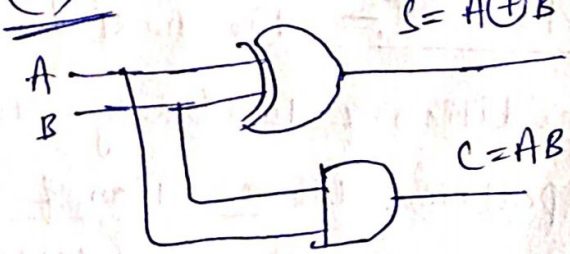


$C = AB$

⇒ **Logic diagram :-**

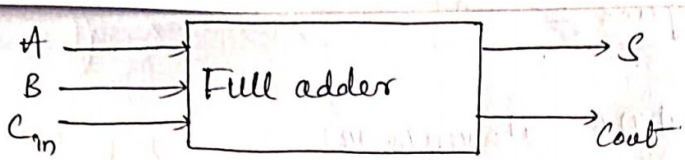


(or)



⇒ **Full Adder :-** A full adder is a combinational circuit that adds two bit 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}).

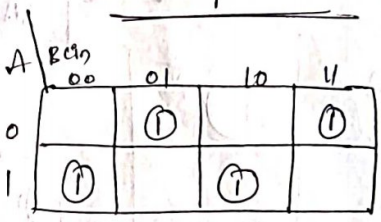


Block diagram

⇒ **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

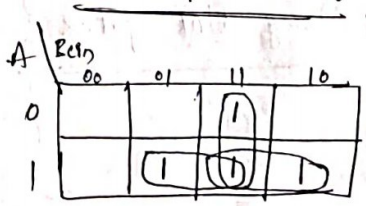


$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$

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

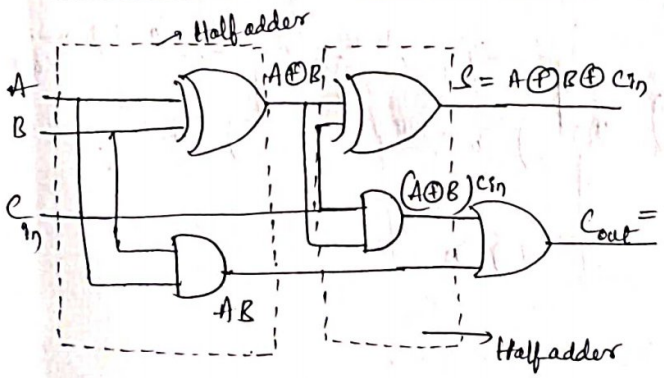
$= A \oplus B \oplus C_{in}$

K-map for Cout :-



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

⇒ **Logic diagram :-** (Using two halfadders)



carry

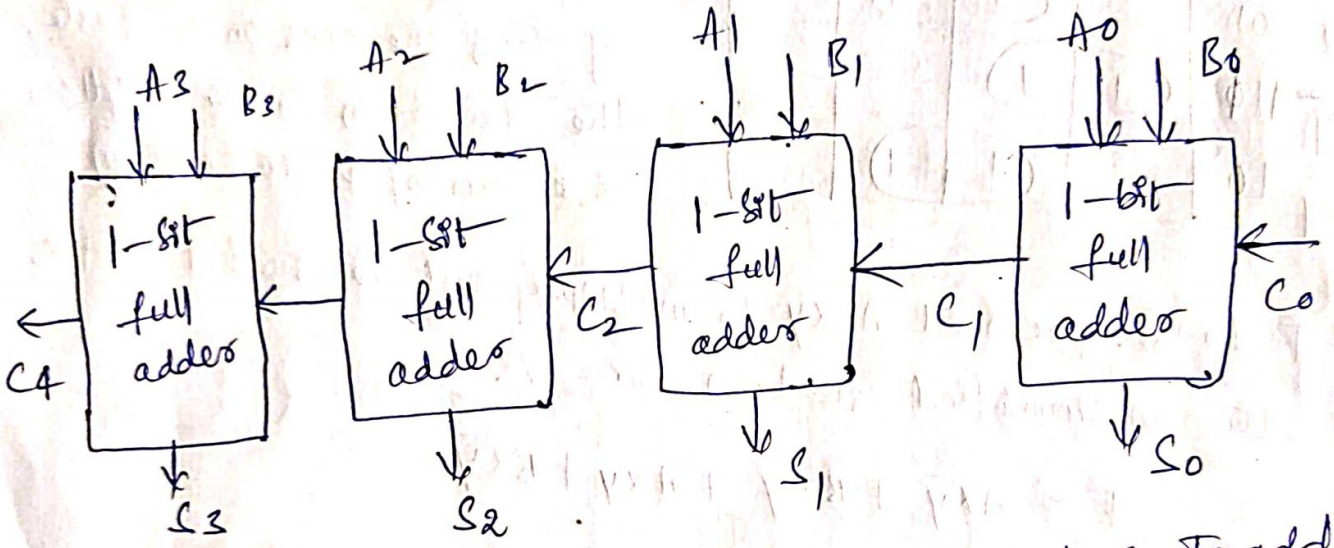
$C = \bar{A}BC + A\bar{B}C + ABC + AB\bar{C}$

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

$= C(A \oplus B) + AB$

⇒ 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)

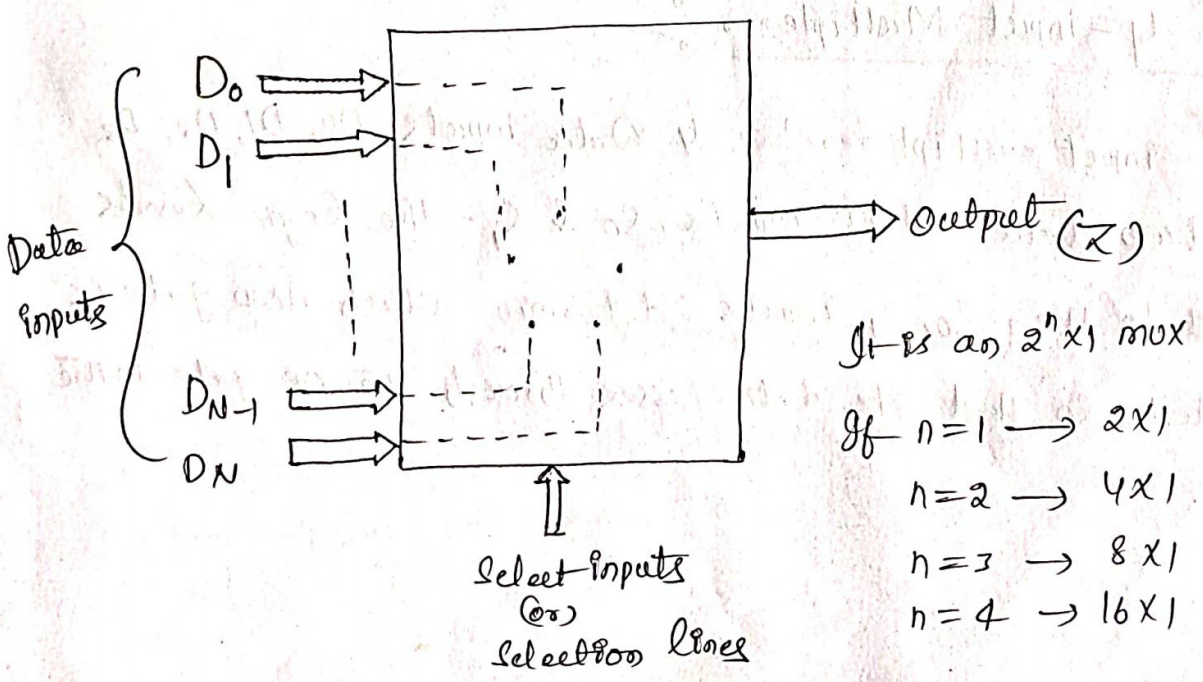
→ A multiplexer is a combinational circuit that selects binary information from one to many input lines and direct to a single output line.

→ 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'



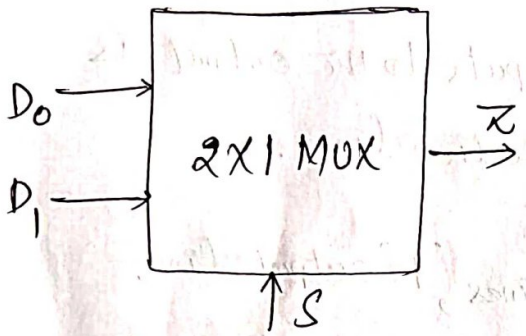
It is an $2^n \times 1$ mux

If $n=1 \rightarrow 2 \times 1$
$n=2 \rightarrow 4 \times 1$
$n=3 \rightarrow 8 \times 1$
$n=4 \rightarrow 16 \times 1$

⇒ Basic 2-Input multiplexer :-

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

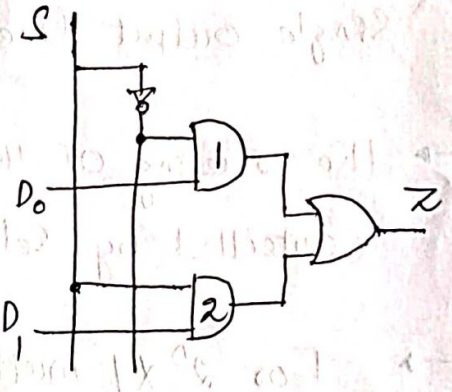
Block diagram



Truth Table

S	Z
0	D_0
1	D_1

Logic diagram



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

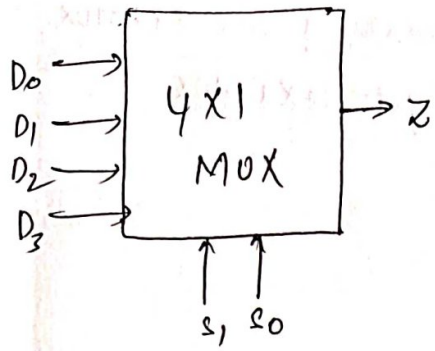
→ when $S=0$, AND gate '1' is enabled & AND gate '2' is disabled so $Z = D_0$.

→ 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.

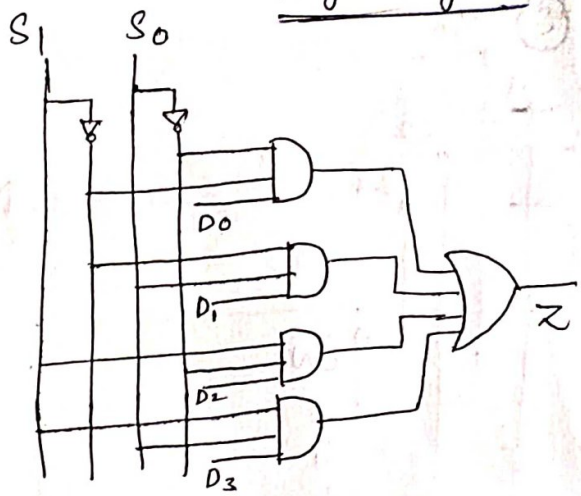
Block diagram



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



$\therefore Z = \bar{S_1}\bar{S_0}D_0 + \bar{S_1}S_0D_1 + S_1\bar{S_0}D_2 + S_1S_0D_3$

⟹ 16-Input multiplexer from two 8-Input multiplexer :-

→ To design a 16-input multiplexer from two 8-input multiplexer 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 $S_3 = 0$, 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 $S_3 = 1$ the right multiplexer is enabled and S_2, S_1 & S_0 inputs select one of its data inputs for passage to output X.

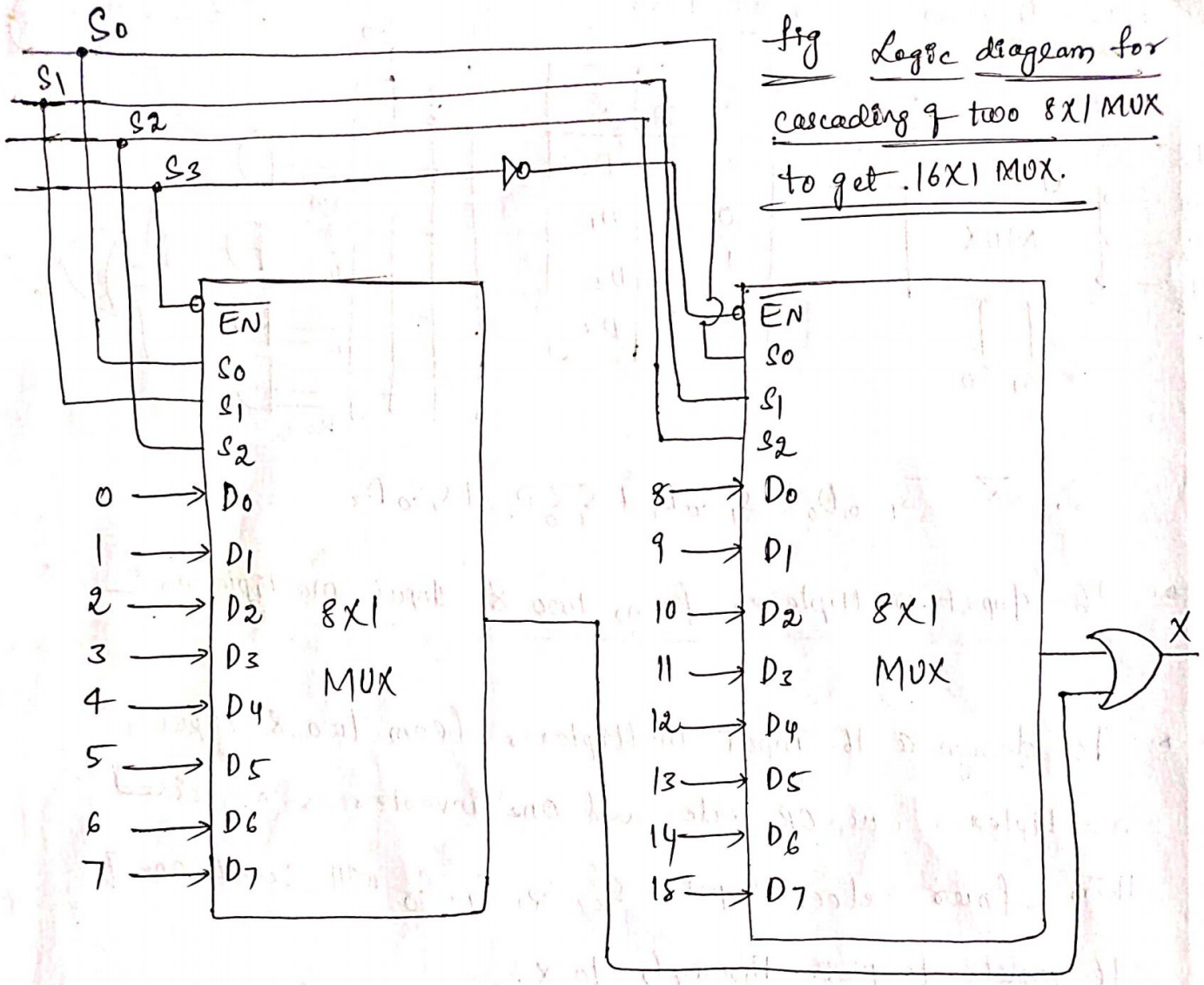


fig Logic diagram for cascading of two 8X1 MUX to get 16X1 MUX.

⇒ 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 S_1 & S_0 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 S_1 & S_0.

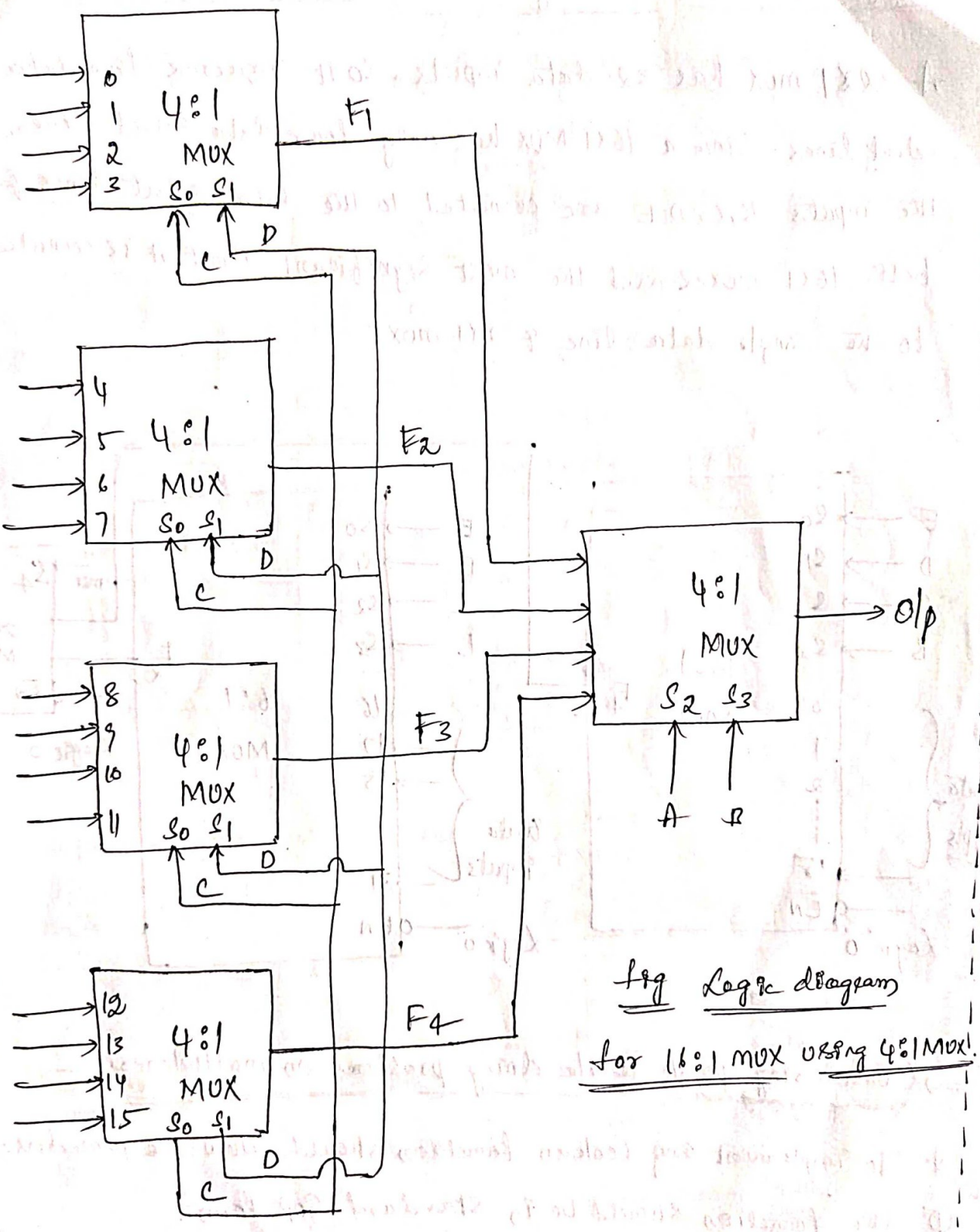
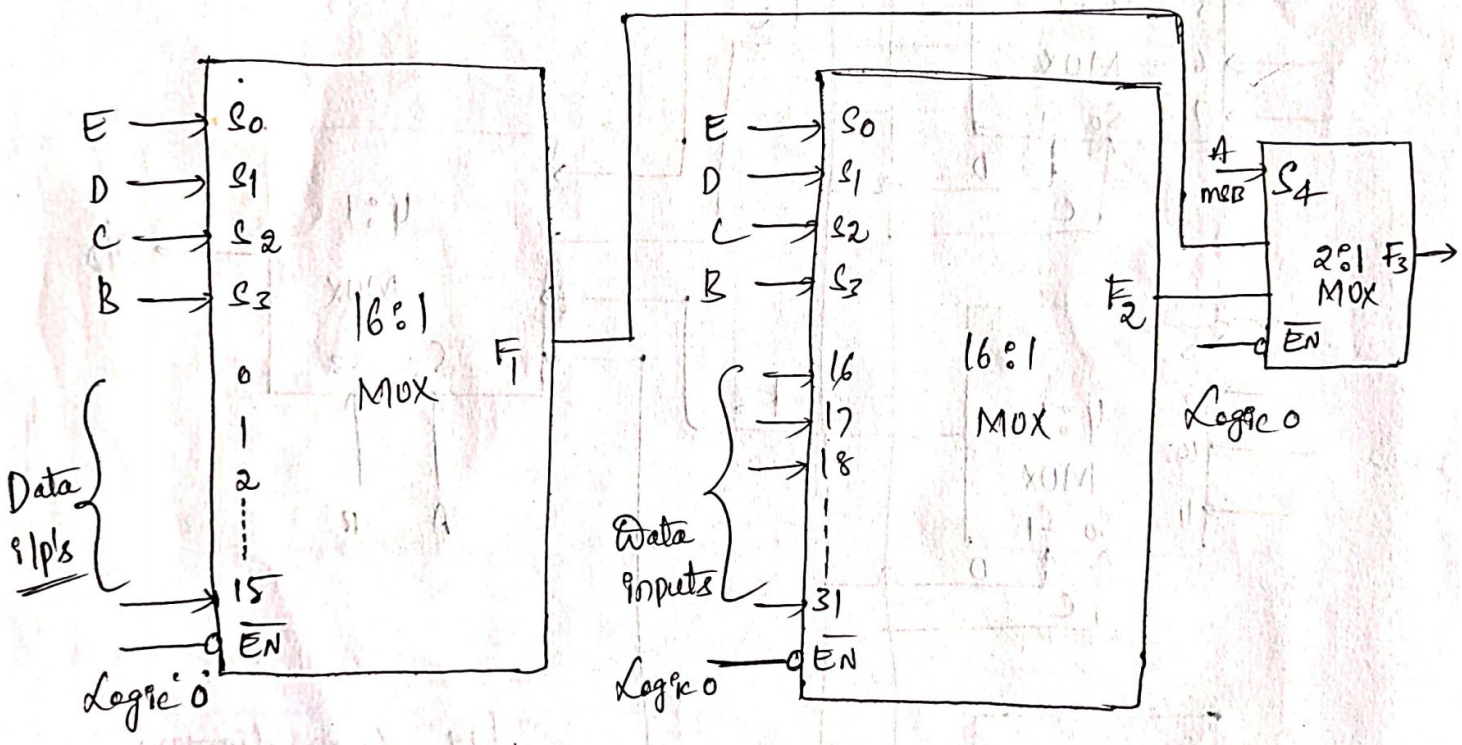


Fig Logic diagram for 16:1 MUX using 4:1 MUX.

⟹ 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.

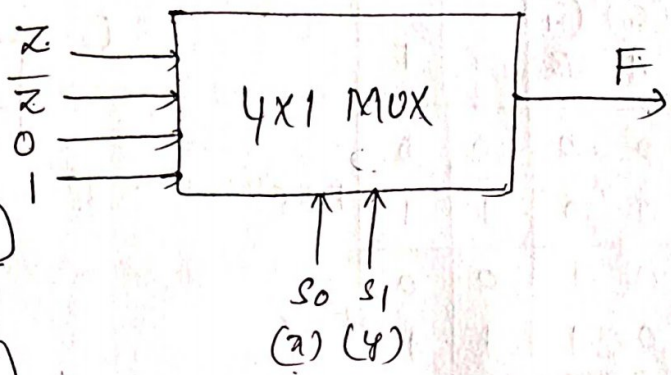
③ Take any two variables as a selection lines.

Q① $F = \Sigma m(1,2,6,7)$

Truth Table

S_0 (x)	S_1 (y)	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = \bar{z}$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

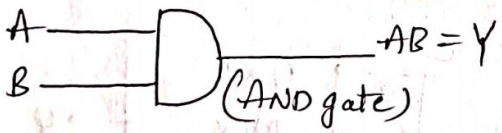
Block Diagram



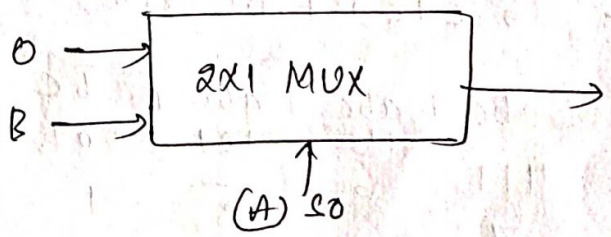
Implemented truth table

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	$y = 0$
0	1	0	
1	0	0	$y = B$
1	1	1	

Implementation truth table

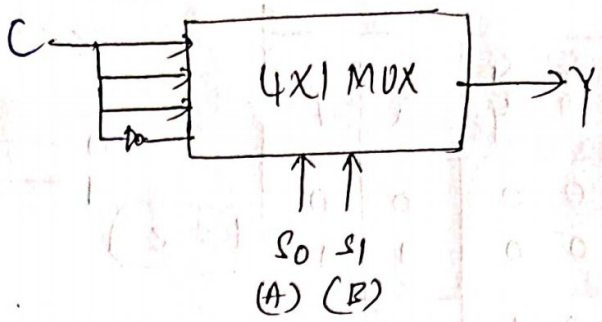
S_0	y
0	0
1	B

(Q3) Using 4X1 MUX, implement the Logic function $F(A,B,C) = \Sigma m(1,2,5,6)$

Sol

(S_0) A	(S_1) B	C	F	
0	0	0	0	F = C
0	0	1	1	
0	1	0	0	F = C
0	1	1	1	
1	0	0	0	F = C
1	0	1	1	
1	1	0	1	$F = \bar{C}$
1	1	1	0	

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.

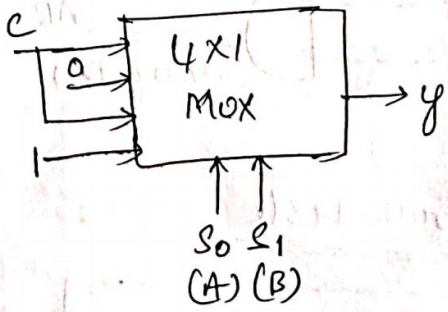
Sol $ab(c+\bar{c}) + \bar{b}c(a+\bar{a})$

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

111 (7), 110 (6), 101 (5), 001 (1)

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

Block Diagram

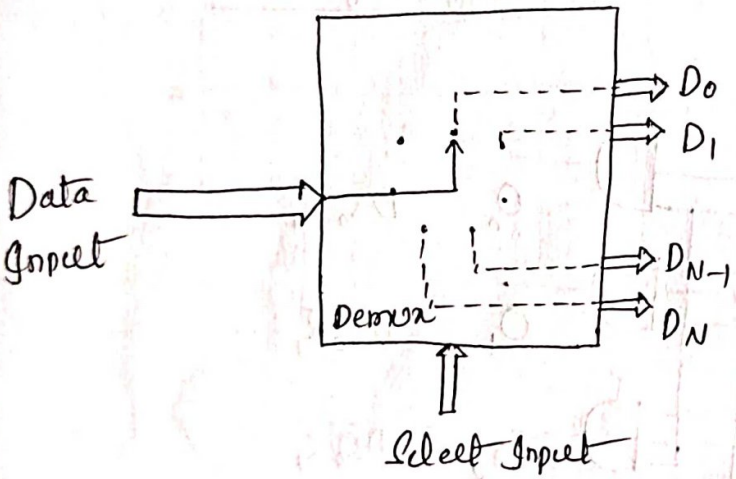


⇒ Implementation Table :-

S_0	S_1	y
0	0	C
0	1	0
1	0	C
1	1	1

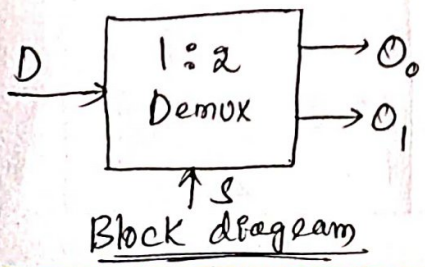
⇒ Demultiplexer :- (Data Distributor)

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 Demultiplexer :-

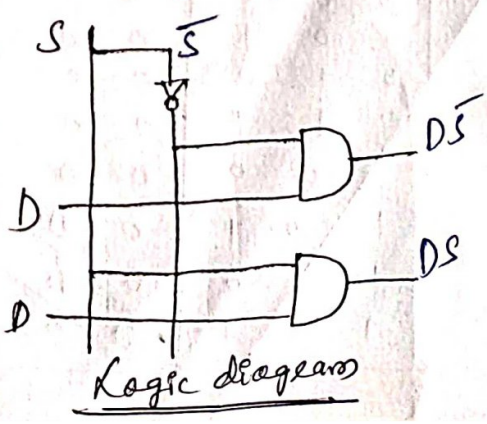
The input data line goes to all of the AND gates. The select line enable only one gate at a time, and the data appearing on the input will pass through the selected gate to the associated output line.



Block diagram

S	O
0	O_0
1	O_1

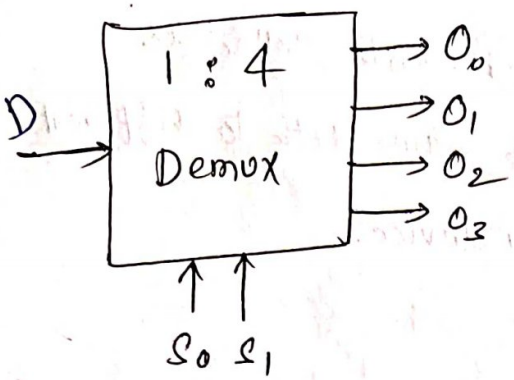
Truth Table



Logic diagram

⇒ 1 Line to 4 Line Demultiplexer :—

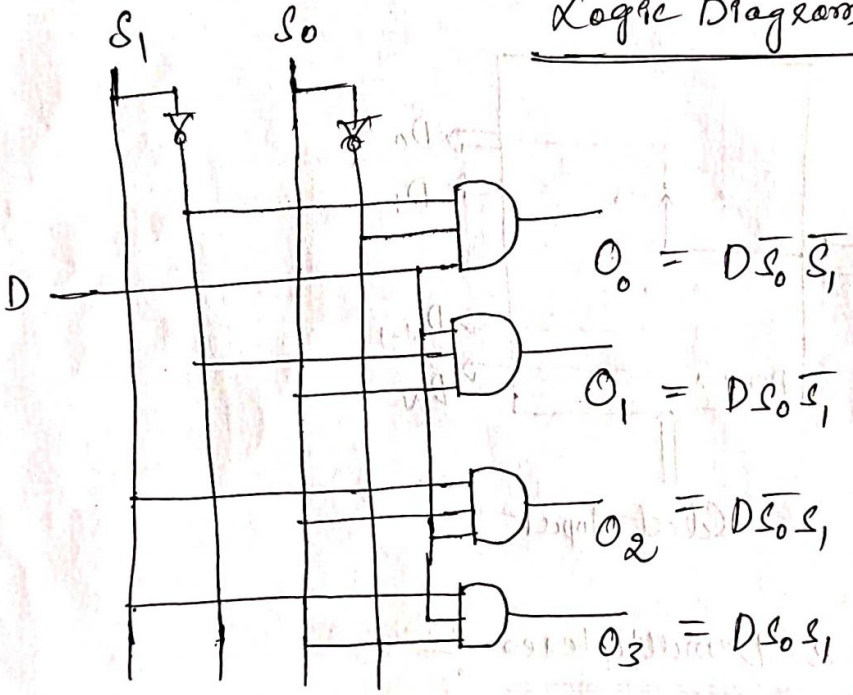
Block diagram



Truth table

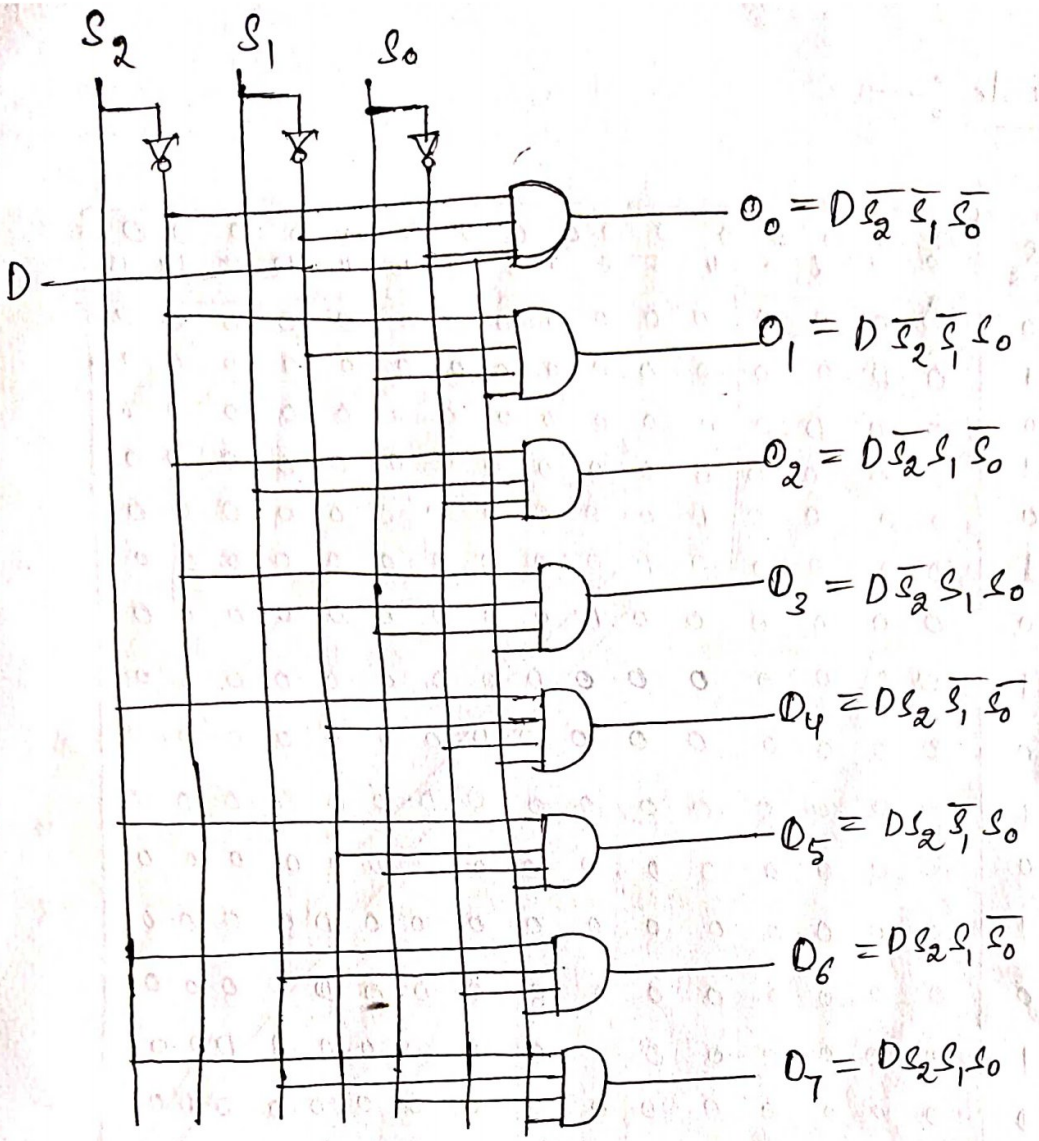
S_1	S_0	O_3	O_2	O_1	O_0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

Logic Diagram

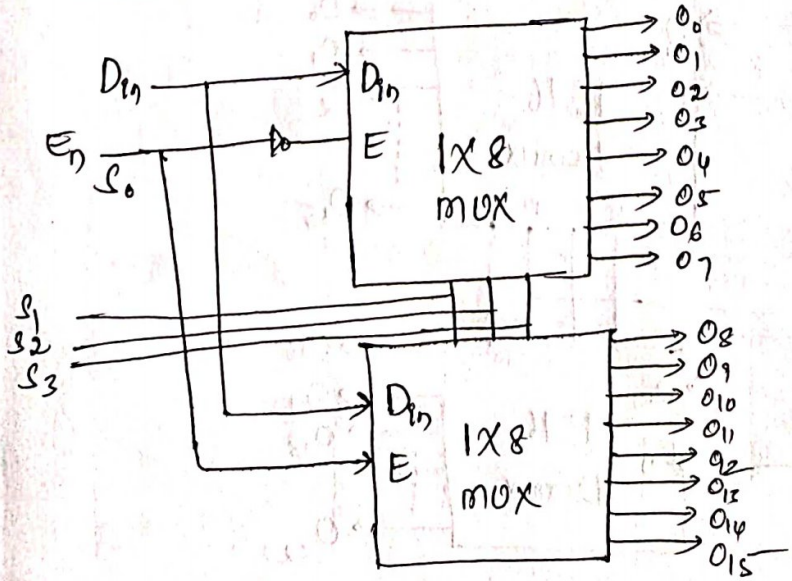


⇒ 1 Line to 8 Line Demultiplexer :—

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



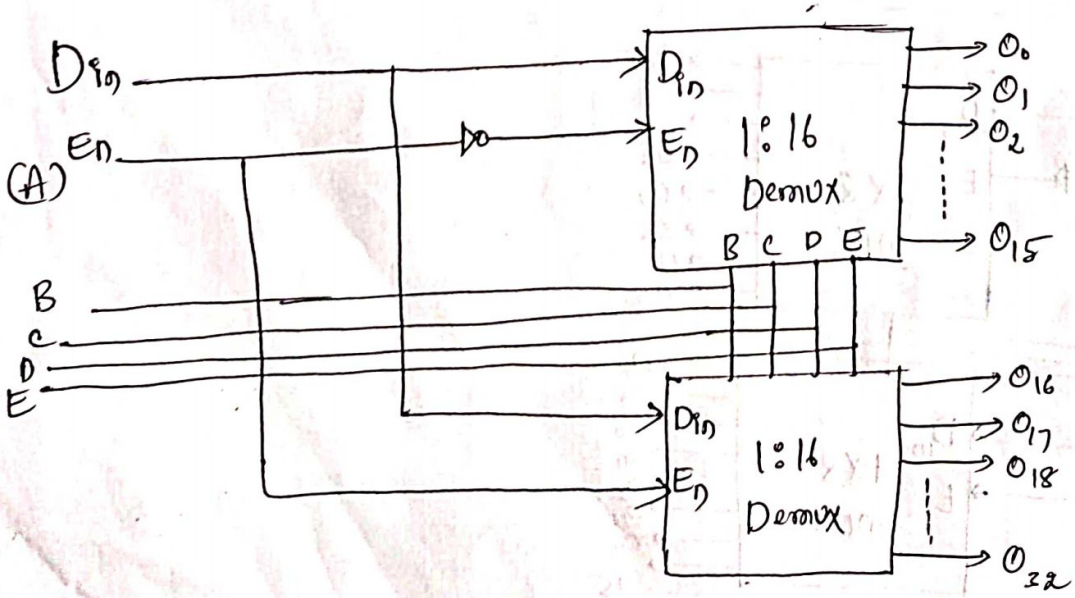
⇒ Design and Implementation of 1X16 Demultiplexer using 1X8 Demultiplexer :-



⇒ Truth Table :—

S_0	S_1	S_2	S_3	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}
0	0	0	0	D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	D	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	D	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	D	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	D	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	D	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	D	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	D	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	D	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	D	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	D	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	D	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	D	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	D	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	D	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	D

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



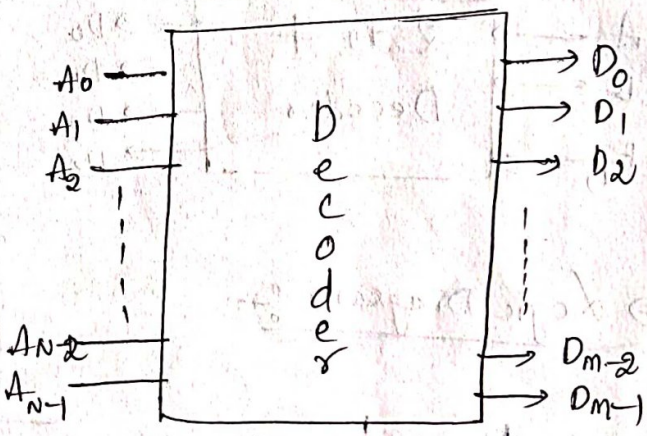
Decoder :-

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 o/p, while all the other outputs remains high.

$N \times 2^N$ Decoder



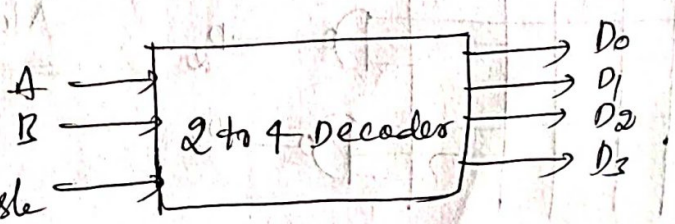
2^N input code figure

* $n = 3 \Rightarrow 3 \times 2^3 \Rightarrow$ 3X8 Decoder
* $n = 2 \Rightarrow 2 \times 2^2 \Rightarrow$ 2X4 Decoder

(Q4) Design and Implementation of 2 to 4 Decoder with active high output (or) using AND gates.

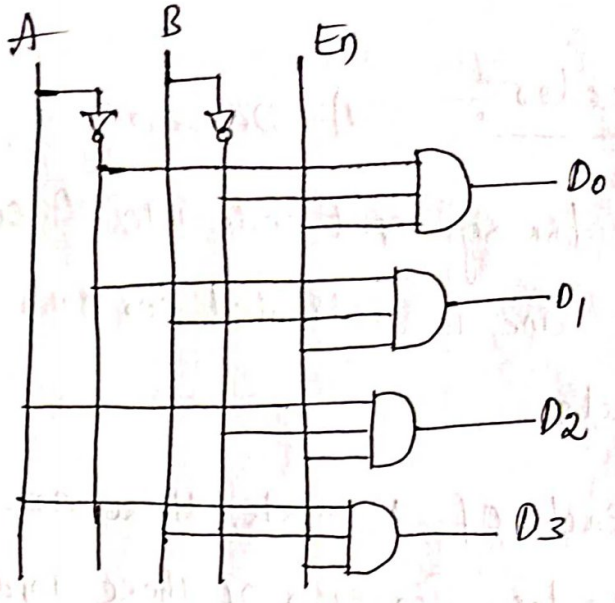
Sol

Block diagram →



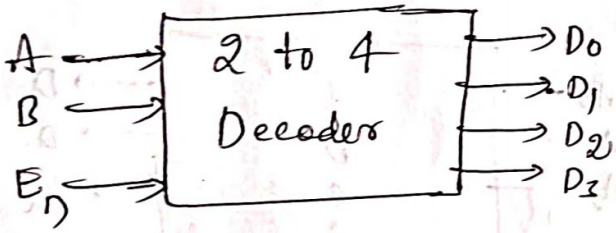
Truth Table

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



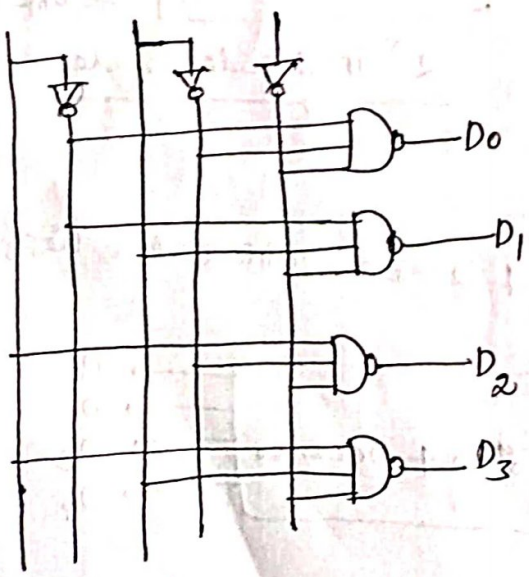
Q2 Design & Implementation of 2 to 4 Decoder with active low output (or) using NAND gates.

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.

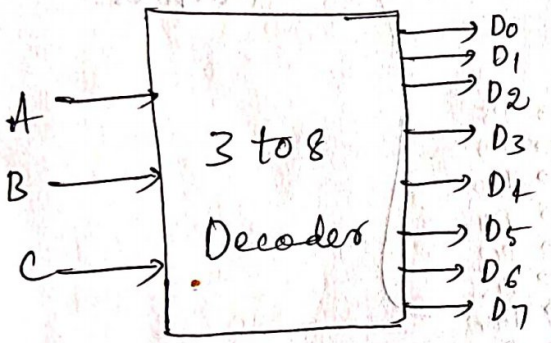
Note ②

For active Low outputs NAND gates are used.

⇒ 3X8 Decoder :-

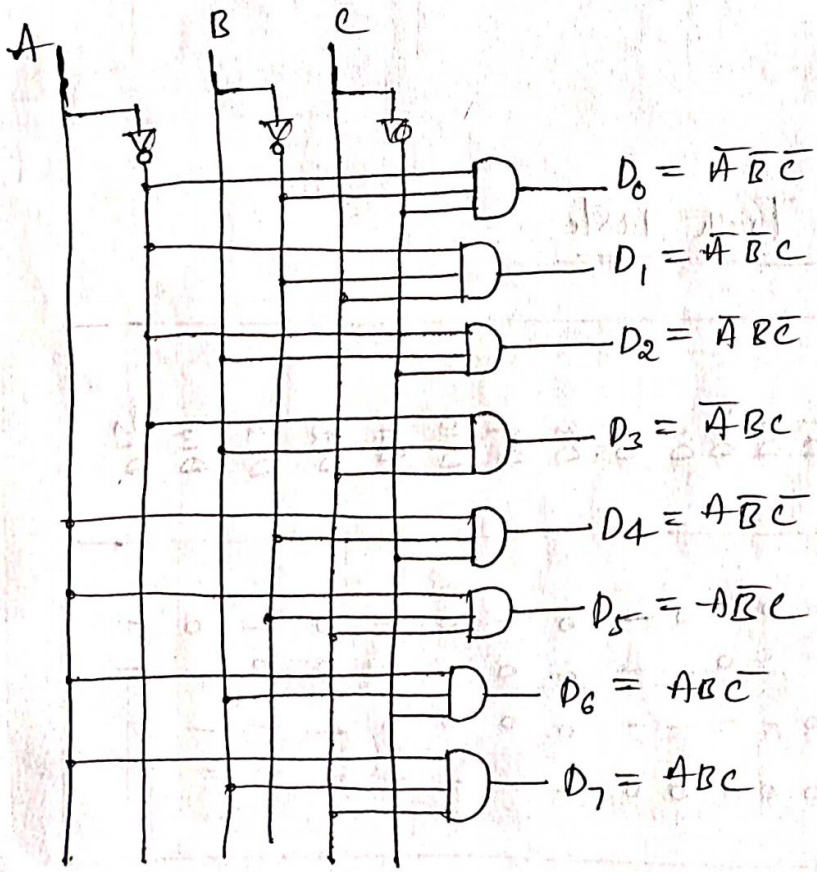
A 3 to 8 Decoder has 3 inputs and 8 outputs. It uses all AND gates, therefore the outputs are active highs. For active low outputs, NAND gates are used. It can be called a 3line to 8 line decoder because it has three input lines and eight output lines. It can also be called as a binary to Octal Decoder.

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

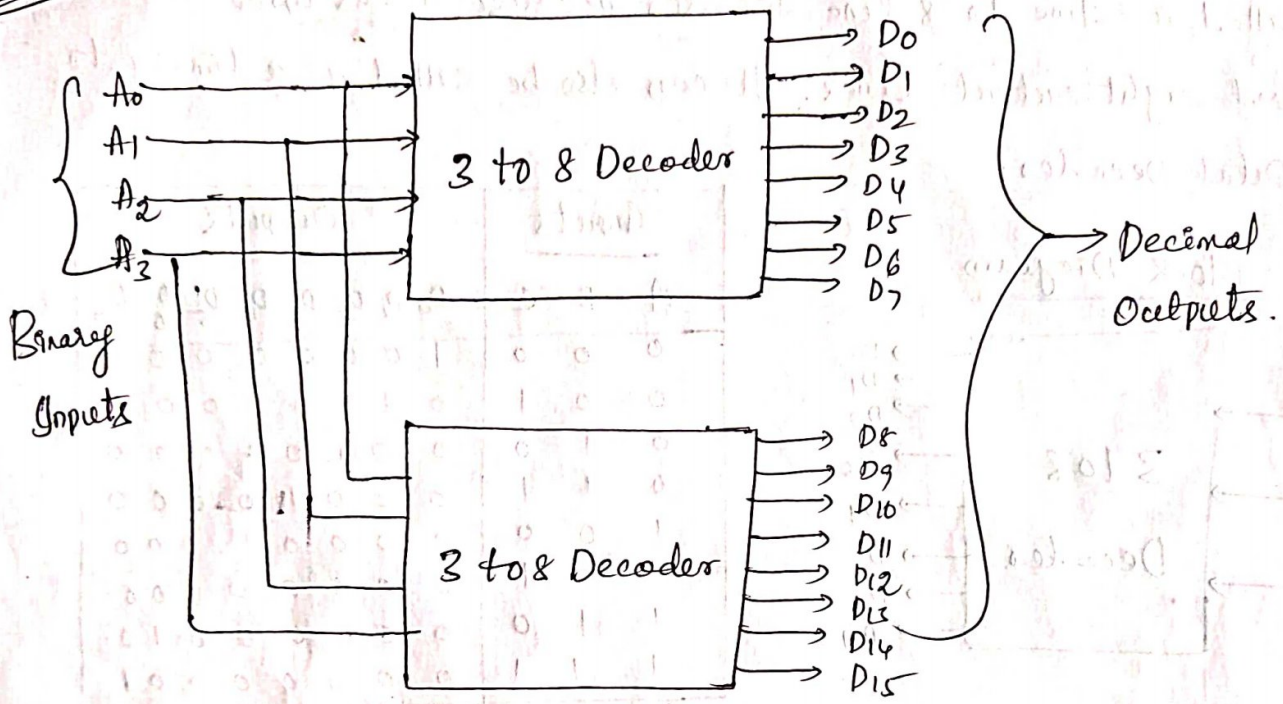


← Logic diagram

Q1 Design & Implement 4 to 16 line Decoder from two 3 to 8 line Decoders.

Sol

Block diagram



Truth Table

Binary Inputs				Decimal Output													
A_3	A_2	A_1	A_0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}
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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

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 is provided by feedback from the output back to the input.

Examples of Sequential circuits are

* Counters
* Sequence generators
* Shift registers
* Serial adders

⇒ **Block diagram :-**

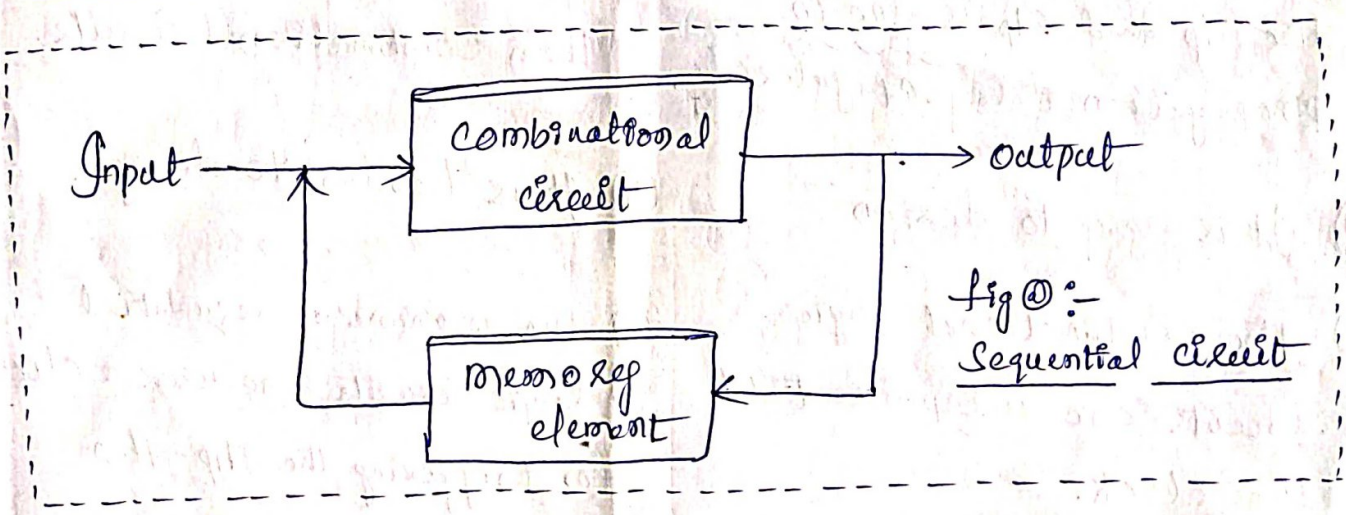


fig ① :- Sequential circuit

→ 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.

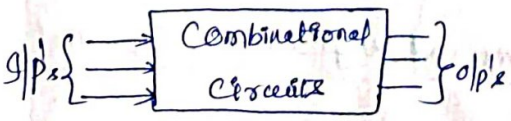
④ It is easy to design

⑤ the combinational logic circuits are independent of the clock.

⑥ the combinational digit circuit don't require any feedback.

⑦ It's 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.

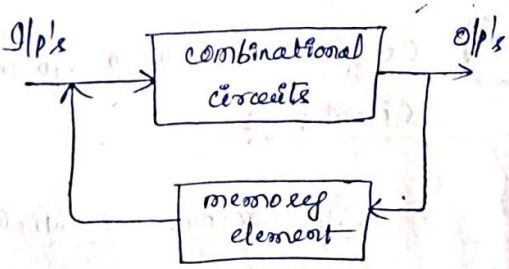
④ It is harder to design.

⑤ the maximum sequential logic circuits are uses a clock for triggering the flip-flop operation.

⑥ the sequential digital logic circuits utilize the feedbacks from outputs to inputs.

⑦ It's behaviour is described by the set of next state fun's and the set of o/p functions.

⑧ Block diagram :-



Q Comparision between Synchronous and Asynchronous Sequential Circuits :-

Synchronous Sequential circuits	Asynchronous Sequential circuits
① In Synchronous circuits, memory elements are Clocked Flip-Flop's (FF's).	① In Asynchronous circuits, memory elements are either Unclocked Flip-Flops (or) time delay elements.
② In this, the change in i/p signals can affect memory elements upon activation of clock signal	② In this, change in i/p signals can affect memory elements at any instant of time.
③ The maximum Operating speed of the clock depends on time delays involved	③ Because of the absence of the clock, asynchronous circuits can Operate faster than Synchronous circuits.
④ They are easier to design	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 doesnot 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.

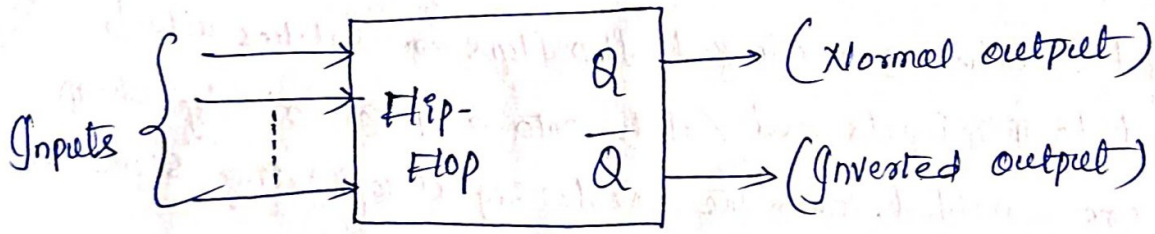
⟹ **Flip-Flops** :- The most important memory element is the flip-flop, which is made up of an assembly of logic gates.

→ 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 i/p signals which command the flip-flop to change state are called excitations.

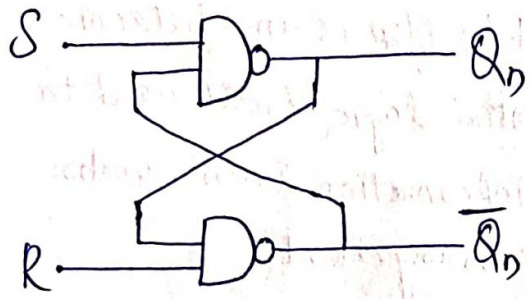
→ The applications of flip-flops are serves as a storage device It stores a '1' when its output 'Q' is a '1' and it stores a '0' its output Q is '0'. These are mainly used in Registers and counters.

Q Difference between Latches & Flip flops :-

Latch	Flip Flop
① A Latch is an electronic sequential logic circuit used to store information in an asynchronous arrangement.	① A Flip flop is an electronic Sequential logic circuit used to store information. In an synchronous arrangement
② One Latch can store one bit information, but output state changes only in response to data input.	② One flip-flop can store one bit data, but output state changes with clock pulse only.
③ Latch is an asynchronous device and it has no clock input.	③ Flip-Flop has clock input and its output is synchronised with clock pulse.
④ Latch holds a bit value and remains constant until new inputs force it to change.	④ Flip Flops holds a bit value and it remains constant until clock pulse is received.
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	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 :-

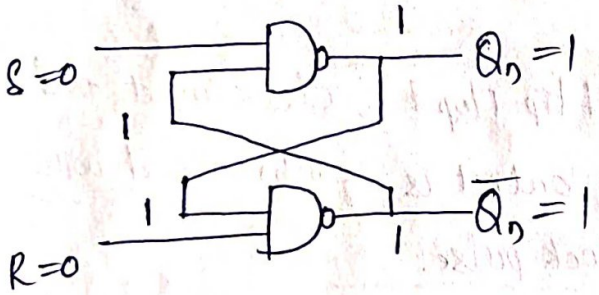


Truth Table

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) :-

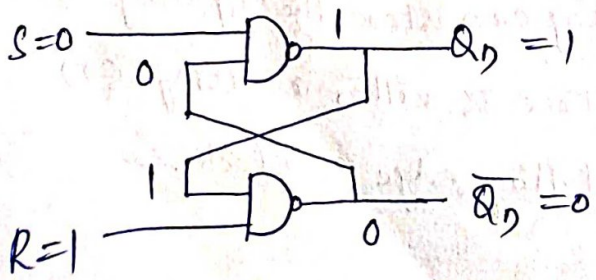
$S=0, R=0$



When $S=0, R=0$ the outputs are $Q_n=1$ & $\bar{Q}_n=1$. So, this is Invalid state / Indetermined.

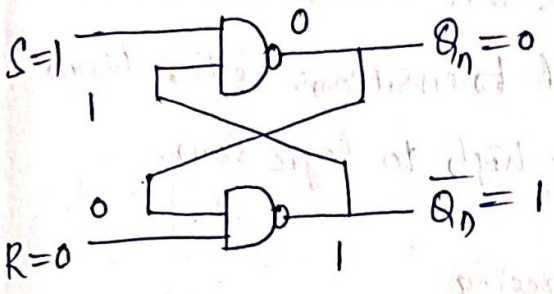
Case ii) :-

When $S=0, R=1$



∴ When $S=0, R=1$ the outputs are $Q_n=1$ & $\bar{Q}_n=0$.

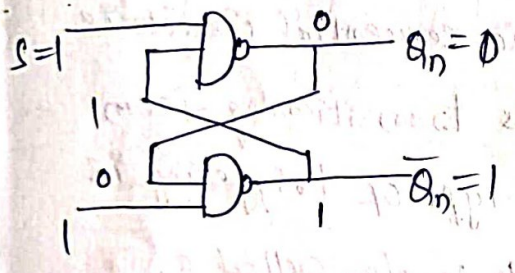
Case (iii) :- $S=1, R=0$



∴ When $s=1, R=0$ the outputs are $Q_n=0$; $\bar{Q}_n=1$

Case (iv) :-

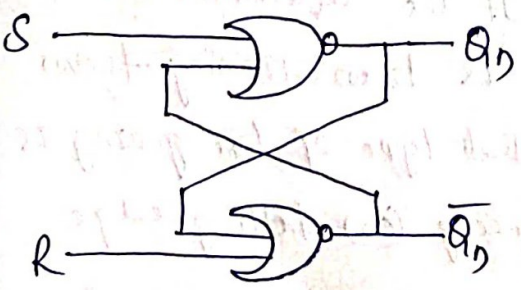
$S=1$; $R=1$



∴ 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 NOR gates.

⟹ The above procedure is same here also. Once we do same 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.

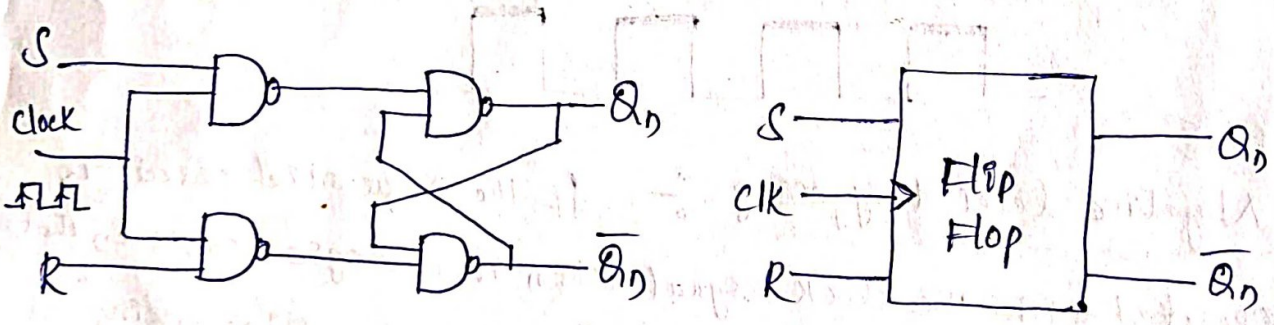
→ Flip-Flops are classified into 4 types

① SR Flip-Flop ③ JK Flip-Flop

② D Flip-Flop ④ T Flip-Flop.

→ Clocked SR FlipFlop :-

⇒ Block diagram of SR Flip-Flop :-



ⓐ Circuit diagram ⓑ Block diagram

ⓒ Truth table

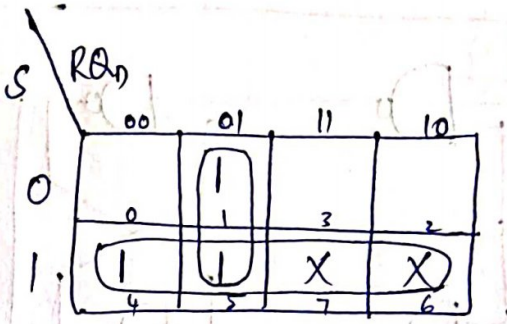
CLK	S	R	Q_n $\bar{Q}_n$
1	0	0	No change
1	0	1	0 1 (Reset)
1	1	0	1 0 (Set)
1	1	1	Invalid state

(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

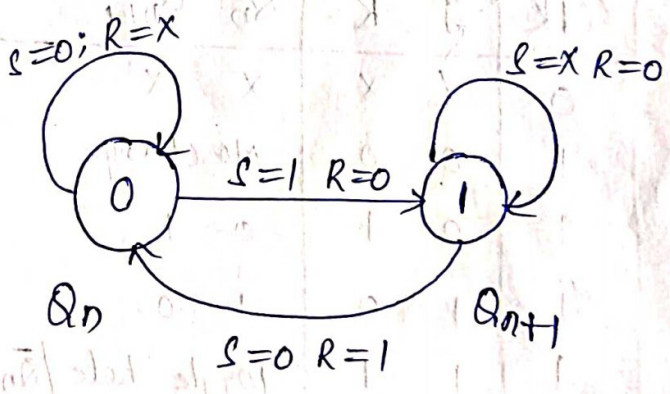


$$Q_{n+1} = S + \bar{R} Q_n$$

(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



① $Q_n\ Q_{n+1}$: 0 0 → S R : 0 0, 0 1 → 0 X ✓

② $Q_n\ Q_{n+1}$: 0 1 → 1 0 ✓

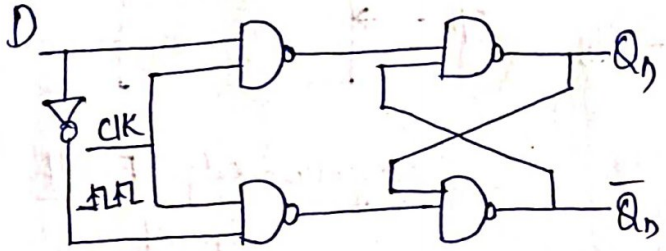
③ 1 0 → 0 1 ✓

④ 1 1 → 0 0, 1 0 → X 0 ✓

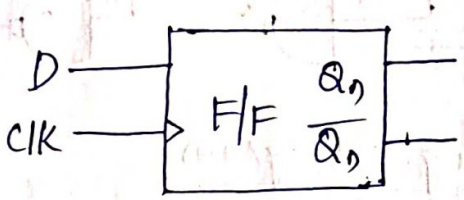
* In SR Flip Flop S=1 & R=1 then the state is indeterminate state. This drawback is avoided in JK Flip Flop.

⇒ 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

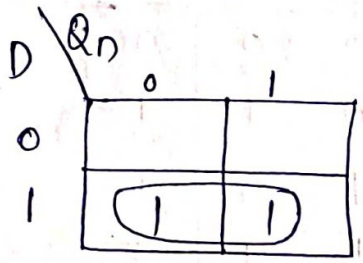
→ D Flip-Flop is also called as Delay (or) Data Flip Flop. It is used to find delay b/w the set and Reset.

(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

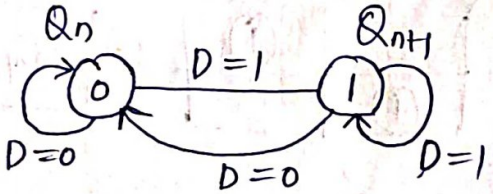


$$Q_{n+1} = D$$

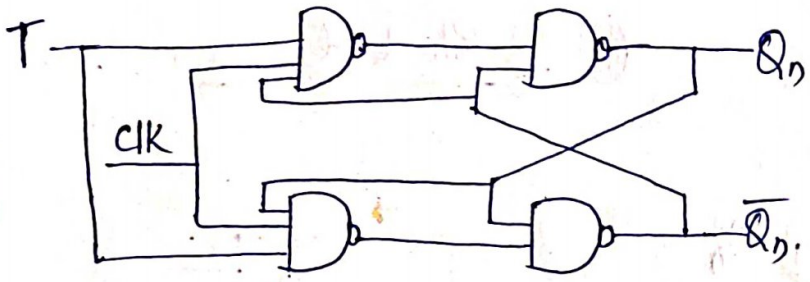
(f) Excitation Table

Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

(g) State diagram :-



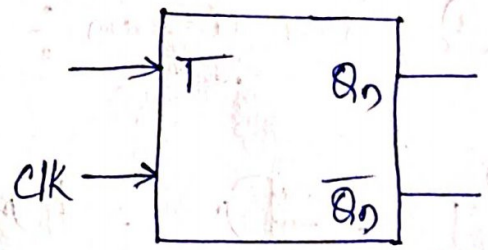
⇒ T- Flip Flop :-



(a) Logic diagram

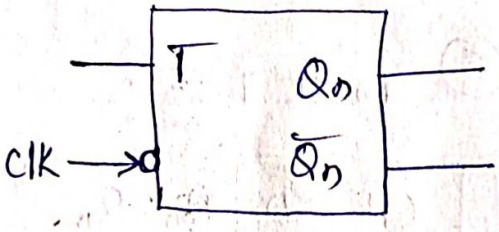
(b) positive edge T-F/F

Symbol



Negative edge T-F/F

Symbol



(c) Truth Table

CLK	T	Q_n $\overline{Q_n}$	
1	0	Q_n $\overline{Q_n}$	→ no change state
1	1	Toggle state	

(d) 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

(f) Excitation Table

Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

(e) Characteristic equation

T \ Q_n	0	1
0		①
1	①	

$$Q_{n+1} = T\overline{Q_n} + \overline{T}Q_n$$

COUNTERS :—

①

→ A digital counter is a set of flip-flop (FFs) whose state change in response to pulses applied at the input to the counter.

→ 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	synchronous 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.	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 not clocked simultaneously	2. All the FFs are clocked simultaneously.
3. Design and implementation is very simple even for more number of states	3. Design and implementation becomes tedious and complex as the number of states increases
4. Main drawback of these counters is their low speed as the clock is propogated through a number of FFs before it reaches the last FFs	4. Since clock is applied to all the FFs simultaneously the total propogation delay is Equal to the propogation delay of only one FF. Hence they are faster.

Asynchronous Counters :-

→ 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 $\overline{R}$ using K-map or any other method.

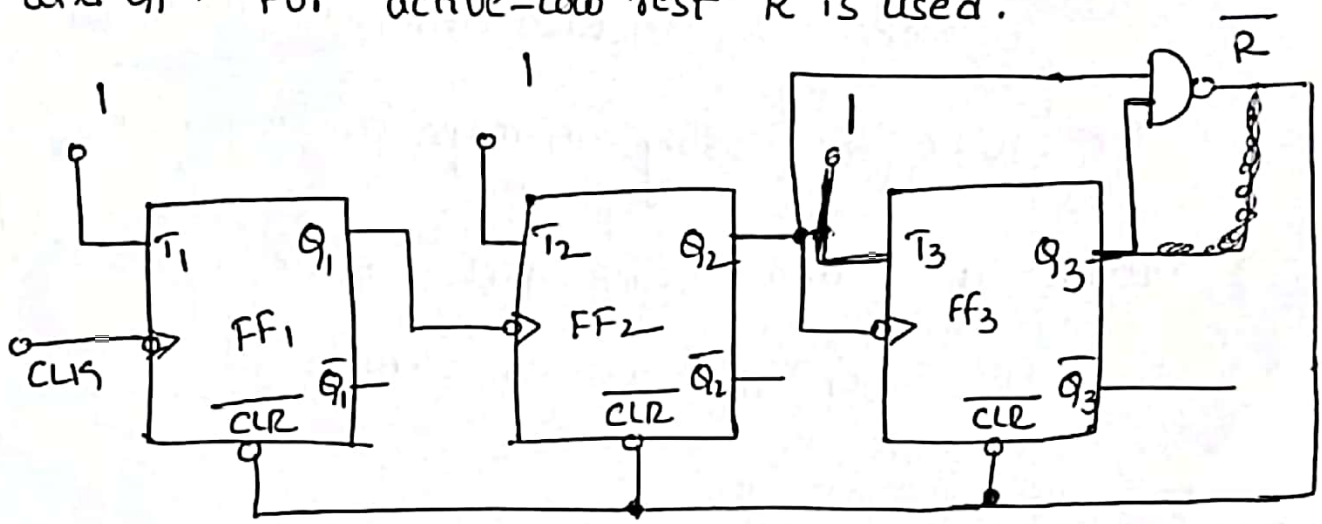
→ Provide a feedback such that R or $\overline{R}$ resets all the FFs after the desired count.

Design of a mod-6 asynchronous counter using T FFs

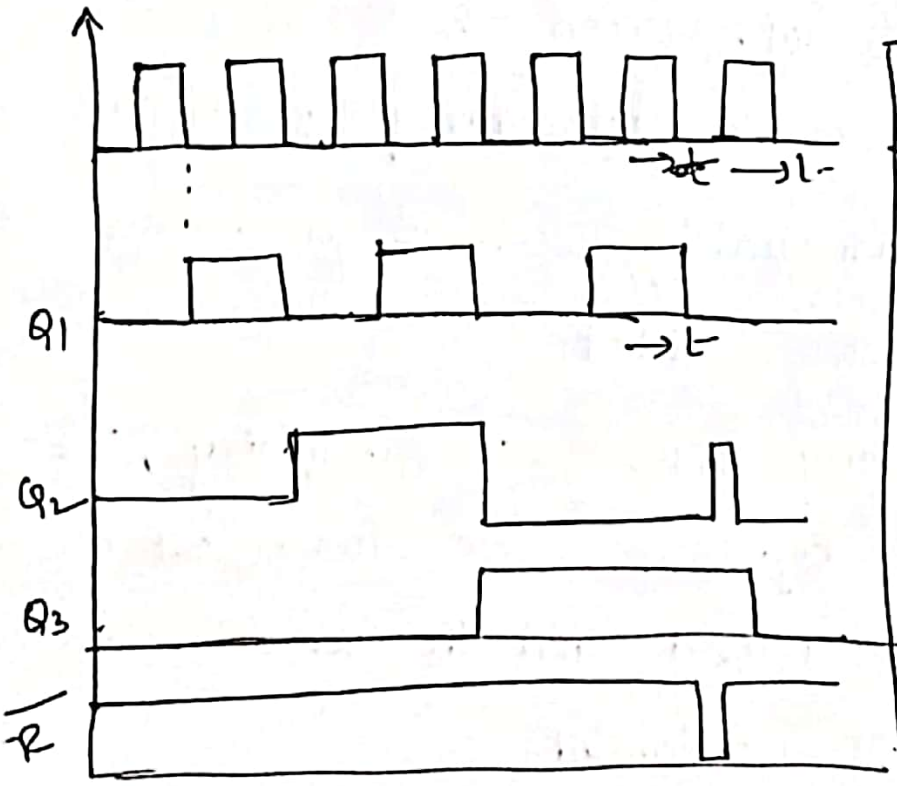
A mod-6 counter has six stable states 000, 001, 010, 011, 100, and 101. When the sixth clock pulse is applied, the counter temporarily goes to 110 state, but immediately resets to 000 because of the feed back provided.

→ Here we are using 3 FFs 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 rest $\overline{R}$ is used.



Logic diagram.



After Pulses	State Q_3	Q_2	Q_1	R
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 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.

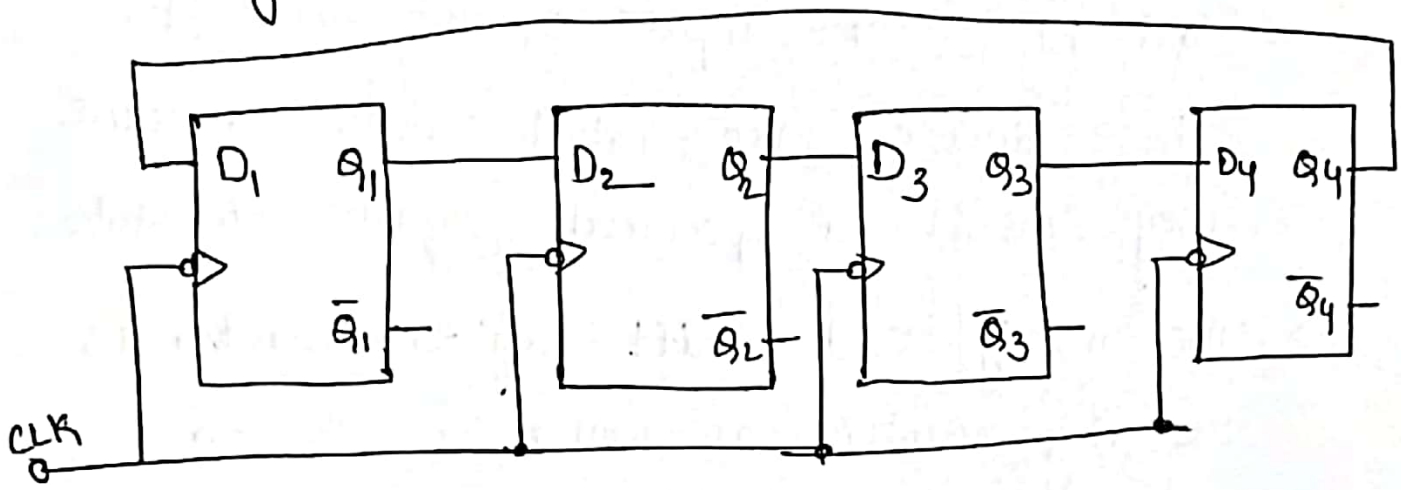
These devices are called counters because they Exhibit a specified sequence of states.

→ The mostly used shift register counter is the ring counter as well as the twisted ring counter.

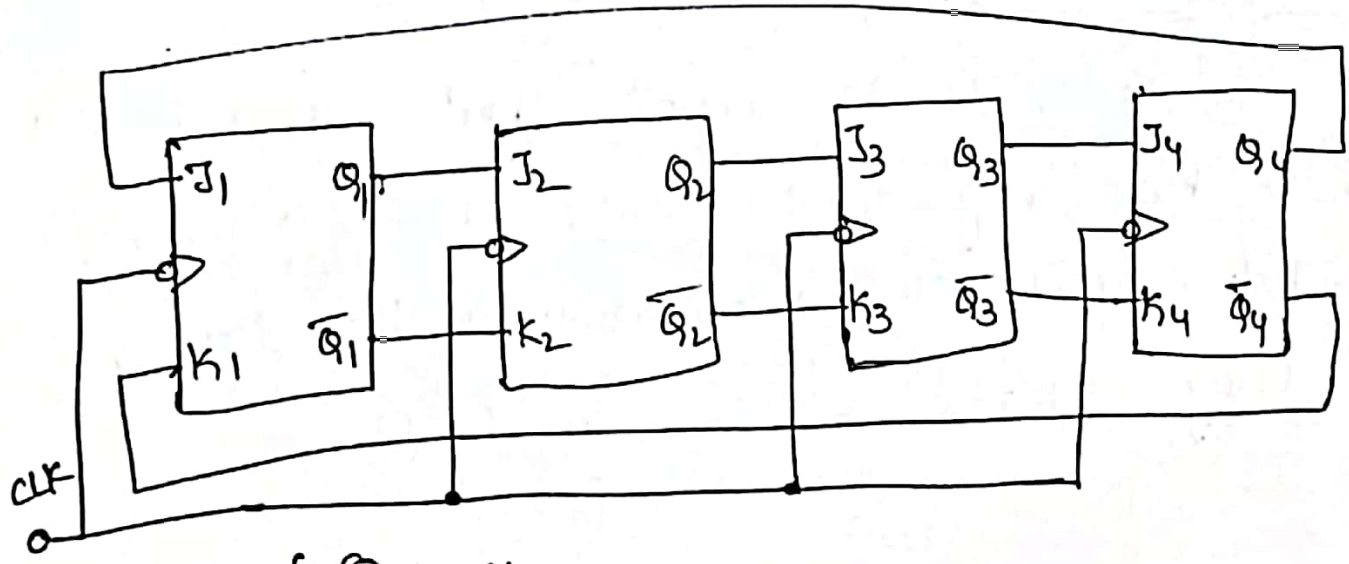
Ring counter :-

→ This is the simplest shift register counter. the basic ring counter using DFFs is shown below fig @. The realization of this counter using J-K FFs is shown in fig ⓑ.

→ 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 therfore, 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, where as n-bit ripple counter can count 2^n ~~states~~ bits.

→ It is Entirely a synchronous operation and requires no gates Exterahal to FFs, it has the further advantage of being very fast.

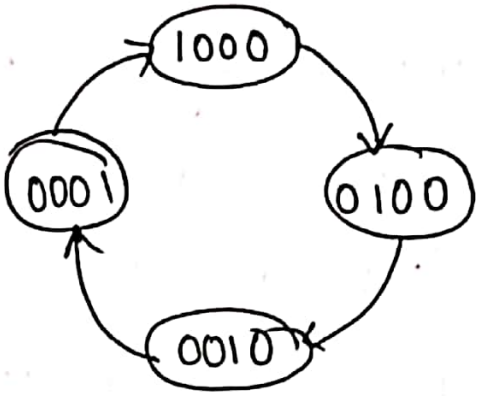
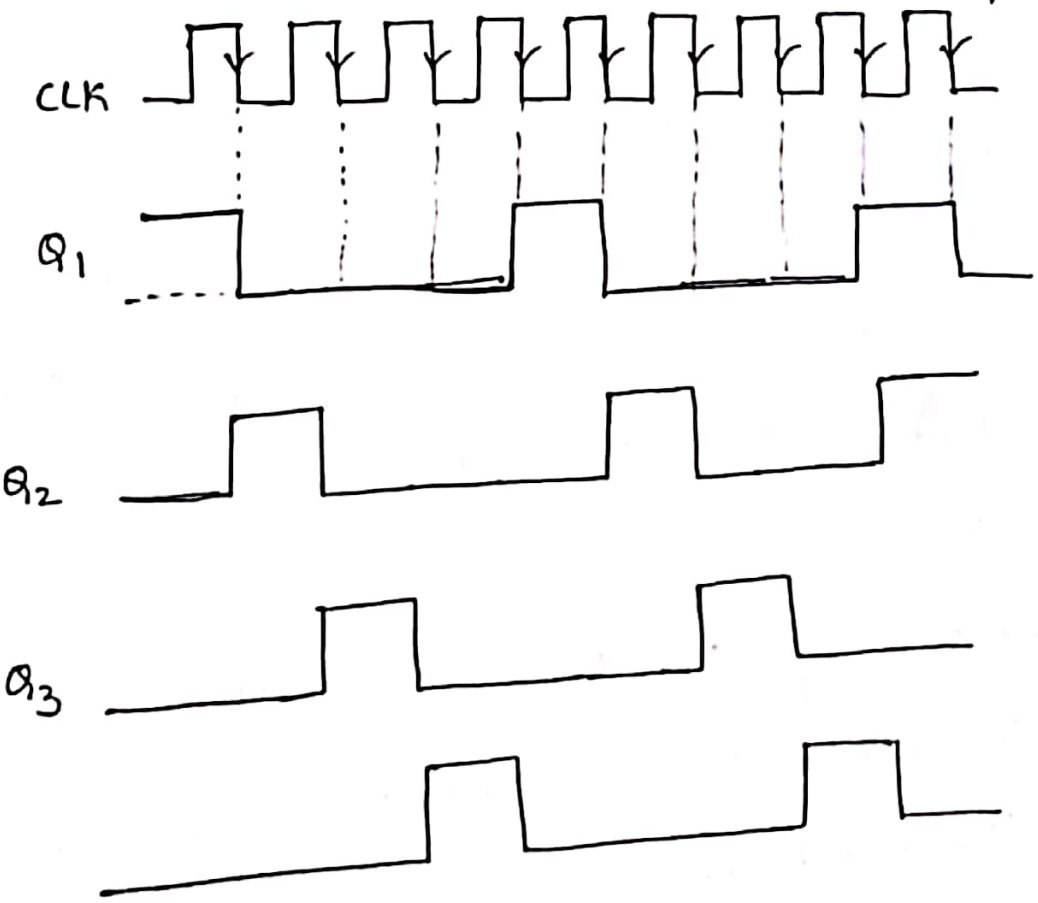


fig :- State diagram.

Q_1	Q_2	Q_3	Q_4	After clockpulse
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 Aritnmetic:

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 bulk 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. A:n 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)

i. When the signs of A and B are identical ,add the two magnitudes and attach the sign of A to the result.

ii. 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.

iii. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Algorithm: (Subtraction with Signed-Magnitude Data)

i. 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.

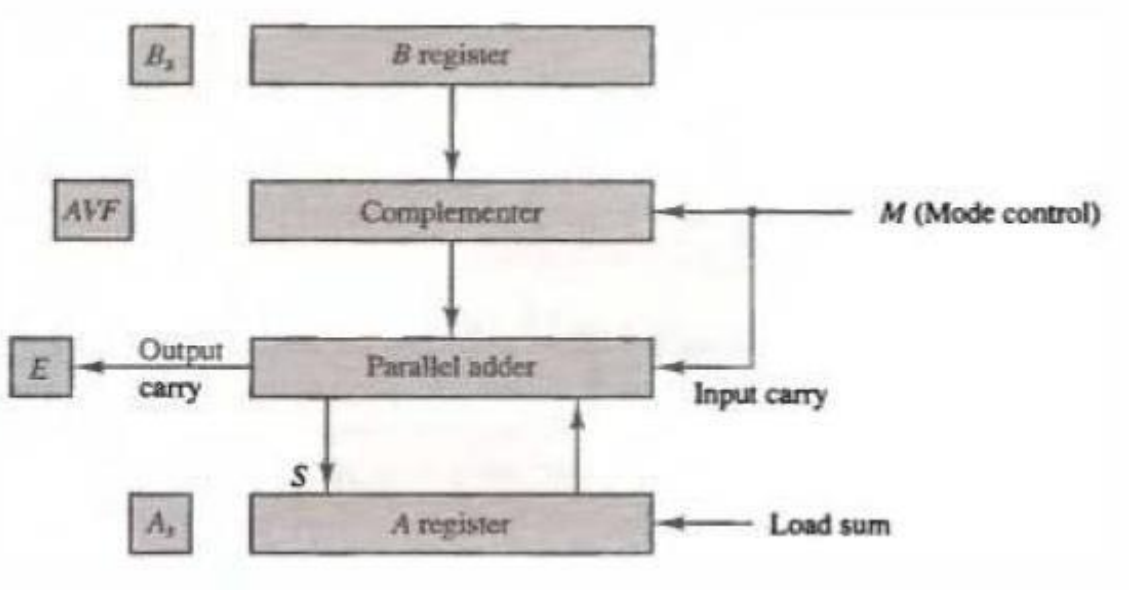
Consider now the hardware implementation of the algorithms above.

- 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 l's complement of B is applied to the adder, the input carry is 1, and output $S = A + \overline{B} + 1$. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A - B.

Hardware Algorithm

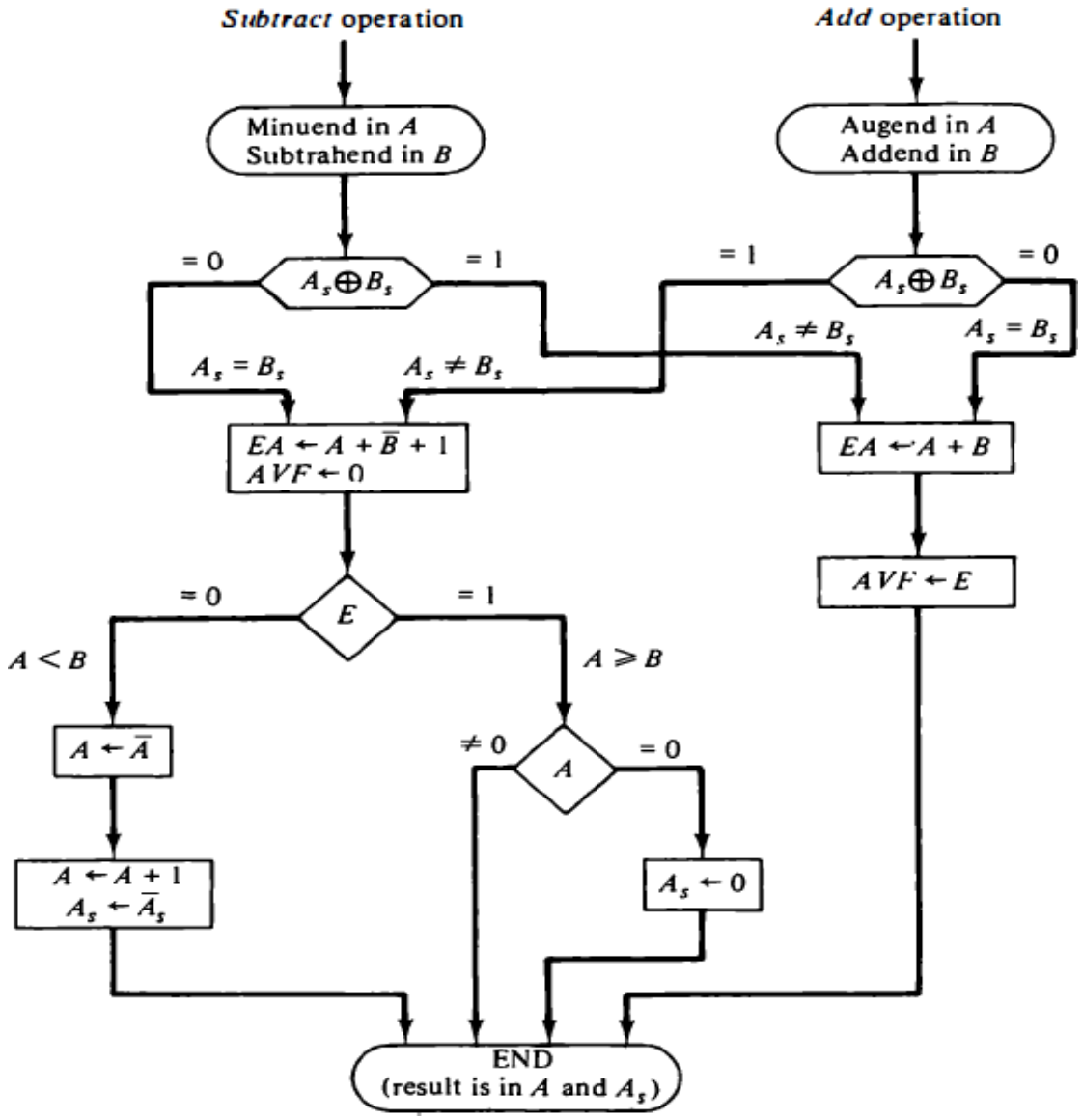
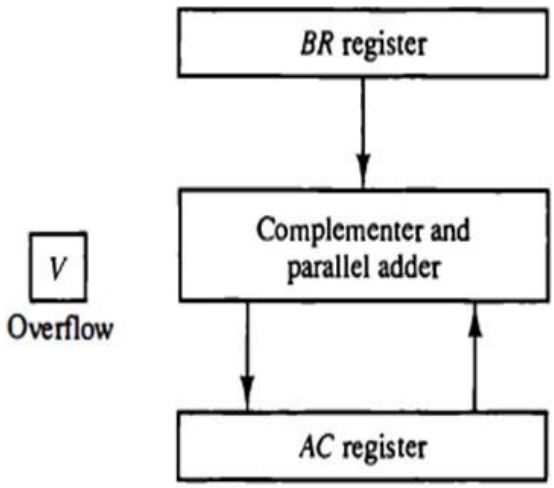


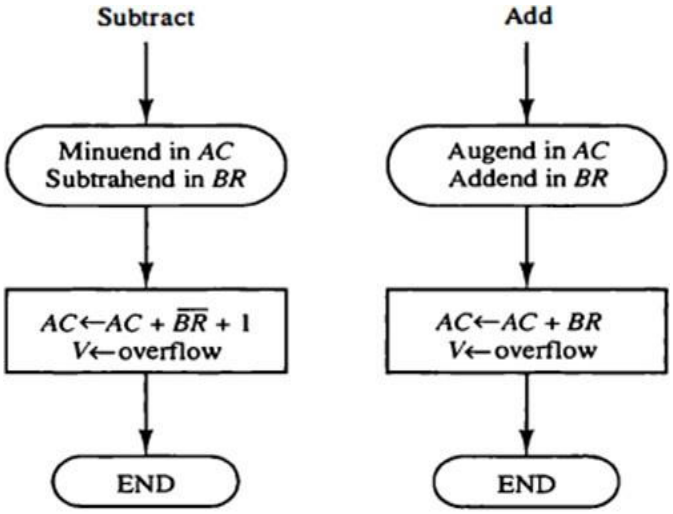
Figure (j): Flowchart for add and subtract operations

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.



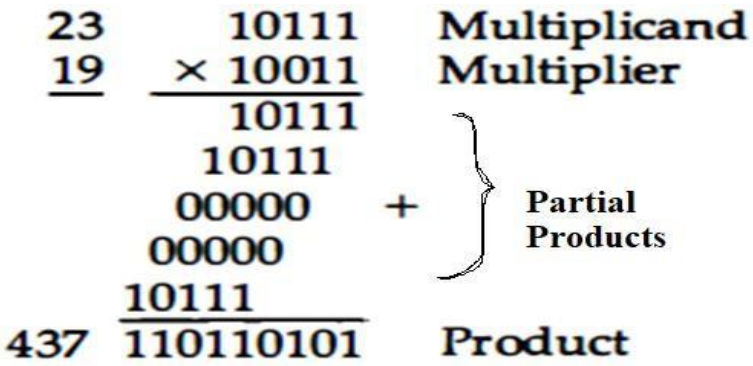
Figure(a): Hardware for addition & subtraction of 2's complement numbers



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.



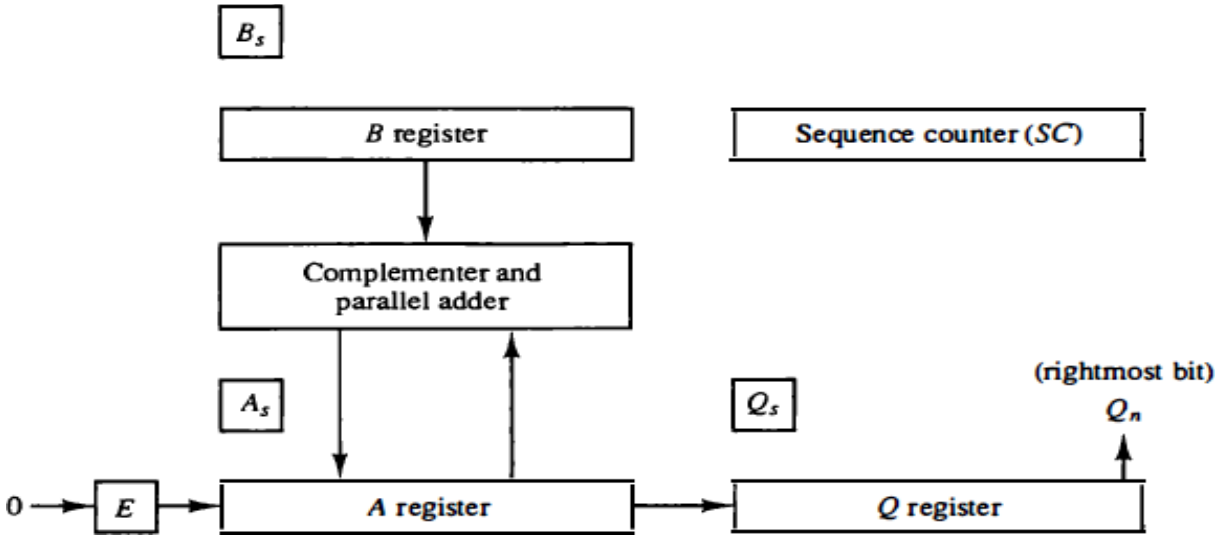
The process of multiplication:

- 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 Bs and Qs, 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 Bs and Qs, 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.

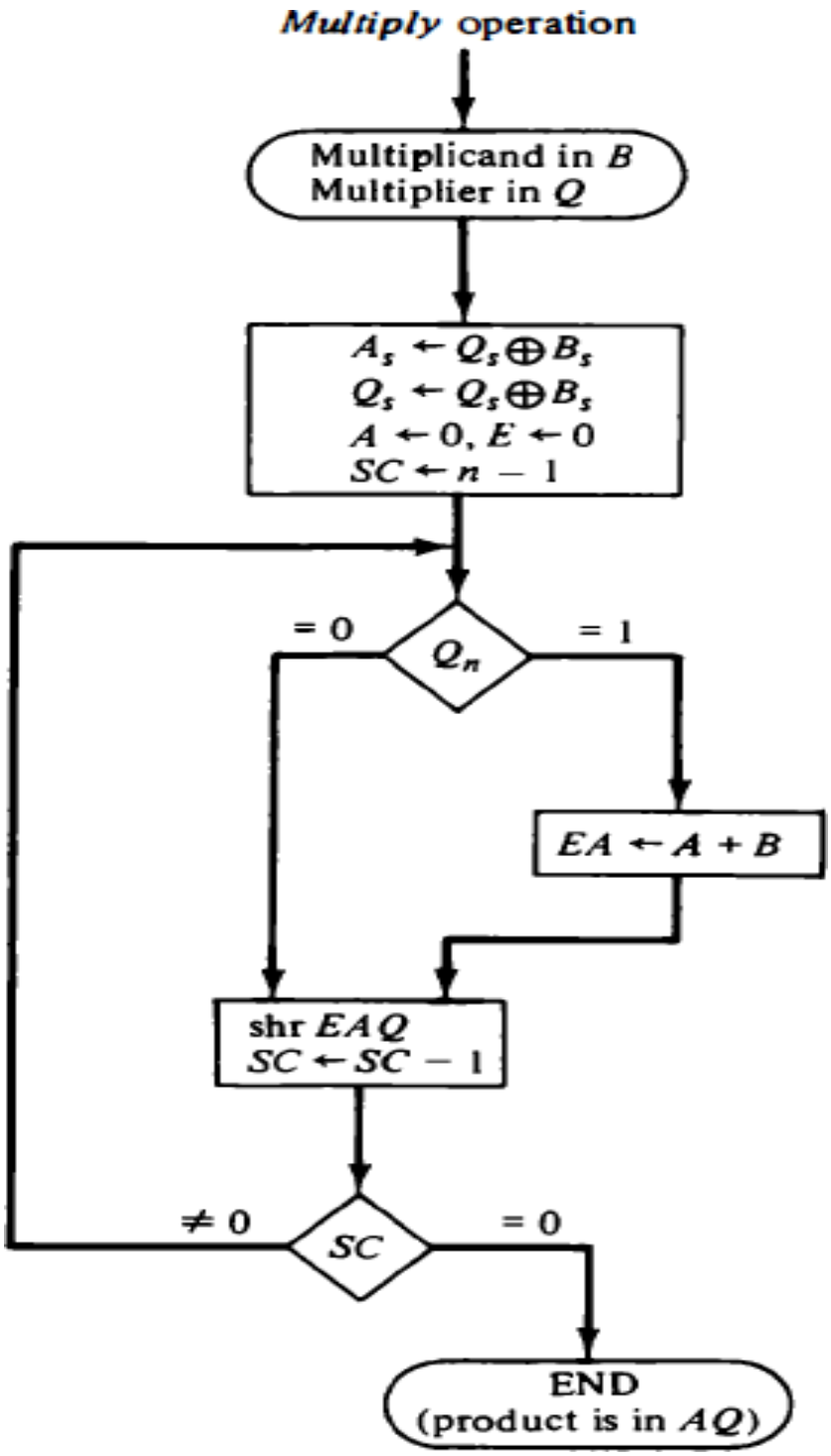
→After the initialization, the low-order bit of the multiplier in Qn is tested.

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(l): Flowchart for multiply operation.

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
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		10111		
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 O'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:

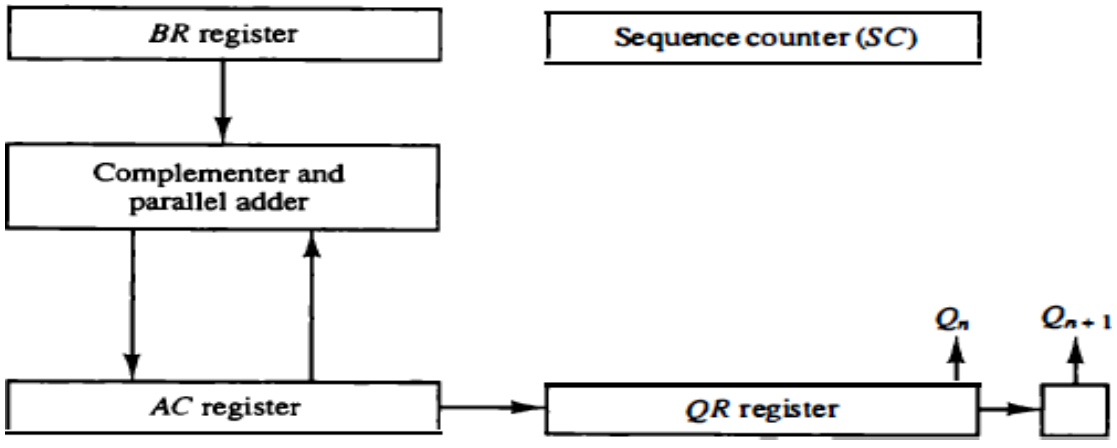


Figure (n): Hardware for Booth Algorithm

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.

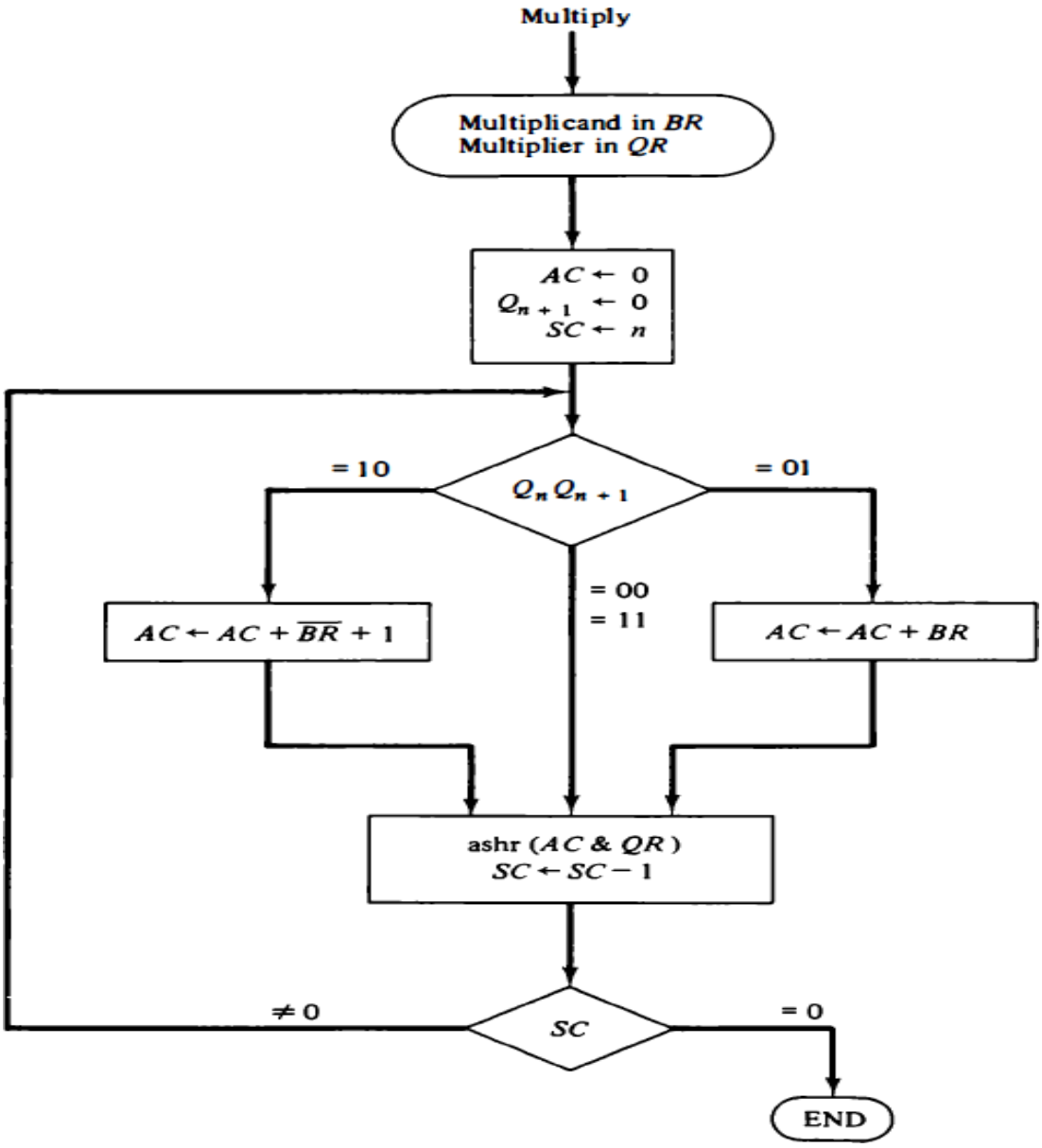


Figure (o): Booth Algorithm for multiplication of 2's complement numbers

Example: multiplication of (- 9) x (- 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	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		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:		
$B = 10001$	11010	Quotient = Q
	)0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Figure (g): 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 Qn 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 o f the quotient is **plus**. If they are unalike, 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 Qn 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, Qn 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 Qn.

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

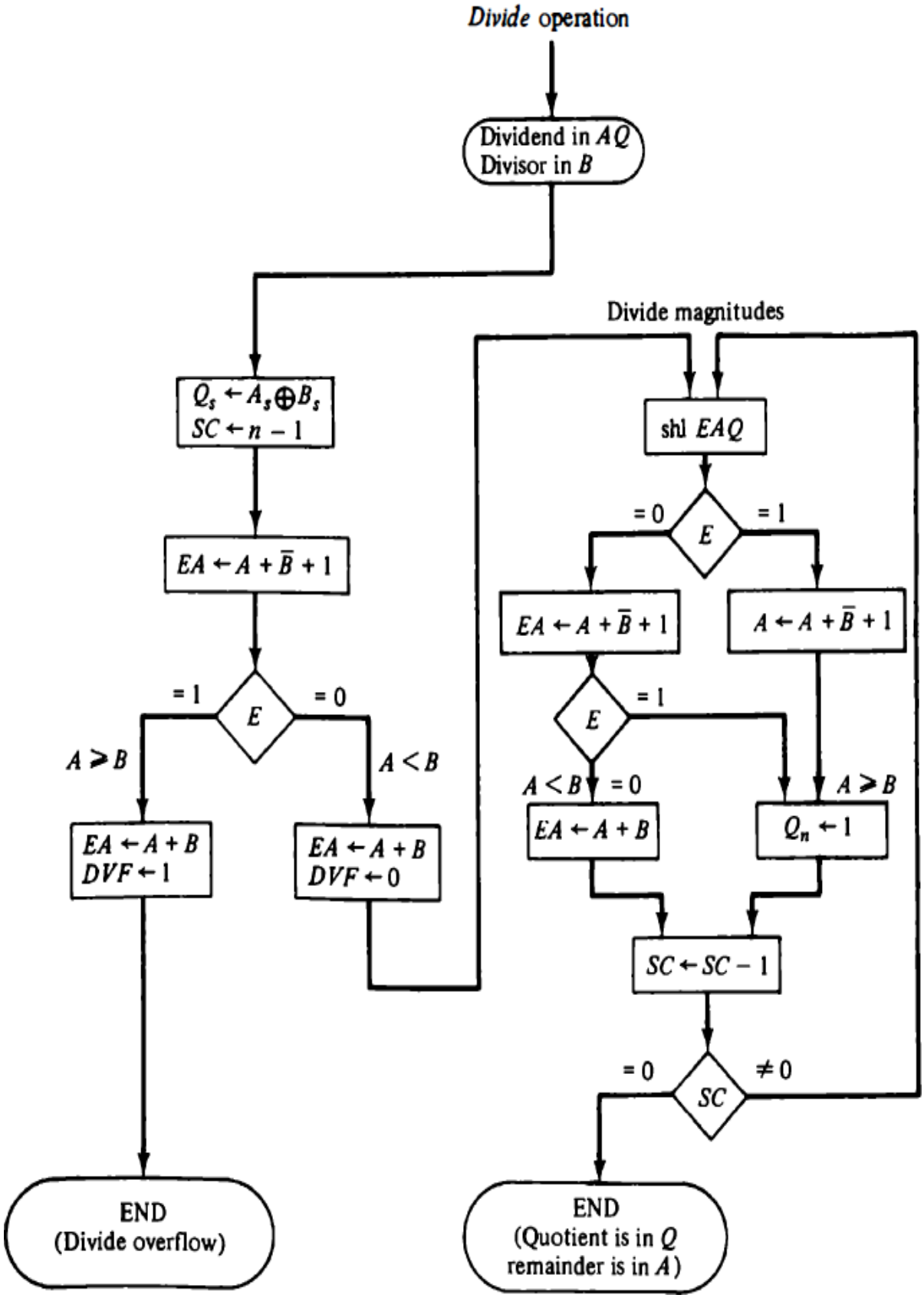


Figure (r): Flowchart for Divide operation

Divisor $B = 10001$, $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
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.

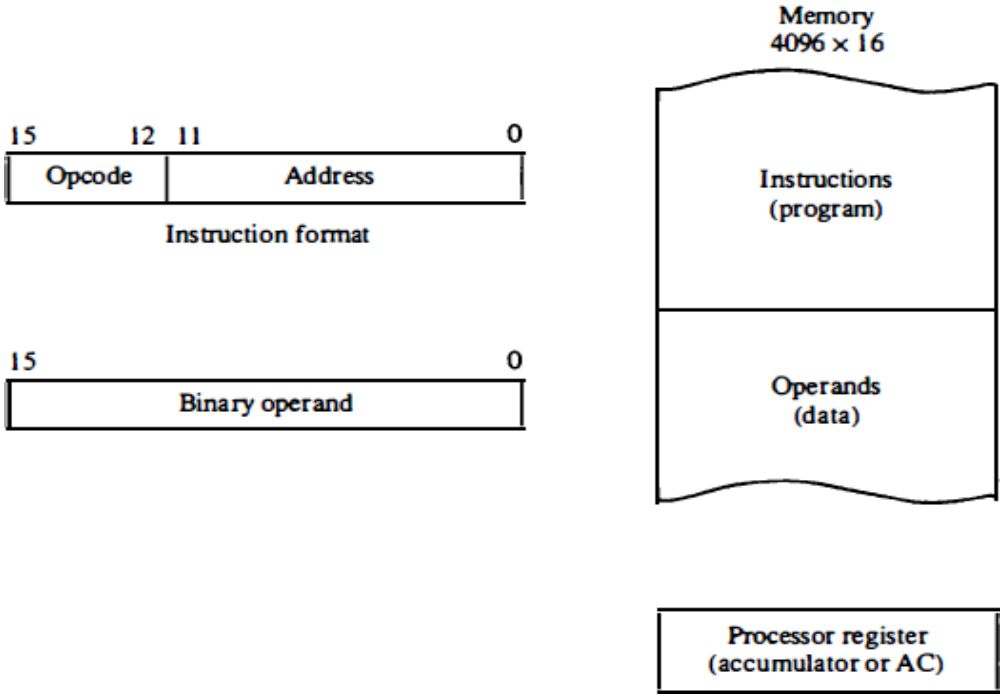


Figure (k): Stored program organization

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**.

- When the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found, the instruction is said to **an indirect address**. One bit of the instruction code can be used to distinguish between a direct and an indirect address.
- An **effective address** is the address of the operand.

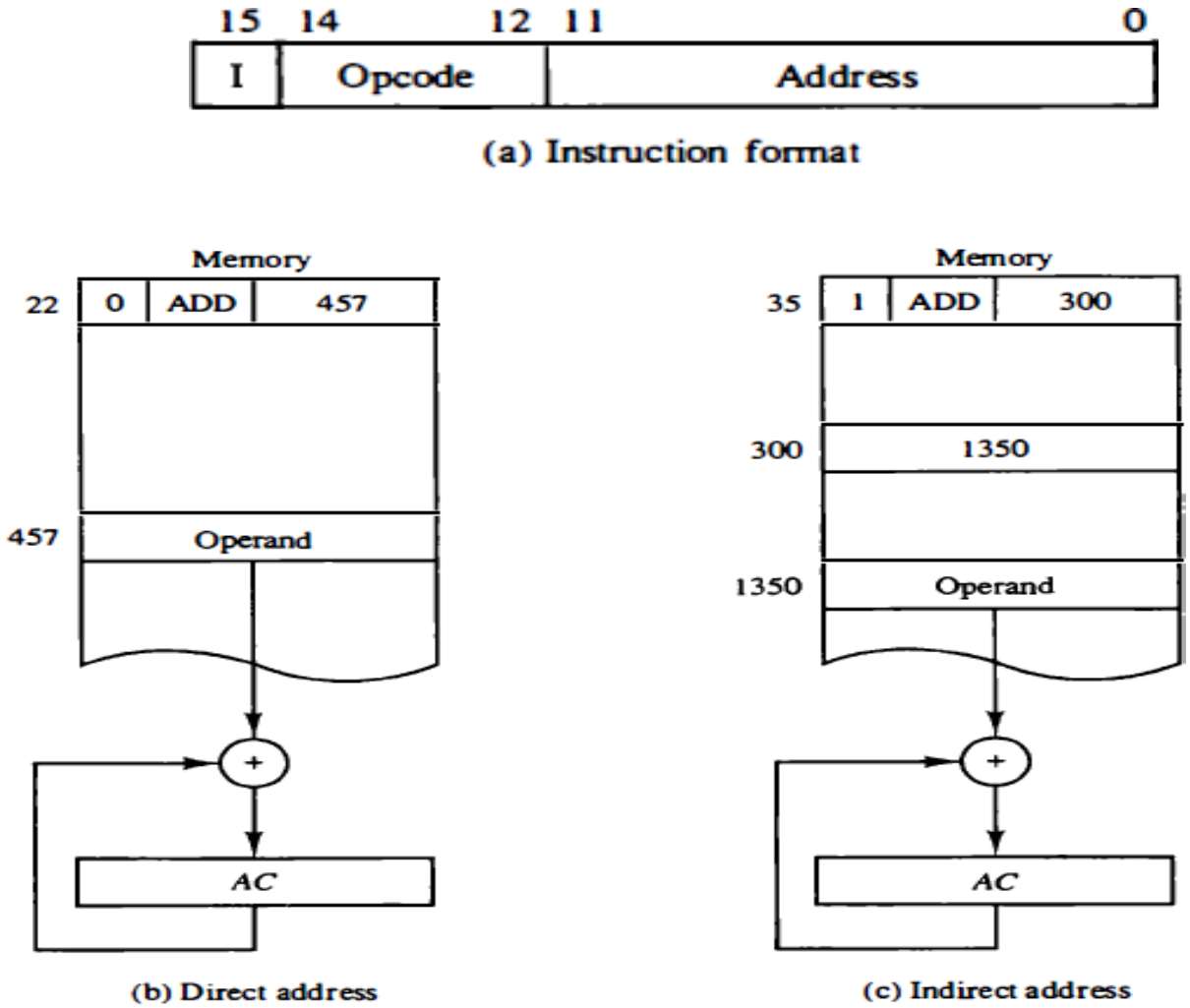


Figure (l): Demonstration of direct and indirect address.

Computer Registers:

Computer instructions are normally stored in 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 (f), a brief description of their function and the number of bits that they contain also given.

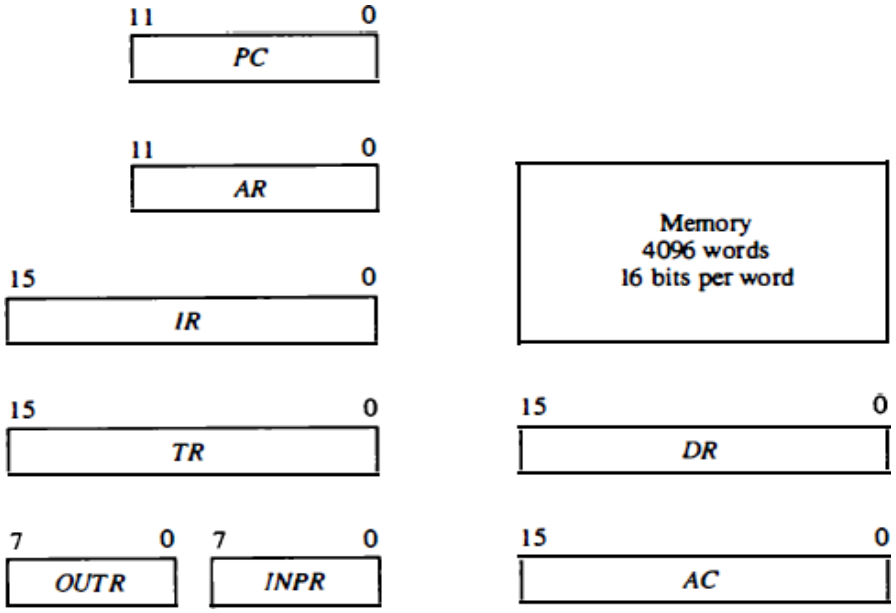


Figure (m): Basic computer registers and memory.

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 (f): 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).

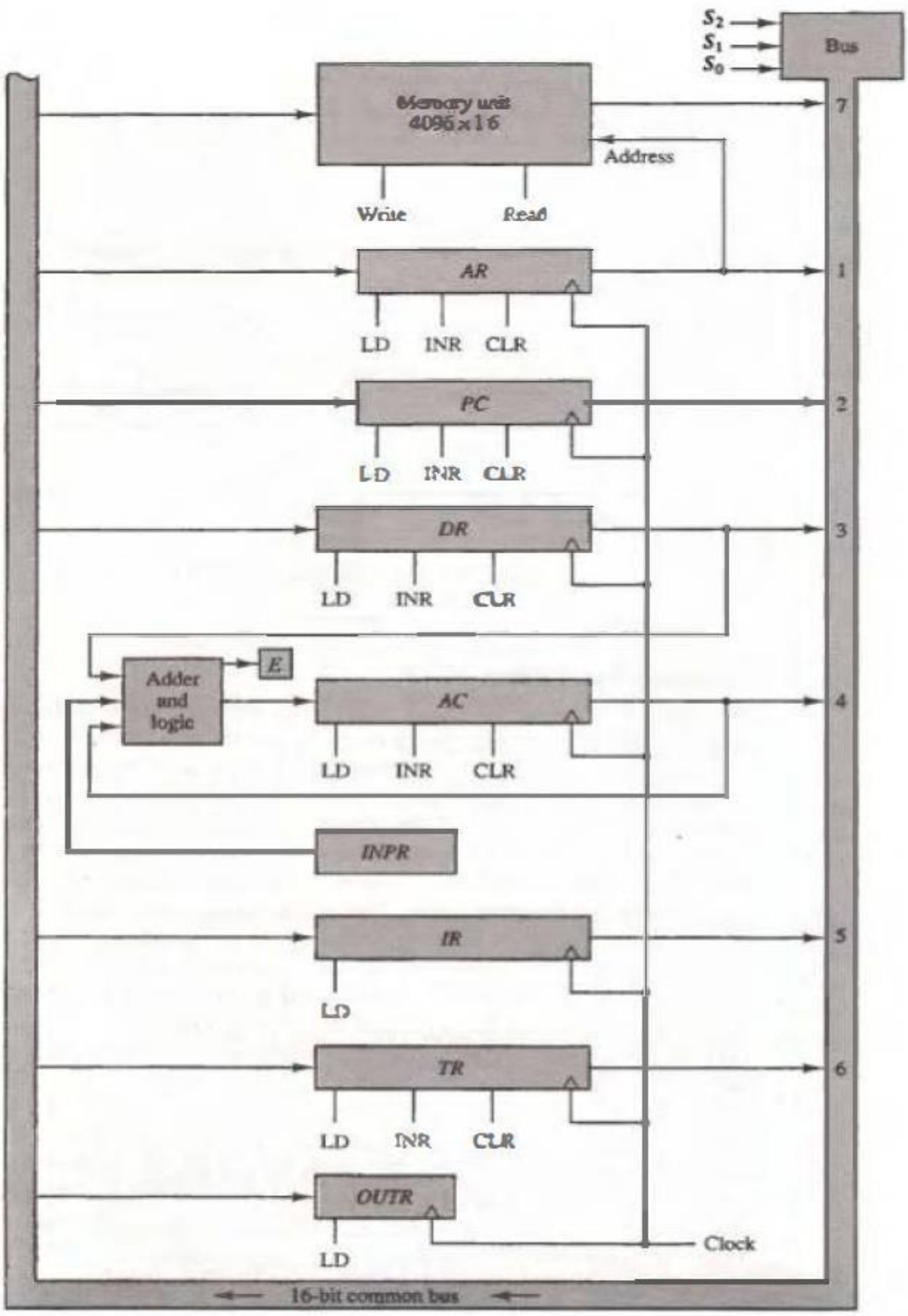


Figure (n): Basic computer registers connected to a common bus.

- 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 example1, 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 example2. 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 rnicrooperations

DR ← AC and AC ←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.

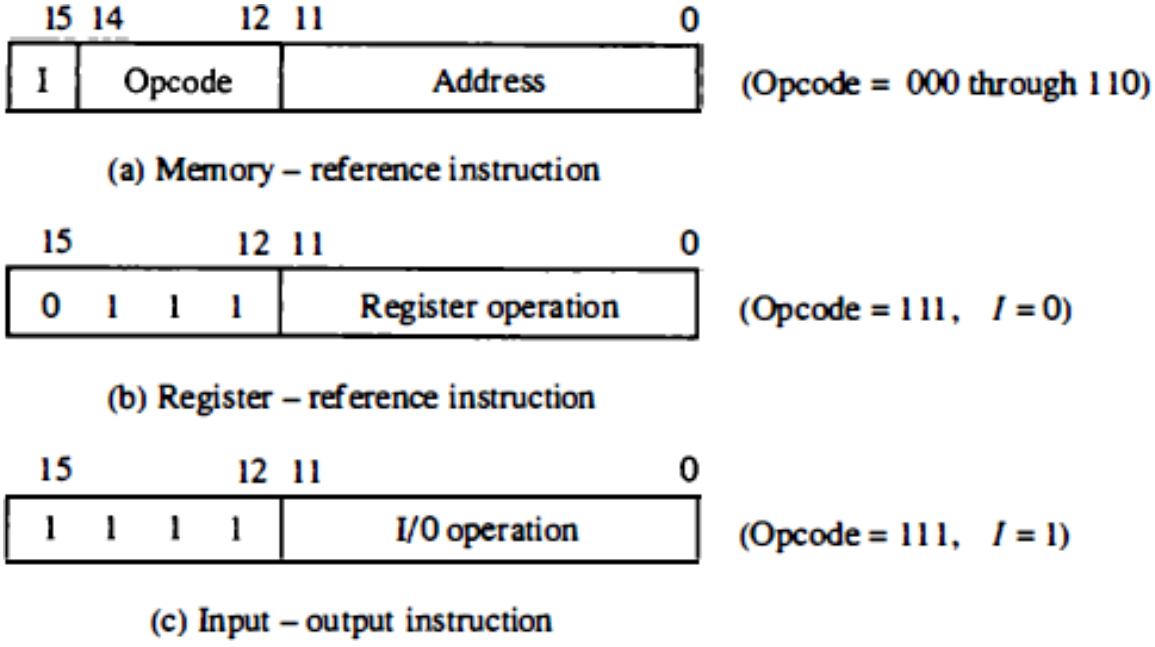


Figure (n): Basic computer instruction formats

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 $I = 0$	Hexadecimal code $I = 1$	Description
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 AC
OUT	F400	Output character from AC
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 = I, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 (1000) to E (1110) since the last bit is I.

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 rnicrooperations for the fetch and decode phases can be specified by the following register transfer statements.

T0: AR ← PC ($S_0S_1S_2$=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1 (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← 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.

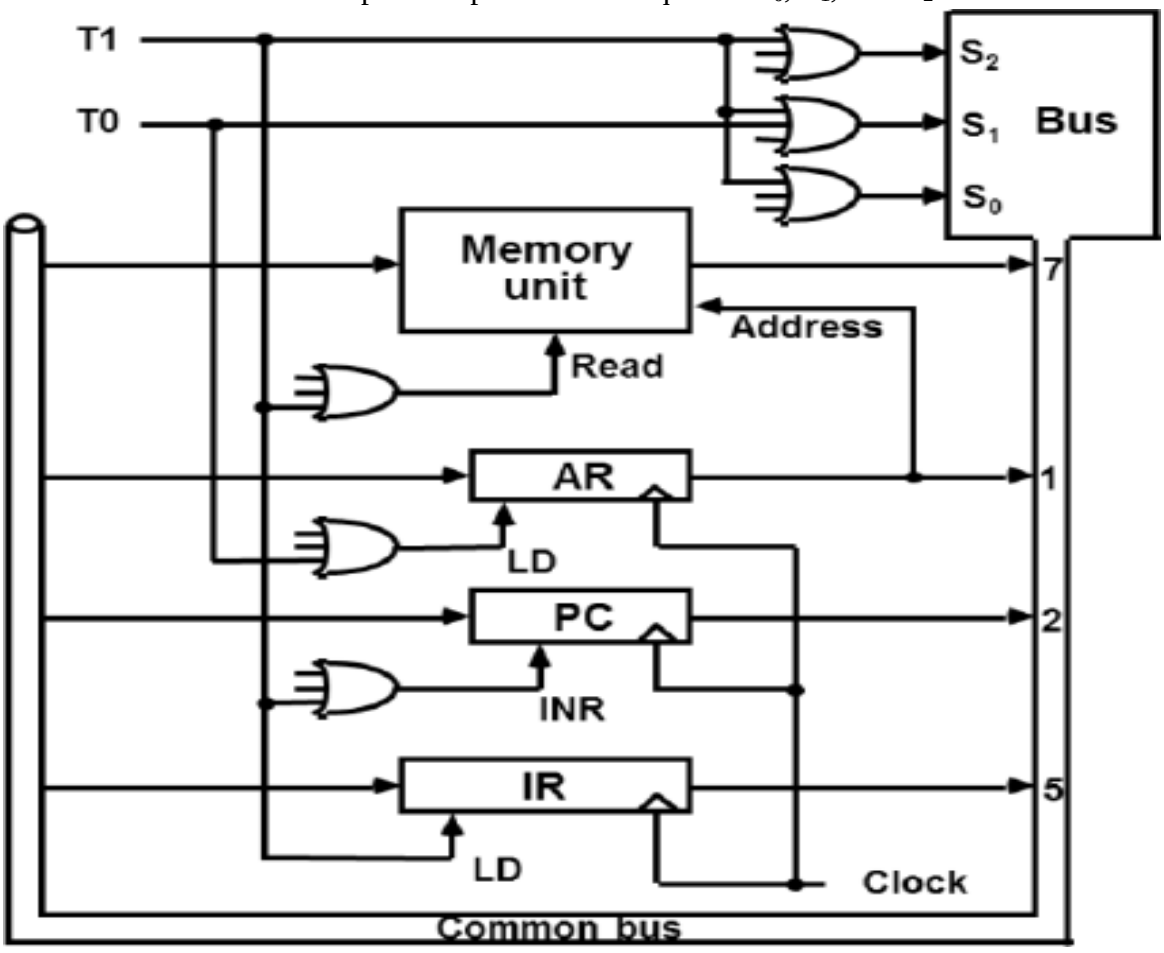


Figure (o): Register transfers for the fetch phase

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←M[AR], PC ← 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-0utput 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:

$D_7' I T_3$: $AR \leftarrow M[AR]$
$D_7' I' T_3$: Nothing
$D_7 I' T_3$: Execute a register-reference instruction
$D_7 I T_3$: 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 T4. 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

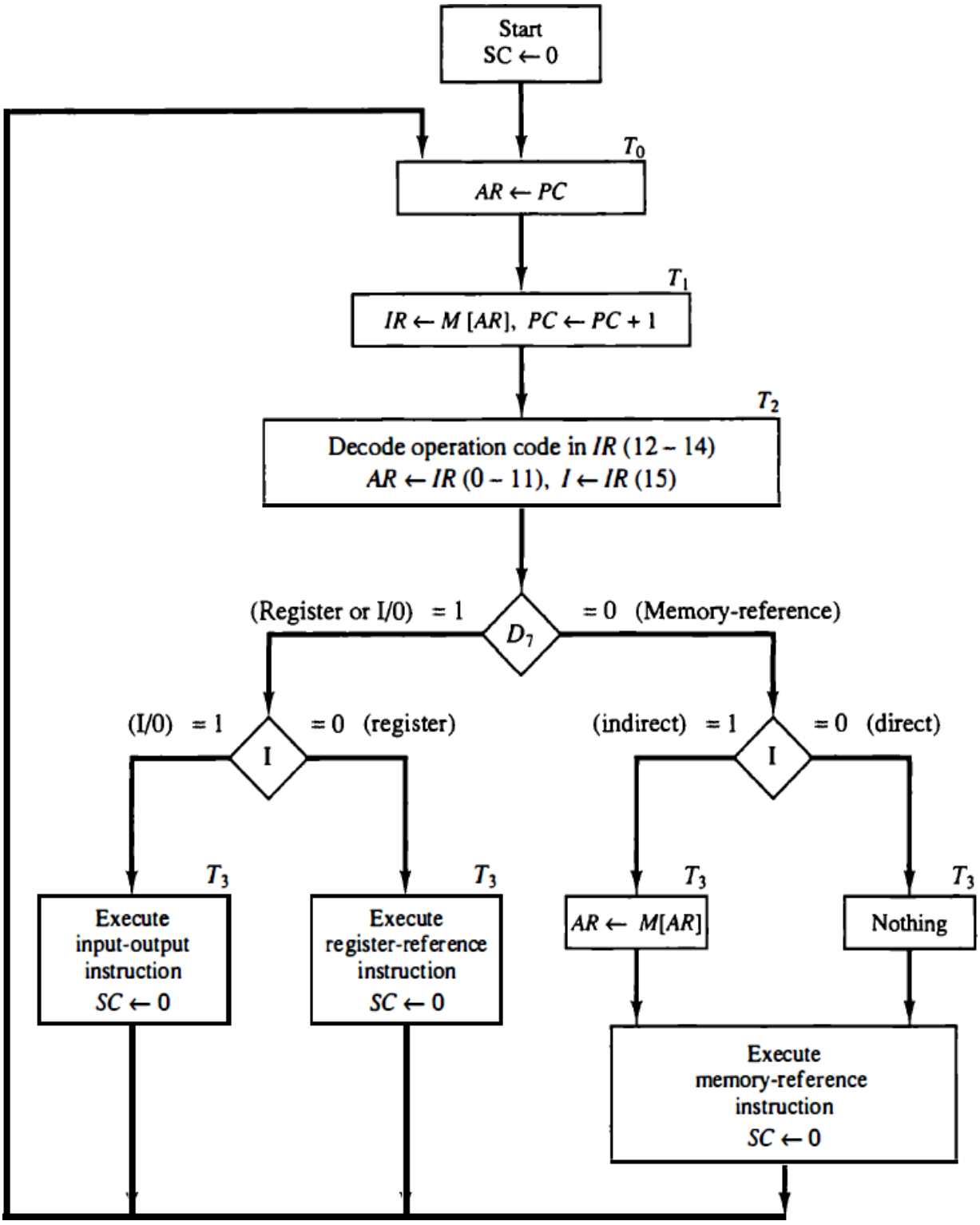


Figure (p): Flowchart for instruction cycle (initial configuration).

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 I' 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 I'.
  ii. The next three bits constitute the operation code and are recognized from decoder output D7.
  iii. Bit 11 in IR is 1 and is recognized from B_{11}.

The control function that initiates the rnicrooperation 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:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ 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 T2 when I = 0, or during timing signal T3 when I = 1. The execution of the memory-reference instructions starts with timing signal T4.

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 rnicrooperations that execute this instruction are:

D_0T_4: DR ← M[AR]
D_0T_5: AC ← AC Λ DR, SC ← 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,., is transferred to the E (extended accumulator) flip-flop. The rnicrooperations needed to execute this instruction are:

D_1T_4 : DR ← M[AR]
D_1T_5: AC ← AC + DR, E ← Cout, SC ← 0

LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC . The rnicrooperations needed to execute this instruction are:

D_2T_4: DR ← M[AR]
D_2T_5: AC ← DR, SC ← 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:

$$D_3T_4: \quad M[AR] \leftarrow AC, \quad SC \leftarrow 0$$

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 rnicrooperation:

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 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.

$$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$$

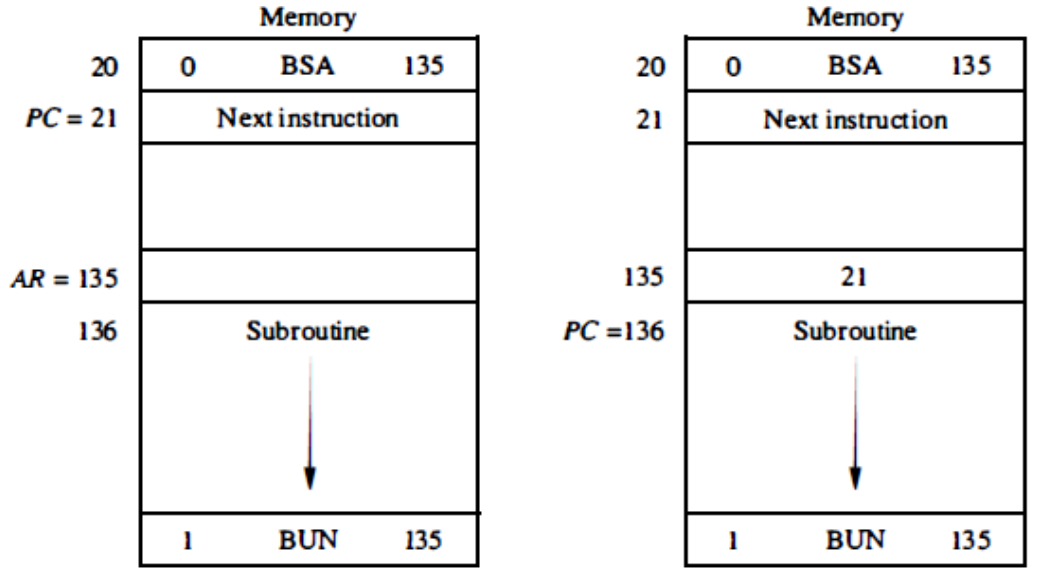
BSA: Branch and Save Return Address

EX:

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:

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$
$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

Timing signal T4 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 T5 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.

$$D_6T_4: \quad DR \leftarrow M[AR]$$
$$D_6T_5: \quad DR \leftarrow DR + 1$$
$$D_6T_6: \quad M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0$$

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 T4, T5, or T6, 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 T0 to start the next instruction cycle.

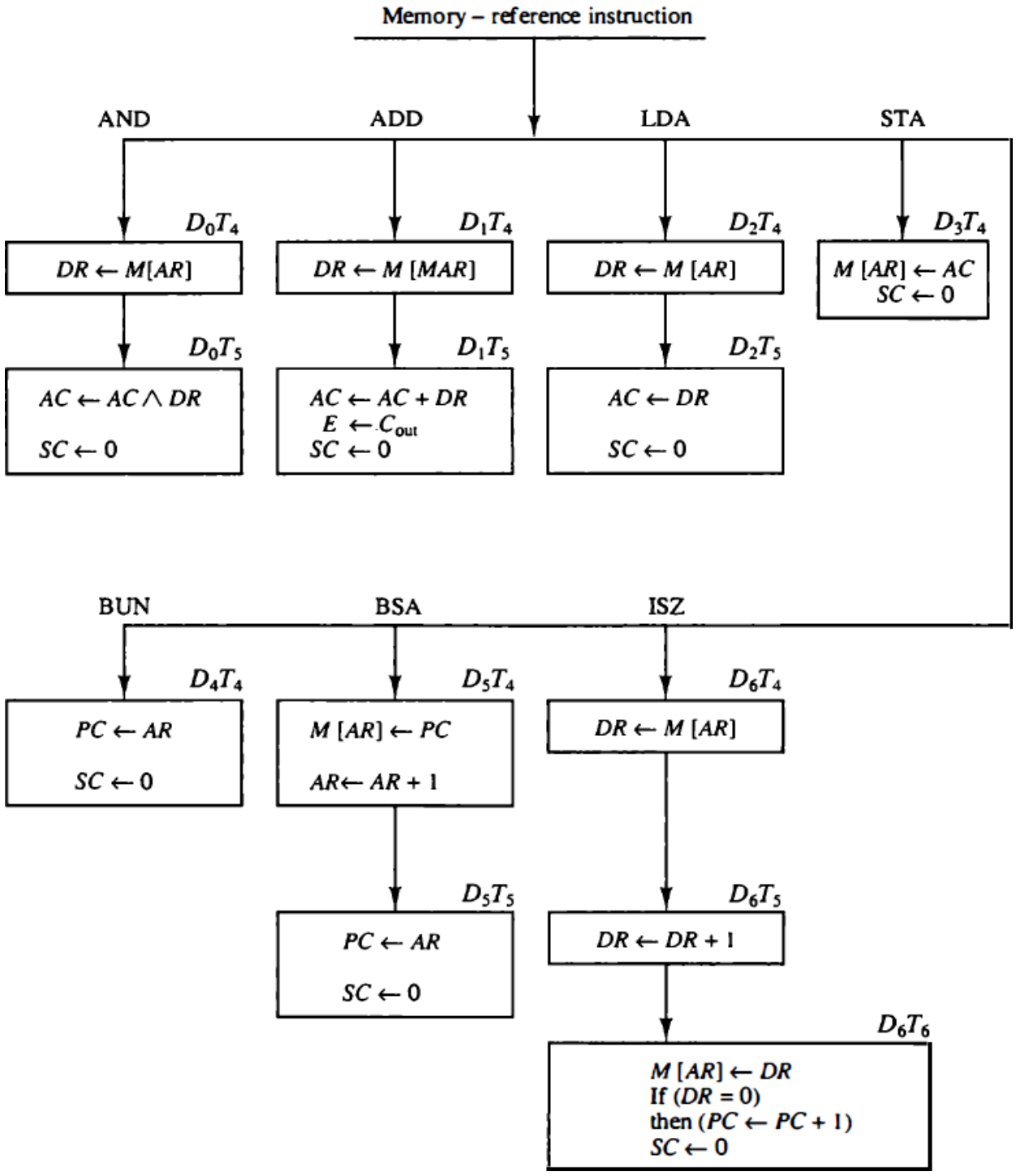


Figure (g): Flowchart for Memory-reference instructions

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.

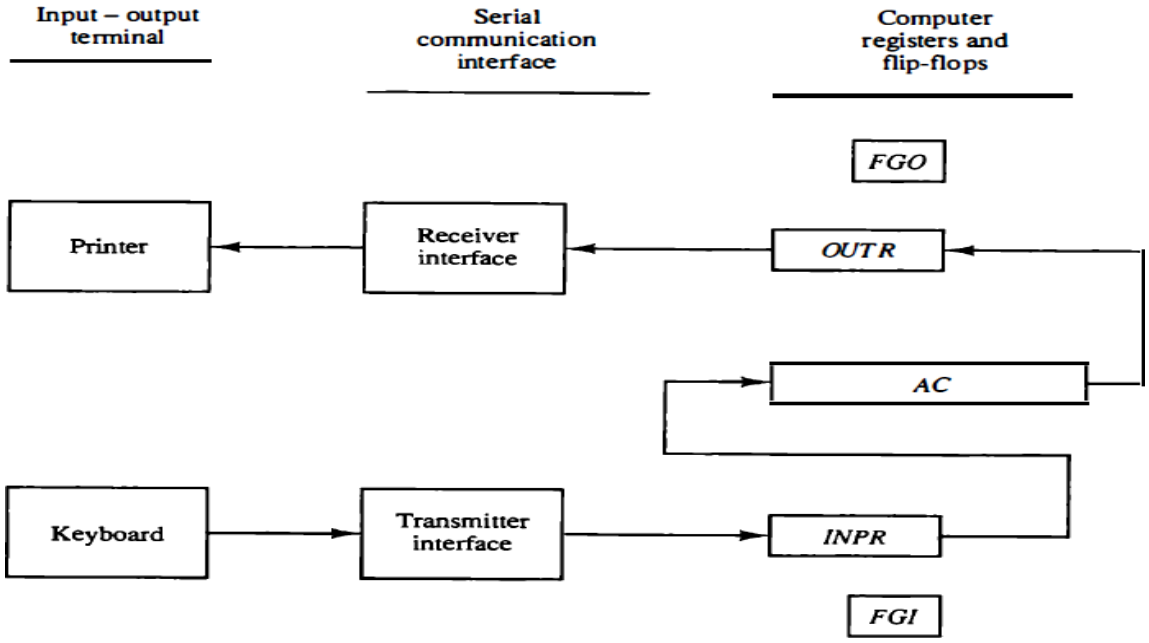


Figure (r): Input-Output configuration

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 OUTR 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 OUTR 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 D7 = 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 (l).

$D_7IT_3 = p$ (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 1μs. 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 μs. 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 lEN can be set and cleared with two instructions (IOF and ION instructions).

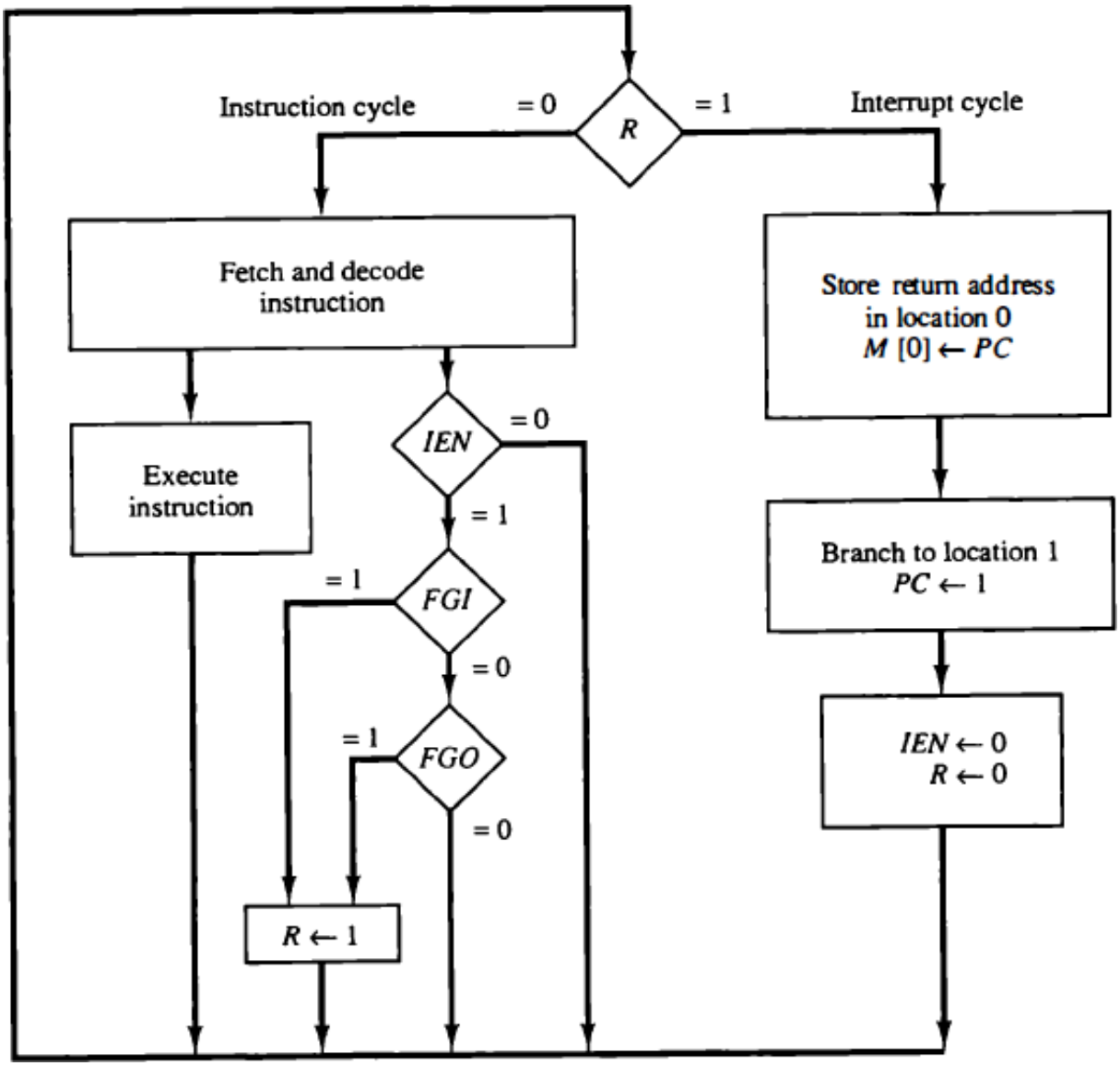


Figure (s): Flowchart for interrupt cycle

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 lEN 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 lEN 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 lEN = 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)** operation.

EX:

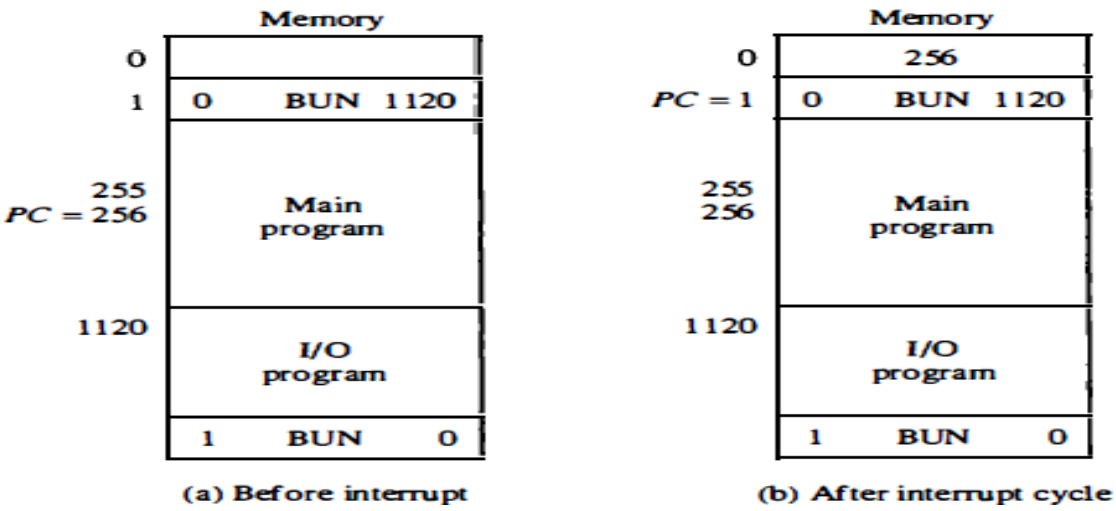


Figure (t): Demonstration of Interrupt Cycle

An example that shows what happens during the interrupt cycle is shown in Figure (t). 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 T0 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 T0, T1 or T2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

$$T_0'T_1'T_2'(IEN)(FGI + FGO): \quad R \leftarrow 1$$

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 lEN 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.

$$\begin{aligned} RT_0&: \quad AR \leftarrow 0, \quad TR \leftarrow PC \\ RT_1&: \quad M[AR] \leftarrow TR, \quad PC \leftarrow 0 \\ RT_2&: \quad PC \leftarrow PC + 1, \quad IEN \leftarrow 0, \quad R \leftarrow 0, \quad SC \leftarrow 0 \end{aligned}$$

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.

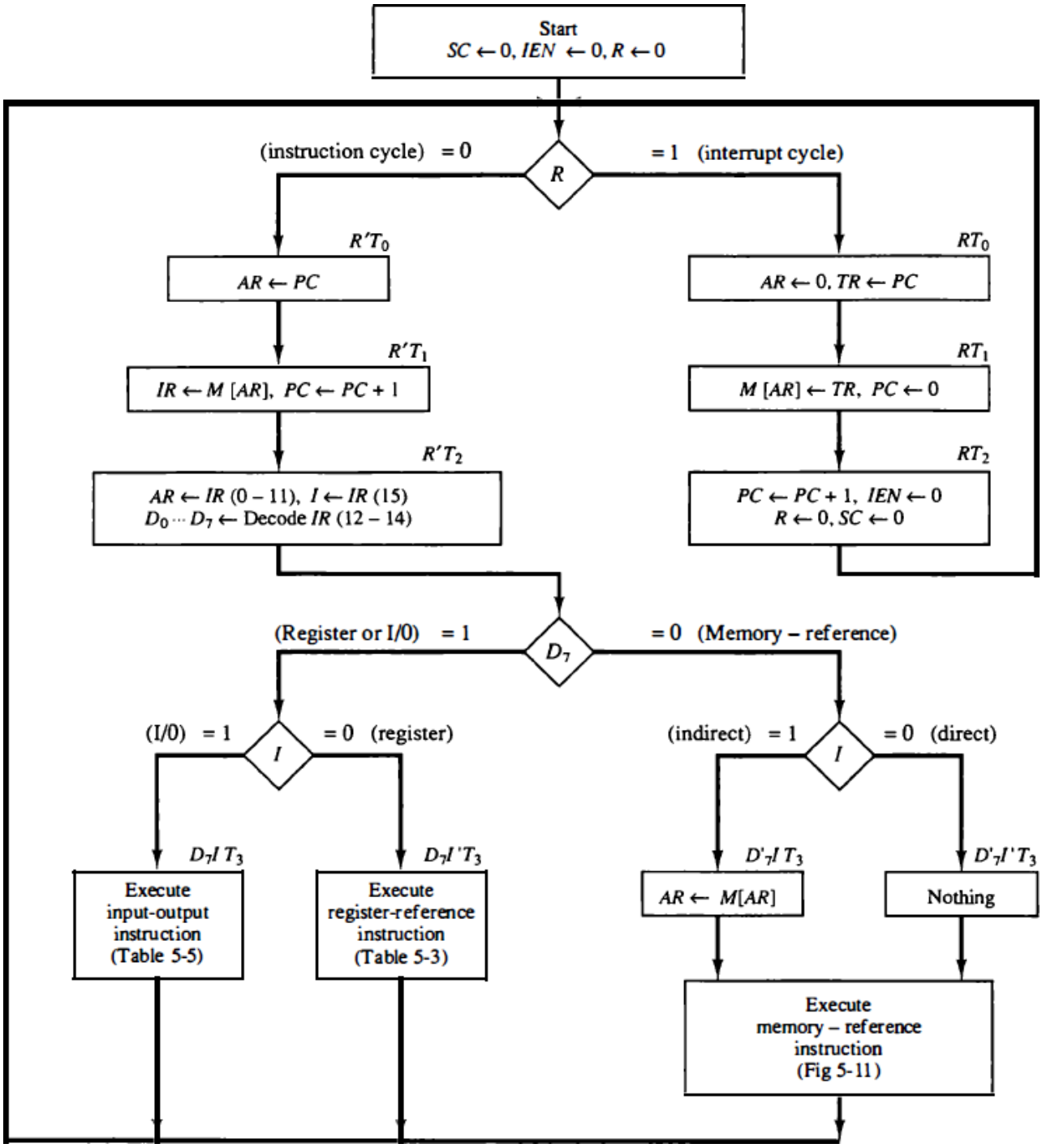


Figure (u): Flowchart for computer operation

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$
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:	$M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
Register-reference:		
		$D_7I'T_3 = r$ (common to all register-reference instructions)
		$IR(i) = B_i$ $(i = 0, 1, 2, \ldots, 11)$
	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:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ 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}:	$AC(0\text{–}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7)$, $FGO \leftarrow 0$
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.

UNIT – 4

Syllabus : Central Processing Unit : General Register Organization, STACK Organization, Instruction Formats, Addressing Modes, Data Transfer and Manipulation, Program Control, Reduced Instruction Set Computer.

Micro programmed Control : Control Memory, Address Sequencing, Micro Program Example, Design of Control Unit.

Central Processing Unit : 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 below.

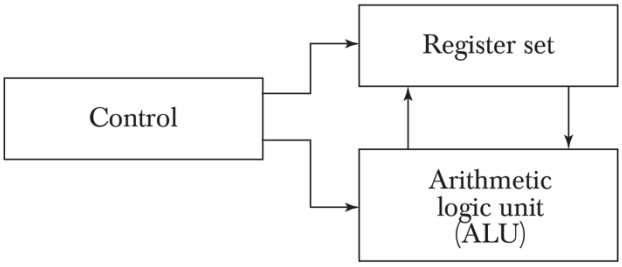


Figure 8-1 Major components of CPU.

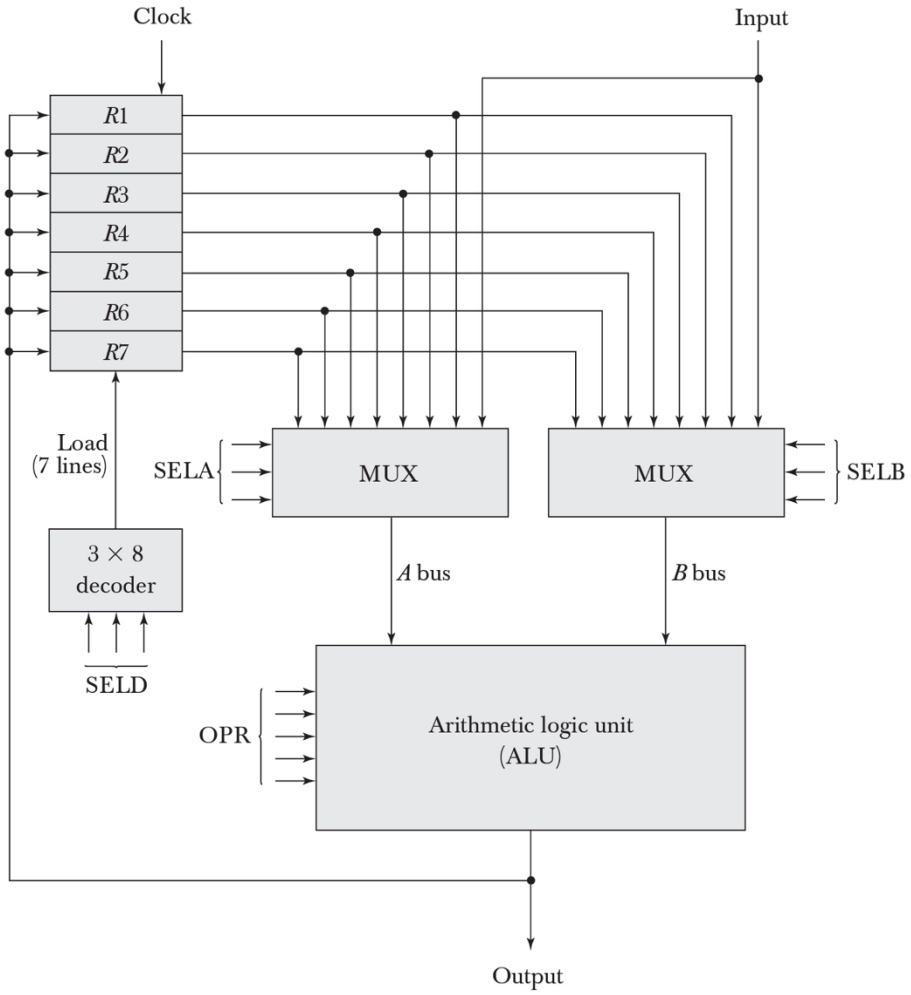
The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

General Register Organization : Referring the memory locations for data is time consuming rather than referring processor registers. 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.

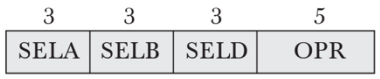
A bus organization for seven CPU registers is shown below. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common 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 ← 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.



(a) Block diagram



(b) Control word

Figure 8-2 Register set with common ALU.

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 shown above. 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.

The encoding of the register selections is specified in the following table. The 3-bit binary code listed in the first column of the table specifies the binary code foreach 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 are available in the external output.

TABLE 8-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R*1	*R*1	*R*1
010	*R*2	*R*2	*R*2
011	*R*3	*R*3	*R*3
100	*R*4	*R*4	*R*4
101	*R*5	*R*5	*R*5
110	*R*6	*R*6	*R*6
111	*R*7	*R*7	*R*7

TABLE 8-2 Encoding of ALU Operations

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

The ALU provides arithmetic and logic operations. In addition, the CPU must provide shift operations. The shifter may be placed in the input of theALU to provide a pre-shift capability, or at the output of the ALU to provide post-shifting capability. In some cases, the shift operations are included with theALU. Different types of operations performed by ALU are listed in above table.

Examples of Micro-Operations : 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 ← 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. 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

Some examples of micro operations are given below.

TABLE 8-3 Examples of Microoperations for the CPU

	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
$R1 \leftarrow R2 - R3$	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
Output ← *R*2	R2	–	None	TSFA	010 000 000 00000
Output ← Input	Input	–	None	TSFA	000 000 000 00000
$R4 \leftarrow$ sh1 $R4$	R4	–	R4	SHLA	100 000 100 11000
$R5 \leftarrow 0$	R5	R5	R5	XOR	101 101 101 01100

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 stack in digital computers is essentially a memory unit with an address register that can count only. 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. The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack'. These operations are simulated by incrementing or decrementing the stack pointer register.

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. The following diagram shows the organization of a 64-word register stack.

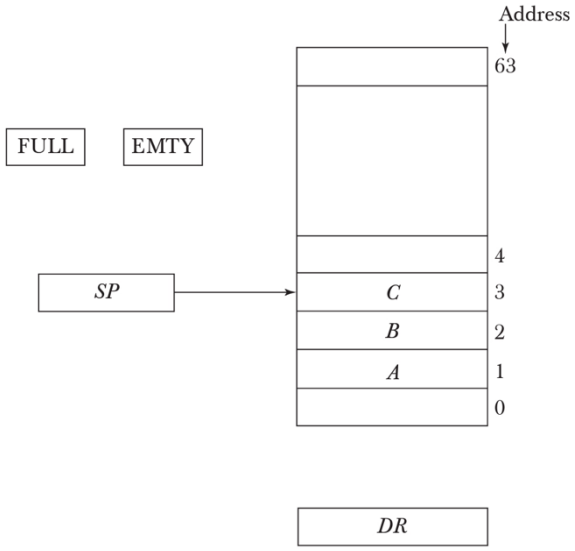


Figure 8-3 Block diagram of a 64-word stack.

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. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2.

To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack. Note that item C has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

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 = 1000000 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 EMTY 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.

Initially, SP is cleared to 0, EMTY 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
$EMTY \leftarrow 0$	Mark the stack not empty

A new item is deleted from the stack if the stack is not empty (if EMTY = 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 $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

Memory Stack : Stack operation can also be implemented in computer memories. This can be done by assigning few segments of the main memory to store the stack, data and program instructions. 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. The following diagram shows a portion of computer memory partitioned into three segments: program, data, and stack.

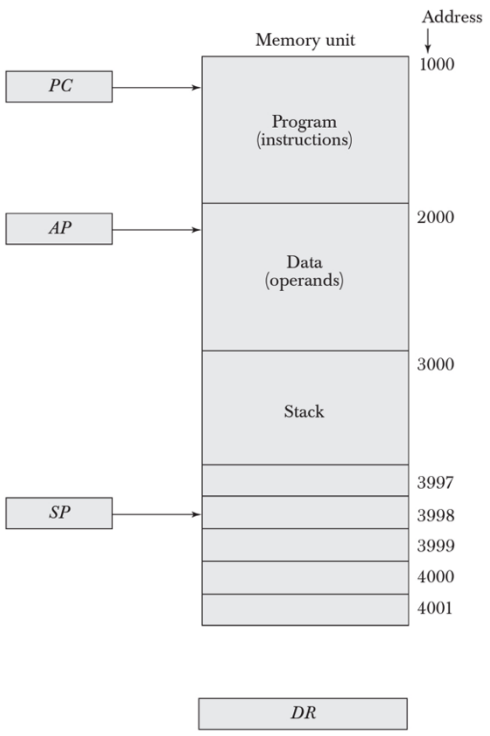


Figure 8-4 Computer memory with program, data, and stack segments.

The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data. The stack pointer SP points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

The initial value of Stack Pointer is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999 and the last address that can be stored for the stack is 3000. A new item is inserted with the PUSH operation as follows.

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow DR$$

The stack pointer is decremented so that it points at the address of next word. A memory write operation inserts the word from DR into the top of the stack. A new item is deleted with a POP operation as follows:

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

Here the top item is read from the stack into DR. the stack pointer is then incremented to point at the next item in the stack.

The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case). After a push operation, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.

A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. Henceforth, SP is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation : A stack organization is very effective for evaluating arithmetic expressions. The common arithmetic expressions are written in infix notation, with each operator written between the operands. Consider the simple arithmetic expression A + B can be represented in the following three notations.

$A + B$	Infix notation
$+AB$	Prefix or Polish notation
$AB +$	Postfix or reverse Polish notation

The Reverse Polish notation is suitable for stack manipulation. The expression A * B + C * D is written in reverse Polish notation as AB*CD*+ and is evaluated as follows: Scan the expression from left to right. When an operator is reached, perform the operation with the two operands found on the left side of the operator. Remove the two operands and the operator and replace them by the number obtained from the result of the operation. Continue to scan the expression and repeat the procedure for every operator encountered until there are no more operators.

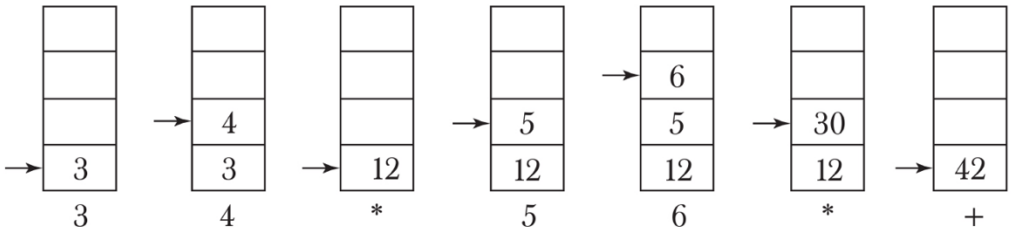
Evaluating Arithmetic Expressions : To evaluate an arithmetic expression follow the below steps.

1. Scan Reverse Polish Notation from left to right.
2. While scanning, if operand is occurred push it into the stack.
3. If operator is occurred pop top two elements from the stack and apply the operator. Again the result is pushed into the stack
4. Continue this process until end of the arithmetic expression

Consider the following arithmetic expression. : (3 * 4) + (5 * 6).

In reverse Polish notation, it is expressed as 3 4 * 5 6 * +. The following represents stack operations to evaluate (3 * 4) + (5 * 6).

Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.



Instruction Formats : The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. 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.

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. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, will designate register R5.

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. **Single Accumulator Organization :** In this 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 ← AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

2. **General Register Organization :** In this Organization, computer will make use of two or three address fields within the instruction format. For example,

   ADD R1, R2, R3 /* R1 ← R2 + R3 */

   This instruction will add the data present in register R1 with the data present in register R3 and the result is stored in register R1.

   ADD R1, R2 /* R1 ← R1 + R2 */

   This instruction will add the data present in register R2 with the data present in register R1 and the result is stored in register R1.

   ADD R1, X /* R1 ← R1 + M[X] */

   This instruction will add the data present in register R1 with the data present at address X in memory location and the result is stored in register R1.

3. **Stack Organization :** In this organization, computer uses only one address filed to performs all its operations. Computers with stack organization would have PUSH and POP instructions which require an address field. For example,

   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. For example, 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.

Types of Address Instructions : The following are different types of address instructions.

1. **Three Address Instructions :** Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates X = (A + B) * (C + D) using three address instructions is shown below.

```
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
```

   The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

2. **Two Address Instructions :** Two-address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate X = (A + B) * (C + D) using two address instructions is as follows:

```
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
```

   The MOV instruction moves or transfers the operands to and from memory and processor registers.

3. **One Address Instructions :** One-address instructions use an implied accumulator (AC) register for all data manipulation. The program to evaluate X = (A + B) * (C + D) using one address instruction is shown below.

```
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
```

   All operations are done between the AC register and a memory operand. Here, T is the address of a temporary memory location required for storing the intermediate result.

4. **Zero Address Instructions :** A stack organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) * (C + D) will be written for a stack organized computer. (TOS stands for top of stack).

```
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
```

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 following are different types of addressing modes:

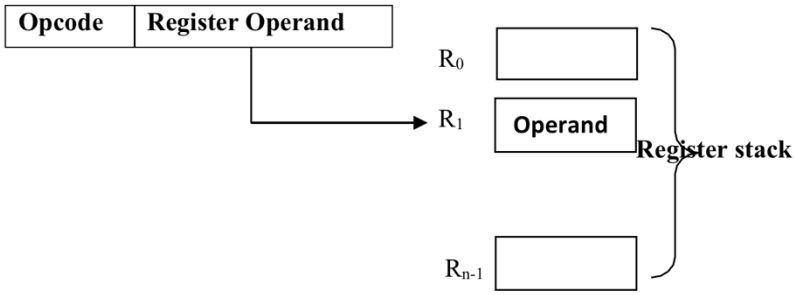
1. **Implied Mode :** In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, 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. Instruction format with implied address mode is as shown below.

Opcode	Mode	Address

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. Immediate-mode instructions are useful for initializing registers to a constant value.

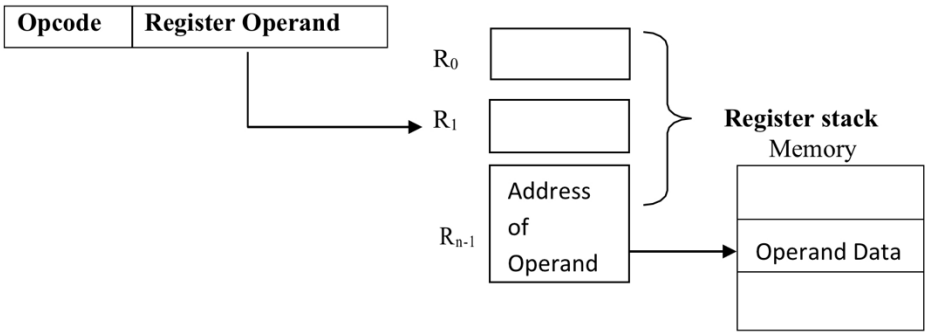
Opcode	Immediate Operand

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. A k-bit field can specify any one of 2^k registers.

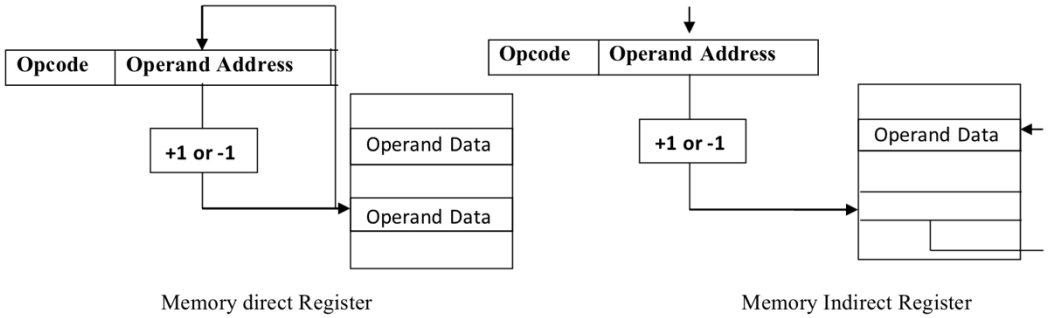


4. **Register Indirect Mode :** In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. A reference to the register is then equivalent to specifying a memory address.

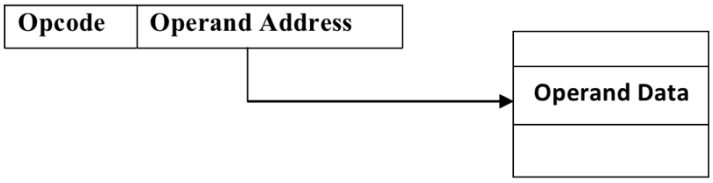
The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.



5. **Auto-increment or Auto-decrement Mode :** In this addressing mode, when the address in the register is used to access memory, the value in the register is incremented or decremented by 1 automatically. It can be implemented as memory direct or memory indirect register.



6. **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.

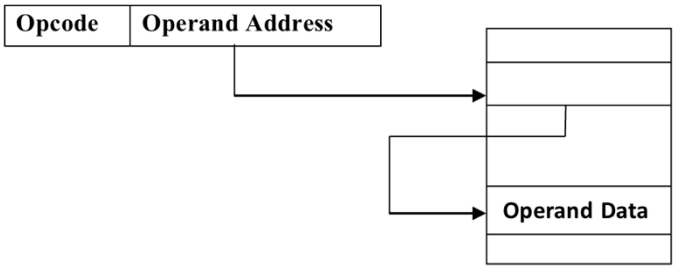


7. **Indirect Address Mode :** In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

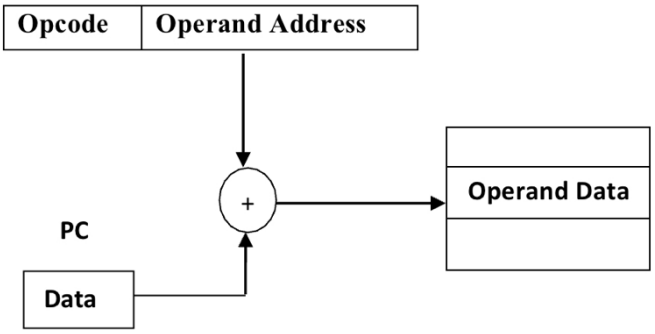
   A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation:

   effective address = address part of instruction + content of CPU register

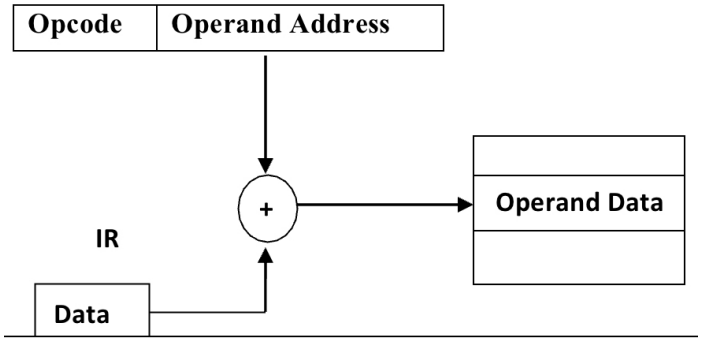
   The CPU register used in the computation may be the program counter, an index register, or a base register.



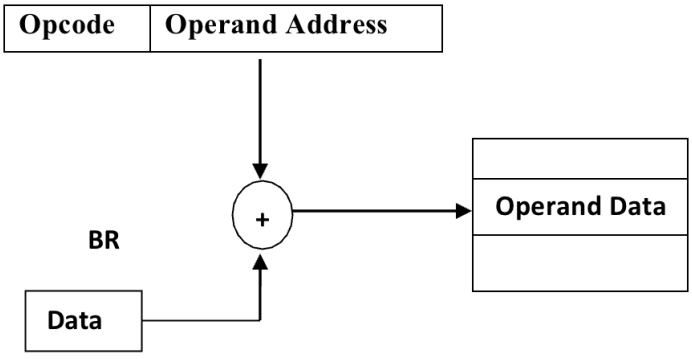
8. **Relative Addressing Mode :** In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.



9. **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. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. The distance between the beginning address and the address of the operand is the index value stored in the index register.



10. **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. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.



Example : Consider a two-word instruction at address 200 and 201 (load to AC) with an address field equal to 500. The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. PC has the value 200 for fetching this instruction. The content of processor register R1is 400, and the content of an index register XR is 100. AC receives the operand after the instruction is executed.

The mode field of the instruction can specify any one of a number of modes. For each possible mod, calculate the effective address and the operand that must be loaded into AC.

- In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800.
- In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201.)
- In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300.
- In the relative mode the effective address is 500 + 202 = 702 and the operand is 325.
- In the index mode the effective address is XR + 500 = 100 + 500 = 600 and the operand is 900.
- In the register mode the operand is in R1 and 400 is loaded into AC.
- In the register indirect mode the effective address is 400, equal to the content of R1 and the operand loaded into AC is 700.
- The auto increment mode is the same as the register indirect mode except that R1 is incremented to 401 after the execution of the instruction.
- The auto decrement mode decrements R1 to 399 prior to the execution of the instruction. The operand loaded into AC is now 450.

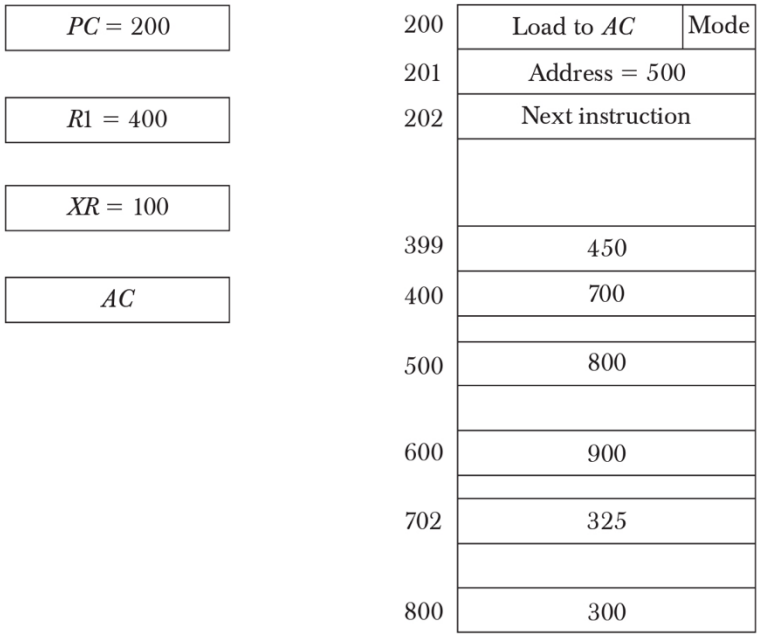


Figure 8-7 Numerical example for addressing modes.

TABLE 8-4 Tabular List of Numerical Example

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

Data Transfer and Manipulation : The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields. The actual operations available in the instruction set are not very different from one computer to another. 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.

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. The following table gives a list of eight data transfer instructions used in many computers.

TABLE 8-5 Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

- 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.

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:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

1. **Arithmetic Instructions :** The four basic arithmetic operations are addition, subtraction, multiplication, and division. A list of typical arithmetic instructions is given in the following table.

   The Increment instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all 1's when incremented produces a result of all 0's. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all 0's, when decremented, produces a number with all 1's.

TABLE 8-7 Typical 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 8-8 Typical Logical and Bit Manipulation Instructions

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

2. **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 separately and treat it as a Boolean variable. By proper application of the logical instructions it is possible to change bit value, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words. Some typical logical and bit manipulation instructions are listed in the above table.
   - The Clear instruction causes the specified operand to be replaced by 0's.
   - The Complement instruction produces the l's complement by inverting all the bits of the operand.
   - The AND instruction is used to clear a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships x AND 0 = 0 and x AND 1= x, dictate that a binary variable ANDed with a 0 produces a 0; but the variable does not change in value when ANDed with a 1. Therefore, the AND instruction can be used to clear bits of an operand selectively by ANDing the operand with another operand that has 0's in the bit positions that must be cleared. The AND instruction is also called a mask because it masks or inserts 0's in a selected portion of an operand.
   - The OR instruction is used to set a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships x OR 1 = l and x OR 0 = x dictate that a binary variable ORed with a 1 produces a 1; but the variable does not change when ORed with a 0. Therefore, the OR instruction can be used to selectively set bits of an operand by ORing it with another operand with 1's in the bit positions that must be set to 1.
   - Similarly, the XOR instruction is used to selectively complement bits of an operand. This is because of the Boolean relationships x XOR 1 = x` and x XOR 0 = x. Thus a binary variable is complemented when XORed with a 1 but does not change in value when XORed with a 0.

- Individual bits such as a carry can be cleared, set, or complemented with appropriate instructions. Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

3. **Shift Instructions :** 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 either logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left. The following table lists different shift instructions.

TABLE 8-9 Typical Shift Instructions

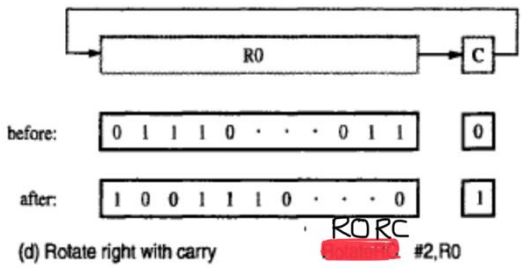
Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left.

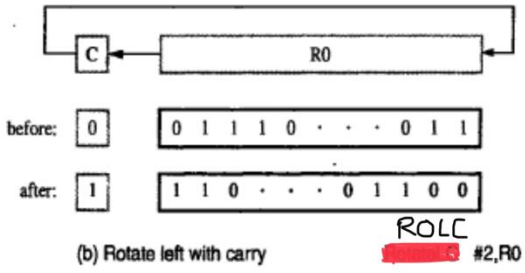
Arithmetic shifts usually conform with the rules for signed-2's complement numbers. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end.

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, shifts the entire register to the left.



(d) Rotate right with carry



(b) Rotate left with carry

Program Control Instructions : 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 causes a break in the sequence of instruction execution. Program Control instructions are used to switch the control to different locations. The following table lists different types of program control instructions.

TABLE 8-10 Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic name for an address. When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR.

Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

The skip instruction does not need an address field and is therefore a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is met. If the condition is not met, control proceeds with the next instruction in sequence.

The call and return instructions are used in conjunction with subroutines.

The compare instruction performs a subtraction between two operands, but the result of the operation is not retained. However, certain status bit conditions are set as a result of the operation.

Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands. The status bits are the carry bit, the sign bit, a zero indication, and an overflow condition.

Stratus Bit Conditions : Status bits are also called condition-code bits or flag bits. The following diagram shows the block diagram of an 8-bit ALU with a 4-bit status register.

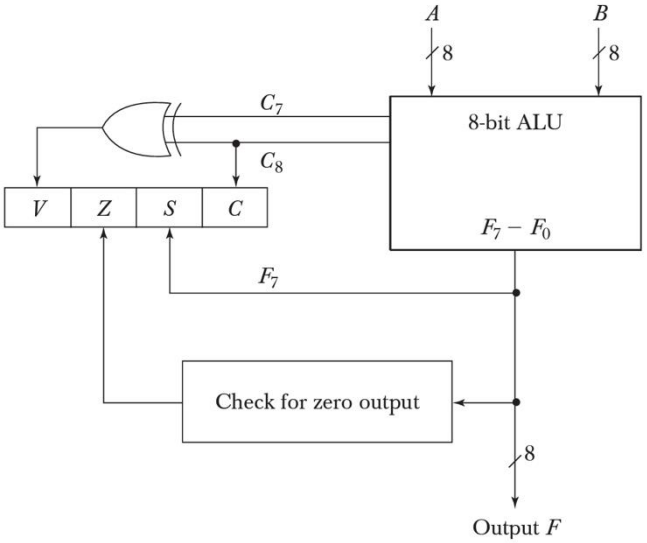


Figure 8-8 Status register bits.

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 F7 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.

Conditional Branch Instructions : The following table gives a list of the most common branch instructions.

TABLE 8-11 Conditional Branch 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	$V = 0$
Unsigned compare conditions $(A - B)$		
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	$A \neq B$
Signed compare conditions $(A - B)$		
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	$A \neq B$

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 is usually initiated by an internal or external signal rather than from the execution of an instruction. The address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and an interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

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. Typically, it includes the status bits from the last ALU operation and it specifies the interrupts that are allowed to occur and whether the CPU is operating in a supervisor or user mode.

The hardware procedure for processing an interrupt is very similar to the execution of a subroutine call instruction. The state of the CPU is pushed into a memory stack and the beginning address of the service routine is transferred to the program counter. The beginning address of the service routine is determined by the hardware rather than the address field of an instruction. Some computers assign one memory location where interrupts are always transferred. The service routine must then determine what caused the interrupt and proceed to service it.

The CPU does not respond to an interrupt until the end of an instruction execution. Just before going to the next fetch phase, control checks for any interrupt signals. If an interrupt is pending, control goes to a hardware interrupt cycle. During this cycle, the contents of PC and PSW are pushed onto the stack. The branch address for the particular interrupt is then transferred to PC and a new PSW is loaded into the status register. The service program can now be executed starting from the branch address and having a CPU mode as specified in the new PSW.

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.

Types of Interrupts : There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

- External interrupts
- Internal interrupts
- 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, elapsed time of an event, or power failure.

Internal interrupts arise from illegal or erroneous use of an instruction or date. 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.

The difference between internal and external interrupts is that the internal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event. Internal interrupts are synchronous with the program while external interrupts are asynchronous.

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 : A computer with a large number of instructions is classified as a complex instruction set computer, abbreviated CISC. Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

A computer with a few number of instructions is classified as a Reduced instruction set computer, abbreviated RISC.

CISC Characteristics :

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 a RISC processor 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
8. A relatively large number of registers in the processor unit
9. Use of overlapped register windows to speed-up procedure call and return
10. Efficient instruction pipeline
11. Compiler support for efficient translation of high-level language programs into machine language programs.

Micro programmed Control : Control Memory, Address Sequencing, Micro Program Example, Design of Control Unit.

Control Memory : The major functional parts in a digital computer are Central Processing Unit (CPU), Memory, and Input–output. The main digital hardware functional units of CPU are control unit, arithmetic and logic unit, and registers. The function of the control unit in a digital computer is to initiate sequences of microoperations. Two methods of implementing control unit are hardwired control and micro-programmed control.

The design of hardwired control involves the use of fixed instructions, fixed logic blocks of and/or arrays, encoders, decoders, etc. The key characteristics of hardwired control logic are high speed operation, expensive, relatively complex, and no flexibility of adding new instructions. Example CPU's with hardwired logic control are Intel 8085, Motorola 6802, Zilog 80, and any RISC (Reduced Instruction Set Computer) CPUs.

When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired. Microprogramming is a second alternative for designing the control unit of a digital computer. The principle of microprogramming is an elegant and systematic method for controlling the microoperation sequences in a digital computer. For example,CPUs with microprogrammed control unit are Intel 8080, Motorola 68000, and any CISC (Complex Instruction Set Computer) CPUs.

The control function that specifies a microoperation is a binary variable. When it is in one binary state, the corresponding microoperation is executed. The active state of a control variable may be either the 1 state or the 0 state, depending on the application.

The control unit initiates a series of sequential steps of microoperations. During any given time, certain microoperations are to be initiated, while others remain idle. The control variables at any given time can be represented by a string of 1's and 0's called a control word. As such, control words can be programmed to perform various operations on the components of the system. A control unit whose binary control variables are stored in memory is called a microprogrammed control unit.

Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes a microprogram. The use of a microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operations. The content of the word in ROM at a given address specifies a microinstruction.

Dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory that is part of a control unit is referred to as a control memory.

A computer that employs a microprogrammed control unit will have two separate memories: a main memory and a control memory. The main memory is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data.

In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations. Each machine instruction initiates a series of microinstructions in control memory. These microinstructions generate the microoperations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.

The general configuration of a microprogrammed control unit is shown in below diagram. The control memory is assumed to be a ROM, within which all control information is permanently stored. The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.

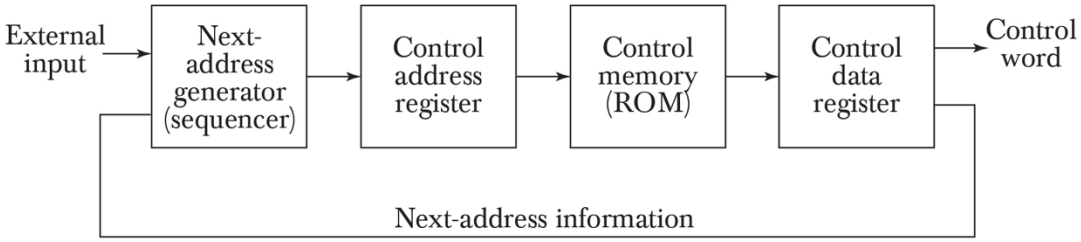


Figure 7-1 Microprogrammed control organization.

The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions.

While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

The next address generator is sometimes called a microprogram sequencer, as it determines the address sequence that is read from control memory. The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.

The control data register holds the present microinstruction while the next address is computed and read from memory. The data register is sometimes called a pipeline register. It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established, there should be no need for further hardware or wiring changes. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

Address Sequencing : Microinstructions are stored in control memory in groups, with each group specifying a routine. Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another.

An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. At the end of the fetch routine, the instruction is in the instruction register of the computer.

The control memory next must go through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers. The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction. When the effective address computation routine is completed, the address of the operand is available in the memory address register.

The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor registers depend on the operation code part of the instruction. Each instruction has its own microprogram routine stored in a given location of control memory. The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a mapping process.

A mapping procedure is a rule that transforms the instruction code into a control memory address. Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register. Microprograms that employ subroutines will require an external register for storing the return address.

When the execution of the instruction is completed, control must return to the fetch routine. This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine. In summary, the address sequencing capabilities required in a control memory are:

- Incrementing of the control address register.
- Unconditional branch or conditional branch, depending on status bit conditions.
- A mapping process from the bits of the instruction to an address for control memory.
- A facility for subroutine call and return.

Following shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.

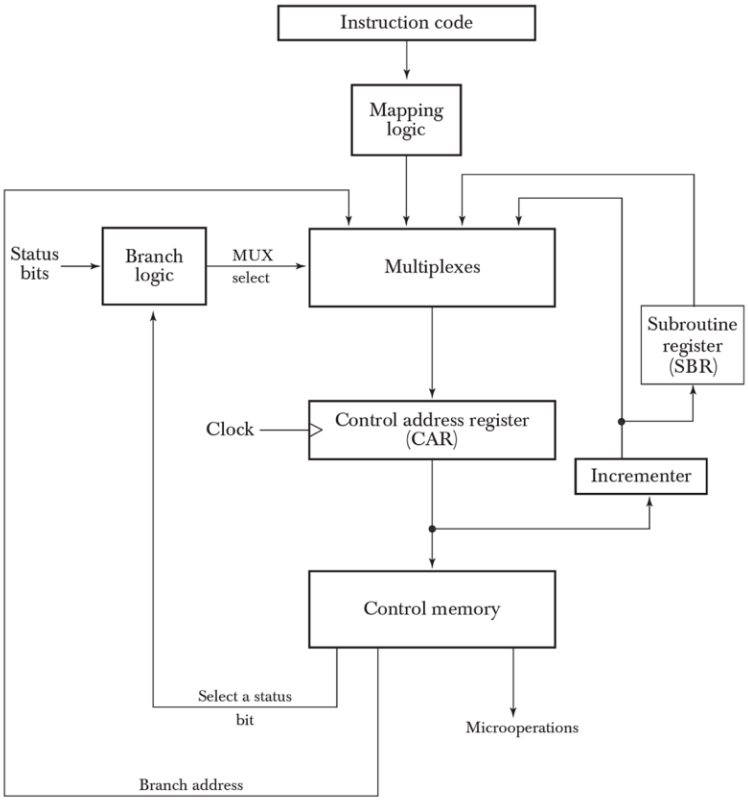


Figure 7-2 Selection of address for control memory.

The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained. The diagram shows four different paths from which the control address register (CAR) receives the address. The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence. Branching is achieved by specifying the branch address in one of the fields of the microinstruction. Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition. An external address is transferred into control memory via a mapping logic circuit. The return address for a subroutine is stored in a special register whose value is then used when the microprogram wishes to return from the subroutine.

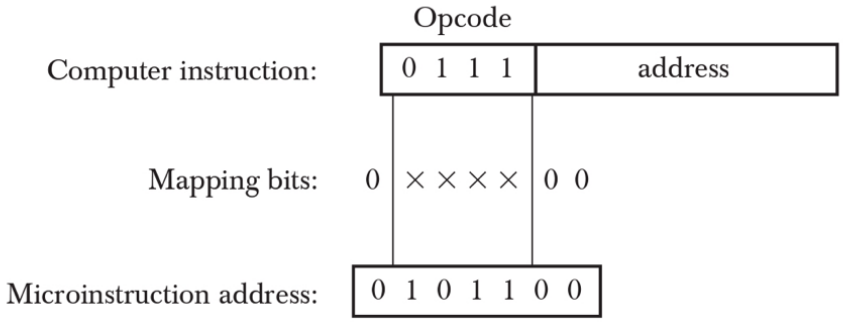
Conditional Branching : The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions. Information in these bits can be tested and actions initiated based on their condition: whether their value is 1 or 0.

The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented. This can be implemented with a multiplexer.

An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register. This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to 1. A reference to this bit by the status bit select lines from control memory causes the branch address to be loaded into the control address register unconditionally.

Mapping of Instruction : A special type of branch exists when a microinstruction specifies a branch to the first word in control memory. The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in below diagram has an operation code of four bits which can specify up to 16 distinct instructions.

Figure 7-3 Mapping from instruction code to microinstruction address.



Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a microprogram routine in control memory that executes the instruction. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in above diagram. This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111.

Micro Program Example :

Computer Configuration : The block diagram of the computer is shown below.

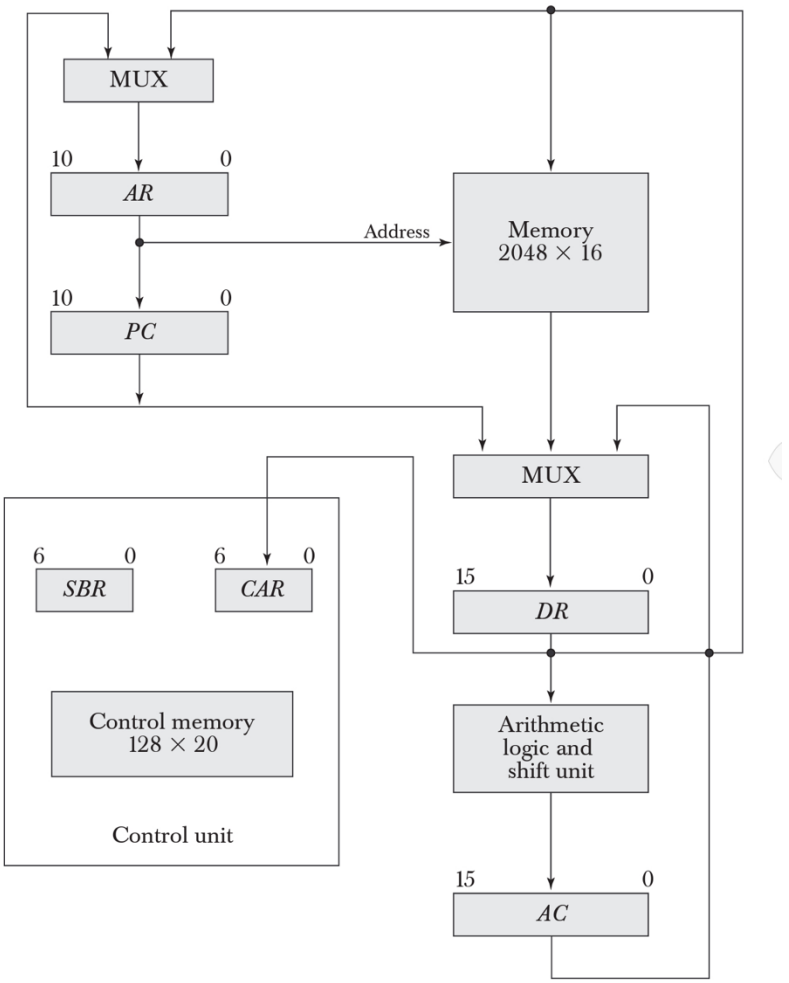
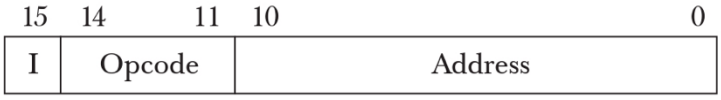


Figure 7-4 Computer hardware fonfiguration.

It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram. Four registers are associated with the processor unit and two with the control unit. The processor registers are program counter PC, address register AR, data register DR, and accumulator register AC. The control unit has a control address register CAR and a subroutine register SBR.

The transfer of information among the registers in the processor is done through multiplexers rather than a common bus. DR can receive information from AC, PC, or memory. AR can receive information from PC or DR. PC can receive information only from AR. The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC. Note that memory receives its address from AR. Input data written to memory come from DR, and data read from memory can go only to DR.

The computer instruction format is shown below.



(a) Instruction format

It consists of three fields: a 1-bit field for indirect addressing symbolized by I, a 4-bit operation code (opcode), and an 11-bit address field.

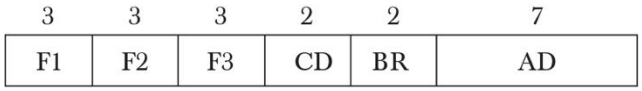
The following figure lists four of the 16 possible memory-reference instructions. The ADD instruction adds the content of the operand found in the effective address to the content of AC. The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of AC into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

Microinstruction Format : The microinstruction format for the control memory is shown in the following figure. The 20 bits of the microinstruction are divided into four functional parts. The three fields Fl, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type or branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has $128 = 2^7$ words.



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure 7-6 Microinstruction code format (20 bits).

The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations as listed in the following tables.

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0-10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0-10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow AC$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

This gives a total of 21 microoperations. No more than three microoperations can be chosen for a microinstruction, one from each field. If fewer than three microoperations are used, one or more of the fields will use the binary code 000 for no operation. For example,

$$DR \leftarrow M[AR] \qquad \text{with F2} = 100$$

and $$PC \leftarrow PC + 1 \qquad \text{with F3} = 101$$

The nine bits of the microoperation fields will then be 000 100 101.

For example, a microoperation field 010 001 000 has no meaning because it specifies the operations to clear AC to 0 and subtract DR from AC at the same time.

The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed below. The first condition is always a 1, so that a reference to CD = 00 (or the symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation. The indirect bit I is available from bit 15 of DR after an instruction is read from memory. The sign bit of AC provides the next status bit. The zero value, symbolized by Z, is a binary variable whose value is equal to 1 if all the bits in AC are equal to zero.

The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction.

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2-5) \leftarrow DR(11-14)$, $CAR(0,1,6) \leftarrow 0$

When BR = 00, the control performs a jump (JMP) operation (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation. The two operations are identical except that a call microinstruction stores the return address in the subroutine register SBR. The jump and call operations depend on the value of the CD field. It the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register CAR. Otherwise, CAR is incremented by 1.

The return from subroutine is accomplished with a BR field equal to 10. This causes the transfer of the return address from SBR to CAR. The mapping from the operation code bits of the instruction to an address for CAR is accomplished when the BR field is equal to 11. The bits of the operation code are in DR(11–14) after an instruction is read from memory.

Symbolic Microinstructions : A symbolic microprogram can be translated into its binary equivalent by means of an assembler. The simplest and most straightforward way to formulate an assembly language for a microprogram is to define symbols for each field of the microinstruction and to give users the capability for defining their own symbolic addresses.

Each line of the assembly language microprogram defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following information:

1. The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2. The microoperations field consists of one, two, or three symbols, separated by commas. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.
3. The CD field has one of the letters U, I, S, or Z.
4. The BR field contains one of the four symbols.
5. The AD field specifies a value for the address field of the microinstruction in one of three possible ways:
    a. With a symbolic address, which must also appear as a label.
    b. With the symbol NEXT to designate the next address in sequence.
    c. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

Note : Symbol ORG is used to define the origin, or first address, of a microprogram routine. Thus the symbol ORG 64 informs the assembler to place the next microinstruction in control memory at decimal address 64, which is equivalent to the binary address 1000000.

The Fetch Routine : The control memory has 128 words, and each word contains 20 bits. To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words. The first 64 words (addresses 0 to 63) are to be occupied by the routines for the 16 instructions. The last 64 words may be used for any other purpose.The microinstructions needed for the fetch routine are

$$AR \leftarrow PC$$
$$DR \leftarrow \mathrm{M}[AR], \quad PC \leftarrow PC + 1$$
$$AR \leftarrow DR(0-10), \quad CAR(2-5) \leftarrow DR(11-14),\ CAR(0,1,6) \leftarrow 0$$

The fetch routine needs three microinstructions, which are placed in control memory at addresses 64, 65, and 66. The symbolic microprogram for the fetch routine as follows:

```
            ORG  64
FETCH:      PCTAR            U     JMP     NEXT
            READ, INCPC      U     JMP     NEXT
            DRTAR            U     MAP
```

The translation of the symbolic microprogram to binary produces the following binary microprogram.

Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

Symbolic Microprogram : The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address 0xxxx00, where xxxx are the four bits of the operation code. For example, if the instruction is an ADD instruction whose operation code is 0000, the MAP microinstruction will transfer to CAR the address 0000000, which is the start address for the ADD routine in control memory. The first address for the BRANCH and STORE routines are 0 0001 00 (decimal 4) and 0 0010 00 (decimal 8), respectively. The first address for the other 13 routines are at address values 12, 16, 20, . . . , 60.

Sample Microprogram :

TABLE 7-2 Symbolic Microprogram (Partial)

Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

The execution of the ADD instruction is carried out by the microinstructions at addresses 1 and 2. The first microinstruction reads the operand from memory into DR. The second microinstruction performs an add micro-operation with the content of DR and AC and then jumps back to the beginning of the fetch routine.

The BRANCH instruction should cause a branch to the effective address 0. The AC will be less than zero if its sign is negative, which is detected from status bit 5 being a 1. The BRANCH routine starts by checking the value of S. IfS is equal to 0, no branch occurs and the next microinstruction causes a jump back to the fetch routine without altering the content of PC. If S is equal to 1, the first JMP microinstruction transfers

control to location OVER. The microinstruction at this location calls the INDRCT subroutine if I = 1. The effective address is then transferred from AR to PC and the microprogram jumps back to the fetch routine.

The STORE routine again uses the INDRCT subroutine if I = 1. The content of AC is transferred into DR. A memory write operation is initiated to store the content of DR in a location specified by the effective address in AR.

The EXCHANGE routine reads the operand from the effective address and places it in DR. The contents of DR and AC are interchanged in the third microinstruction. The original content of AC that is now in DR is stored back in memory.

Binary Microprogram : The symbolic microprogram is a convenient form for writing microprograms in a way that people can read and understand. But this is not the way that the microprogram is stored in memory. The symbolic microprogram must be translated to binary either by means of an assembler program.

The equivalent binary form of the symbolic microprogram shown above is listed below. The addresses for control memory are given in both decimal and binary.

TABLE 7-3 Binary Microprogram for Control Memory (Partial)

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	Fl	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

Design of Control Unit : The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching. The number of control bits that initiate microoperations can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2^k microoperations. Each field requires a decoder to produce the corresponding control signals.

The following figure shows the three decoders and some of the connections that must be made from their outputs.

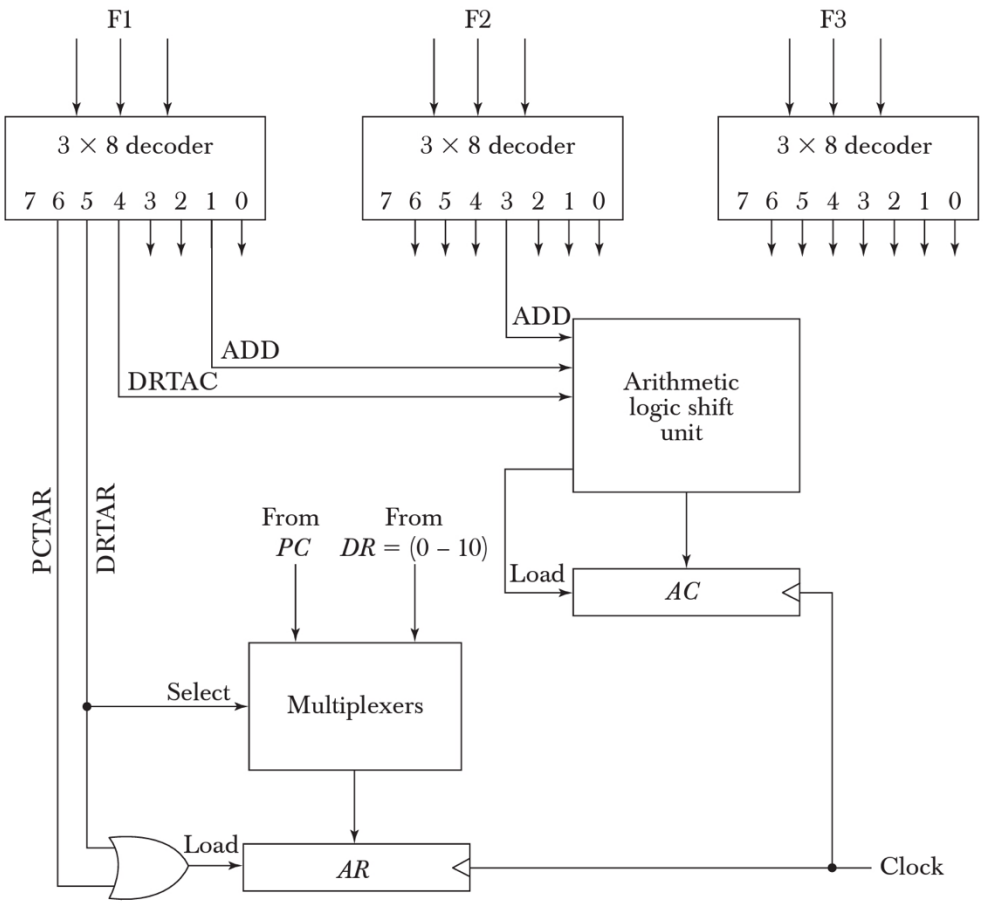


Figure 7-7 Decoding of microoperation fields.

Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 * 8 decoder to provide eight outputs. For example, when F1=101 (binary 5), the next clock pulse transition transfers the content of DR(0–10) to AR (symbolized by DRTAR). Similarly, when F1=110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR).

Outputs 5 and 6 of decoder F1 are connected to the load input of AR, so that when either one of these outputs is active, information from the multiplexers is transferred to AR. The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.

Micro Program Sequencer : The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed. The next-address logic of the sequencer determines the specific address source to be loaded into the control address register. The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction.

The block diagram of the microprogram sequencer is shown below. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register CAR. The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit. The output from CAR provides the address for the control memory. The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register SBR. The other

three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output of SBR, and from an external source that maps the instruction.

The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer. If the bit selected is equal to 1, the T (test) variable is equal to 1; otherwise, it is equal to 0. The T value together with the two bits from the BR (branch) field go to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit. Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operation.

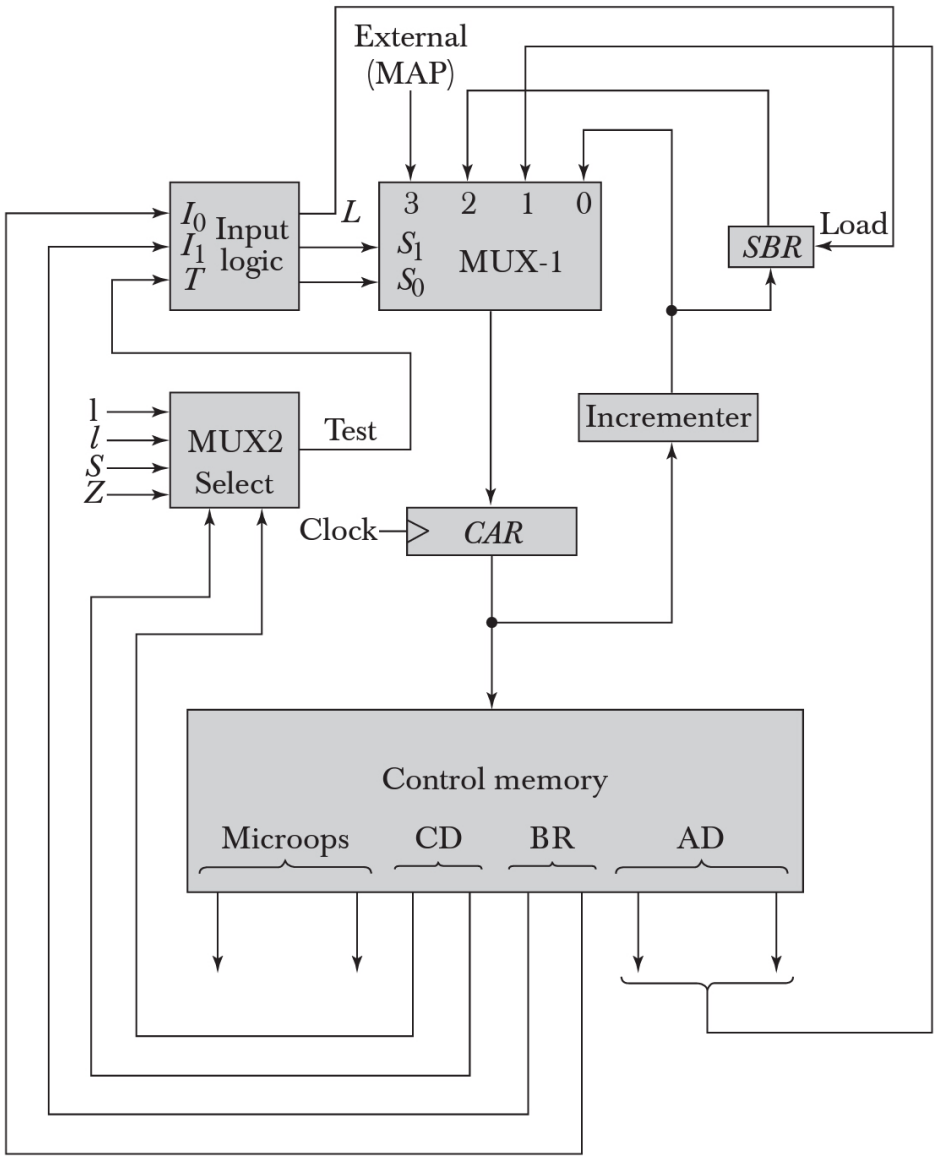


Figure 7-8 Microprogram sequencer for a control memory.

The input logic circuit has three inputs, I_0, I_1, and T, and three outputs, S_0, S_1, and L. Variables S_0 and S_1 select one of the source addresses for CAR. Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexer. For example, with $S_1S_0 = 10$, multiplexer input number 2 is selected and establishes a transfer path from SBR to CAR.

The truth table for the input logic circuit is shown below. Inputs I_1 and I_0 are identical to the bit values in the BR field. The bit values for S_1 and S_0 are determined from the stated function and the path in the

multiplexer that establishes the required transfer. The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR=01) provided that the status bit condition is satisfied (T=1).

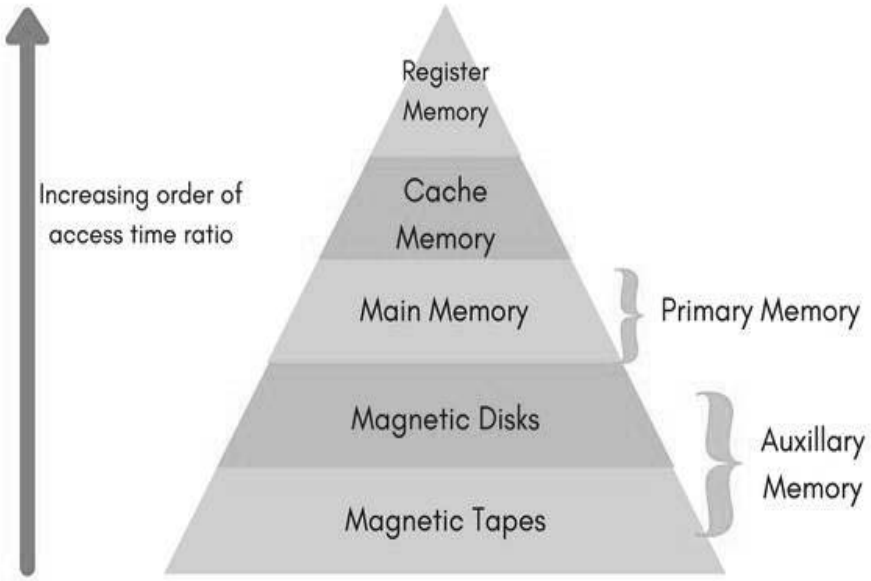
TABLE 7-4 Input Logic Truth Table for Microprogram Sequencer

BR Field		Input			MUX 1		Load *SBR*
		I_1	I_0	T	S_1	S_0	L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	1	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0

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



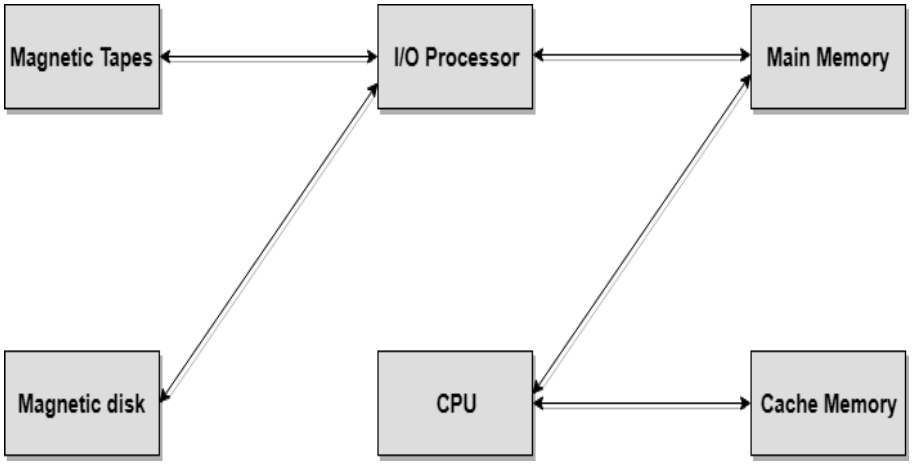
The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.

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 methods 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, Auxillary 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 holing 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: Flash memory.

- 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 accomodate 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.

Hit Ratio = Hit/(Hit + 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.

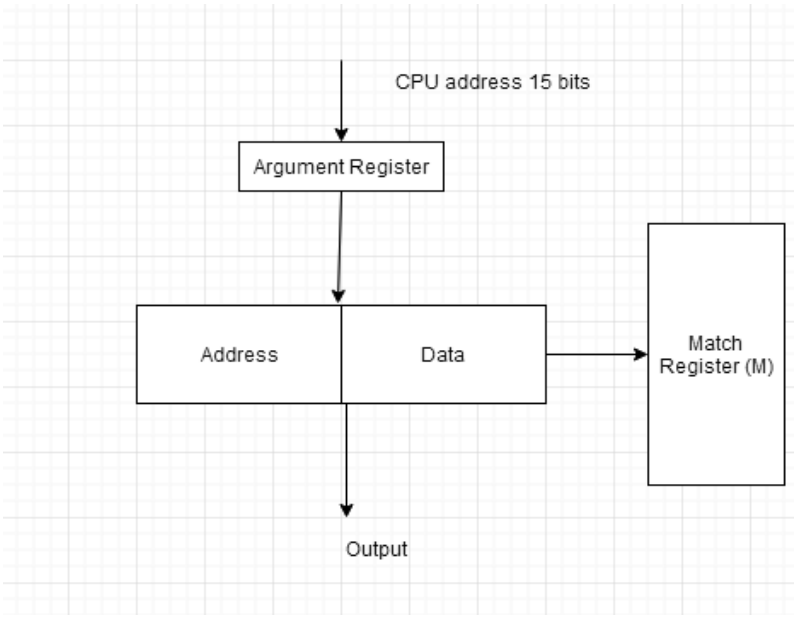
Memory Mapping and Concept of Virtual Memory

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

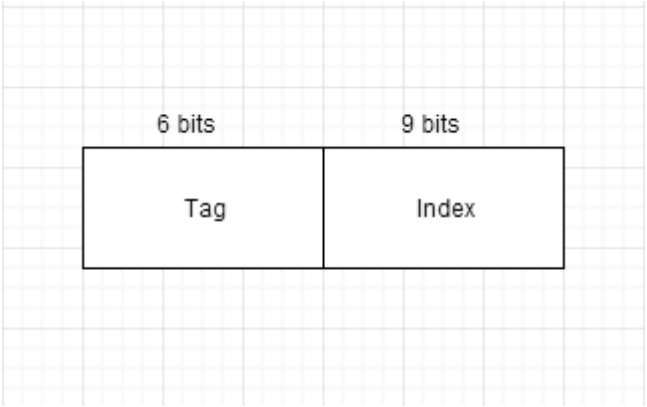
Associative Mapping

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

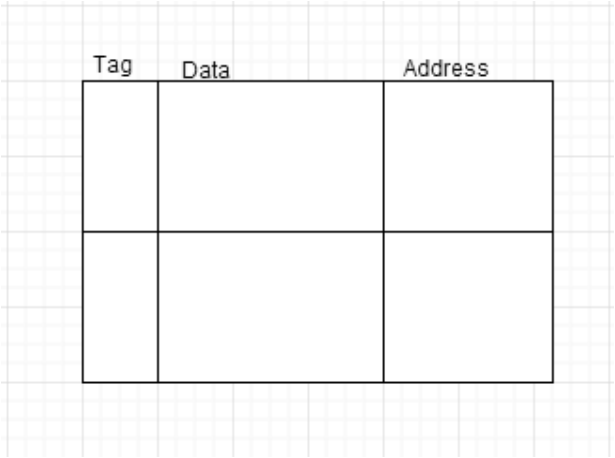
The CPU address of 15 bits is divided into 2 fields. In this the 9 least significant bits constitute the **index** field and the remaining 6 bits constitute the **tag** field. The number of bits in index field is equal to the number of address bits required to access cache memory.



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.



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 computer system. With write-through, the main memory always has an up-to-date copy of the line. So when a read 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 writing back since memory already has an up-to-date copy of the line.

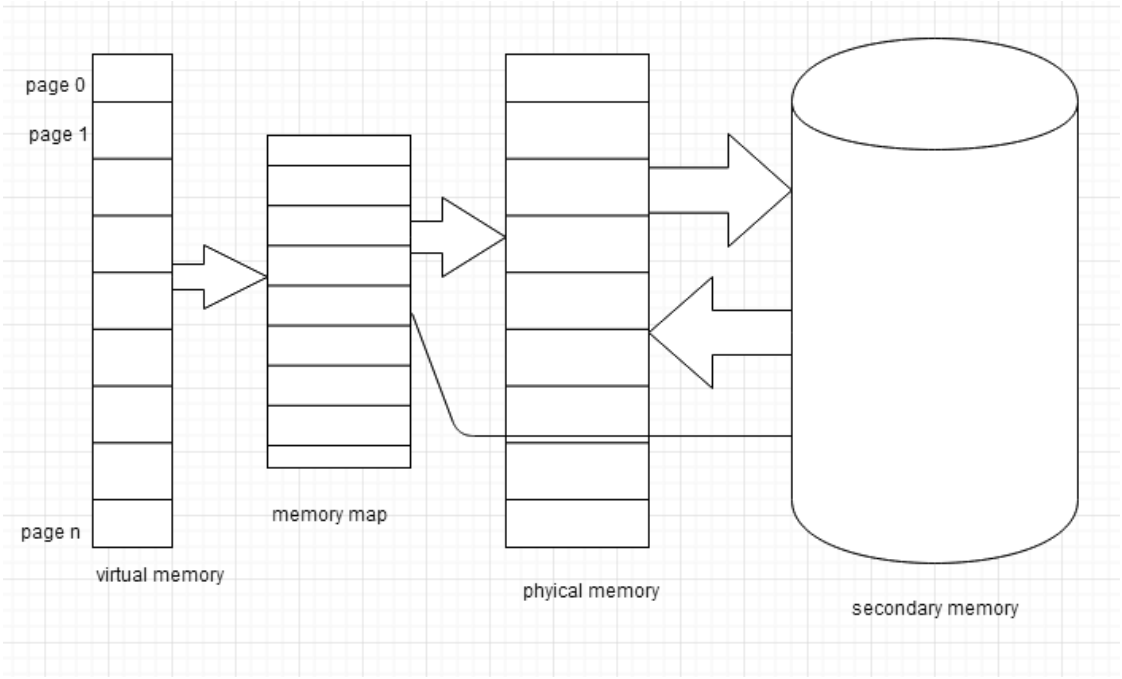
Cache Coherence:

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.

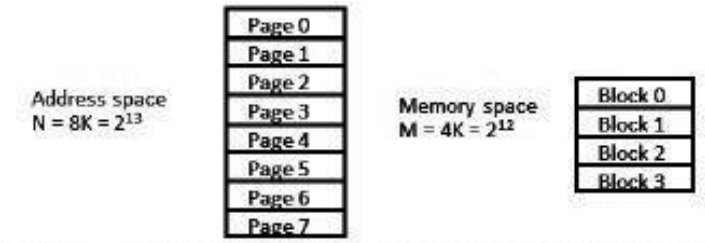
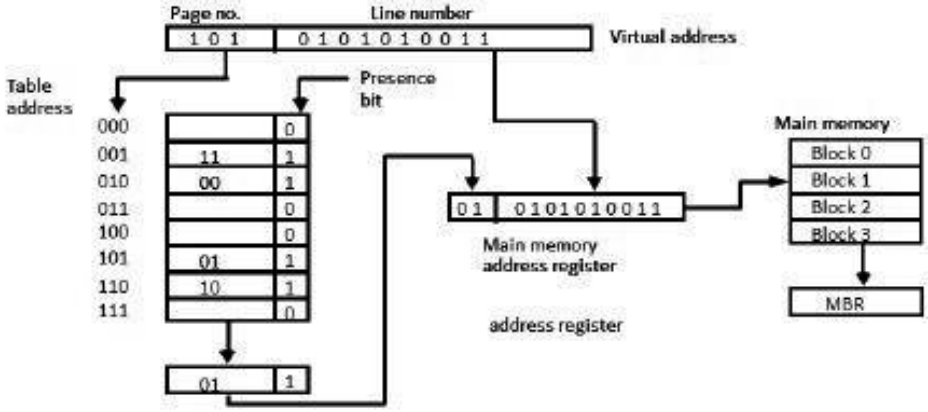


Figure Address and Memory space split into group of 1K words



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 outised world to computer) as well as, output(from computer to the outside world). Example: Touch screen etc.

Interfaces

Interface is a shared boundary btween 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 Inteface
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 or 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 occasion 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.

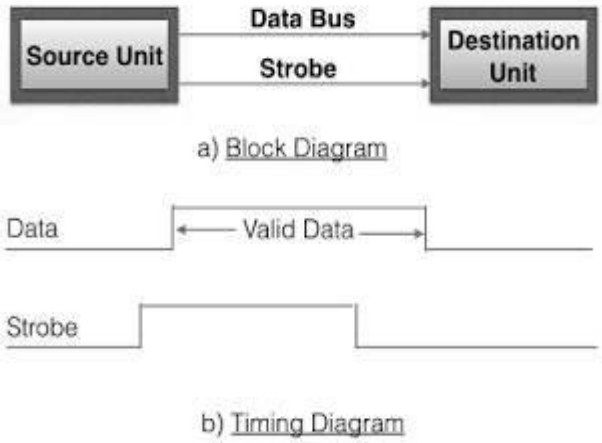
Now, we will discuss each method of asynchronous data transfer in detail one by one.

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 initiate transfer.

Source initiated strobe for data transfer:

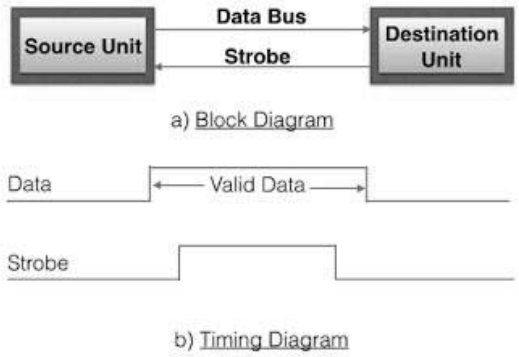
The block diagram and timing diagram of strobe initiated by source unit is shown in figure below:



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 per determine 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.

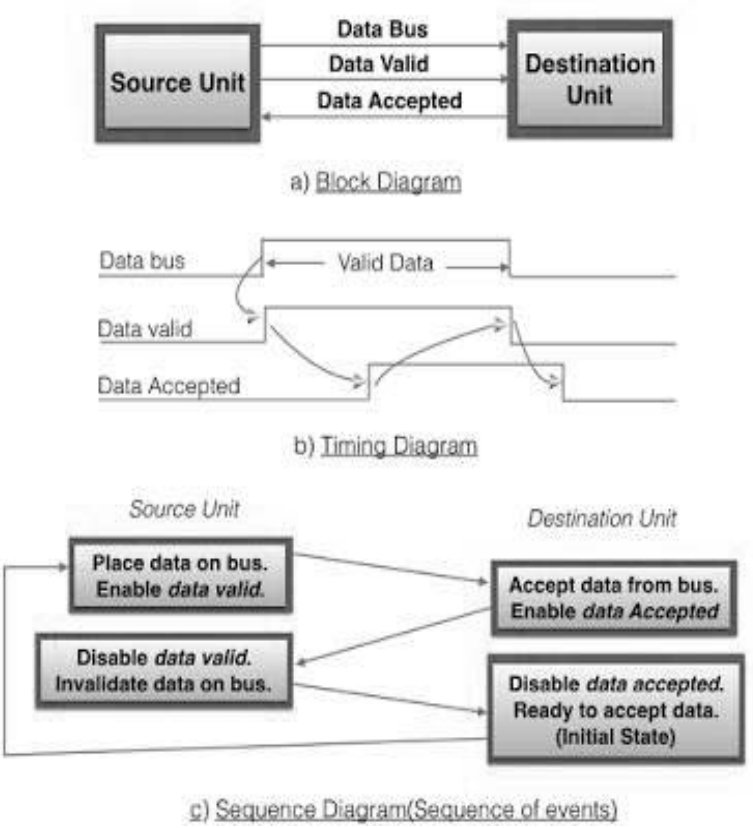
This problem can be solved by handshaking method.

Hand shaking method introduce 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:



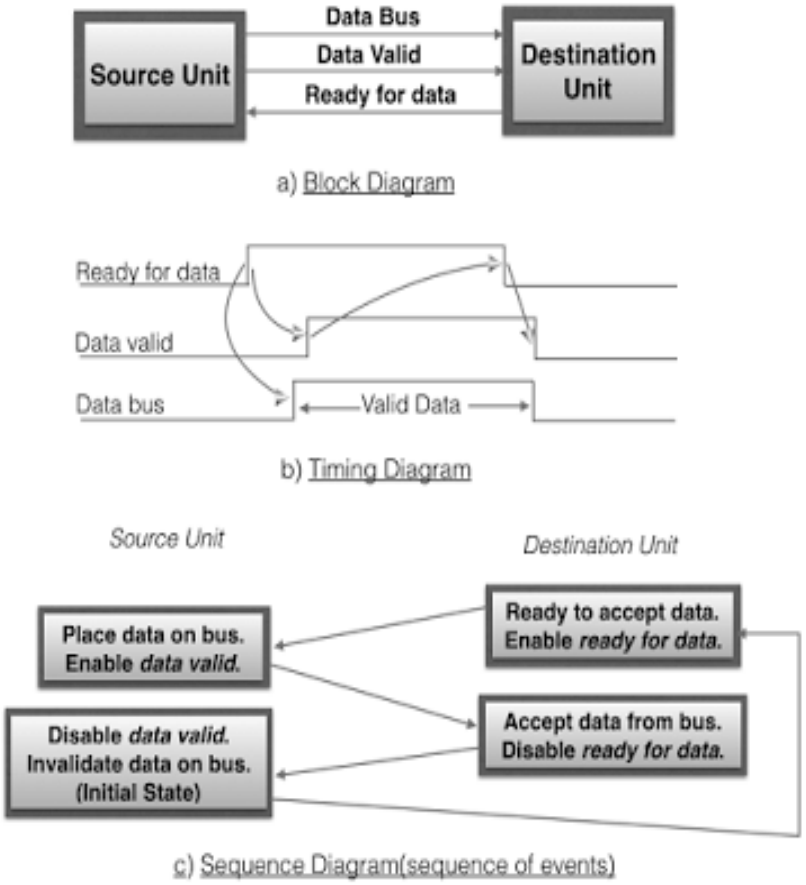
In its block diagram, we se that two handshaking lines are "data valid", which is generated by the source unit, and "data accepted", generated by the destination unit.

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 disable 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:



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 some 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 sound cards 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

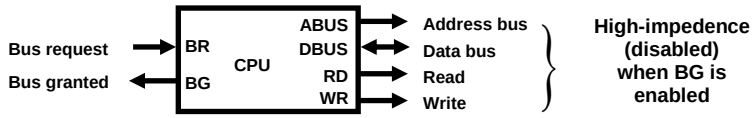
A priority interrupt is a system which decides the priority at which various devices, which generates the interrupt signal at the same time, will be serviced by the CPU. The system has authority to decide which conditions are allowed to interrupt the CPU, while some other interrupt is being serviced. 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.

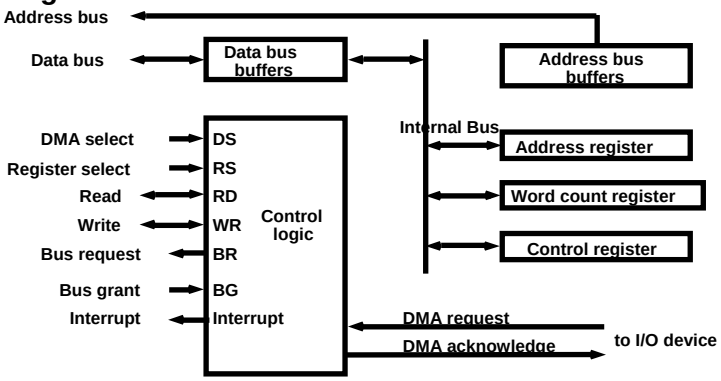
DIRECT MEMORY ACCESS

Block of data transfer from high speed devices, Drum, Disk, Tape

CPU bus signals for DMA transfer



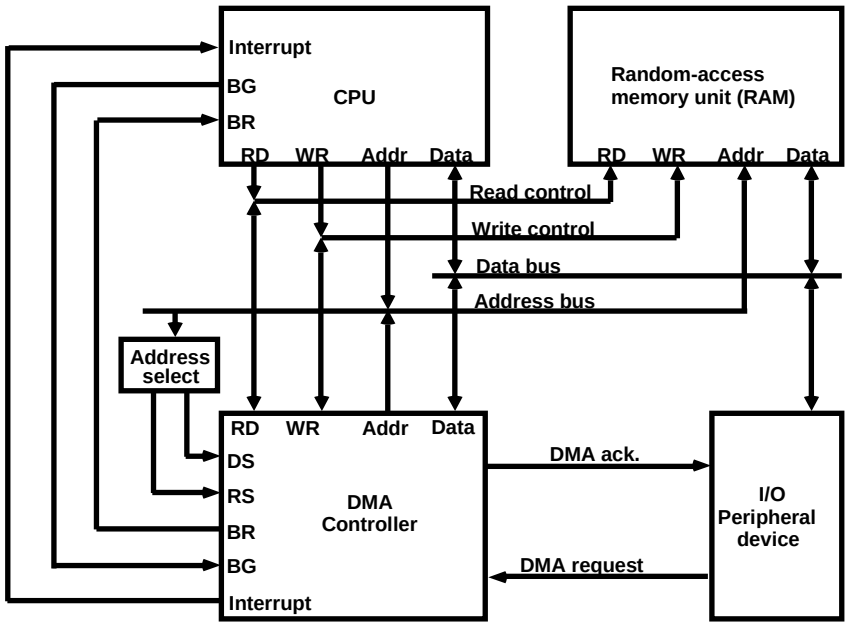
Block diagram of DMA controller



***** 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.

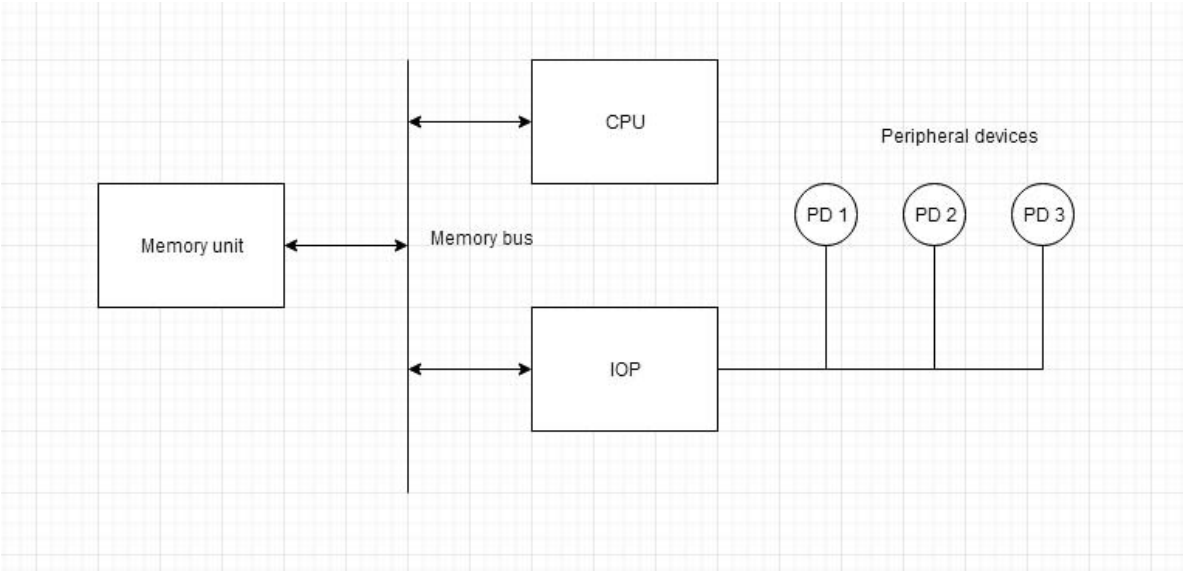
Each IOP controls and manage the input-output tasks. The IOP is similar to CPU except that it handles only the details of I/O processing. The IOP can fetch and execute its own instructions. These IOP instructions are designed to manage I/O transfers only.

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.