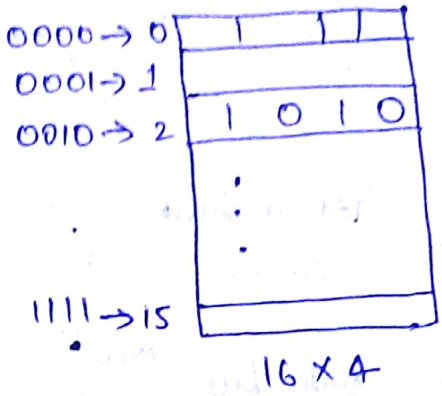
## ⊛ System bus : (Interconnection b/w CPU and memory)

1) Address bus
2) Data bus
3) Control bus




$N = 16$

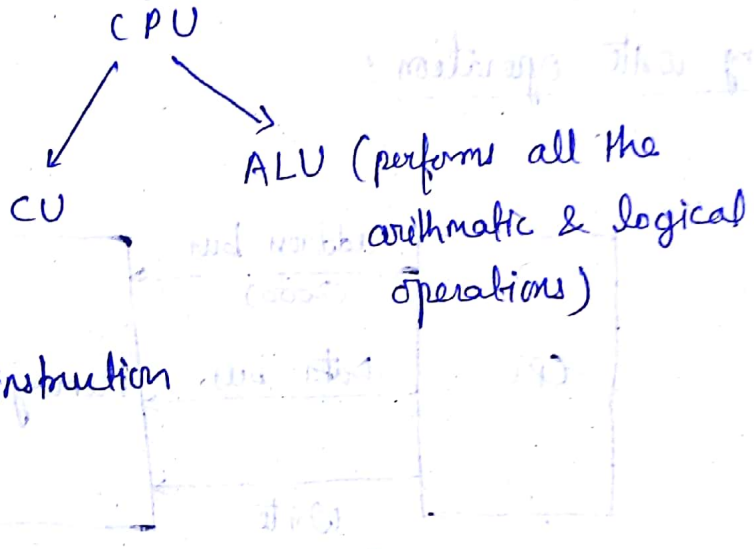
$\log_2 N$

$= \log_2 16$

$= 4 \Rightarrow$ size of address bus

## ⊛ Size of address bus

$= 4 \Rightarrow$ because here each location requires 4 bits.



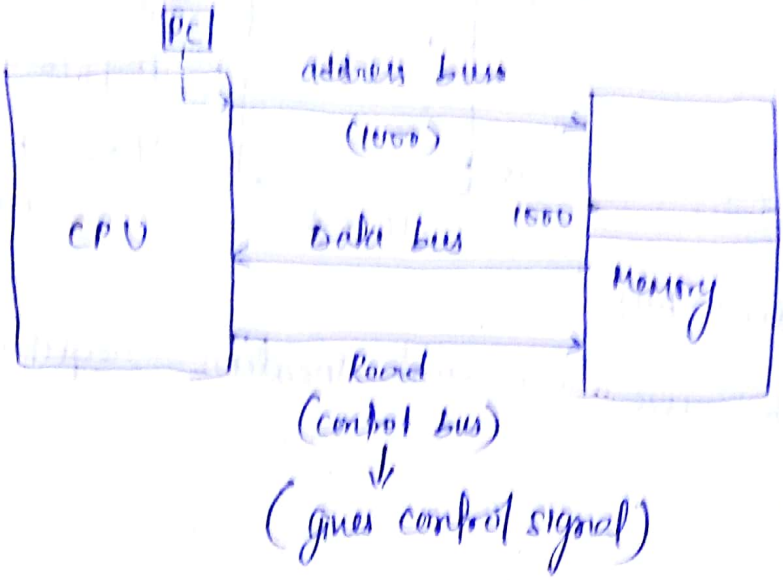

⇒ Initially the instructions are stored in the memory in binary form.

## ⊛ Steps for instruction execution cycle :-

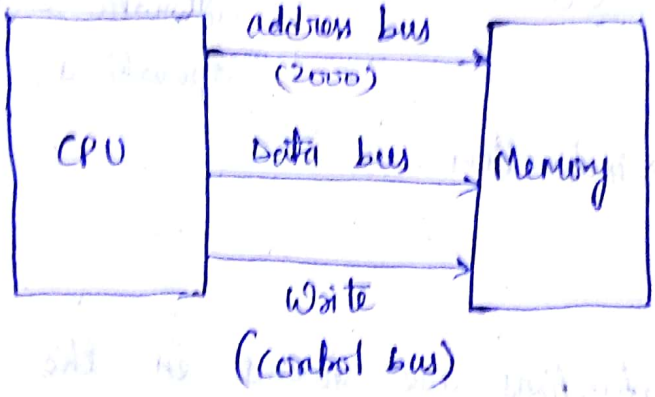
1) Fetch / Read the instructions from memory.
2) Decoding (After decoding the instructions we know that this is 'add' instruction)
3) Fetch the operand (access the content of a & b)
4) Execute the operation.
5) storing the result.

⑧ Program Counter → A special type of register which stores the starting address of a program.

⑨ Memory Read :

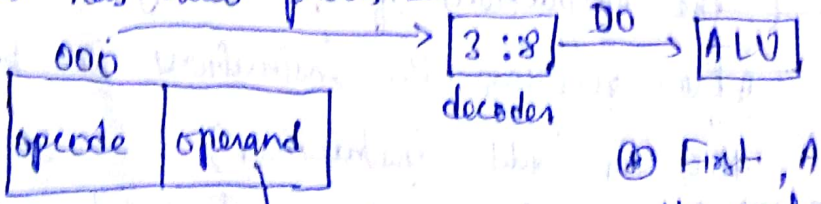



⑩ Memory write operation :




⇒ Address bus and control are unidirectional, but Data bus is bi-directional.

⑪ Instructions has two parts –



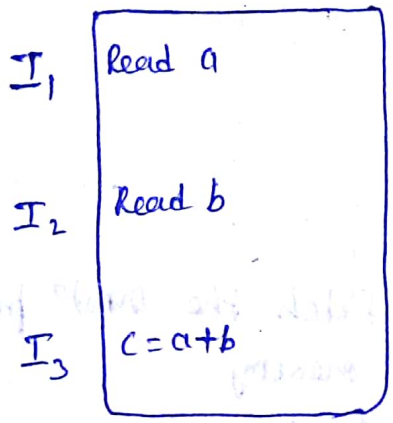

000 Add
001 Sub
010 Mul
011 Div

Address field or operand

⑫ First, ALU gets the control signal from CU and then it performs all the arithmatic operation.

18/1

# Van Neumann Machine (also called IAS computer)




→ Stored program concept

→ Developed in Institute of Advanced study, Princeton

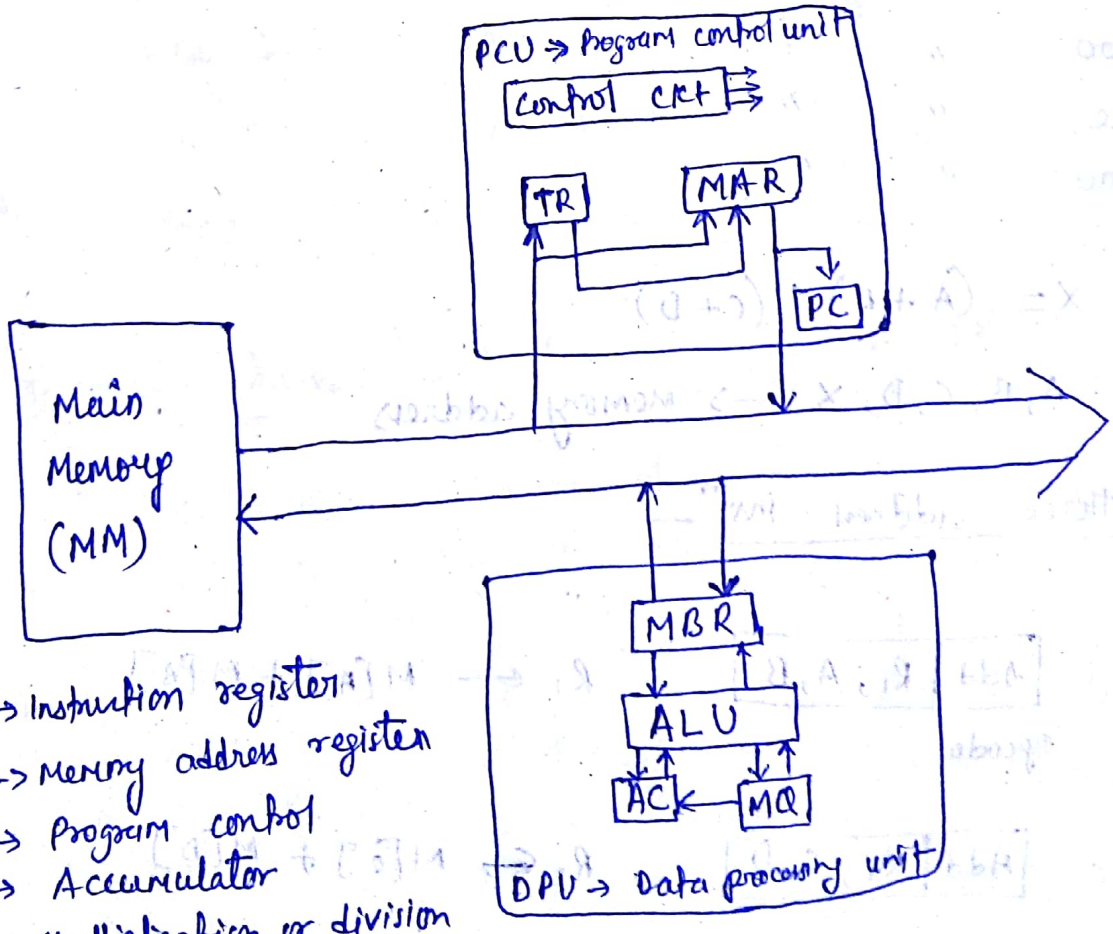
⇒ Three concepts for Von neumann machine —

1.) Data & instⁿ will be stored in single memory.

2.) Content of memory will be accessed by the address not by type of data

3.) Sequential execution.

## ⊛ Structure of IAS computer




IR → Instruction register

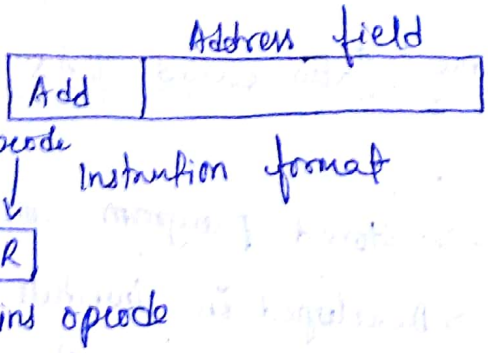
MAR → Memory address register

PC → Program control

AC → Accumulator

MQ → Multiplication or division

MBR → Memory buffered register. or DR or Data register




④ PCU –

PC → MAR → Address bus → Fetch the instⁿ from memory → opcode in IR

# Instruction format

→ The nos. of address field in the instruction format of a computer depends on the internal organisation.

→ There are four types –

① Three address instⁿ
② Two " "
③ One " "
④ Zero " "

Ex. $X = (A+B) * (C+D)$

A, B, C, D, X → Memory address

① three address instⁿ –

| Add | $R_1$, A, B |
|---|---|

opcode

$R_1 \leftarrow M[A] + M[B]$

| Add | $R_2$, C, D |
|---|---|

$R_2 \leftarrow M[C] + M[D]$

| MUL | X, $R_1$, $R_2$ |
|---|---|

$M[X] \leftarrow R_1 * R_2$

## ② Two address Inst$^n$

load

| | | |
|---|---|---|
| MOV | $R_1$, A | $R_1 \leftarrow M[A]$ |
| Add | $R_1$, B | $R_1 \leftarrow R_1 + M[B]$ |
| MOV | $R_2$, C | $R_2 \leftarrow M[C]$ |
| Add | $R_2$, D | $R_2 \leftarrow R_2 + M[D]$ |
| MUL | $R_1$, $R_2$ | $R_1 \leftarrow R_1 * R_2$ |
| MOV | X, $R_1$ | $M[X] \leftarrow R_1$ |

store

## ③ One address Inst$^n$

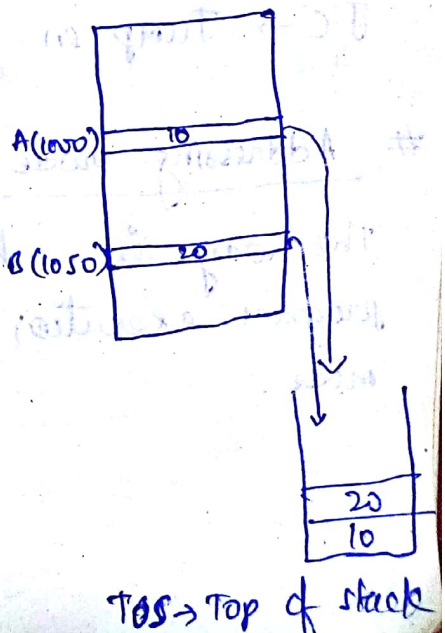
→ In case of one address Inst$^n$, accumulator is used

| | |
|---|---|
| LOad A | $AC \leftarrow M[A]$ |
| Add B | $AC \leftarrow AC + M[B]$ |
| STORE $T_0$ | $M[T_0] \leftarrow AC$ |
| Load C | $AC \leftarrow M[C]$ |
| Add D | $AC \leftarrow AC + M[D]$ |
| MUL T | $AC \leftarrow Ac * T$ |
| STORE X | $M[X] \leftarrow AC$ |

24/1

## ④ Zero address Inst$^n$ / stack organised Inst$^n$

| | |
|---|---|
| PUSH A | $TOS \leftarrow M[A]$ |
| PUSH B | $TOS \leftarrow M[B]$ |
| ADD | |
| PUSH C | $TOS \leftarrow M[C]$ |
| PUSH D | $TOS \leftarrow M[D]$ |
| ADD | |
| MUL | |
| POP X | $M[X] \leftarrow TOS$ |



## Types of instructions —

I) Data transfer instruction —

MOV, LOAD, STORE, PUSH, POP

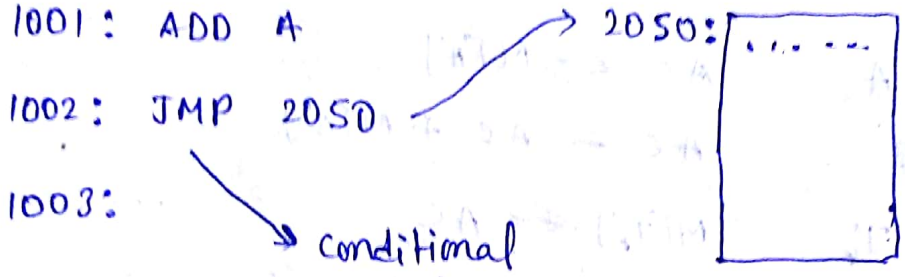
II) Data processing instruction —

(a) Arithmatic & logical

ADD, MUL, DIV, SUB  AND, OR, NOT

III) Program control instruction —

→ Branch instruction




1001 : ADD A
1002 : JNC 2050
1003 :
conditional

⇒ if (C == 0) ← carry bit
goto 2050
else
goto 1003

JNC → Jump not carry

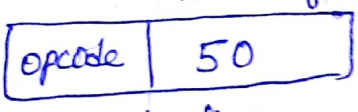
JC → Jump on carry

# Addressing mode :-

The way in which operands are choosen during program execution is dependent on the addressing mode.

I) Immediate : When address field contains data directly.
(Faster)

address field



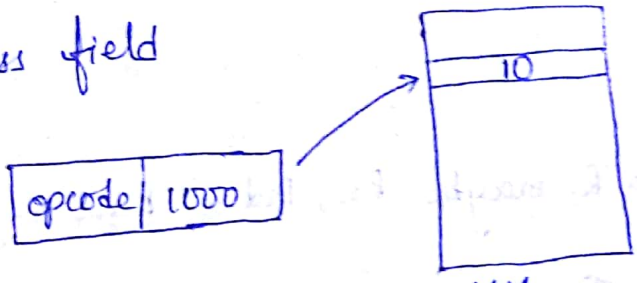

instruction

Ex: ADD 50

$AC \leftarrow AC + 50$

II) Direct addressing : When address field contains address.

Effective address = address field



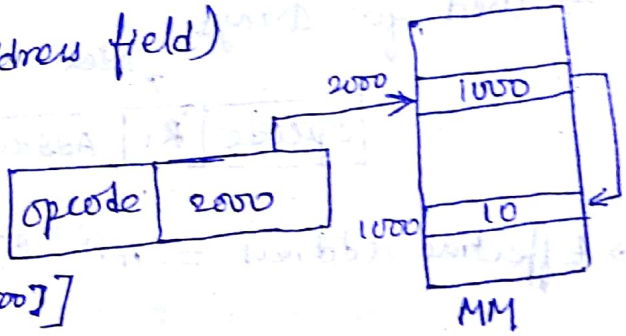

Ex: ADD 1000

$AC \leftarrow AC + M[1000]$

ADD 1000, $R_1$

III) Indirect : content of content of address field.

Effective address = (address field)




Ex. ADD (2000)

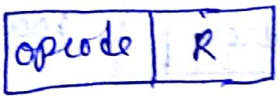
$AC \leftarrow AC + M[M[2000]]$

→ Have to access memory twice

⇒ can access more memory location using single instruction.

IV) Register addressing : Data present in register.

ADD R

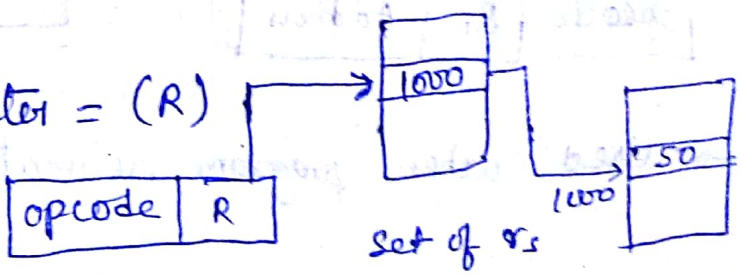



R → register

$AC \leftarrow AC + R$

V) Register indirect :

Effective a register = (R)

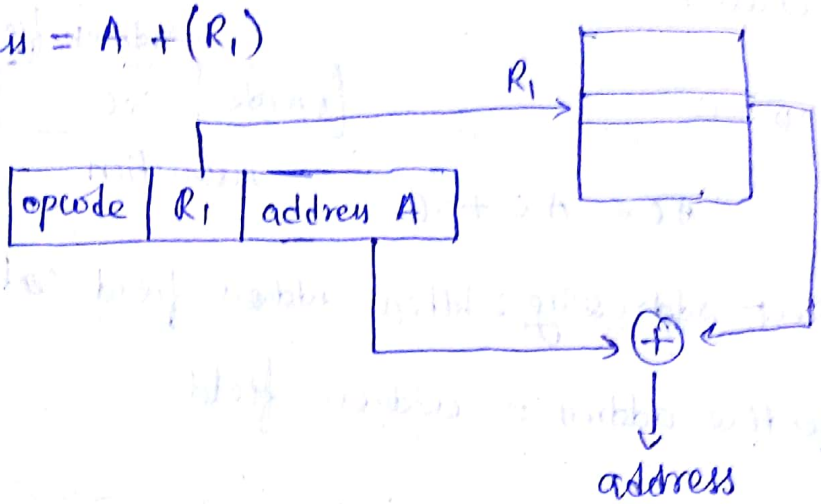



ADD (R)

$AC \leftarrow AC + M[R]$

## VI) Displacement addressing :

Effective address = A + ($R_1$)

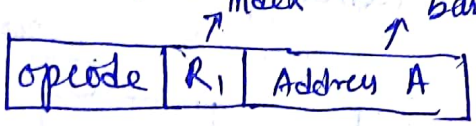



→ $R_1$ maybe PC, Index, base register

ⓐ Index addressing —

$R_1$ → index register

→ used for arrays.

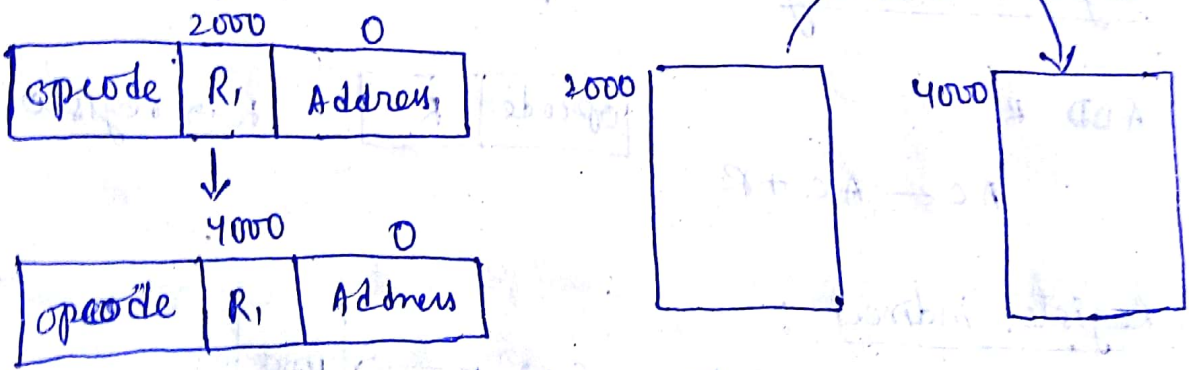



→ Effective address = A + R* size of data type.

ⓑ Base addressing —

$R_1$ → Base register

Effective address = A + Base




→ used when program relocation is required

(E) Relative addressing —

R → PC

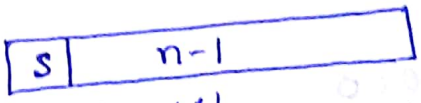
Effective address = PC + A

25||1

(*) Unsigned and signed number

Unsigned → No specific bit for sign

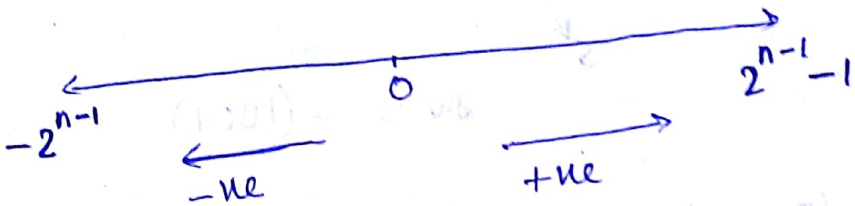
signed → 1 bit is used for sign



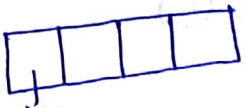

n bit

S = 0 ⇒ +ve

S = 1 ⇒ −ve




n = 4 bit



−8 to +7

used for sign

(*) +8 cannot be stored in 4-bit systems, this is known as overflow.

```
      S
      ↑
      1 1 1
+3    0 0 1 1
+5    0 1 0 1
      ---------
      0 1 0 0 0
      ↓
      S
```

To identify overflow –

| $C_n$ | $C_{n-1}$ | $C_{n-2}$ | $C_{n-3}$ | | |
|---|---|---|---|---|---|
| $C_4$ | $C_3$ | $C_2$ | $C_1$ | $C_0$ | +3 |
| | | | | | +8 |
| 0 | 1 | 1 | 1 | 0 | |

$C_n \oplus C_{n-1} = 1$ → for overflow.

Ex. −7 + −2

```
  000
  1001
  1110
 10111
```

$C_4 \oplus C_3 = 1$ ∴ overflow

S ↙ Ans = −(1001)

Ex. +7 + −3

```
  111
  0111
  1101
 10100
```

$C_4 \oplus C_3 \neq 1$ ∴ no overflow

S ↙ (0100) ↗ Ans

Ex. +7 + +2

```
  1
  0111
  0010
  1001
```

$C_4 \oplus C_3 = 1$ ∴ overflow

S ↙

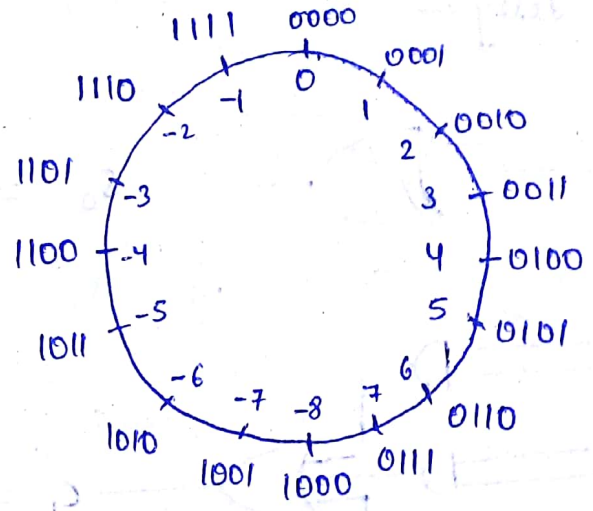
## # Addition modulo system

N = 16
4 bit **Addition modulo 16**

0 to 15
0 to N−1

→ For +ve nos.
move clockwise

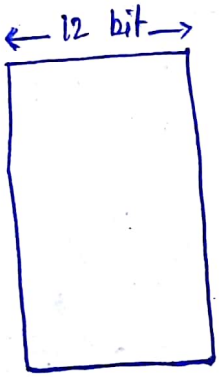
→ For -ve nos.
move anticlockwise




Range (-8 to +7)

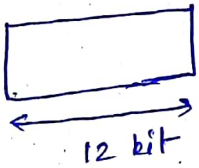
31/1

Ex




Address bus size = 16
Data bus = 12

data size



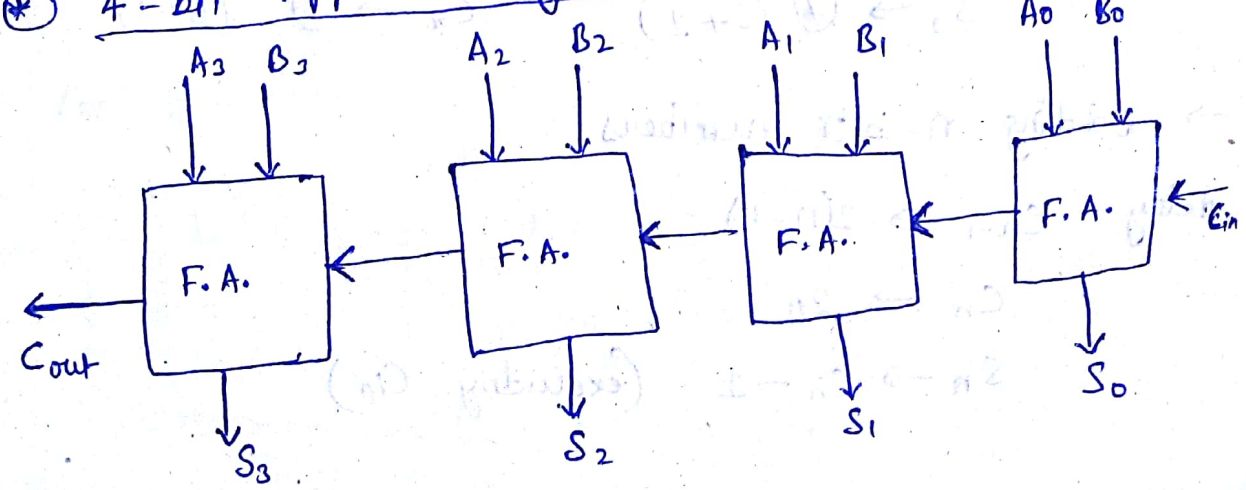

PC → 16 bits
MAR → 16 bits
MDR → 12 bits
IR → 12 bits

Ex $2^{20} \times 10$

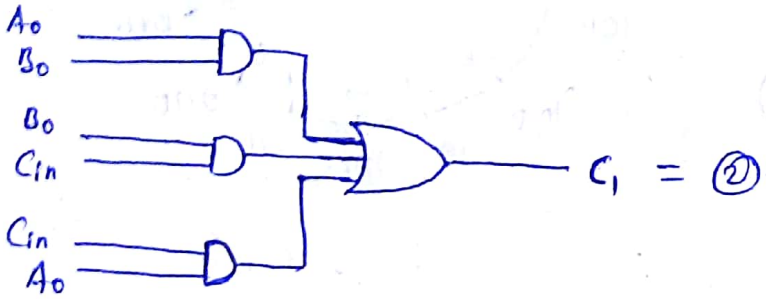
PC = 20 bits
MAR = 20 bits
MDR = 10 bits
IR = 10 bits

# Arithmatic unit

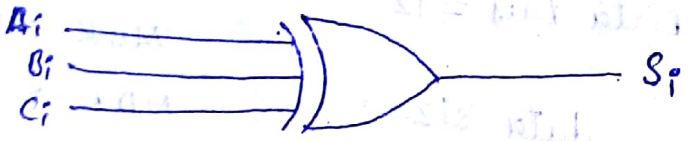
## (*) 4-bit ripple carry adder

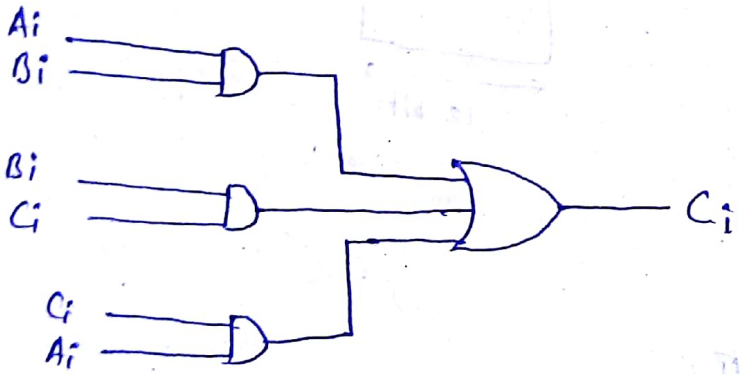


Gate delay –


General




$S_0 \rightarrow (1)$

$S_1 \rightarrow (3)$ $(2+1)$

$S_2 \rightarrow (5)$ $(4+1)$

$S_3 \rightarrow (7)$ $(6+1)$

$C_1 \rightarrow (2)$

$C_2 \rightarrow (4)$ $(2+2)$

$C_3 \rightarrow (6)$ $(4+2)$

$C_4 \rightarrow (8)$ $(6+2)$

→ Adding n-bit numbers

delay $C_{n-1} \Rightarrow 2(n-1)$

$C_n \rightarrow 2n$

$S_n \rightarrow C_n - 1$ (excluding $C_{in}$)

→ For overflow condition

$C_n \oplus C_{n-1} = 1$

↓

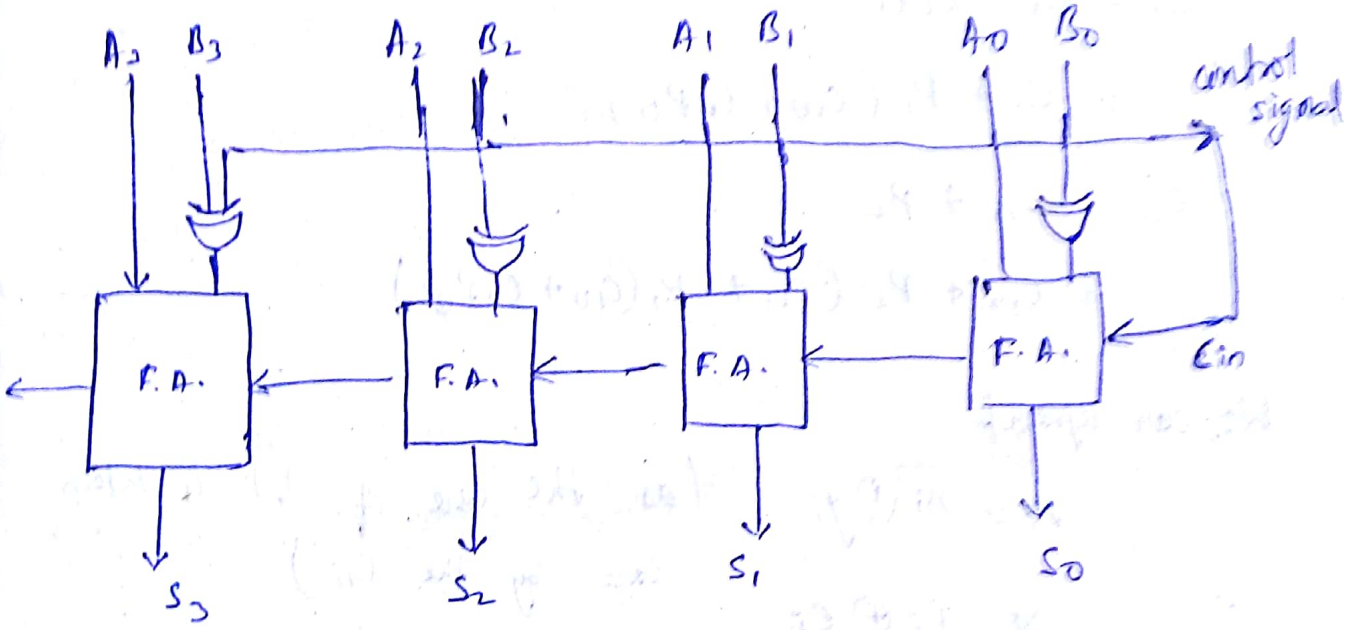
$2n$

$(2n+1)$ → detecting overflow

## (A1) Combinational adder/subtractor

control signal = 0 → add

= 1 → sub.




2's complement of $B = (\overline{B} + 1)$

Gate delay -

$C_n = 2n+1$

$S_n = C_n - 1$ (excluding $C_{in}$)

For overflow

delay $= 2n + 2$

## (4) Carry look ahead adder

$$C_{i+1} = x_i y_i + y_i c_i + c_i x_i$$

$$= x_i y_i + c_i (x_i + y_i)$$

$$= G_i + c_i P_i$$

$$G_i = x_i y_i \quad ; \quad P_i = x_i + y_i$$

$$C_1 = G_0 + C_0 P_0$$

$$C_2 = G_1 + C_1 P_1$$

$$= G_1 + P_1 (G_0 + C_0 P_0)$$

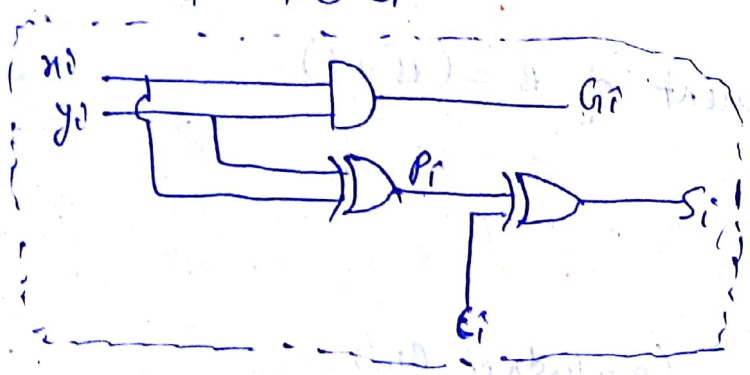
$$C_3 = G_2 + P_2$$

$$= G_2 + P_2 (G_1 + P_1 (G_0 + C_0 P_0))$$

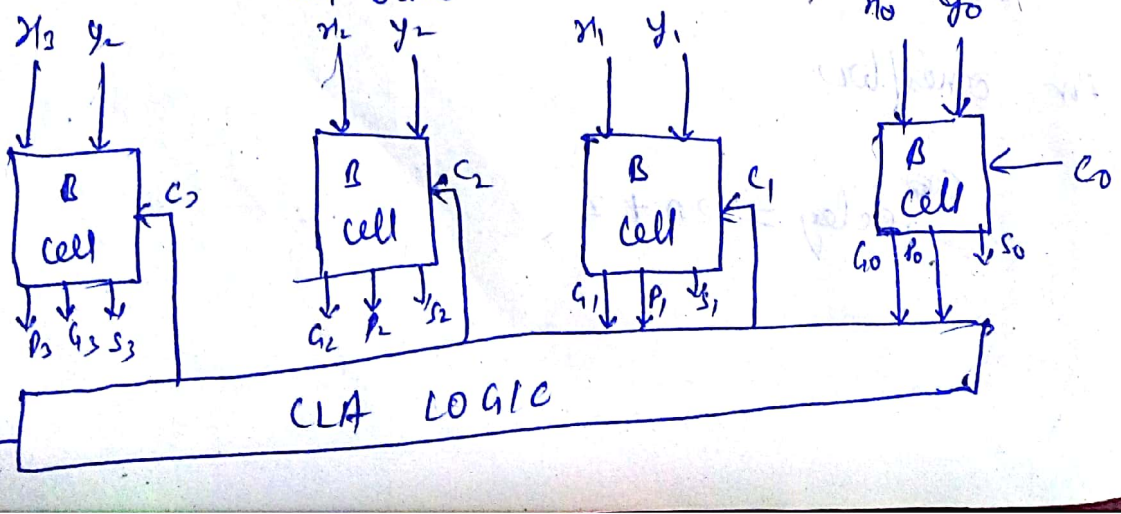
We can represent

$P_i = x_i \oplus y_i$ (as the case of 1,1 is taken care by the $G_i$)

$S_i = P_i \oplus C_i$

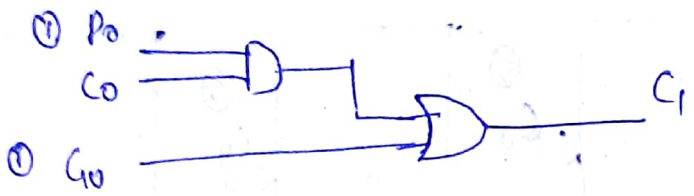



Bit store cell



Gate delay:

$C_1 = G_0 + P_0 C_0$




$C_1 = ① + ① + ① = ③$

$S_0 = P_0 \oplus C_0 = ②$

$\rightarrow P_i, G_i \rightarrow ① \rightarrow C_i \rightarrow ③$

$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$

$S_2 = P_2 \oplus C_2 \rightarrow ④$

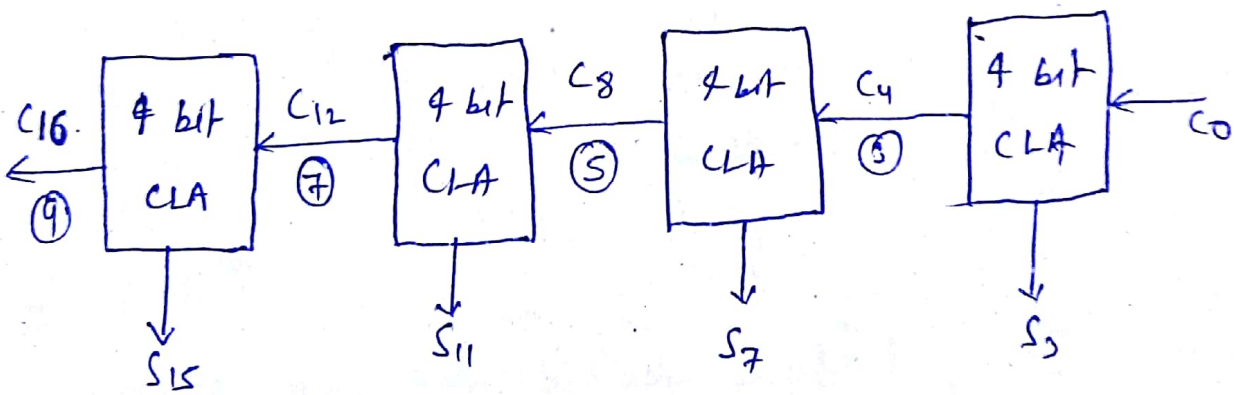
$S_1, S_-, S_3 \rightarrow ④$

(*) 16-bit CLA adder:

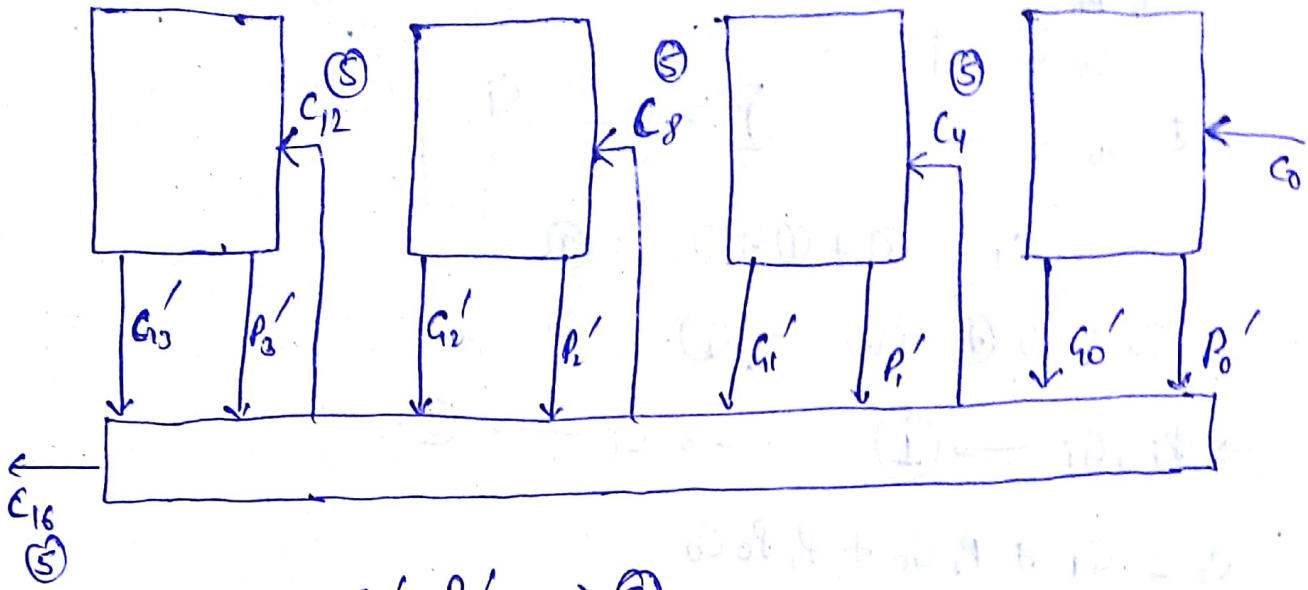
I) Cascading CLA

II) Higher level carry propagation & generation

I)



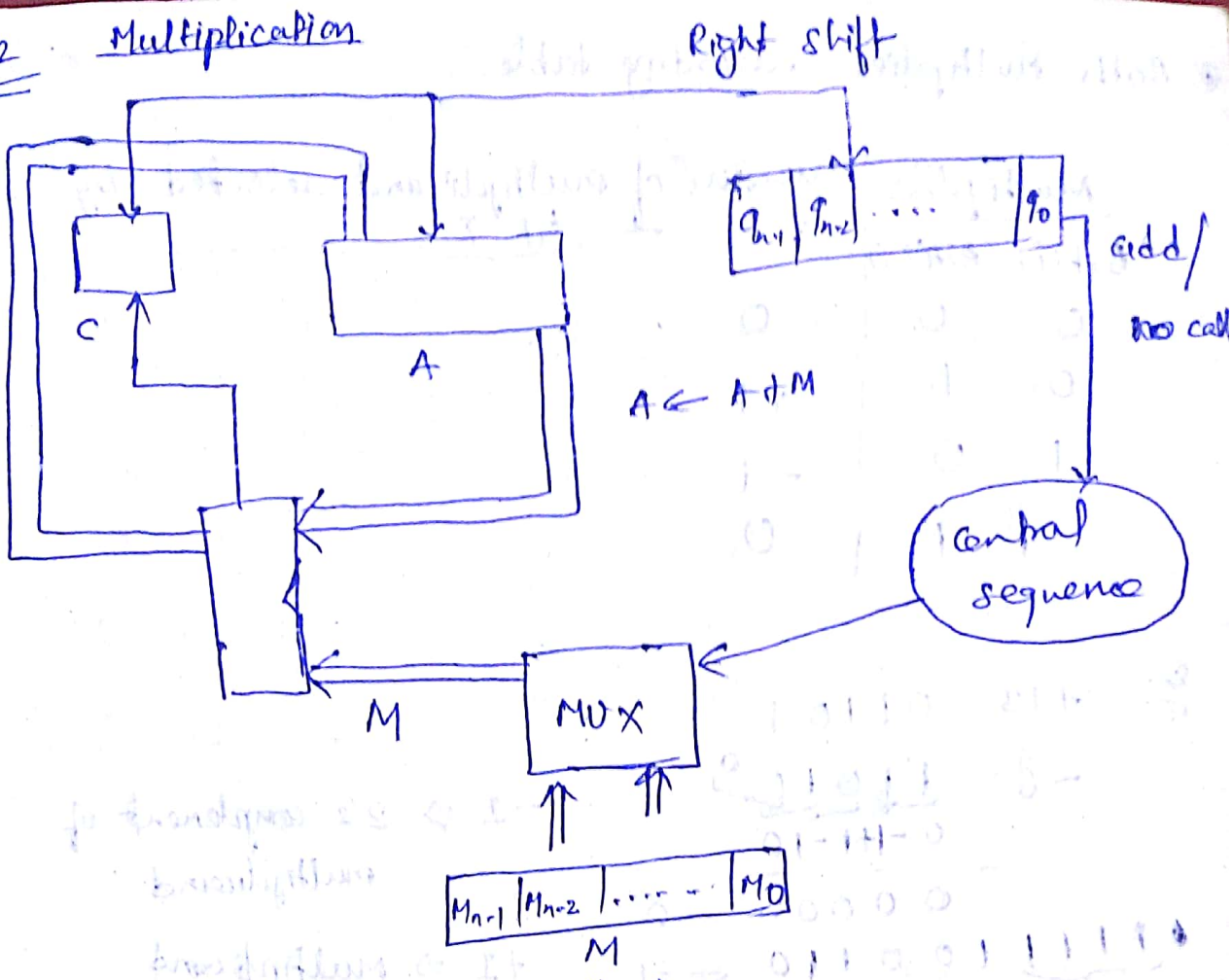
II) $C_4 = \underbrace{G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0}_{G_0'} + \underbrace{P_3 P_2 P_1 P_0}_{P_0'} C_0$




$G_0', P_0' \longrightarrow$ (1)

$G_1', P_1' \longrightarrow$ (2)

7/2 Multiplication




| | | | |
|---|---|---|---|
| 1 | 1101 | 1011 | → add |
| 0 | 0110 | 1101 | → shift |
| 1 | 0011 | 1101 | → add |
| 0 | 0100 | 1110 | → shift |
| 0 | 0100 | 1111 | |
| 1 | 0001 | 1111 | → add |
| 0 | 1000 | 1111 | → shift |

1101
1011

## ⊛ Booth's Algo

→ Extend the sign bit of partial product to the left as far as the product will extend.

→ Applicable for +ve as well as -ve multiplier.

④ Both Multiplier recording table –

| Multiplier | | version of multiplicand selected by bit i |
|---|---|---|
| Bit(i) | Bit(i-1) | |
| 0 | 0 | 0 |
| 0 | 1 | +1 |
| 1 | 0 | -1 |
| 1 | 1 | 0 |

Ex.

+13 01101

-6 11010 0

0 -1 +1 -1 0

-1 ⇒ 2's complement of multiplicand

+1 ⇒ multiplicand

00000 ← 0

11110011 0 ← -1

sign bit extention

00001101 ← +1

11100111

1 1 11 01100 10 ← -1

Cout ⇓ ignore

sign bit

2's comp. = (001001110)

Ans = - ( " )

⑤ Bit pair recording multiplication

→ This technique reduces the number of partial products by two times.

→ For n-bit operand, nos. of partial product will be $n/2$ and it is directly derived from the booth's algo by grouping the multiplier bits in pair.

Table:

| pair | | result |
|---|---|---|
| -1 | 0 | -2 |
| +1 | 0 | +2 |
| -1 | +1 | -1 |
| +1 | -1 | +1 |
| 0 | 0 | 0 |
| 0 | -1 | -1 |
| 0 | +1 | +1 |

+13 0 1 1 0 1

-6 1 1 0 1 0 0

By Booth's algo ← 0 -1 +1 -1 0

-1 -2

-2 → 2's complement and left shift

+2 → left shift

-1 → 2's complement

-9 = 1 0 1 1 1

+3 = 0 0 0 1 1 0

0 +1 0 -1

+1 -1

0 0 0 0 0 0 1 0 0 1

1 1 1 1 1 0 1 1 1

(1) 1 1 1 1 1 0 0 1 0 1

sign

$A_{11} = -(0000110111)$

8/2

1 1 1 1 1 0 0 1 1 0 ← -2

1 1 1 1 1 0 0 1 1 ← -1

(1) (1) 1 1 0 1 1 0 0 1 0

carry sign

Carry save adder

1 1 0 1

1 1 1 1

(1) 1 0 1

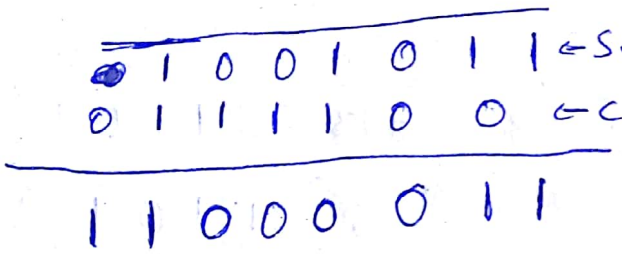
1 1 0 1

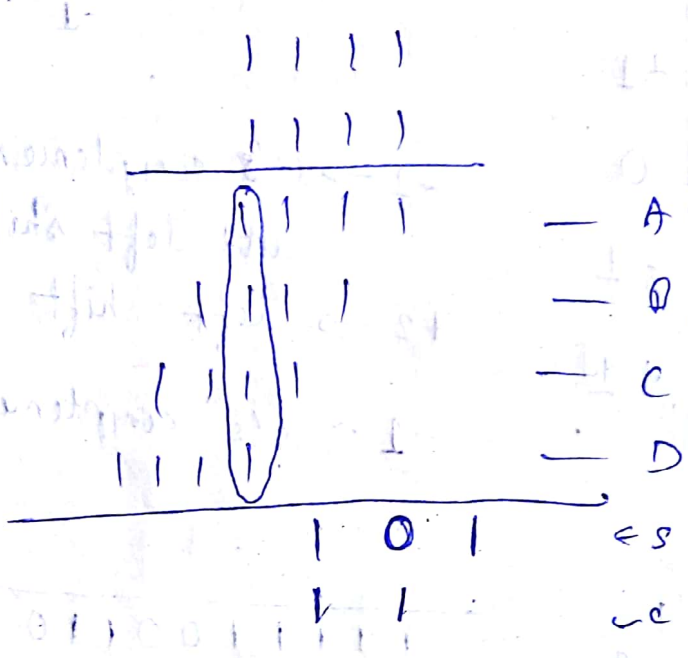
1 1 0 1

1 1 0 1

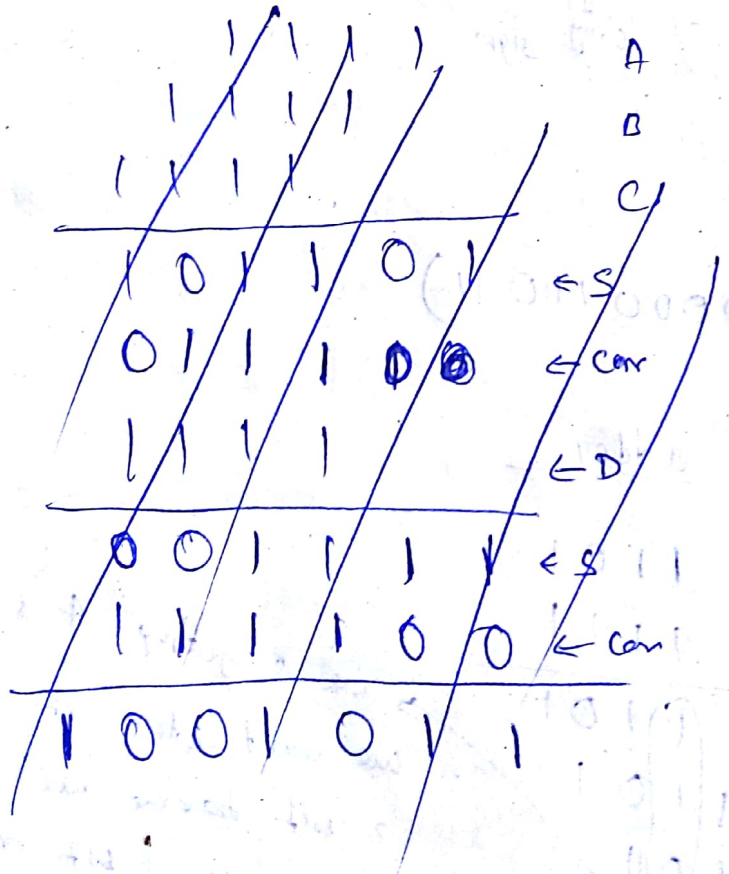
0 1 1 1 0 1 1

we r getting 4 & 1s we can't store it using 2 bit bcoz we are getting 3 bit result as 100.


1 0 0 1 0 1 1 ← S
0 1 1 1 1 0 0 ← C
1 1 0 0 0 0 1 1

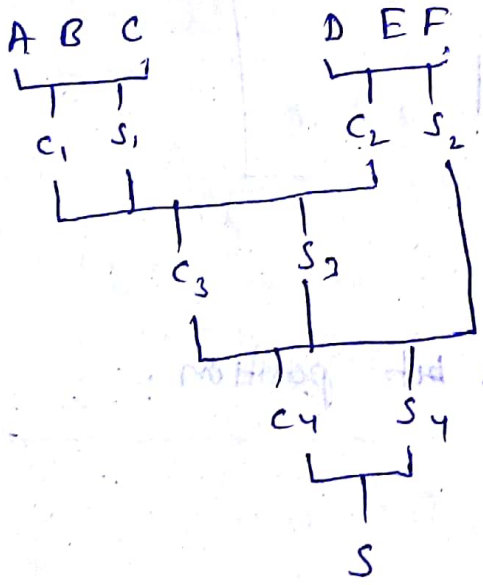

A
B
C
D
1 0 1 ← S
1 1 ← C

Label 1 : A + B + C


A
B
C
1 0 1 0 1 ← S
0 1 1 1 0 0 ← Carr
1 1 1 1 ← D
0 0 1 1 1 1 ← S
1 1 1 1 0 0 ← Carr
1 0 0 1 0 1

```
    1 1 1 1      A
  1 1 1 1        B
  1 1 1 1        C
-----------------
  1 0 1 1 0 1  ← S1
0 1 1 1 1 0    ← C1
    1 1 1 1      D
-----------------
  1 1 0 1 0 0 1  ← S2
0 1 1 1 1 0 0    ← C2
-----------------
1 1 1 0 0 0 0 1
```

For 6 bit adder.

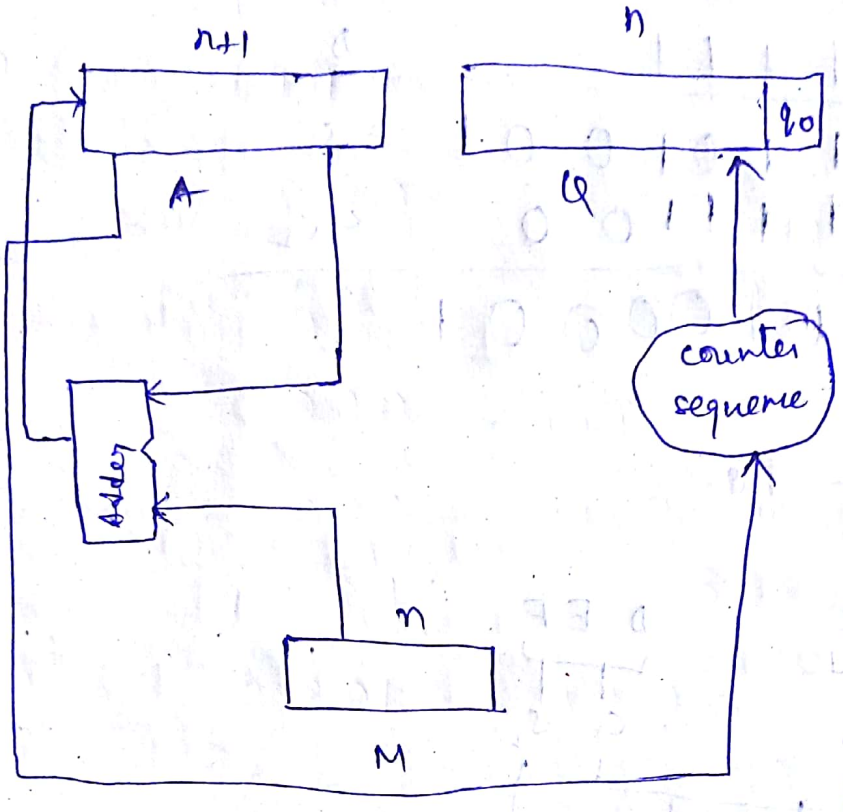



```
      1 1 1 1 1 1
    1 0 1 0 1 0
-------------------
    0 0 0 0 0 0    A
  1 1 1 1 1 1      B
0 0 0 0 0 0        C
  1 1 1 1 1 1      D
0 0 0 0 0 0        E
1 1 1 1 1 1        F
```

14/2 Division

```
11 | 1000 | 010 → Q
     11
     ---
     10 → remaind
```

② Restoring technique




Steps:

i) shift A and Q left one bit position.

ii) $A \leftarrow A - M$

iii) IF sign of A is 1

$q_0 \leftarrow 0$, $A \leftarrow A + M$

else

$q_0 \leftarrow 1$

Ex. m → divisor → 11

Q → dividend → 1000

M

| 0 | 0 | 1 | 1 |
|---|---|---|---|

| A | | | | | Q | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |

initial

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| shift | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | □ |
| A ← A-M | (1) | 1 | 1 | 1 | 0 | | | | |
| $q_0$ ← 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| A ← A+M | ignore | | | | | | | | |

cycle 1

---

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| shift | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | □ |
| A ← A-M | (1) | 1 | 1 | 1 | 1 | | | | |
| $q_0$ ← 0 | | | | | | | | | |
| A ← A+M | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | ignore | | | | | | | | |

cycle 2

---

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| shift | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | □ |
| A ← A-M | (1) | (0) | 0 | 0 | 0 | 1 | | | |
| | carry | sign | | | | | | | |
| $q_0$ ← 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

cycle 3

---

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| shift | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | □ |
| A ← A-M | (1) | 1 | 1 | 1 | 1 | | | | |
| $q_0$ ← 0 | | | | | | | | | |
| A ← A+M | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| | remainder | | | | | quotient | | | |

cycle 4

n = 1 0 0

| | A | Q | |
|---|---|---|---|
| | 0 0 0 0 | 1 1 0 1 | |
| shift | 0 0 0 1 | 1 0 □ | cycle 1 |
| A ← A − M | ① 1 0 1 | | |
| $q_0 \leftarrow 0$ | | | |
| A ← A + M | 0 0 0 1 | 1 1 0 | |
| shift | 0 0 1 1 | 1 0 □ | |
| A ← A − M | ① 1 1 1 | | |
| $q_0 \leftarrow 0$ | | | cycle 2 |
| A ← A + M | 0 0 1 1 | 1 0 0 | |
| shift | 0 1 1 1 | 0 0 □ | |
| A ← A − M | ⓪ 0 1 1 | 0 0 1 | |
| $q_0 \leftarrow 1$ | | | |

(a) Non-restoring

Step 1: Do if n times

i) If the sign of A is 0

shift A and Q left one bit position

$A \leftarrow A - M$

else

left shift A and Q

$A \leftarrow A + M$

ii) If the sign of A is 0

$q_0 \leftarrow 1$

else

$q_0 \leftarrow 0$

Step 2: If the sign of A is 1

$A \leftarrow A + M$

M → 0011

| | A | Q | |
|---|---|---|---|
| | 0000 | 1000 | |
| Step 1: shift | 0000 1 | 000 □ | cycle 1 |
| $A \leftarrow A - M$ | ① 1110 | 0000 | |
| $q_0 \leftarrow 0$ | | | |
| shift | 1110 0 | 000 □ | cycle 2 |
| $A \leftarrow A + M$ | ① 1111 | 0000 | |
| $q_0 \leftarrow 0$ | | | |
| shift | 1111 0 | 000 □ | cycle 3 |
| $A \leftarrow A + M$ | 1 ⓪ 0001 | 0001 | |
| $q_0 \leftarrow 1$ | | | |
| shift | 0001 0 | 001 □ | cycle 4 |
| $A \leftarrow A - M$ | ① 1111 | 0010 | |
| $q_0 \leftarrow 0$ | | | |
| Step 2: $A \leftarrow A + M$ | 1 0001 0 | 0010 | |
| | rem. | Quo. | |

M 00101 01011

| | | |
|---|---|---|
| | 0 0000 | 1100 |
| Shift | 00001 | 100 □ |
| A ← A-M | 11100 | 1000 |
| $Q_0 \leftarrow 0$ | | |
| Shift | 11001 | 000 □ |
| A ← A+M | 11110 | 0000 |
| $Q_0 \leftarrow 0$ | | |
| Shift | 11100 | 000 □ |
| A ← A+M | 10000 1 | 0001 |
| $Q_0 \leftarrow 1$ | sign | |
| Shift | 00010 | 001 □ |
| A ← A-M | 11101 | 0010 |
| $Q_0 \leftarrow 0$ | | |
| Step 2 | | |
| A ← A+M | 10010 | 0010 |
| | rem | Quo. |

15/2

## Floating point representation

→ Fractional no.

→ Position of decimal point is not fixed.

① Exponent

② Mantissa

③ sign of exponent

④ sign of mantissa

④ Normalized form -

Position of decimal point after one non-zero digit.

Ex: 4.567

Ex: 0.75

$0.75 \times 2 = 1.50 \rightarrow 1$

$0.5 \times 2 = 1.00 \rightarrow 1$

$= (0.11)_2$

$= (1).1 \times 2^{-1}$ → normalized form

(1) → hidden bit

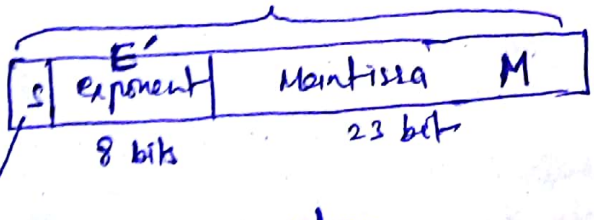
→ In case of binary system, 1st non-zero digit is 1, so no need to represent it.

## ⑤ IEEE 754 Standard

① Single precision → 32 bit

② Double " → 64 bit

### ① Single precision




E' → biased exponent

$E \Leftarrow E' - 127$

↓

actual exponent

S = 0 ⇒ +ve mantissa

= 1 ⇒ -ve mantissa

bias = 127

as K = 8

→ Bias

$bias = 2^{K-1} - 1$

(for exponent)

$0 \leq E' \leq 2^8 - 1$

$0 \leq E' \leq 255$

$E' = 0$ } Reserved
$E' = 255$

So actual range

$$1 \le E' \le 254$$

Cases -

① $E' = 0$ , $M = 0$

$\Rightarrow N = 0$

N → N number

② $E' = 0$ , $M \neq 0$

De normalized form

Ex. 0.0110

③ $E' = 255$ , $M = 0$

$\Rightarrow N = \infty$

④ $E' = 255$ , $M \neq 0$

$\Rightarrow N = NaN$ eg. $\frac{0}{0}$ , $\sqrt{-1}$

range of E

$$1 - 127 \le E \le 254 - 127$$

$$-126 \le E \le 127$$

Ex. $4.5 = 100.1 = 1.001 \times 2^2$

$E = 2$

$E' = 2 + 127 = 129$

| 0 | 10000001 | 00100... |
|---|---|---|
| | 8 bit | 23 bit |

Ex. $0.75 = 1.1 \times 2^{-1}$ $\quad E' = 126$

| 0 | 01111110 | 1000... |
|---|---|---|

* Range of single precision format

$E' = 0$ } reserved.
$E' = 255$ }

Min. $E' = 00000001$ $\quad$ min. $N = 1.0 \times 2^{-126}$ (+ve)

Min. $M = 000...0$

| 0 | 00000001 | 000... |
|---|---|---|

Min. $N = -1.0 \times 2^{-126}$ (-ve)

| 1 | 00000001 | 000... |
|---|---|---|

Max. $E' = 254$

$E = 254 - 127 = 127$

Max. $M = \underbrace{111...1}_{23}$

$111...1 \rightarrow 2 - 2^{-23}$

$000...1 \quad 2^{-23}$

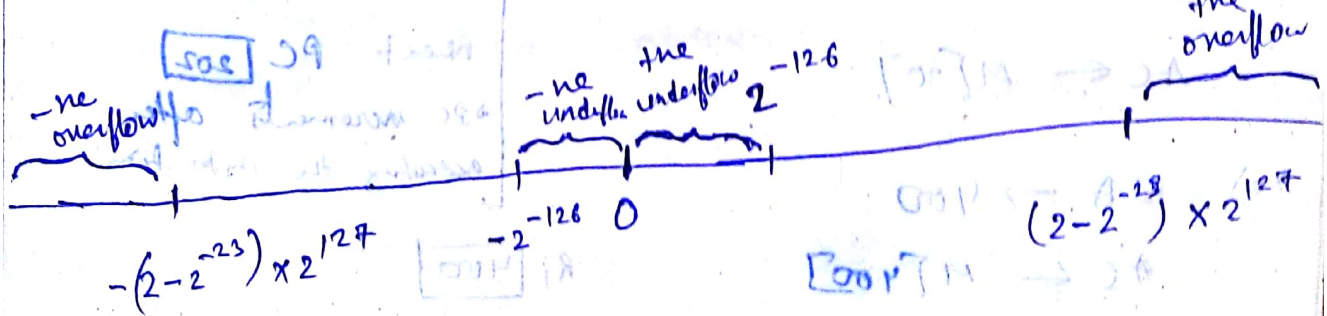
$10.000... \rightarrow 2$

| 0 | 11111110 | 111... |
|---|---|---|

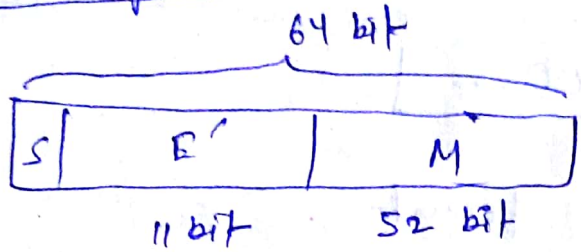
Max. $N = (2 - 2^{-23}) \times 2^{127}$ (+ve)

| 1 | 11111110 | 111... |
|---|---|---|

Max. $N = -(2 - 2^{-23}) \times 2^{127}$ (-ve)



④ Double precision




19/2

Ans: The two word instruction "LOAD AC" is stored at location 200 with address field at location 201. The address field contains the value 500. The register $R_1$ contains 400. Evaluate the effective address and content of AC if the following addressing mods are used -

i) Direct ii) immediate iii) Indirect

iv) Register indirect v) Relative vi) Index

Sol: LOAD AC 500

i) EA → 500

AC ← M[500] = 30

ii) EA → 201

AC ← 500

iii) EA → 30

AC ← M[30]

iv) EA → 400

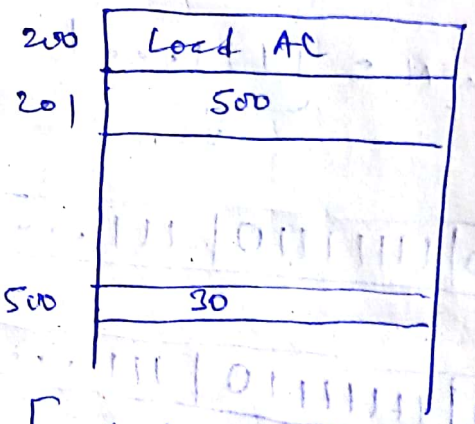
AC ← M[400]

v) EA = 202 + 500

AC ← M[702]

vi) EA = 400 + 500

AC ← M[900]




initial PC 200

next PC 202

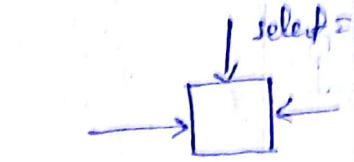
*PC increments after executing the instruction

$R_1$ 400

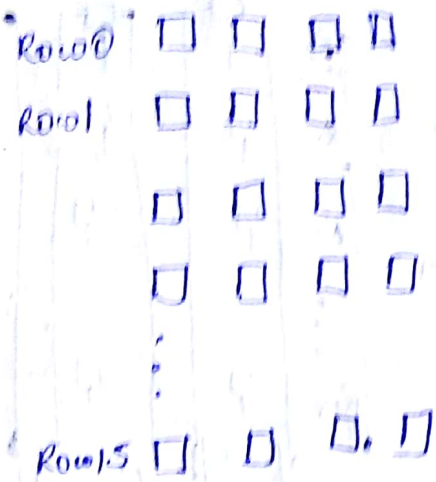
21/2 Main memory organization

Read = 1
write = 0

Properties of cell




16×4




→ group of n bits cell = word
→ word length = 4
→ word size = 4

→ size of adress bus = 4
→ size of data bus = 4

→ Nos. of locations = M
Nos. of bits required for a location = $\log_2 m$

32×8

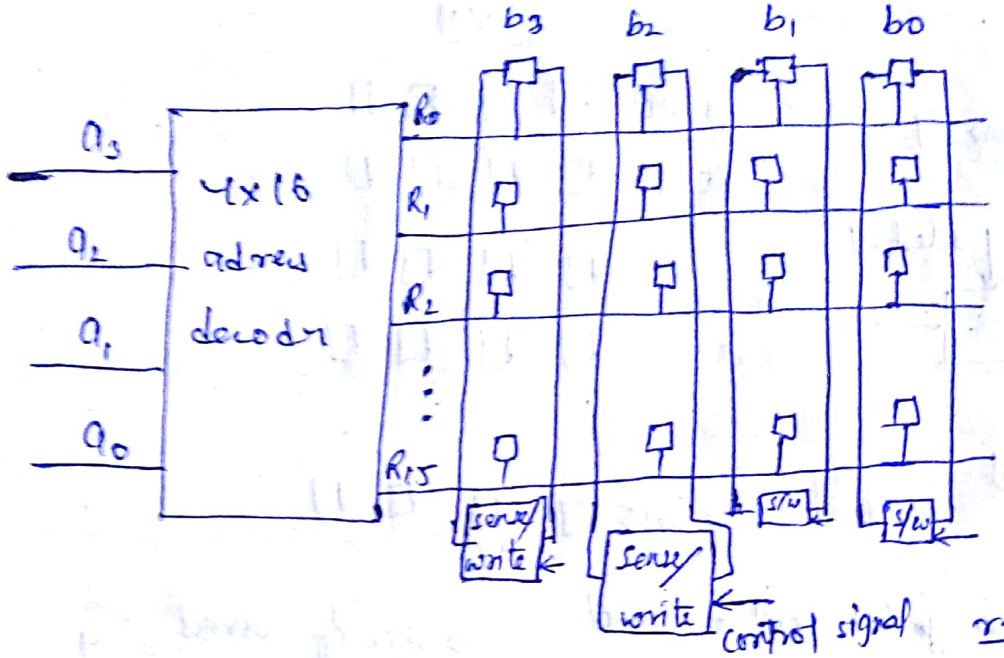
→ size of address bus = $\log_2 32 = 5$
→ size of data bus = 8

Ques Adress bus = 10, Data bus = 8

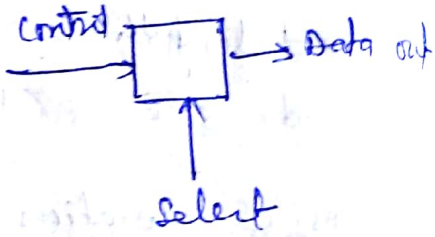
soln Memory = $2^{10} \times 8 = 1024 \times 8$

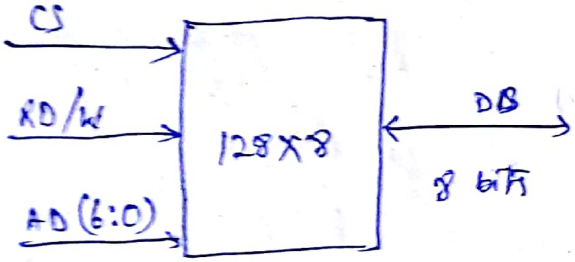
① Internal organization of memory chips.

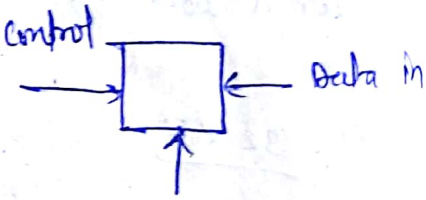
16 x 4



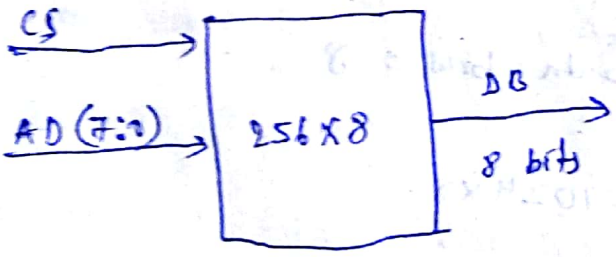

RAM chip




ROM chip



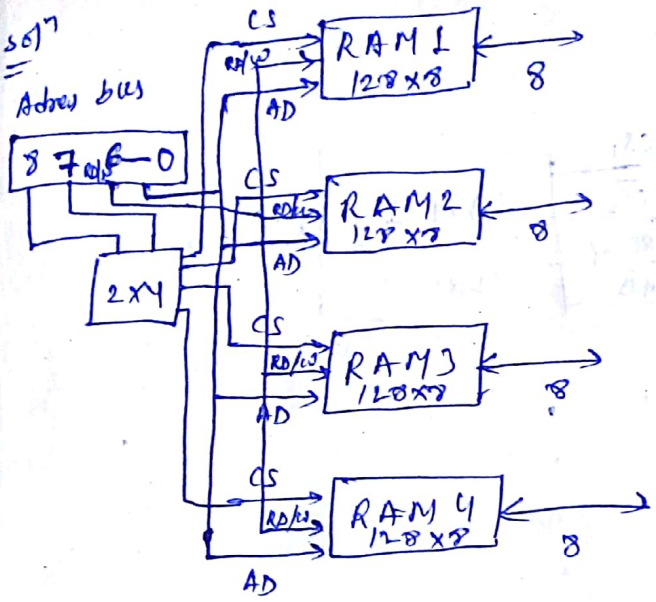

CS → chip select line

AD → Address bus

DB → Data bus

RD/W → Read/write

Ans = Your computer system required 512 bytes of RAM and 512 bytes of ROM. Size of each RAM is 128x8. Design the system.




→ Memory address map :

Memory address map specifies the address assigned to each chip.

| | | 8 | 7 | 6 — 0 | |
|---|---|---|---|---|---|
| RAM 1 | Min → | 0 | 0 | 0000000 | → 00 |
| | Max → | 0 | 0 | 1111111 | → 7F |

Range → 00 — 7F

| | | | | | |
|---|---|---|---|---|---|
| RAM 2 | | 0 | 1 | 000 | 0000 |
| | | | | 8 | 0 |
| | | 0 | 1 | 111 | 1111 |
| | | | | F | F |

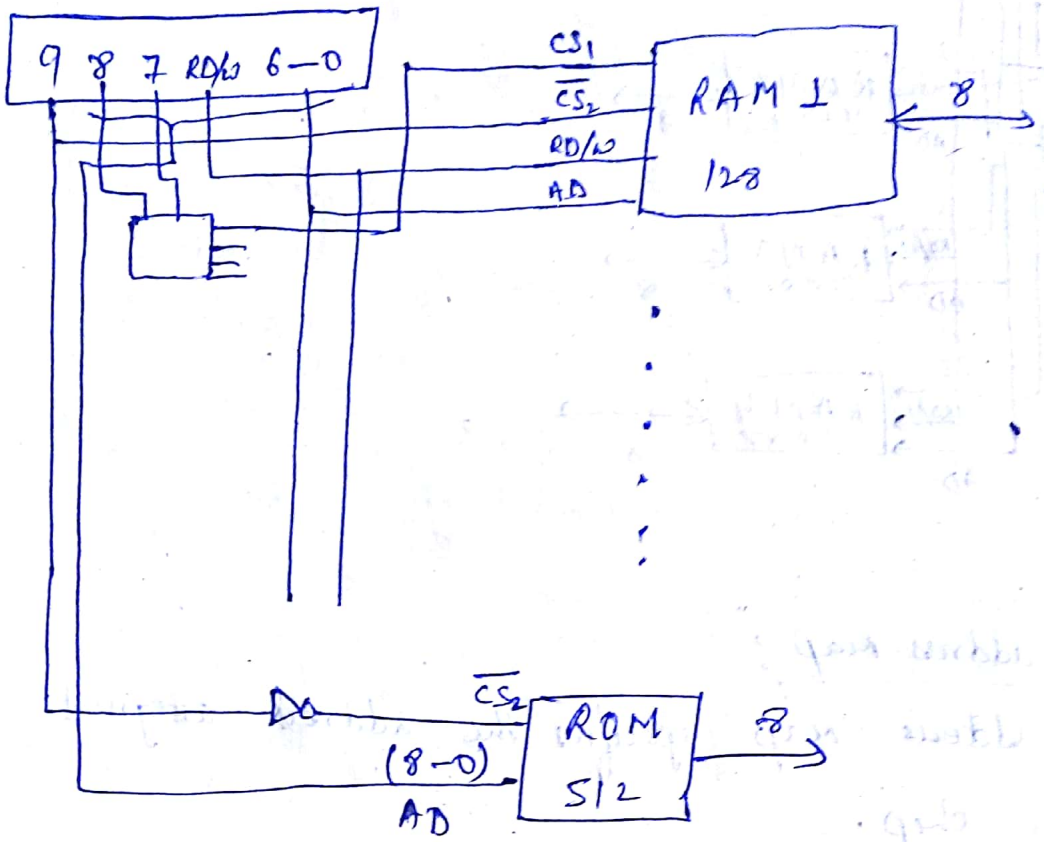
Range → 80 — FF

| | | | | |
|---|---|---|---|---|
| RAM 3 | 1 | 0 | 0 | 000000 |
| | 1 | 0 | 1 | 111111 |

Range → 100 — 17F

| | | | | |
|---|---|---|---|---|
| RAM 4 | 1 | 1 | 0 | 0000000 |
| | 1 | 1 | 1 | 1111111 |

Range → 180 — 1FF




Address bus = 10 bits

9–0

Data bus = 8

9th bit = 0 ⇒ RAM

= 1 ⇒ ROM

512 × 8

512 × 8

1028 × 8

for ROM

9 8–0

min – 1 0 0000000

2 0 0

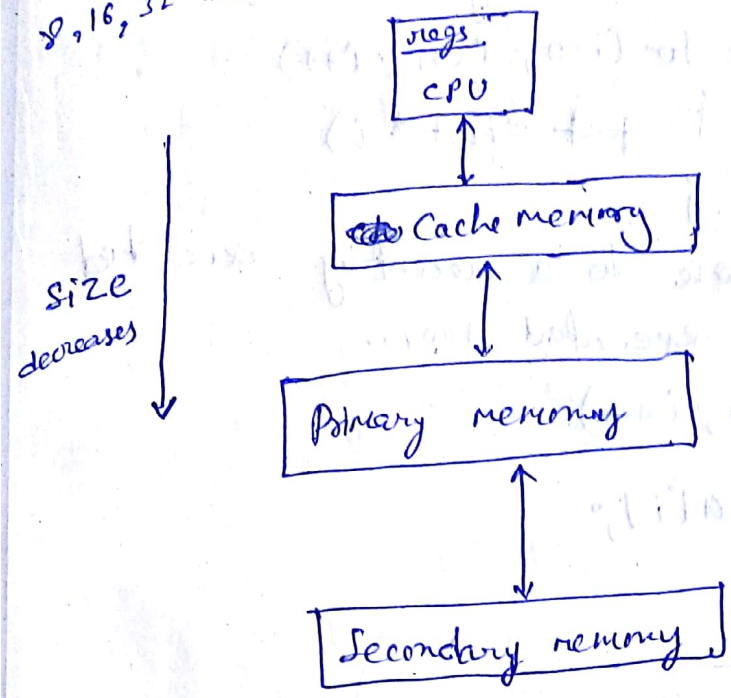
max 1 1 1111111

3 F F

Range 200 – 3FF

22/2 Memory Hierarchy

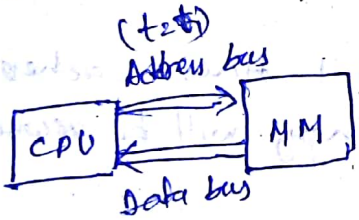
8, 16, 32 bits




size of regs < size of cache

speed → access time

speed of regs > speed of cache > speed of PM > speed of SM




Access time = $t_2 - t_1$

-> Cache memory: stores the portion of program & data which is frequently needed by CPU.

=> Locality of reference property

Analysis of a large nos. of typical programs has shown that references to memory within fixed interval of time tend to be confined within a few localized area of memory. This property is called locality of reference property.

(*) Temporal locality

Recently executed inst$^n$ are likely to be executed again very soon.

```
for (i=0; i<n; i++)
{ fact = fact * i)
}
```

(*) Spatial locality

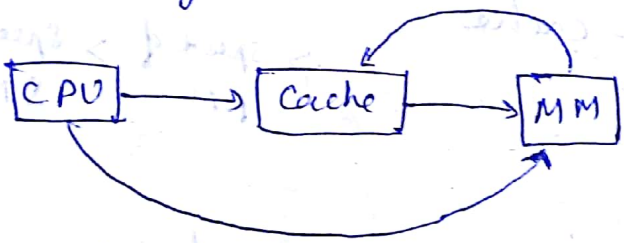
Inst$^n$ residing close to a recently executed inst$^n$ are likely to be executed soon.

ex. `for (i=0, i<n, i++)`

`S+ = a[i];`

→ Basic operation of cache :-

CPU generates some address




After finding data in MM, it transfers to cache.

only 1 times MM will be accessed

n-1 times cache memory will be accessed

→ Data is in cache memory → called Hit

→ If not → called miss

(*) Hit ratio (HR)

$$= \frac{\text{nos. of hits}}{\text{nos. of hits + nos. of missed}}$$

(*) Avg. memory access time

$$= (HR \times \text{cache access time}) + (\text{miss rate} \times \text{miss penalty})$$

$$(1 - HR)$$

miss penalty = cache access time + MM access time

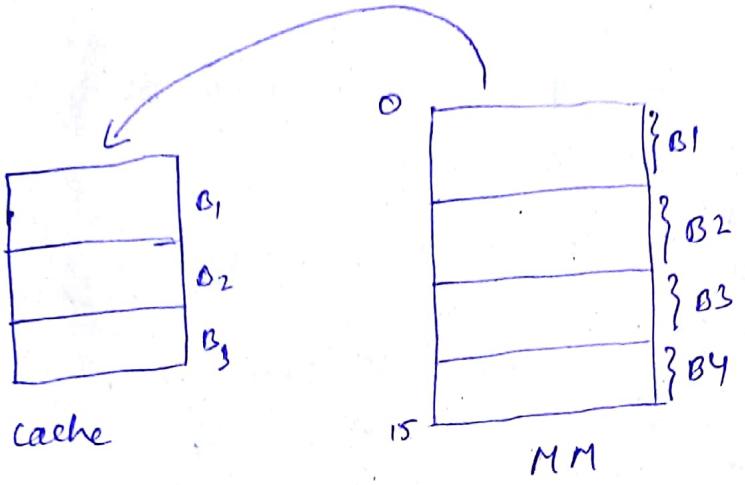
our cache access time = 100 ns
MM " " = 1000 ns

H.R = 0.9

Soln

26/2

(2) (a) Cache mapping




size of MM = $2^4 = 16$

" " blocks = $4 = 2^2$

No. of blocks = $\frac{2^4}{2^2} = 4$

No. of blocks in cache = C

" " " " MM = M

$C << M$

(b) Mapping Techniques

1) Direct mapping :

$$a = b \bmod m$$

a = cache block nos. m = nos. of blocks in cache memory.

b = MM " "

Ex. Cache size = 2K bytes

MM " = 64K bytes

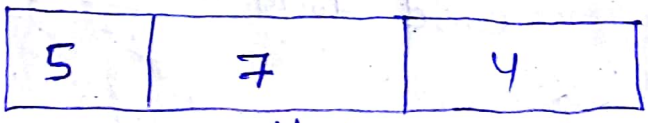
block " = 16 bytes

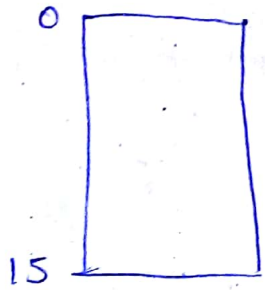
Sol. No. of blocks in cache = $\frac{2 \times 2^{10}}{2^4} = 2^7$

No. of blocks in MM = $\frac{2^6 \times 2^{10}}{2^4} = 2^{12}$

| M.M. block | Cache block |
|---|---|
| 0 | 0 |
| 1 | 1 |
| ⋮ | ⋮ |
| 127 | 127 |
| 128 | 0 |
| 129 | 1 |
| ⋮ | ⋮ |
| 255 | 127 |
| 256 | 0 |

Sol.




16 bits

↓

size of address bus (processor ex)

ii) Associative Mapping:

Any block from MM can be mapped to any block of cache memory.

Sol: cache size = 2 K bytes

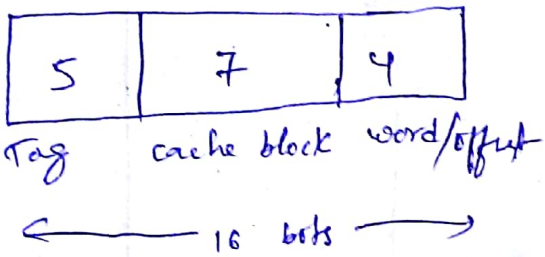
MM size = 64 kB

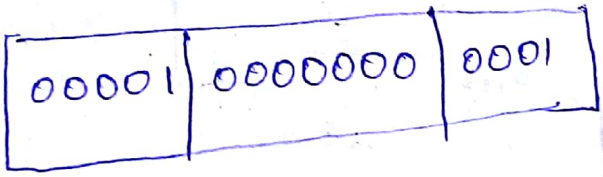
block size = 16 bytes = $2^4$

nos. of blocks in cache = $\frac{2K}{16} = 2^7 = 128$

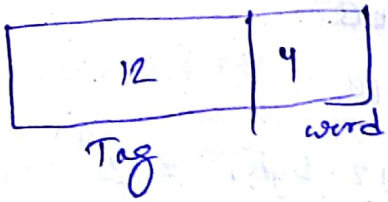
size of MM address bus = 16 bits

(a) direct




(b) Associative




- Nos. of comparisions in case of direct = 1
  " " " " associative = 128 (in this case)
  " " " (comparisions will be 11 of)

drawback ⇒ (cost will increase)

iii) Set Associative mapping :

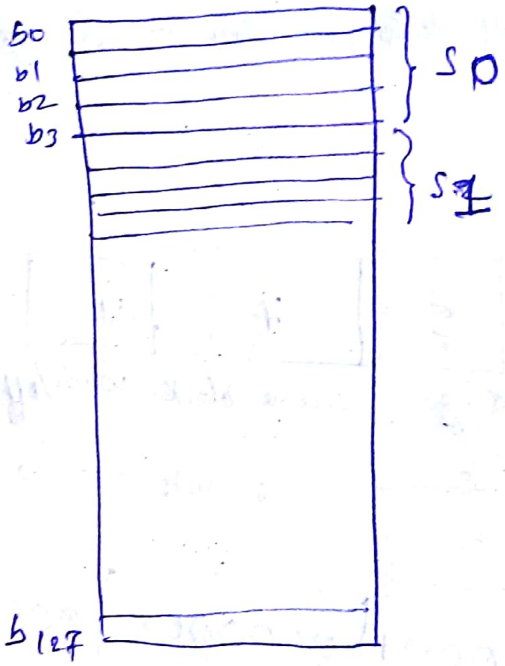
cache memory is divided into some sets and each set contains some blocks.

K-way set associative —

Each set contains k nos. of blocks.

Ex =

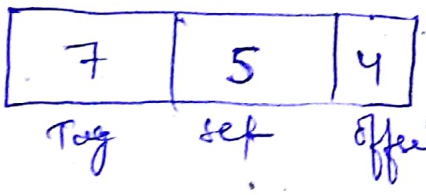
4-way




No. of sets = $\frac{128}{4}$

= 32

No. of comparisons = 4




Ex =

Cache size = 32 kB

MM size = 16 MB

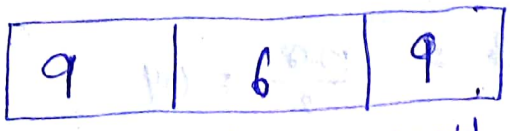
block size = 512 bytes = $2^9$

Show that partition of MM address into tag, index, offset

nos. of blocks in cache $= \frac{2^{15}}{2^9} = 2^6$

size of address bus = 24 bits

direct →

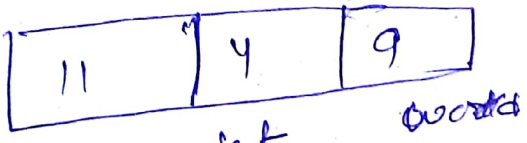



Tag. cache block word

← 24 bits →

Set associative → 4-way

nos. of sets $= \frac{64}{4} = 16$




Tag set words

Q3 Consider a direct map cache with 64 blocks and block size = 16 words. Find out cache block nos. which may contain the main memory address 1200

Sol$^n$ 1200 ⟹ MM block $= \frac{1200}{16} = 75$

$a = b \bmod m$

$\Rightarrow a = 75 \bmod 64$

$= 11$

Qu consider two way set associative cache with 128 block & block size = 16 b words. Find out the set which will contain MM address 1500.

Sol no. of cache set = $\frac{128}{2} = 64$

no. of MM blocks = $\frac{1500}{16} = 93.75 = 93$ blocks

$a = b \bmod S$

set no = $93 \bmod 64$

$= 29$

14/3

$T_{CA}$ → Cache Access time

$T_{MS}$ → MM " "

⇒ Write back

Miss penalty

Read :- $T_{CA} + T_{MS} + P_d T_{MS}$

↳ Prob. that flag is set

Avg. memory access time for read $= H \times T_{CA} + (1-H)\{T_{CA} + (1+P_d) T_{MS}\}$

write :- $2T_{CA} + T_{MS} + P_d T_{MS}$

Avg. memory access time for write $= H \times T_{CA} + (1-H)\{2T_{CA} + (1+P_d) T_{MS}\}$

Ques: Cache access time = 90 ns

MM " " = 500 ns

80% of the memory request are for read operation and 20% for write operation. H.R. for read is 0.9, H.R. for write is 0.8. Compute i) avg. memory access time for read operation for both write back and write through.

ii) Avg. memory access time for both read and write for both write back & write through.

($P_d = 0.5$)

Sol i) For write back -

$$T_A = 0.9 \times 90 + 0.1\,(90 + (1+0.5)\,500)$$
$$= 0.9 \times 90 + 0.1\,(90 + 1.5 \times 500)$$
$$= 81 + 84 = 165 \text{ ns}$$

For write through –

$T_{\pi} = H \times T_{CA} + (1-H)(T_{CA} + T_{MS})$

$= 0.9 \times 90 + 0.1 (90 + 500)$

$= 81 + 59 = 140$ ns

ii) For write back –

$T_w = 0.8 \times 90 + (1-0.8)(2 \times 90 + (1+0.5)500)$

$= 72 + 0.2 (180 + 750)$

$= 72 + 186 = 258$ ns

$T_{avg.} = P_{\pi} \times T_{\pi} + P_w T_w$

$= 0.8 \times 165 + 0.2 \times 258$

$= 183.6$

$P_{\pi}$ → Probability of read

$P_w$ → Probability of write.

For write through –

$T_w = T_{MS}$

$= 500$ ns

(whether miss occurs or not MM & CM will be updated parallely)

$T_{avg} = 0.8 \times 140 + 0.2 \times 500$

$= 112 + 100 = 212$ ns

H.W.: i) Find out avg. access time for memory read operation

ii) Both read & write

→ only write through

$T_{CA} = 50\,ns$ , $T_{MA} = 50\,ns$

$P_H = 0.8$ , $H_H = 0.9$

## ⑧ Cache Replacement Algorithm

→ In case of MISS, if the cache memory is already full, to write in cache memory, cache replacement algo is required to choose the block to be overwritten

| | |
|---|---|
| $b_0$ | 4 |
| $b_1$ | 5 |
| $b_2$ | 10 |
| $b_3$ | 12 |
| $b_4$ | 15 |

→ only for associative & set-associative.

→ 3 types –

i) FIFO → Replace the block, which has the max. timer value.

ii) LRU → Least recently used

iii) Random

Ex: M.M. block

[ 5, 4, 2, 3, 5, 1, 5, 10, 3

Note: When cache is empty & M.M. is called for first time, Miss is known as compulsory MISS.

i) FIFO →

| 5 | 4 | 2 | 3 | 5 | 1 | 5 | 10 | 3 |
|---|---|---|---|---|---|---|---|---|
| M | M | M | M | H | M | M | M | H |
| | | | | | 5 | 4 | 2 | |

ii) LRU →

| 5 | 4 | 2 | 3 | 5 | 1 | 5 | 10 | 3 |
|---|---|---|---|---|---|---|---|---|
| M | M | M | M | H | M | H | M | H |
| | | | | | 4 | | 2 | |

(*) Types of cache misses —

I) Compulsory

II) Capacity → When program requires more blocks frequently than the capacity of cache.

III) Conflict → direct mapping
More than one block is mapped to the same block.

→ set associative
In 4 way, more than 4 blocks are used frequently which map to the same set.

(*) Techniques to reduce cache misses —

(1) Increasing cache size
(2) Increasing block size
(3) Higher associativity
2 way "
8 way "

(4) Compiler optimization

~~loop interchange~~

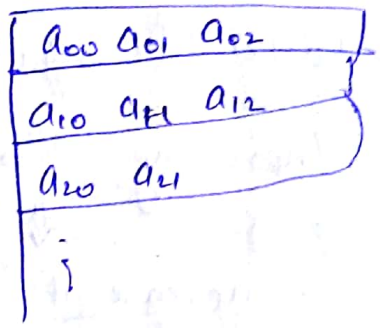
i) loop interchange

```
for (j=0 ; j<1000 ; j++)
    for (i=0 ; i<1000 ; i++)
        a[i][j] = 2 * a[i][j];
```

a[0][0]
a[1][0]
.
.

if instead of accessing block by block data, we access all the data of a block and then next block, then miss will be reduced. For this we have to interchange the loop

```
for (i=0, i<1000, i++)
    for (j=0; j<1000, j++)
        a[i][j] = 2 * a[i][j];
```




ii) Loop fusion

```
for (i=0; i<1000, i++)
    for (j=0, j<100, j++)
        a[i][j] = b[i][j] + c[i][j]
        a[i][j] = b[i][j] * 2;
```

complete all the operation related to a[i][j] and then move to the next i or j.

iii) Merging arrays.

combining two separate arrays (that might conflict for a block in cache memory) into one separate array.

Ex:-

```
int val[10];  → b0
int key[10];  → b16

struct merge
{ int val
  int key
} [0];
```

15/5

## ④ Memory Expansion

### i) Vertical expansion –

Increasing no. of memory location.

⇓

Increase in size of address bus.

### ii) Horizontal expansion –

Increasing word length

⇓

Increase in size of data bus.

Ques. i) How many 256 x 8 RAM chips are needed for the memory of 1K x 16

Soln.

$$\text{no. of chips} = \frac{1K \times 16}{256 \times 8} = \frac{2^{10} \times 2^4}{2^8 \times 2^3}$$

$$= (4) \times (2)$$

(4) → vertical

(2) → horizontal

ii) How many lines of address bus must be used to access 1K x 16 of memory.

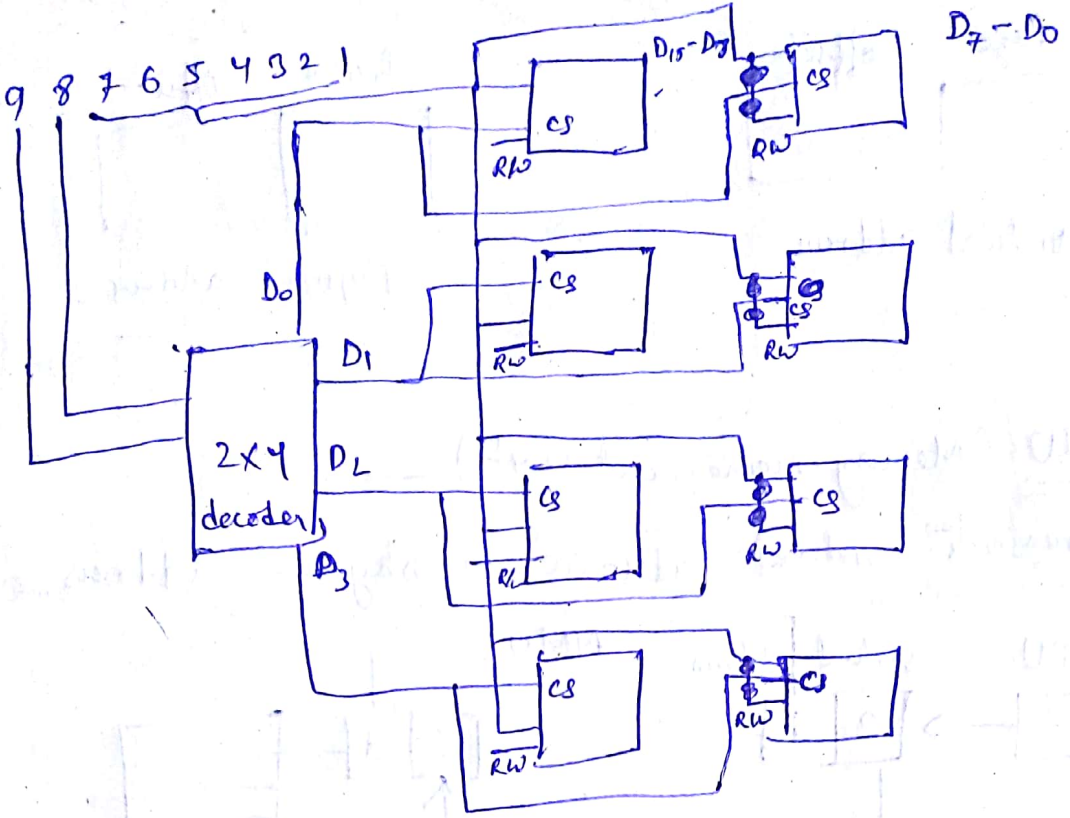
Ans – 10

iii) How many of this lines will be common for all chips.

Ans – 8

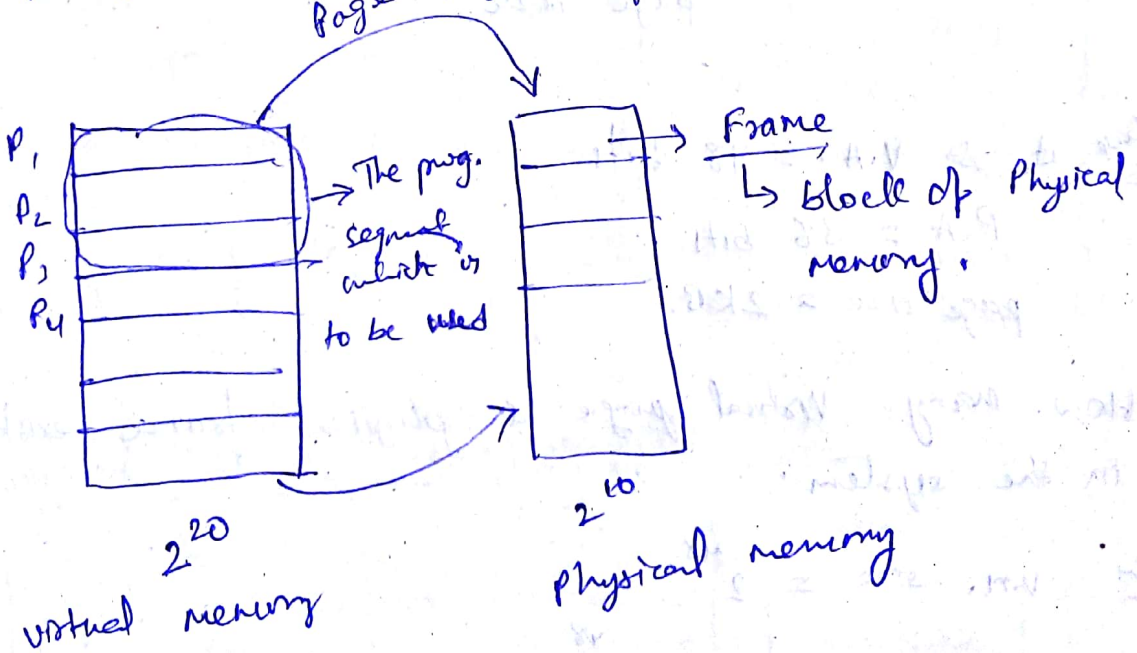
iv) Specify the size of decoder.

Ans – 2 x 4



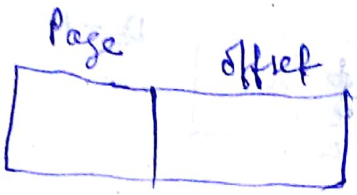

## (*) Virtual Memory

Virtual memory is a concept that allow the ~~exec~~ execution of a program that may not be completely in MM.

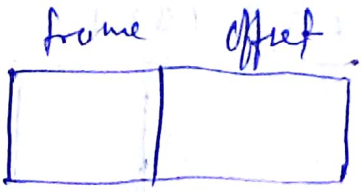



→ Address for virtual ~~address~~ memory → virtual address

→ Verbal memory is also divided into some blocks, called "page".



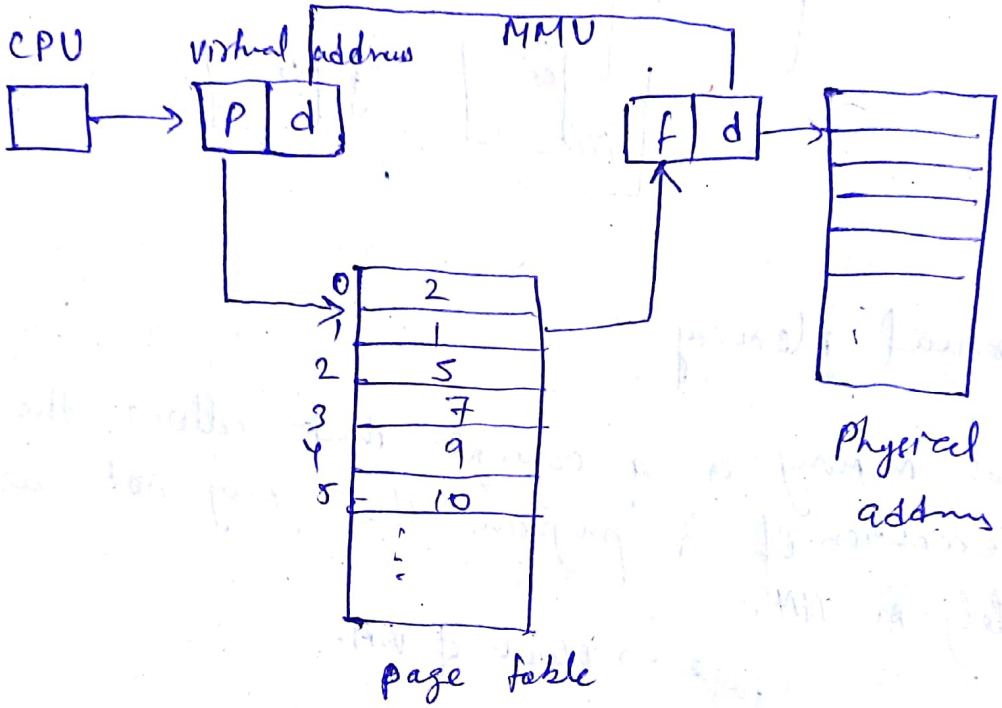

Virtual address




Physical address

(4) MMU (Memory management unit) —

Transmits virtual address to physical address.




Ques V.A. = 48 bits

P.A = 36 bits

page size = 2KB

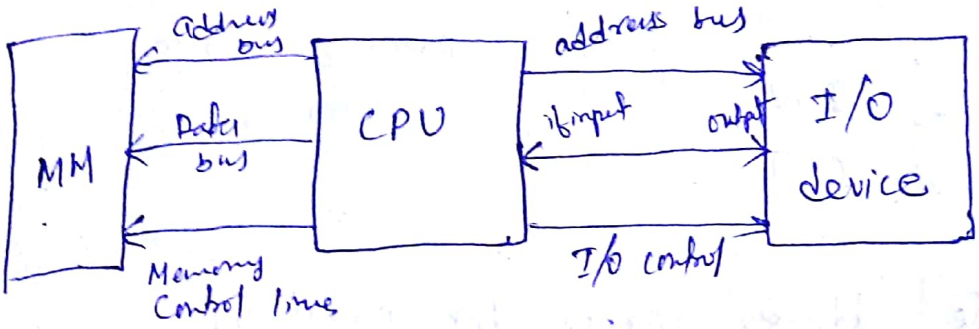
How many virtual page & physical frame exist in the system.

Sol V.M. size = $2^{48}$

No. of pages = $\frac{2^{48}}{2KB} = \frac{2^{48}}{2^{11}} = 2^{37}$

No. of frame = $\frac{2^{36}}{2^{11}} = 2^{25}$

19/3 Input and Output device

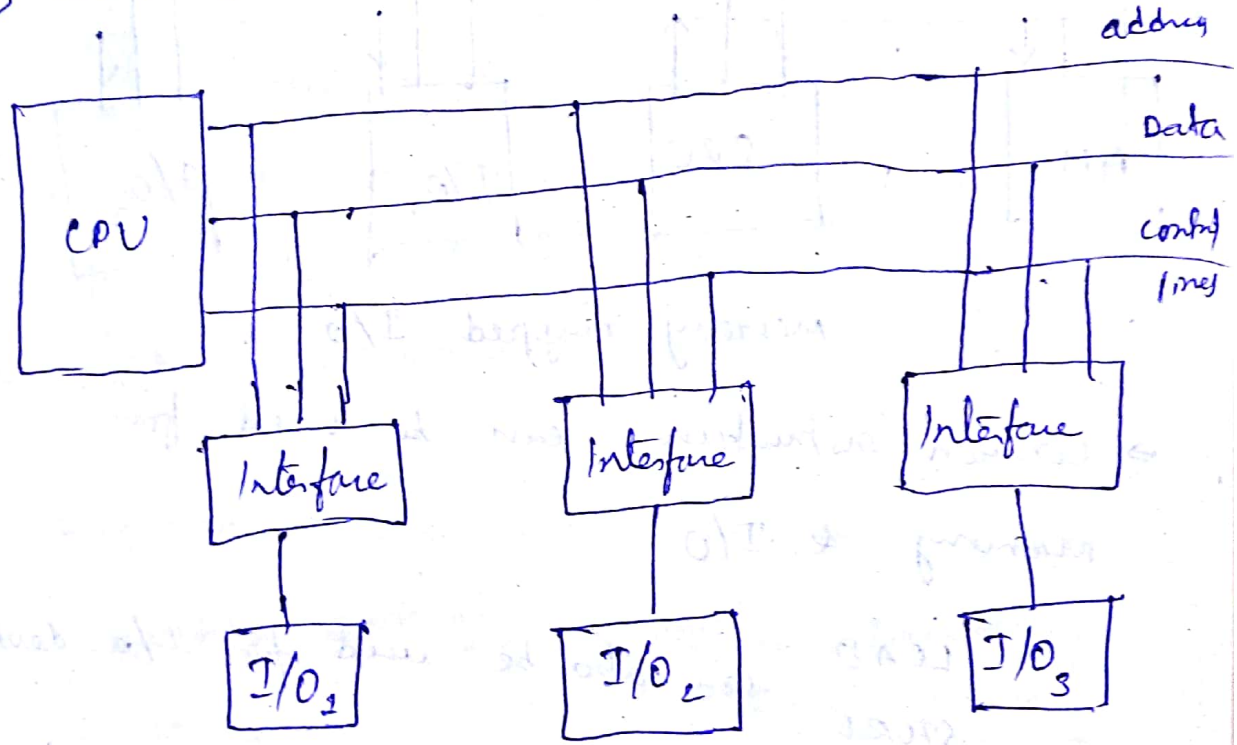



Difference b/w CPU & I/O device

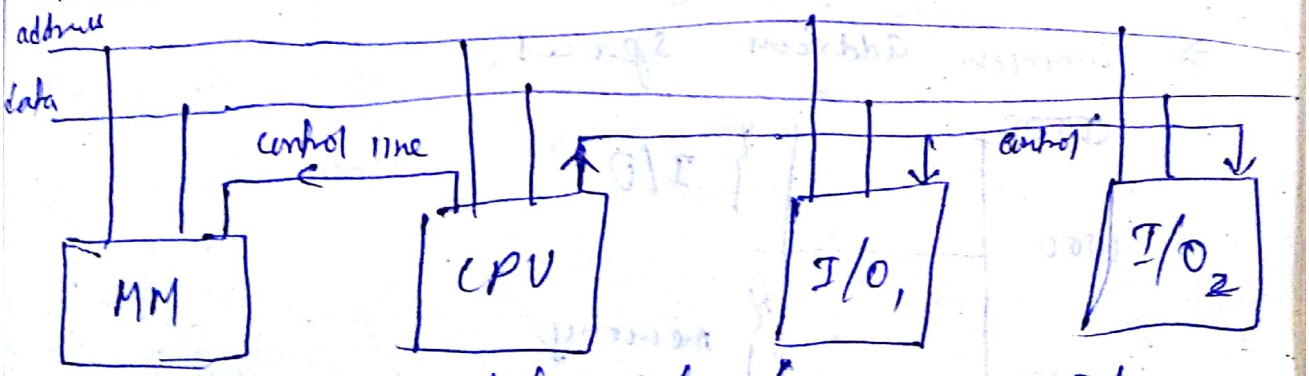
i) I/O electro-mechanical device, CPU is electronics device.

ii) Rate of data transfer (CPU is faster)

iv) Data format




⇒ Isolated I/O Vs memory mapped I/O



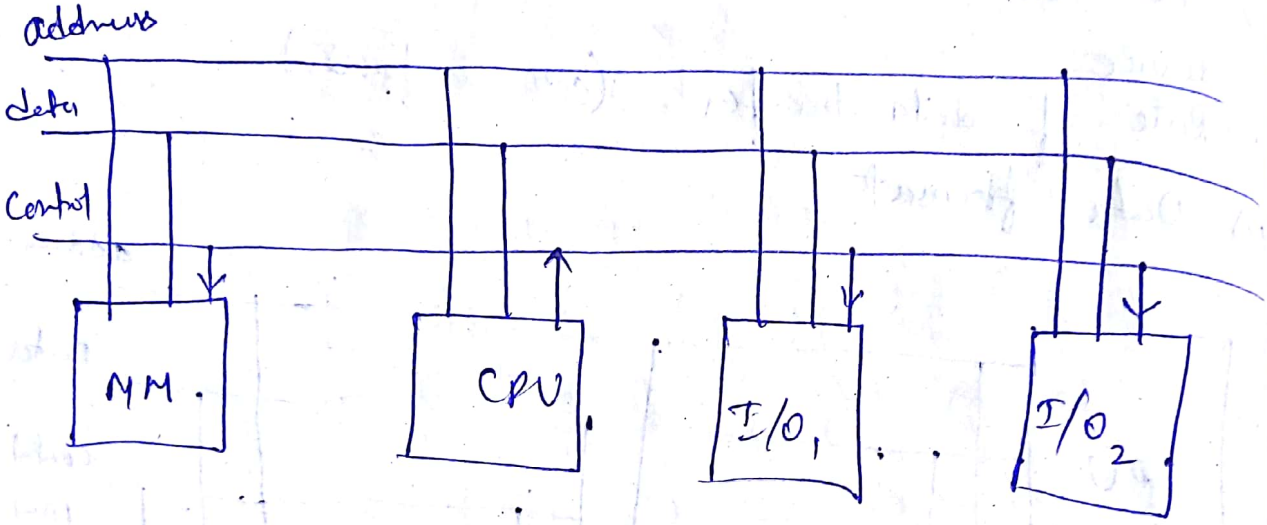

Isolated I/O / I/O mapped I/O

→ Memory ⟵ LOAD
STORE

memory read
memory write

I/O → IN
OUT

→ Separate address spaces for memory and I/O



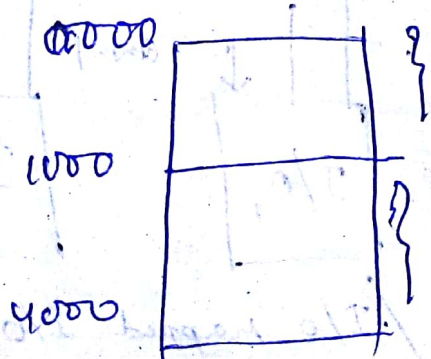

Memory mapped I/O

→ common instruction can be used for memory & I/O

LOAD
STORE
can also be used for I/O device

→ Common control line

→ common address spaces



ee LOAD 500 → for I/O

LOAD 1500 → Memory

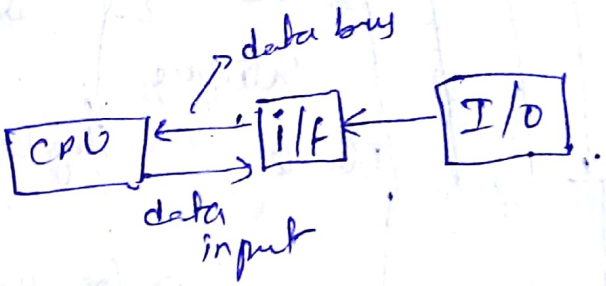
## ⊛ I/O command

i) control

ii) status

iii) Data input

iv) Data output




21/3

Input output device → peripheral device

## ⊛ Modes of data transfer –

I) Programmed I/O

II) Interrupt initiated I/O

III) Direct memory Accew (DMA)

## I. Programmed I/O

### ⊛ Handshaking protocol –

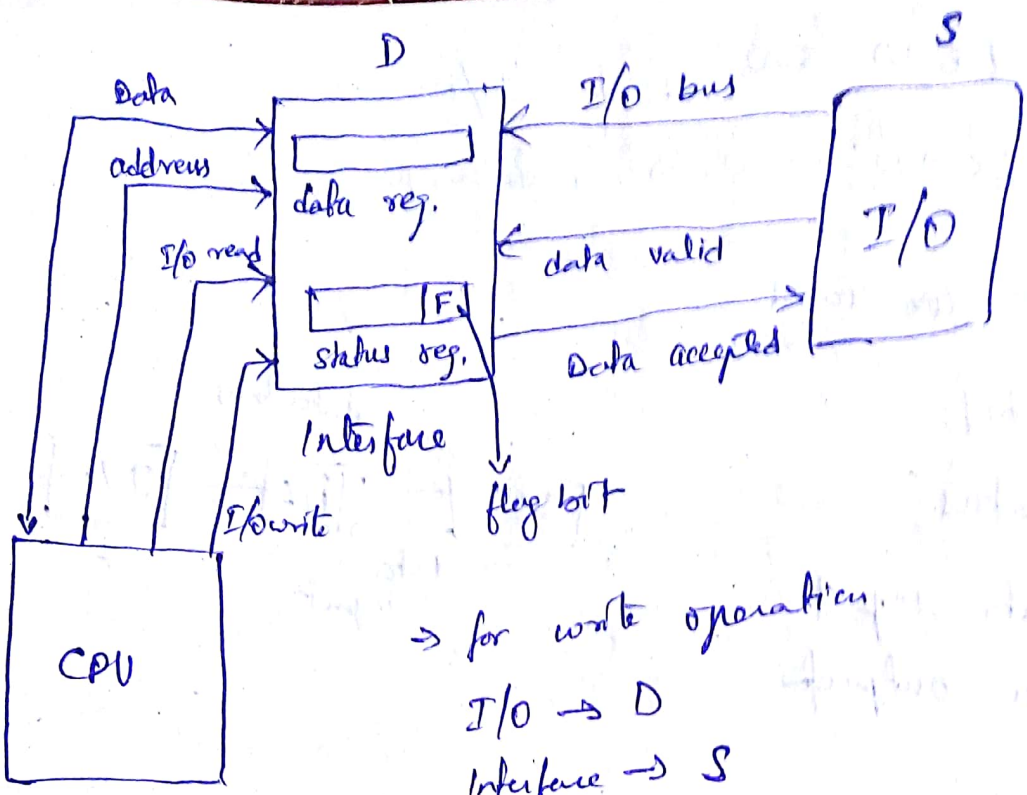
i) Data in data bus

ii) Enables data valid line

iii) Enables data accepted by D

iv) disables data valid lines

v) disables data accepted line




→ for write operation.

I/O → D

Interface → S

F = 1 → means data present in the data reg.

F = 0 → No data in data reg. / I/O device is not ready

If F = 1, CPU accepts data from data reg.

after that ~~data~~ F will be reset.

Steps - (for write operation)

i) Placing address in address bus by CPU

ii) ~~Data~~ Data in I/O bus → data reg.

iii) F = 1, ~~Data valid line~~

iv) Data accepted line enabled by interface

v) CPU checks the flag bit

F = 1 → ready for data transfer

vi) Data ~~flag~~ from data reg. to data bus

vii) Data accepted line disable by interface.

viii) Next set of data.

* Drawbacks of programmed I/O —

i) CPU has to check for flag bit (0 or 1) continuously.

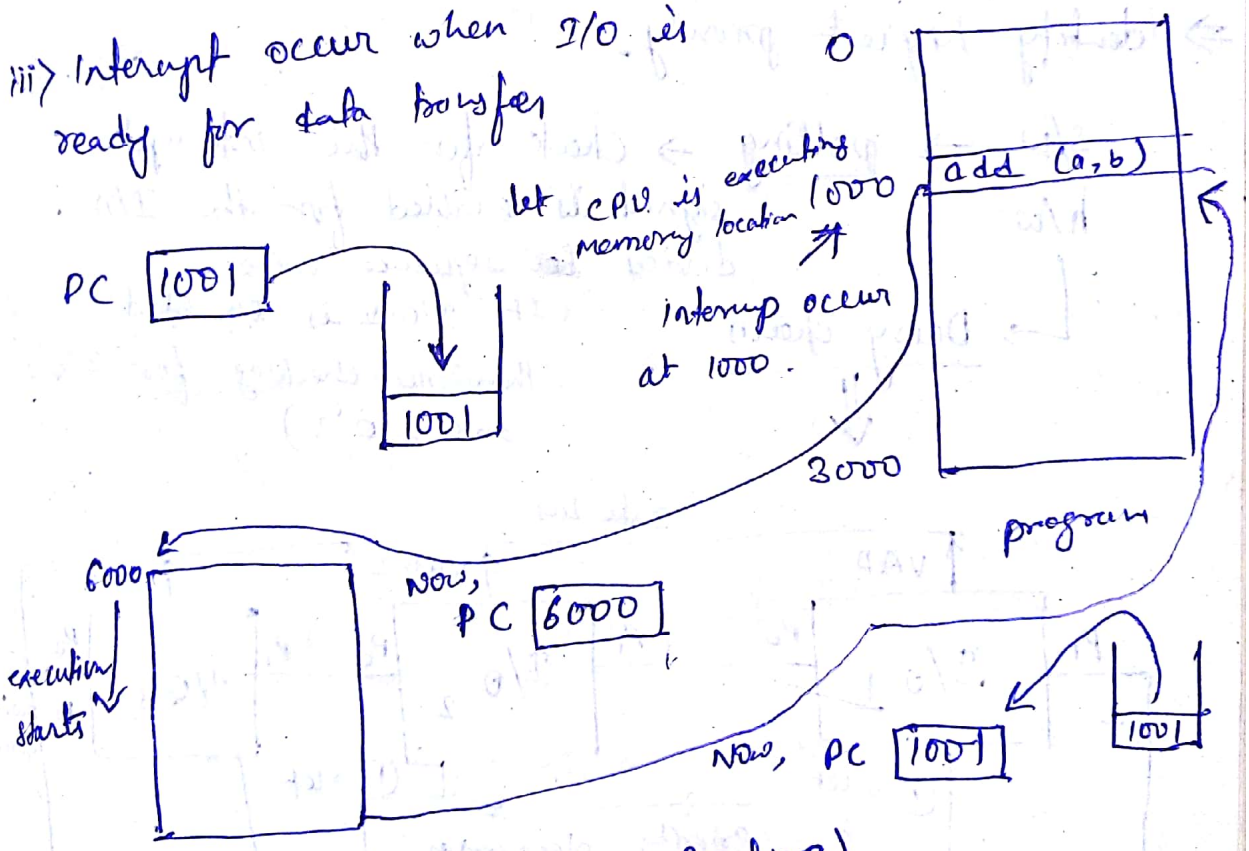
## II Interupt initiated I/O

steps

i) First CPU initiate the I/O operation by placing address of I/O device in address bus.

ii) Never checks the flag bit, continue its normal execution.

iii) Interupt occur when I/O is ready for data transfer




ISR (Interupt service Routine)

↳ program that store what to do when interup occur.

⇒ Ways to chose the address of ISR by CPU -

i) Non - Vectored interupt → Fixed location in memory

ii) Vectored interupt → I/O device gives the address of ISR

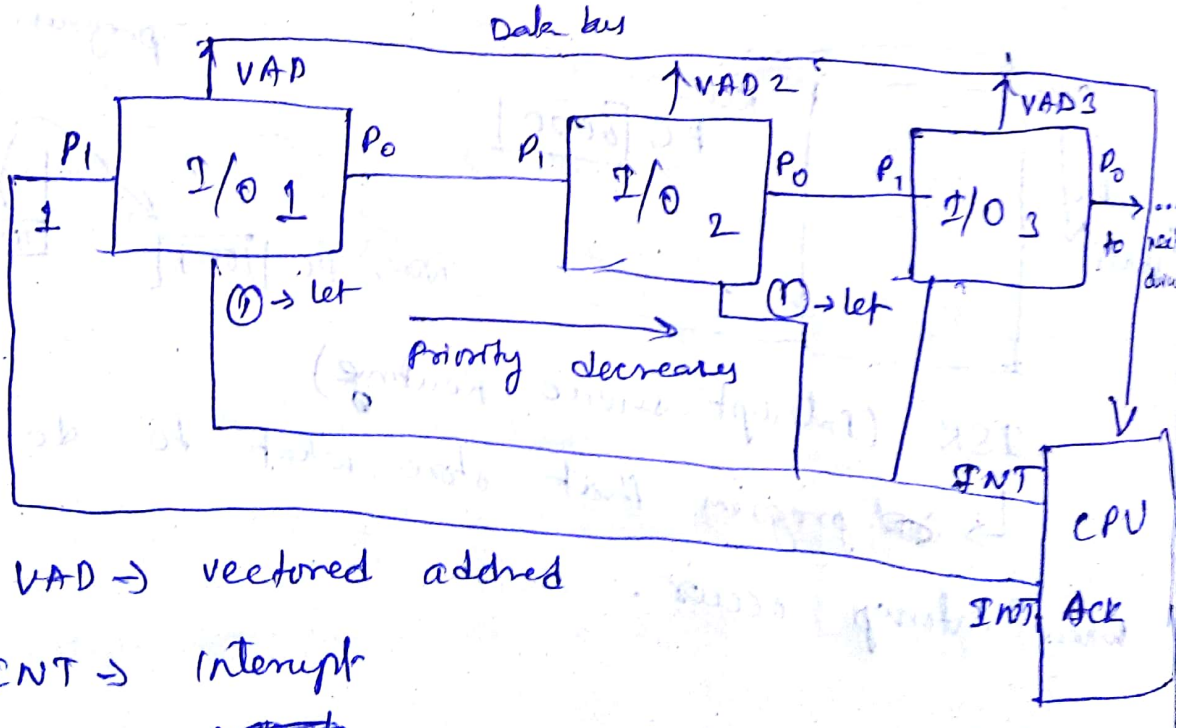
④ Priority Interupt -

I/O 1 , I/O 3 → at some time interupt the CPU, then the I/O having higher priority will be serviced first

⇒ Identify highest priority -

s/w → polling → check for the interupt signal is enabled for the I/O devices ~~too~~ sequence wise.
(IF I/O 1 is enabled then no checking for I/O 2 and I/O 3)

h/w
↳ Daisy chain




VAD → vectored address

INT → interupt

ACK → ~~Acknowledge~~ acknowledgement

PI = 1 for I/O 1
PO = 0
↳ blocks the signal (if Interrupt signal is high) → for I/O 1.

If I/O 1 is not ready for data transfer

for I/O 1
PI = 1
PO = 1 → passes the signal to next I/O

If I/O 2 is ready

∴ for I/O 2
PI = 1
PO = 0

④ Drawbacks

Programmed I/O & Interrupt I/O both required CPU interaction.

22/3

III DMA controller (Memory Map I/O)

Add. reg.

(i) starting address of block of data to be read or written in memory.

(ii) No. of words to be transfered.

previous

$A_1 = 3 = k\,tp$

$A_2 = k\,tp + tp = 4$

$A_3 = \underbrace{k\,tp}_{A_1} + \underbrace{tp}_{A_2} + \underbrace{tp}_{A_3} = 5$

$$SU = \frac{n\,k\,tp}{k\,tp + (n-1)\,tp} = \frac{nk}{\underbrace{k+n-1}_{n}} = \frac{nk}{n} = k$$

for $n >> k-1$

→ You have large no. of operation —

$n \to \infty$

$n >> k-1$

**Speed up $\simeq k$**

Ex: $t_p = 20\,ns$

$k = 4$

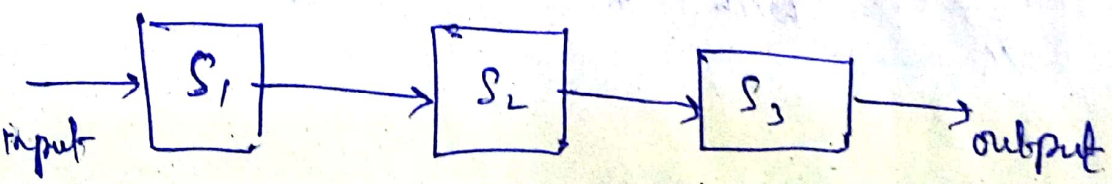
$n = 100$

$$SU = \frac{n\,k\,tp}{k\,tp + (n-1)\,tp} = \frac{nk}{k+n-1} = \frac{100 \times 4}{4 + 100 - 1}$$

$$= \frac{400}{103} = 3.88$$

(R) → Max. speedup = $k$

26/3 pipeline




S → segments.

## ② Data pipeline :

$A_i * B_i + C_i$

$i=1$ $A_1 * B_1 + C_1$

$i=2$ $A_2 * B_2 + C_2$

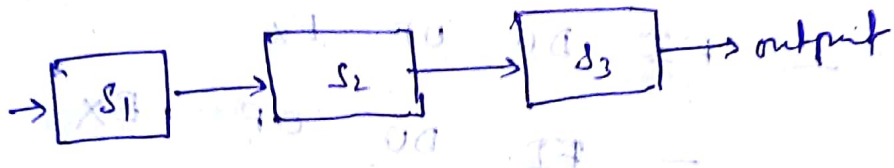
⋮

$i=7$ $A_7 * B_7 + C_7$

$S_1$ : $R_1 \leftarrow A_i$ , $R_2 \leftarrow B_i$

$S_2$ : $R_3 \leftarrow R_1 * R_2$ $R_4 \leftarrow C_i$

$S_3$ : $R_5 \leftarrow R_3 + R_4$




| clock | $S_1$ | | $S_2$ | | $S_3$ |
|---|---|---|---|---|---|
| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
| 1 | $A_1$ | $B_1$ | | | |
| 2 | $A_2$ | $B_2$ | $A_1 * B_1$ | $C_1$ | |
| 3 | $A_3$ | $B_3$ | $A_2 * B_2$ | $C_2$ | $A_1 * B_1 + C_1$ |
| 4 | $A_4$ | $B_4$ | $A_3 * B_3$ | $C_3$ | $A_2 * B_2 + C_2$ |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | | | | $A_7 * B_7 + C_7$ |

Total Cycle = 8

## (iii) Instruction pipeline :

→ Instruction execution cycle

I) Fetch the instruction — FI

II) Decode " " — DE

III) Fetch the operands — OF

IV) Execute the instruction — EX

| Instⁿ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $I_1$ | FI | DO | OF | EX | | | |
| $I_2$ | — | FI | DO | OF | EX | | |
| $I_2$ | — | — | FI | DO | OF | EX | |
| $I_3$ | — | — | — | FI | DO | OF | EX |

without pipeline

$I_1 = 4$ , $I_2 = 4+4$ , $I_3 = 4+4+4$

$I_4 = 16$

with pipeline

$I_4 = 7$

Speed up = $\frac{16}{7}$

## ⑧ Pipeline hazards

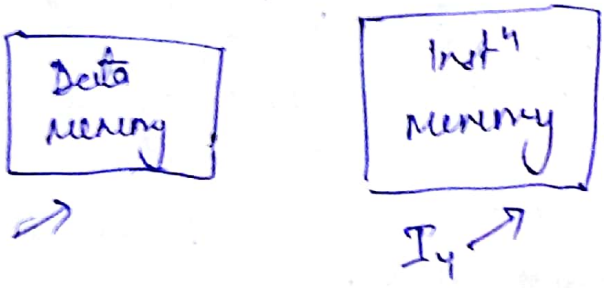
### i) structural hazards (Resource conflict)

- when more than one inst$^n$ want to use the same resource at the same time

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $I_1$ | FI | DO | OF | EX → writing in memory | | | |
| $I_2$ | | FI | DO | OF | EX | | |
| $I_3$ | | | FI | DO | OF | EX | |
| $I_4$ | | | | FI ↓ reading from memory | DO | OF | EX |

$I_1$ & $I_4$ both are using memory at CP-4 for writing & reading respt.

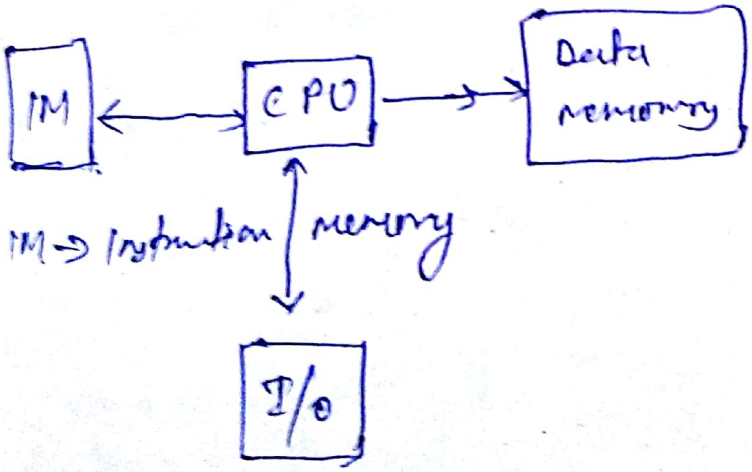
To resolve this




two parts of memory.

⇓

Harvard Architecture



Solution :- ① Delayed Load : (H/w)

Delay loading conflicting operator.

NOP → No operation will be performed, just increase the clock cycle.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| I | FI | DO | OP | EX | | |
| J | NOP | FI | DO | OP | EX | |

$R_1 = 15$ $R_y = 10$

1 CLK delay

② S/w → compiler optimization

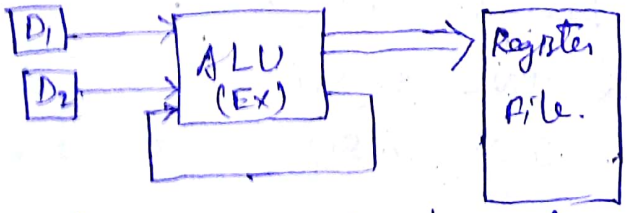
Compiler detect data hazard → rearrange the instructions

$R_1 \leftarrow R_2 + R_3$
$R_4 \leftarrow R_1 * 4$ } RAW ⇒
$R_5 \leftarrow R_6 + R_3$

I : $R_1 \leftarrow R_2 + R_3$
J : $R_5 \leftarrow R_6 + R_3$
K : $R_4 \leftarrow R_1 * 4$

→ independent instn will be inserted b/w.
→ No extra clk pulse is required.

③ Operand Forwarding (H/w)




EX will be performed in ALU

Hardware detect the data hazard → Instead of storing the value in $R_1$, ALU bypass the value
No extra clk pulse is required.

II) WAW :

I : $R_1 \leftarrow R_2 + R_3$

J : $R_2 \leftarrow R_5 * 3$

hazard will occur if J writes ($R_2$) before I reads ($R_2$).

III WAW :

I : $R_1 \leftarrow R_2 + R_3$

J : $R_1 \leftarrow R_4 - R_5$

K : $R_6 \leftarrow R_1 + 10$

Hazard will occur if J writes the variable ($R_1$) before I writes the variable ($R_1$)

## Control hazard :

2000 : 1 : $R_1 \leftarrow M[1000]$

2001 : 2 : JMP 2050 → 2050 (PC)

2002 : 3 : $R_4 \leftarrow R_2 + R_3$

2050 : $R_5 \leftarrow R_6 * 4$

i) Unconditional branch Inst$^n$ → ex: JMP 2050

ii) conditional branch Inst$^n$ →

condition check ⇓ Ex: JNC 2050, JC 2050

At the end of 5th CLK

PC 2050

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $I_1$ | FI | DO | OF | EX | | |
| $I_2$ | | FI | DO | OF | EX | |
| $I_3$ & | | | FI | – | – | FI |

FI from 2002 (Normally)

FI for 2050 (in case of branch)

solution:

(1) Delayed branch:

Insert B the NOP. (6-8) delay

(2) Branch Target Buffer

| Address of branch | Target address |
|---|---|
| 3000 | 2050 |
| 2001 | 2000 |

BTB

Inst$^n$:-

3000: JMP 2050

Ex: 2000 Add A,B → loop

2001: JMP 2000

No need for the gate delay, if the instruction address matches the BTB address of branch then it will start the target instruction execution.

(3) Branch Prediction:

JNC 3000 <P>

P = 1 → goto 3000

P = 0 → next inst$^n$

→ For loop inst$^n$ -

for ( i=0, i<99, i++)

C ← A+B

99 times prediction will be correct

only 1 time miss prediction will occur.

2/4/18

## RISC Architecture

→ Small nos. of instruction sets.
→ Simple instruction
→ Few addressing modes
→ Fixed length inst$^n$ format
→ Memory access is limited to load and store inst$^n$.
→ Large nos. of registers.
All operations are done within reg. of CPU.

Ex. ADD $R_1, R_2$ ← [faster because using reg.]

ADD
→ Single cycle inst$^n$ execution, efficient for pipeline.
→ Hardwired control unit.

## CISC (Complex inst$^n$ set computer)

→ Large nos. of inst$^n$.
→ Complex inst$^n$.
→ Large variety of addressing modes (5-20)
→ Variable length inst$^n$ format
→ Inst$^n$ manipulate operands in memory

$M[z] \leftarrow M[x] + M[y]$

→ Small nos. of reg.
→ Complex inst$^n$ may have multiple clock cycle
→ Micro programmed control unit.

5/4. Control signal

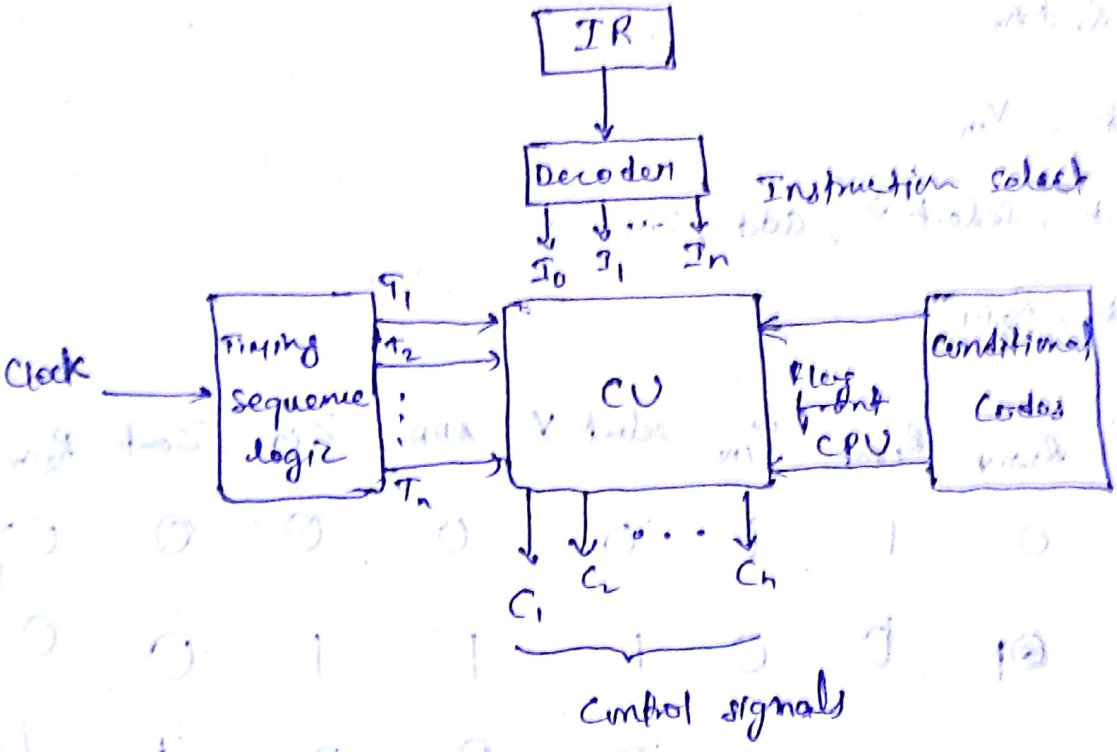
i) Hardwared CU → control signals are generated using a hardware circuit.

ii) Microprogrammed CU

↳ control signals are generated by using a programming.

I Hardwared CU :

→ Decoder, encoder, Functional logic.




Conditional codes is required for the branched instructions.

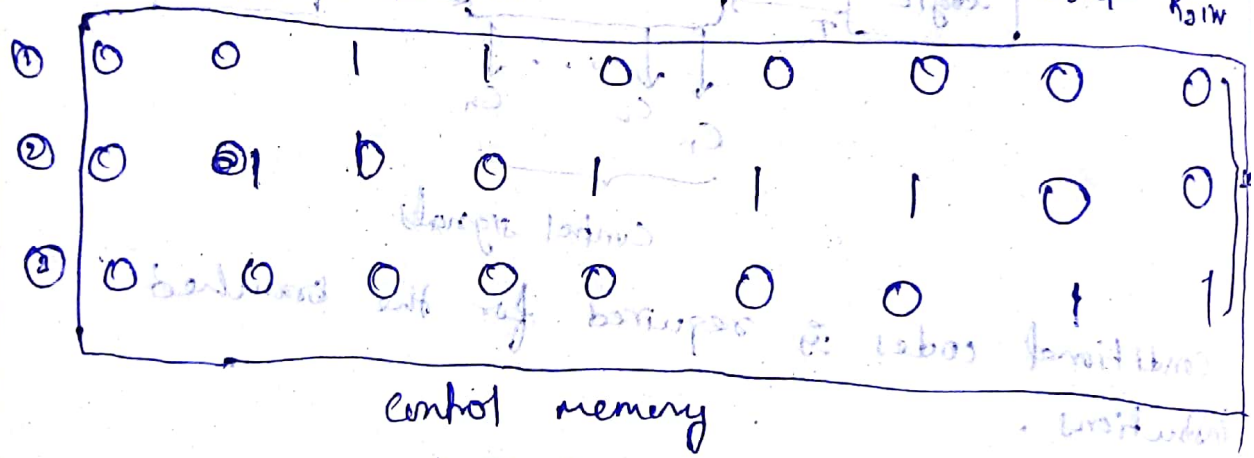
→ Fixed instⁿ sets.
→ Not flexible
→ Dificult to handle complex instⁿ.
→ Complicated design process
→ Speed is faster.
→ Complex decoding &
ex. RISC

## Microprogrammed CU

→ control signals ← programming.
→ Speed is slower
→ more flexible
→ Easier to handle complex Instⁿ.
→ Simple design process
Ex. CISC

$R_3 \leftarrow R_1 + R_2$

① $R_1$ out , $Y_{in}$
② $R_2$ out , select Y , add , $Z_{in}$
③ $Z_{out}$ , $R_3$ in

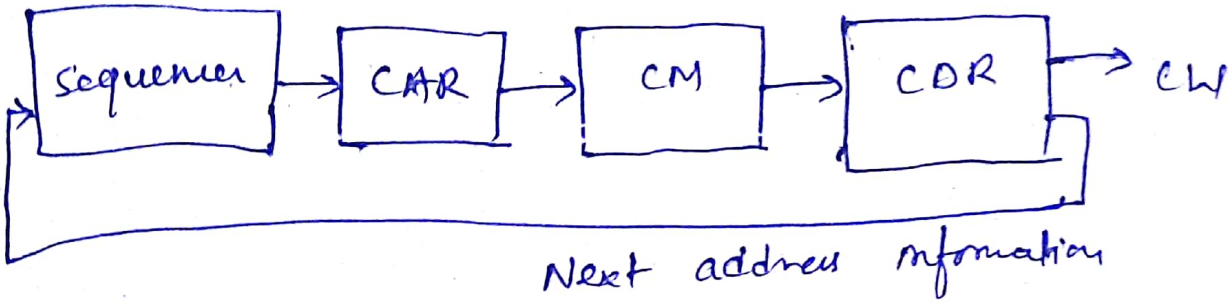



control memory
(ROM)

Micro routine → set of micro instructions that accutually forms a particular machine instⁿ.

control word → ~~contro~~ content of any memory locations of control memory.

Block diagram of microprogrammed CU —




sequencer is also called next address generator.

CAR → control address reg.
CM → control memory
COR → " data reg
CW → " word.

(i) Increment the content of CAR
(ii) Load the branch address
(iii) subroutine call and return
(iv) Mapping process