*. VON NEWMANN ARCHITECTURE :

- The variables, Constants are stored in Main Memory.
- The program / instructions are stored in Main Memory.
- In Main Memory, instruction set & data are placed separately in same Memory (Main)
- Converting Input to output involves set of components:

1. Arithmetic Logic Unit : (ALU)

→ ALU has set of circuits available in it.

→ It uses set of logical operations to perform +, −, *, Add, OR, XOR, shift, etc...

2. Registers :

→ The set of flip flops / sequence of bits make register.

→ After having Memory, we use register to increase speed of ALU.

→ Intermediate results during operation is stored in Register. So it contributes in processing.

3. Control Unit :

→ There are 2 factors → Timing Signal
→ Control Signal

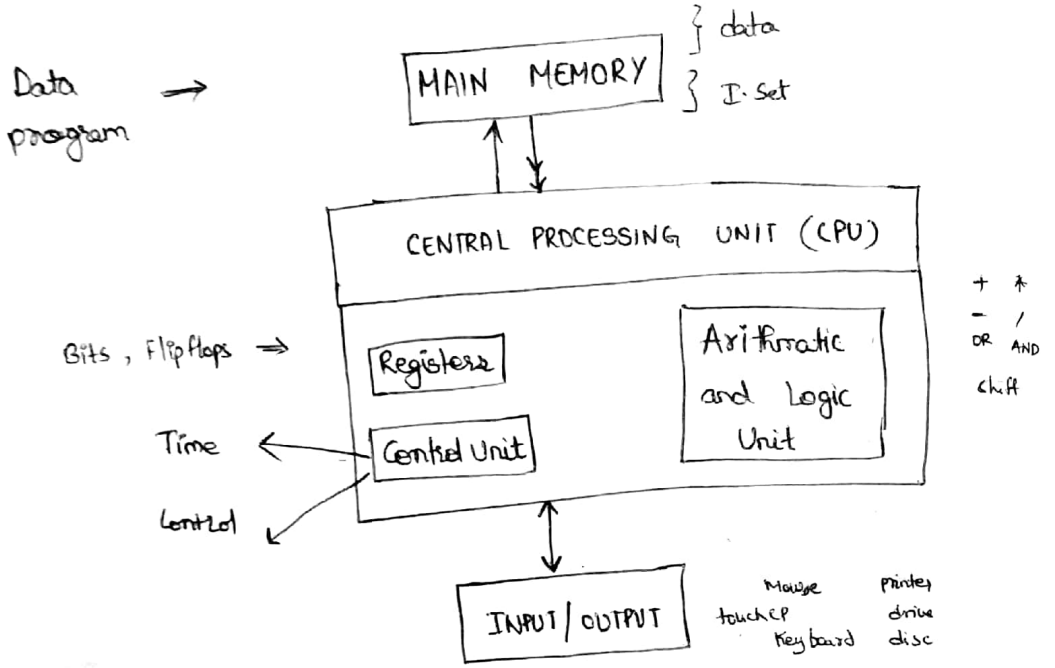
→ To execute instructions in order we use hardware called Control Unit.

→ Control Signals are used in Registers to control read, access in order.

4. Input / output systems :

→ output & Input is generated, processed through input output systems (Mouse, Keyboard, printer, pen drive, etc.)

→ Different Topologies are used to connect those Components. (BUSES).



*. Registers are connected with common busses using Multiplexed circuits.

*. TYPES OF REGISTORS :

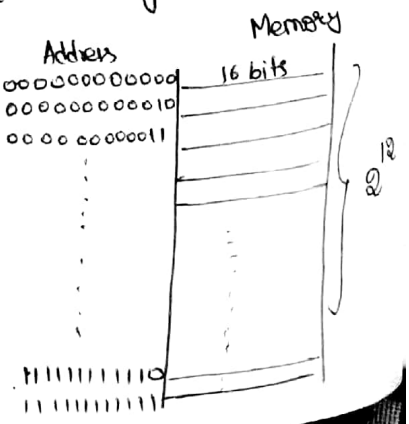
→ They are sequence of bits.

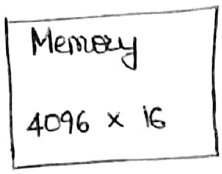
→ RAM : 4096 × 16 (assume) is Memory.

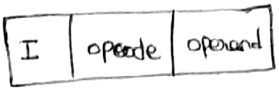
No. of words = 4096
word size = 16 bits
→ 4096 × 16

word - block (memory representation)
or byte

Here word means random bytes.







AR | Address Register

Instruction Reg | IR

DR | Data Register

Temporary Reg | TR

AC | Accumulator

Input Reg | IR

PC | program Counter

output Reg | OR

2. Address Register :

→ To take a variable from memory we need Address of that variable data.

→ This address is stored in Address Register. This is given to Memory to be fetched.

→ Size of AR = 12 bits (0:11)

Since Memory = 4096 = 2^{12}

3. Data Register :

→ Total Size of Data Register = 16 words = 16 bits (0:15)

4. Accumulator :

→ Intermediate data is stored in Accumulator.

→ After fetching memory (data), it is stored in Accumulator inorder to forward it to ALU.

→ Size of Accumulator = 16 bits = (0:15)

5. Program Counter :

→ Used to store address of next instruction.

→ General implementation & fetching ⇒ PC = PC + 1 if there is no jump conditions.

→ Size of PC = 12 bits (0:11)

→ Immediately after fetching address from Main Memory, the Program Counter increaments to next address.

6. Instruction Register :

→ Instruction = opcode + operation

→ opcode ≡ (12:15) operand ≡ (0:11) I ≡ either 0,1 (15)

→ Size of Instruction Register = 16 bits (0:15)

7. Temperory Register → Size = 16 bits (0:15)

8. Input Register :

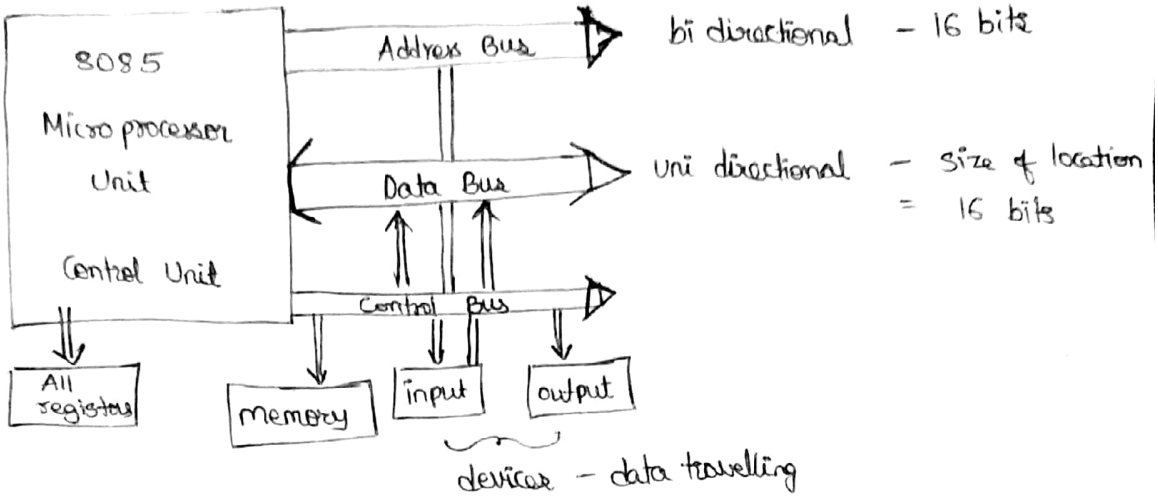
→ Data is taken from Input devices (8 bits Size) variable

9. Output Register :

→ Data is taken from ALU & given to output devices

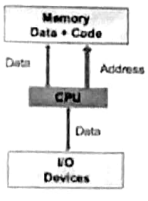
Size = (8 bits - variable).

*. TYPES OF BUSES :

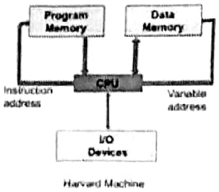


bi directional - 16 bits

uni directional - size of location = 16 bits

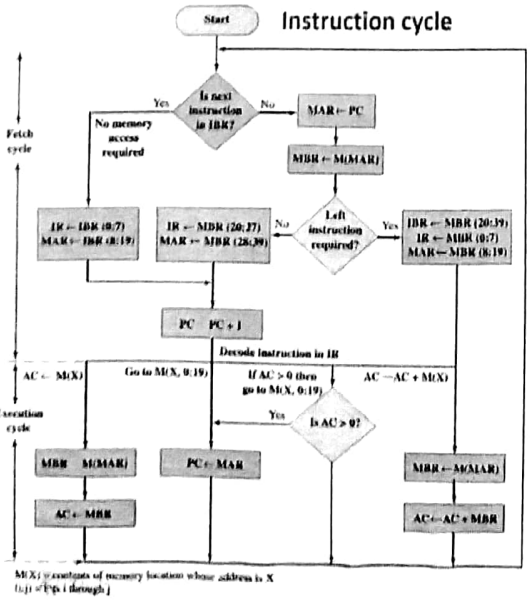


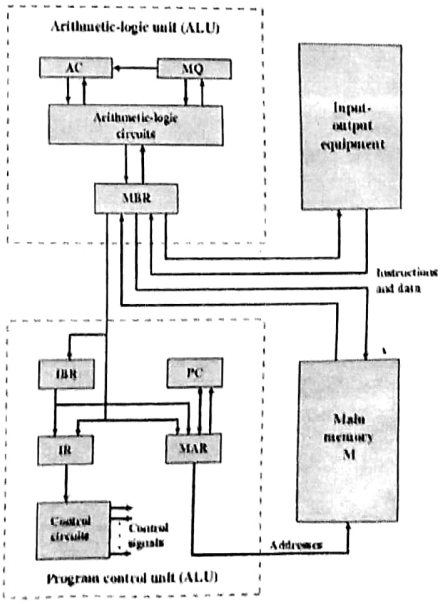
Von Neumann Machine



Harvard Machine

VON NEUMANN ARCHITECTURE	HARVARD ARCHITECTURE
It is ancient computer architecture based on stored program computer concept.	It is modern computer architecture based on Harvard Mark I relay based model.
Same physical memory address is used for instructions and data.	Separate physical memory address is used for instructions and data.
There is common bus for data and instruction transfer.	Separate buses are used for transferring data and instruction.
Two clock cycles are required to execute single instruction.	An instruction is executed in a single cycle.
It is cheaper in cost.	It is costly than von neumann architecture.
CPU can not access instructions and read/write at the same time.	CPU can access instructions and read/write at the same time.
It is used in personal computers and small computers because we need to change the program in RAM.	It is used in micro controllers and signal processing. In these applications the program is already dumped in to the ROM.





*. IAS Architecture :

① Instructions are stored in Main Memory

②

MEMORY

PC	operation	Address	operation	Address
1	Load	M(x) 500	Add	M(x) 501
2	Store	M(x) 500	other instructions	

Step-1 : Set program counter = 1 to start execution.

Step-2 : PC = 1 is copied to MAR.

Step-3 : MAR (address) gives input to Main Memory

Step-4 : In Memory it will see what is there for PC = 1, i.e. Load M(x) 500 Add M(x) 501

Step-5 : Here, Load M(x) 500 Add M(x) 501

Left instruction Right instruction

Step-6 : To perform left instruction, the right instruction is copied to IBR. (Add M(x) 501)

Step-7 : left instruction load M(x) is copied to IR.

Step - 8 : left instruction → Load M(x) - operation
→ 500 - operand address

operation load M(x) is copied to IR.

operand address 500 is copied to MAR.

Now, MAR = 500

Step - 9 : In 500 address stores data 3 → This is given to Main Memory

Step - 10 : The data 3 is copied to MBR

Step - 11 : This data 3 in MBR is now copied to Accumulator (stored).

PC = 1 Ladd M(x) 500 is completed

Step - 12 : The right instruction present in the IBR Add M(x) 501

Step - 13 : Add M(x) 501 → opcode - Add M(x)
→ Address - 501

Step - 14 : opcode Add M(x) is copied to IR and Address is copied to MAR (501)

Step - 15 : Upto Now PC = 1 is completed. Therefore PC = PC + 1 = 1 + 1 = 2 (updated)

Step-16: After updating PC = 2, Address (501) in MAR is sent to Main Memory

Step-17: The value of this Address (data = 4) is sent to MBR from Main Memory.

Step-18: The MBR = 4, AC = 3 → result Store in AC
(both performs operation)

Step-19: The MBR & AC are added & result is stored in Accumulator (AC)
Now AC = 7

With this execution of Add M(x) 501 is completed.

Step-20: As PC = 2 is copied to MAR.

Step-21: PC = 2 from MAR is given to Main memory.

Step-22: The instruction in PC=2 → Store M(x) 500, (other Inst) is sent to MBR.

Step-23: Right instruction (other instructions) is copied to IBR again.

Step - 24 : Store M(x) 500 is left in MBR now.

Step - 25 : Store M(x) is sent to IR and Address 500 is sent to MAR

MAR = 500

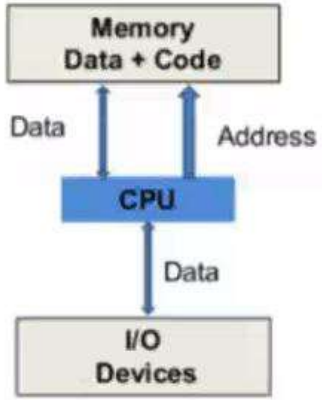
Step - 26 : Now, AC = 7 is copied back to MBR & sent to Main Memory.

Step - 27 : MAR = 500 gives input to Main Memory

Step - 28 : This AC = 7 in Main Memory is stored in 500 Address.

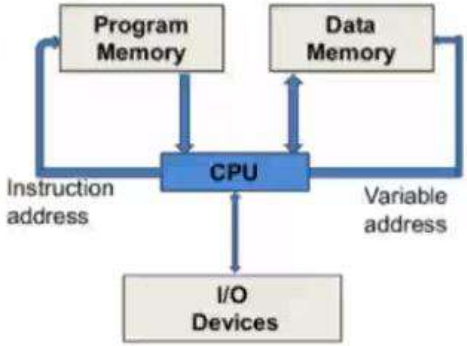
Step - 29 : Now store M(x) 500 is completed.

Step - 30 : It repeats from step - 12 & the process continues ↓ right instruction



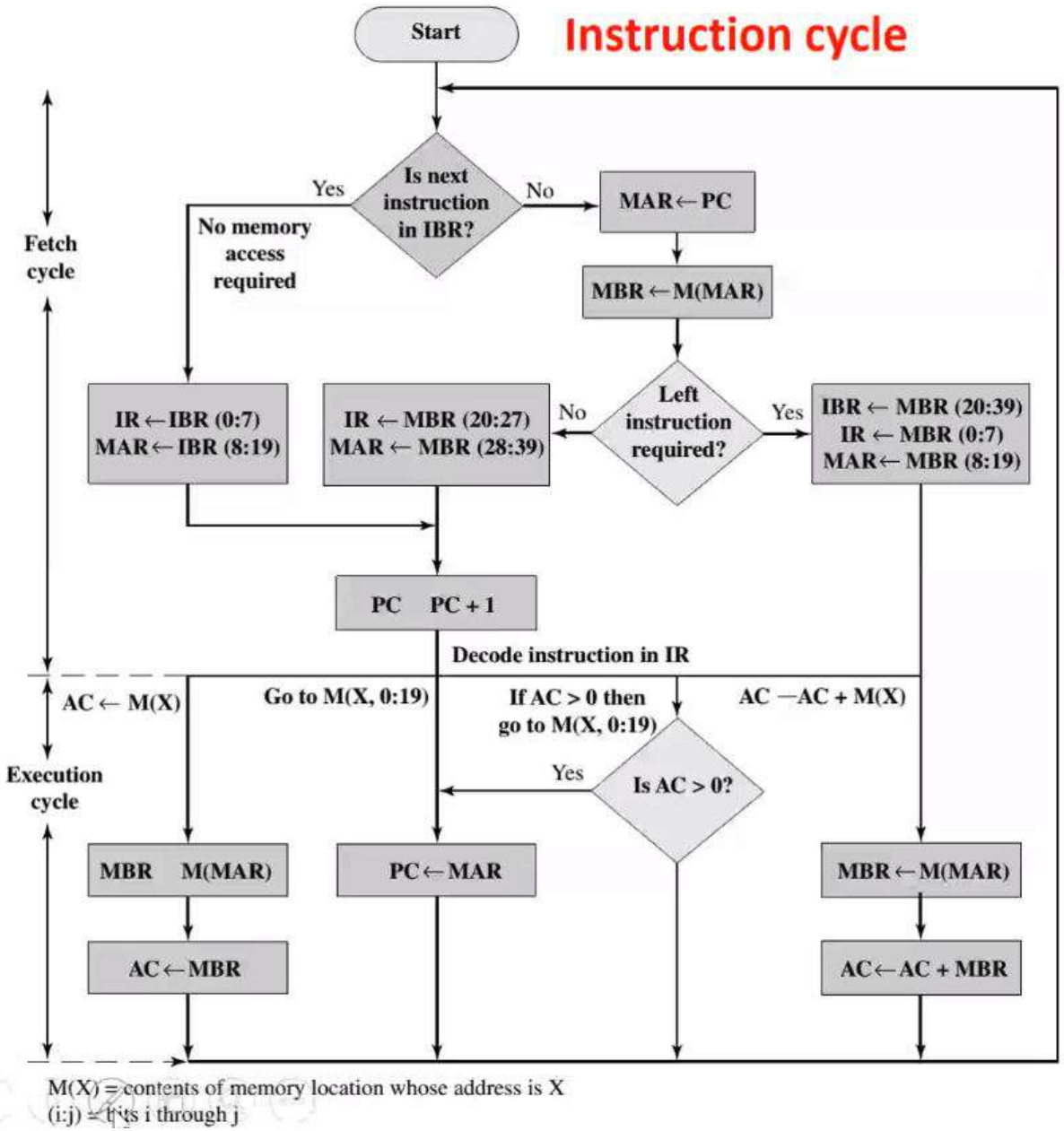
Von Neumann Machine

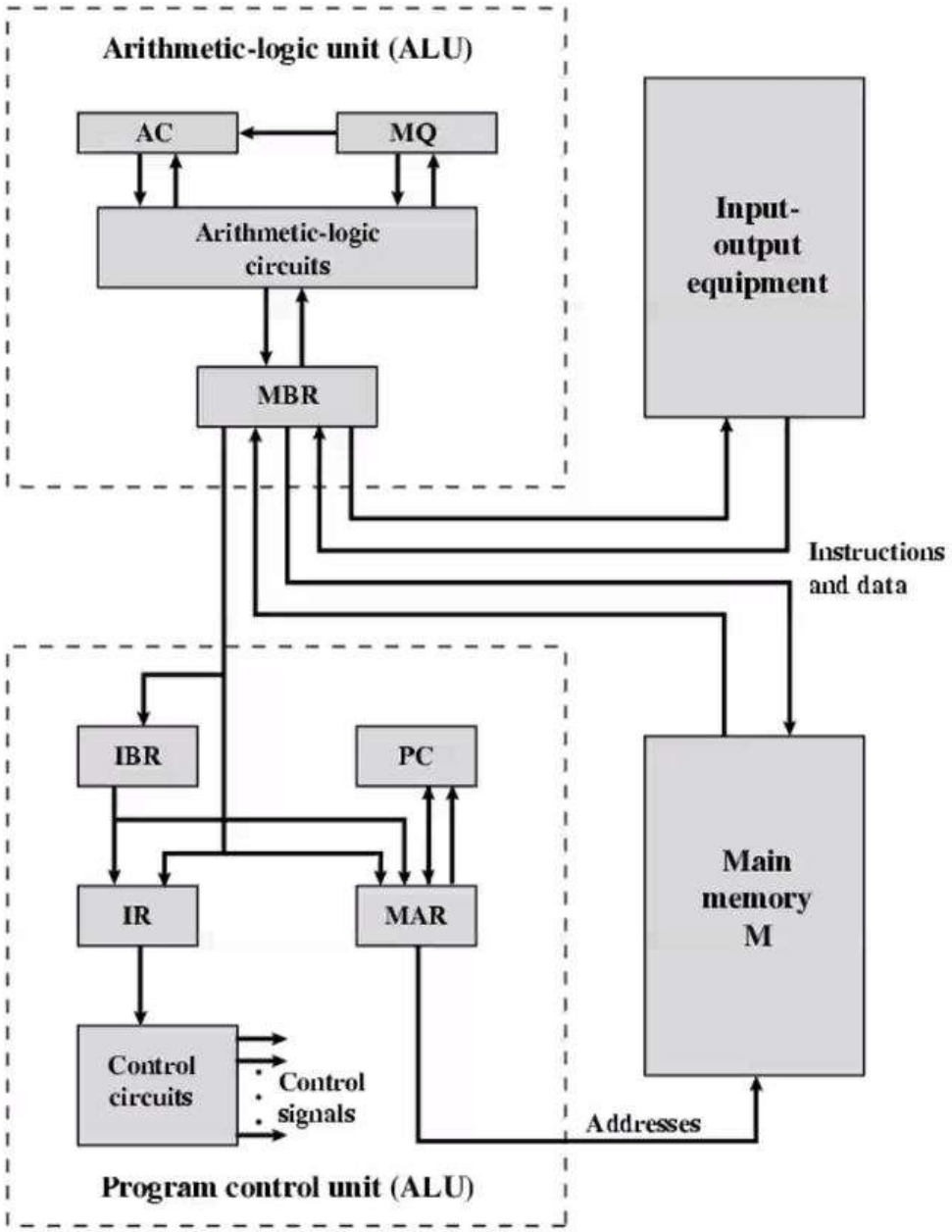
VON NEUMANN ARCHITECTURE	HARVARD ARCHITECTURE
It is ancient computer architecture based on stored program computer concept.	It is modern computer architecture based on Harvard Mark I relay based model.
Same physical memory address is used for instructions and data.	Separate physical memory address is used for instructions and data.
There is common bus for data and instruction transfer.	Separate buses are used for transferring data and instruction.
Two clock cycles are required to execute single instruction.	An instruction is executed in a single cycle.
It is cheaper in cost.	It is costly than von neumann architecture.
CPU can not access instructions and read/write at the same time.	CPU can access instructions and read/write at the same time.
It is used in personal computers and small computers because we need to change the program in RAM.	It is used in micro controllers and signal processing. In these applications the program is already dumped in to the ROM.



Harvard Machine

Instruction cycle





P1 Given the memory contents of the IAS computer shown below,

Address	Contents
08A	010FA210FB
08B	010FA0F08D
08C	020FA210FB

show the assembly language code for the program, starting at address 08A. Explain what this program does.

Instruction Type	Opcode	Symbolic Representation	Description
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD −M(X)	Transfer −M(X) to the accumulator
	00000011	LOAD |M(X)|	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD −|M(X)|	Transfer −|M(X)| to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP+ M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP+ M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)
Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD |M(X)|	Add |M(X)| to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB |M(X)|	Subtract |M(X)| from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2; i.e., shift left one bit position
	00010101	RSH	Divide accumulator by 2; i.e., shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

P1 Given the memory contents of the IAS computer shown below,

Address	Contents
08A	010FA210FB
08B	010FA0F08D
08C	020FA210FB

show the assembly language code for the program, starting at address 08A. Explain what this program does.

Instruction Type	Opcode	Symbolic Representation	Description
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD −M(X)	Transfer −M(X) to the accumulator
	00000011	LOAD |M(X)|	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD −|M(X)|	Transfer −|M(X)| to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP+ M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP+ M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)
Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD |M(X)|	Add |M(X)| to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB |M(X)|	Subtract |M(X)| from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2; i.e., shift left one bit position
	00010101	RSH	Divide accumulator by 2; i.e., shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

P2

On the IAS, what would the machine code instruction look like to load the contents of memory address 2?

How many trips to memory does the CPU need to make to complete this instruction during the instruction cycle?

P3

Let **A** = A(1), A(2), . . . , A(1000) and **B** = B(1), B(2), . . . , B(1000) be two vectors (one-dimensional arrays) comprising 1000 numbers each that are to be added to form an array C such that C(I) = A(I) + B(I) for I = 1, 2, . . . , 1000. Using the IAS instruction set, write a program for this problem. Ignore the fact that the IAS was designed to have only 1000 words of storage.

Answer

P1

This program will store the absolute value of into memory location 0FB.

Address	Contents
08A	LOAD M(0FA)
	STOR M(0FB)
08B	LOAD M(0FA)
	JUMP +M(08D)
08C	LOAD –M(0FA)
	STOR M(0FB)
08D	

P2

OPCODE	OPERAND
00000001	000000000010

First, the CPUmust make access memory to fetch the instruction. The instruction contains the address of the data we want to load. During the execute phase accesses memory to load the data value located at that address for a total of two trips to memory.

Answer

P3

The vectors A, B, and C are each stored in 1,000 continuous locations in memory, beginning at locations 1001, 2001, and 3001, respectively. The program begins with the left half of location 3. A counting variable N is set to 999 and decremented after each step until it reaches -1. Thus, the vectors are processed from high location to low location.

Location	Instruction	Comments
0	999	Constant (count N)
1	1	Constant
2	1000	Constant
3L	LOAD M(2000)	Transfer A(I) to AC
3R	ADD M(3000)	Compute A(I) + B(I)
4L	STOR M(4000)	Transfer sum to C(I)
4R	LOAD M(0)	Load count N
5L	SUB M(1)	Decrement N by 1
5R	JUMP+ M(6, 20:39)	Test N and branch to 6R if nonnegative
6L	JUMP M(6, 0:19)	Halt
6R	STOR M(0)	Update N
7L	ADD M(1)	Increment AC by 1
7R	ADD M(2)	
8L	STOR M(3, 8:19)	Modify address in 3L
8R	ADD M(2)	
9L	STOR M(3, 28:39)	Modify address in 3R
9R	ADD M(2)	
10L	STOR M(4, 8:19)	Modify address in 4L
10R	JUMP M(3, 0:19)	Branch to 3L

Example 3

```
main () {
int a=15, b=5, c;
if (a >= b)
   c = a – b;
else
   c = a + b;
}
```

```
0 15    a
1 5     b
2 c
3 begin
4 . If (a >=b)
4 load M(0)
5 sub M(1)
6 jump+ M(8)
7 jump M(12)
8 .true, c=a-b
8 load M(0)
9 sub M(1)
10 stor M(2)
11 jump M(15)
12 .false c = a+b
12 load M(0)
13 add M(1)
14 stor M(2)
15 halt
```

Example 3 (continued)

- Optimized

```
main () {
int a=15, b=5, c;
if (a >= b)
   c = a – b;
else
   c = a + b;
}
```

```
0 15      a
1 5       b
2 c
3 begin
4 load M(0)
5 sub M(1)
6 jump+ M(9)
7 load M(0)
8 add M(1)
9 stor M(2)
10 halt
```

Example3 (with a > b)

```
main () {
int a=15, b=5, c;
if (a > b)
   c = a – b;
else
   c = a + b;
}
```

```
0 15    a
1 5     b
2 c
3 1
4 begin
5 . a > b
5 load M(0)
6 sub M(1)
7 sub M(3)
8 jump+ M(10)
9 jump M(14)
10 . True, c = a- b
10 load M(0)
11 sub M(1)
12 stor M(2)
13 jump M(17)
14 . False, c = a + b
14 load M(0)
15 add M(1)
16 stor M(2)
17 halt
```

Example 6

```
main () {
int a=2, b=2, I;
I = 1;
while (I < 10) {
   a = a +b;
   I = I +1;
}
}
```

Give it a try.

Example 6 (continued)

```
main () {
int a=2, b=2, I;
I = 1;
while (I < 10) {
   a = a +b;
   I = I +1;
}
}
```

```
0 1
1 10
2 2          a
3 2           b
4 i
5 begin
6 . I =1
7 load M(0)
8 stor M(4)
9 . while (I < 10)
10 load M(4)
11 sub M(1)
12 jump+ M(22)
13 . a = a +b
14 load M(2)
15 add M(3)
16 stor M(2)
17 . I=I+1
18 load M(4)
19 add M(0)
20 stor M(4)
21 jump M(10)
22 halt
```

*. MODULE – 2 : ALU - ARITHEMETIC LOGIC UNIT

Multiplication :

1. Complex 2. Work out partial product for each bit

n bit x n bit = total of 2n bits

→ 4 bits

1 0 1 1 = 11

X 1 1 0 1 = 13 ⇒ 1 0 0 0 1 1 1 1 → 2 x 4 = 8 bits

1 0 1 1

0 0 0 0 +

1 0 1 1 + +

1 0 1 1 + + +

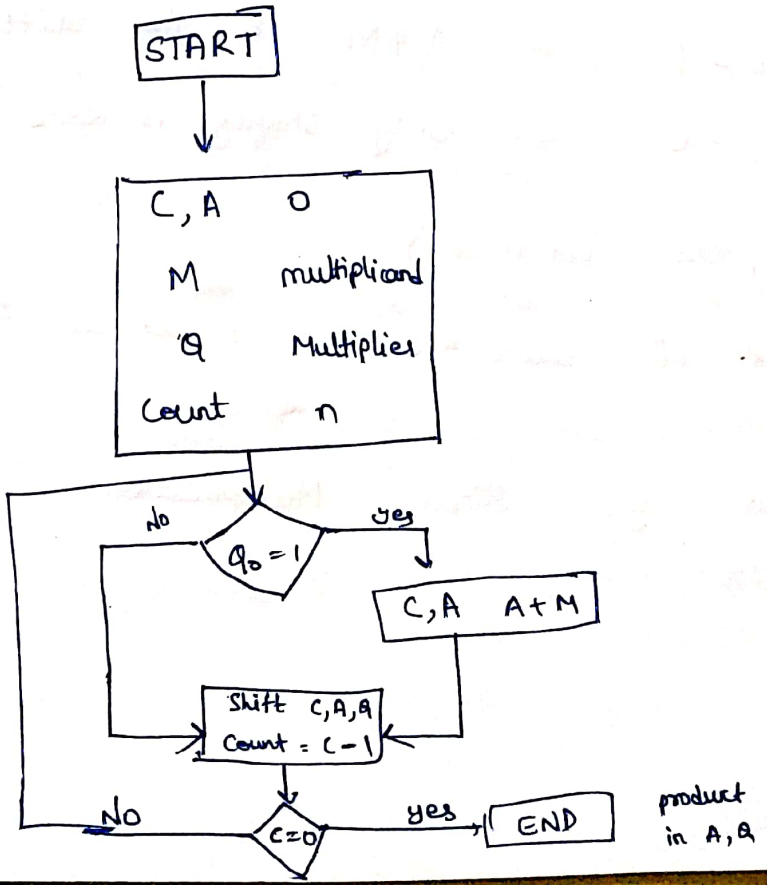
1 0 0 0 1 1 1 1

1 + 2 + 4 + 8 + 128 = 128 + 15 = 143

11
13
33
11+
143

⇒ $(1\,0\,0\,0\,1\,1\,1\,1)_2 = 43_{10}$

- Adders and shift registers only perform this.
- n bits x n bits = output is of 2xn bits.



Multiply negative numbers :

- Convert to positive if required
- Multiply as above
- If signs were different, negate answer

  If signs were same, no change

→ SOLUTION 1

1101
0011
1111
-3

1011
0101
-5

*- BOOTH'S ALGORITHM (for Signed Multiplication)

For Unsigned Binary Multiplication ↓

Multiplier → same bits → M

Multiplicant → Same bits → Q (same bits)

Carry bit → C register = 0

Shift bit → A register = 0 (same bits)

2bit 1bit 0bit

$Q = Q_n \ldots\ldots Q_2\, Q_1\, Q_0$

10001

1+

C	A	Q	M	
0	0000	1101	1011	Initial

If $Q_0 = 1$, ⟹ A + M & then shifting

If $Q_0 = 0$, ⟹ only shifting is done

(This process continues)

At last if Count = 0 → Answer is in A, Q

* For Signed Binary Multiplication :

BOOTH'S ALGORITHM : SOLUTION -2

- Booth's multiplier coding :

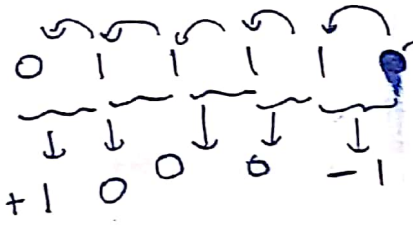
Consider a binary signed number :

0 1 1 1 1 (+15)

If transition is there from right to left : (coding)

(0 0)	0	→ not changing its' sign
(1 1)	0	→ "
(1 0)	-1	→ changing its' sign to -ve no.
(0 1)	+1 (plus 1)	→ " from -ve to +ve.

So, after coding, 0 1 1 1 1 will be



(put a zero to last place)

$\Rightarrow$ +1 0 0 0 -1

Check what it is +1 0 0 0 -1

↓ ↓

16 -1

$\Rightarrow 16 + (-1) = 15$

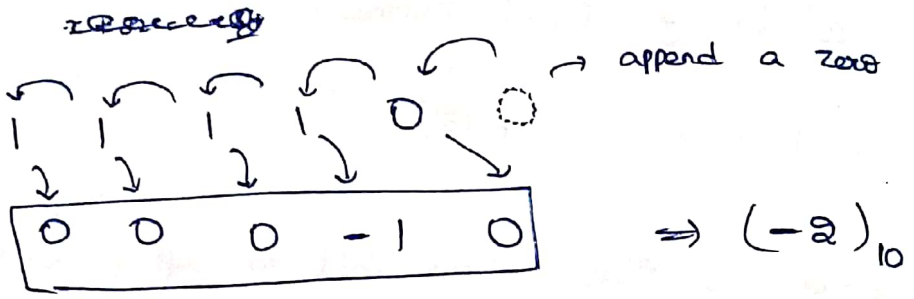
⇓

+1 0 0 0 -1 → after transition booth coding

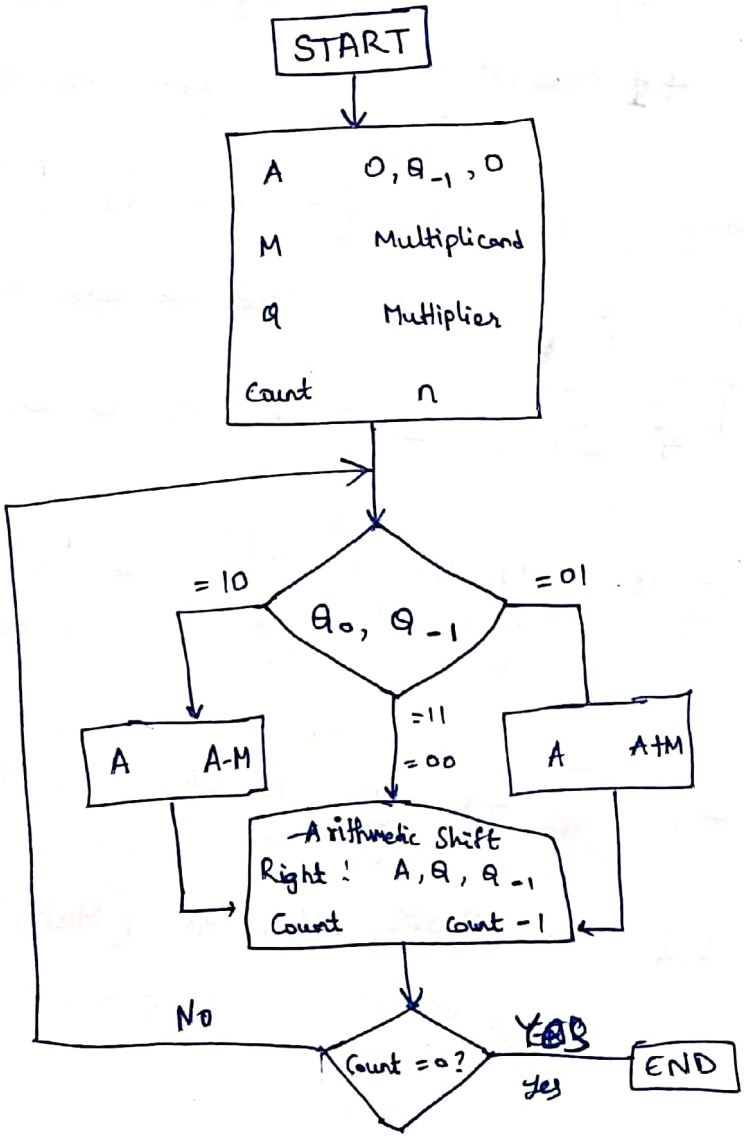
- Booth used this in Booth Algorithm (Multiplier)

1 1 1 1 0

2's ⇒ 0 0 0 1 0 ⇒ −2



* Booth's Algorithm with example :



Example of Booth's Algorithm (7 x 3)

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A	A-M
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	
0101	0100	1	0111	A	A+M
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	

Examples:

- Note: For Q_n & Q_{n-1}
- If 1-1 / 0-0 the value of A will not change i.e. it will remain 0000
- If 1-0 / A = A - M = 0000 - 0111 = 1001
- If 0-1 A = A + M = 0000 + 0111 = 0111
- But for each positive intermediate value, 0s will be appended at left and for negative value, 1s will be appended at left side.

INP

⇒ Try for (7 x 3), (7 x (-3)), ((-7) x 3), ((-7) x (-3)) ✓

0111 x 0101

```
    0 1 1 1
    0 1 0 1
  ---------
    0 1 1 1
  1 0 0 0 0 +
  0 1 1 1 + +
  ---------------
  1 0 0 0 1 1   → value of A in third cycle
```

*. ARRAY MULTIPLICATION :

A_3 A_2 A_1 A_0

B_3 B_2 B_1 B_0

© $B_0 \times A_3$ $B_0 \times A_2$ $B_0 \times A_1$ $B_0 \times A_0$ → done by AND gate

$B_1 \times A_3$ $B_1 \times A_2$ $B_1 \times A_1$ $B_1 \times A_0$

Sum Sum Sum Sum

total $4 \times 4 = 16$ AND gates are needed

. . . . . .

*. Hardware configuration of 4 x 4 Array Multiplier: (Unsigned)

Full adders = 8

Half adders = 4

And Gates = $4 \times 4 = 16$ gates = n^2 bits

n multiplicand bits, n multiplier bits

⇒ For m * n bit multiplier, we need

no. of AND Gates = m * n — m - multiplicand bit

no. of Half Adders = n — n - multiplier bits

no. of Full Adders = (m-2) * n

*. Signed multiplication using array multiplier :

(9) x (3)

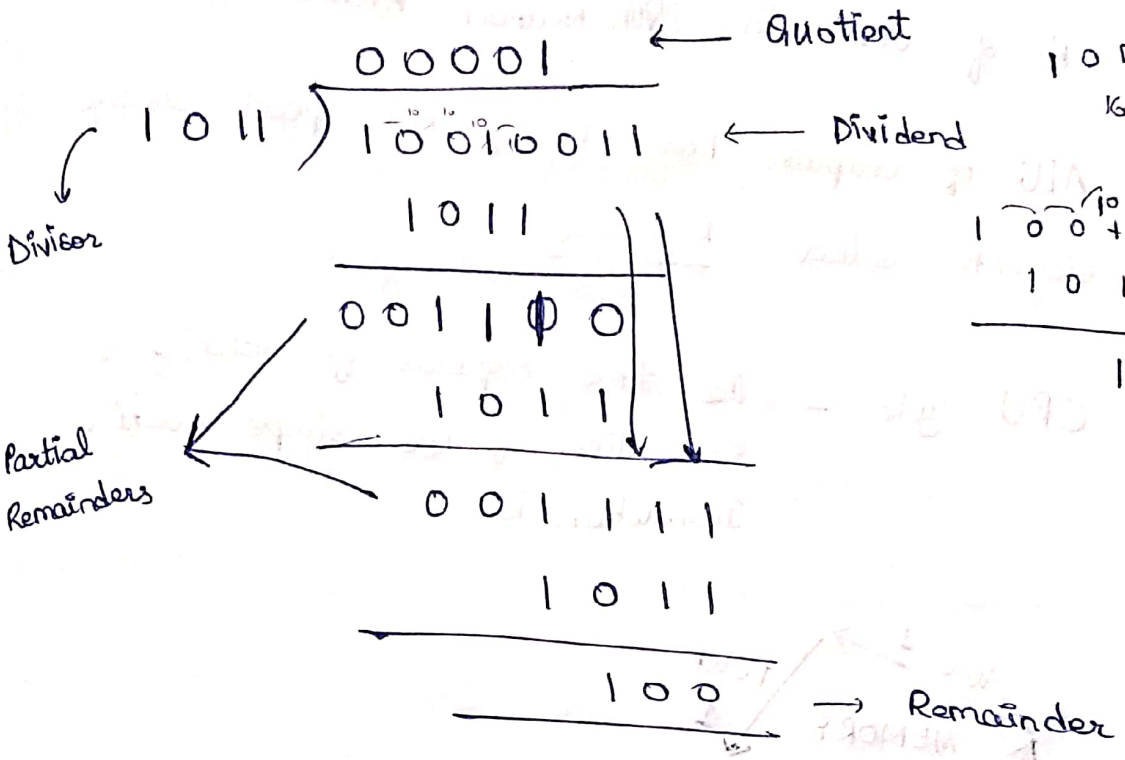
1001 = 9

0011 = 3

(-7) x (3)

1001 = -7

0011 = 3



10010

16+2=18

1 0 0 10

1 0 1 1

1 1

*. RESTORING ALGORITHM :

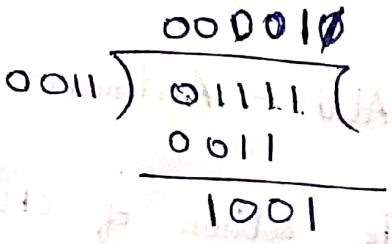
A ← 0

M ← Divisor

Q ← Dividend

Count ← n

7/3



*. (For Signed Numbers) :

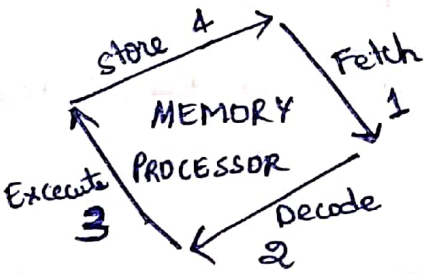
*. Key points :

1. Pointer is not input device (its output device)

2. No. of Units in Von Neumann Architecture = 5

3. ALU of Computer has no. of high speed storage elements called Registers

4. CPU cycle - The time required for fetching & execution of one simple machine Instruction is.



5. Computer Memory consists of → RAM, ROM, PROM

6. ALU - Arithmetic and Logic Unit

7. The section of CPU that selects, interprets and sees to the execution of program instructions Control Unit.

8. RAM is located in Mother Board.

9. Input Unit is responsible for recording information provided by User and sending it to CPU.

10. Registers are used for Small internal storage of CPU.

11. Components that provide internal storage to CPU = Registers.

12. Saving data & instructions to make them readily available is job of Storage Unit.

13. Basic Memory → Primary, Secondary

14. Primary Memory is used to hold running program instructions.

15. DRAM is used in Main memory.

16. Moore's Law :

- Increase operating speed due to short electrical path length.
- Reduction in power & Cooling requirements.
- Cost of a chip has remained virtually unchanged.

17. According to Von Neumann model, data & programs are stored in memory.

18. MAR, AC, PC are ~~is~~ Registers in CPU.

19. Fetch Cycle :
1. PC is incremented
2. PC value copied to MAR
3. The Instruction at MAR is read from RAM.

20. Decode cycle : Instruction in MR → CU

21. Execute cycle : 1. The data at MAR is read from RAM.

2. The ALU performs any calculations required on AC.

3. The value in MDR is written to address in MAR.

22. MAR - memory Address Register.

23. Idea — Stored program Concept.

24. Small high speed storage area within memory and processor → Register.

25. PC ≡ Stores address of next instruction in memory.

26. MAR ≡ Stores address of where next item be fetched from.

27. MDR ≡ Stores data fetched from memory or waiting to be stored in Memory

28. Accumulator ≡ Stores the result of arithmetic calc

29. CPU ≡ Controls the flow of data

30. ALU ≡ Carries out numerical logical decisions.

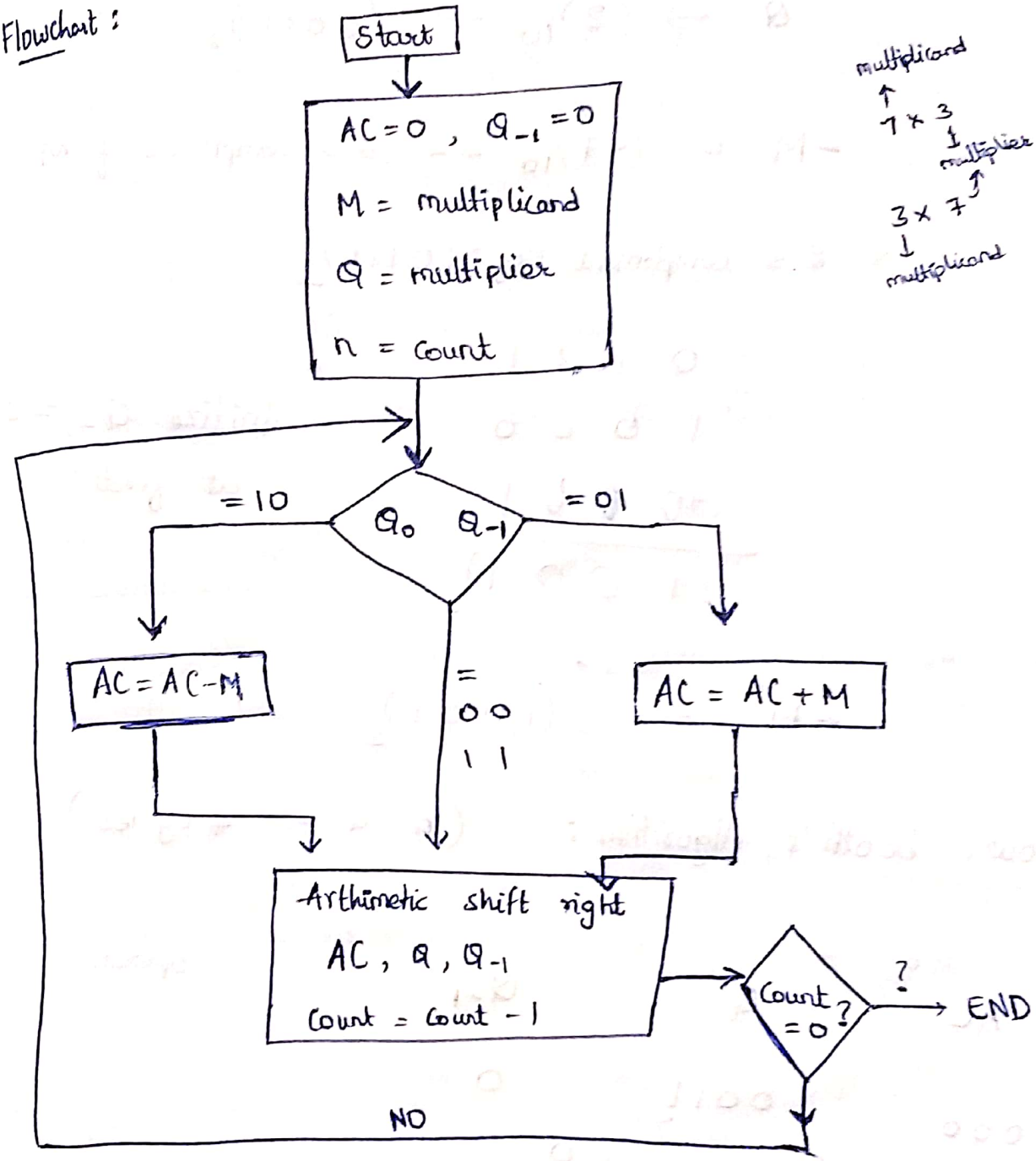
31. Cache ≡ Stores frequently used data & instructions.

32. How CPU process Instructions ?

→ Using Fetch Decode Execute Cycle.

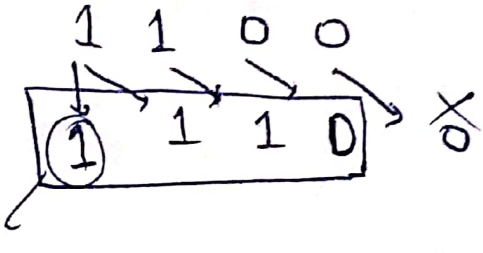
*. MULTIPLICATION - BOOTH'S ALGORITHM :

Flowchart :



multiplicand
↑
1 x 3
↓ multiplier
3 x 7 ↑
↓
multiplicand

Arthimetic shift right means :



write as it is whatever is there

0 0 1 1 0

⇓

Ans: 0 0 0 1 1

* . Multiply 7 and 3 using Booth's Algorithm :

Register size = n = 4

$M \rightarrow (7)_{10} \rightarrow (0111)_2$

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

$-M = (-7)_{10} \rightarrow$ 2's complement of M

2's complement of $(0111)_2$

$$\begin{array}{r} 0\ 1\ 1\ 1 \\ 1\ 0\ 0\ 0 \\ +\ 0\ 0\ 0\ 1 \\ \hline (1\ 0\ 0\ 1)_2 \end{array}$$

initize $Q_{-1} = 0$ at first

$-M \longrightarrow (1001)_2$

Now, booth's Algorithm : ($n = 4 \Rightarrow$ 4 cycles)

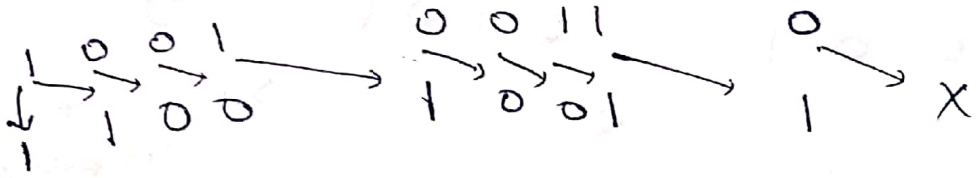
AC	Q	Q_{-1}	operation
0000	0011	0	
		10	

↓

$AC = AC - M$ = AC + (2's complement of M)

$$\begin{array}{r} AC = \quad 0\ 0\ 0\ 0 \\ +\ 1\ 0\ 0\ 1 \\ \hline 1\ 0\ 0\ 1 \end{array}$$

Now, do Arithmetic shift right.



1 1 0 0 1 0 0 1 1

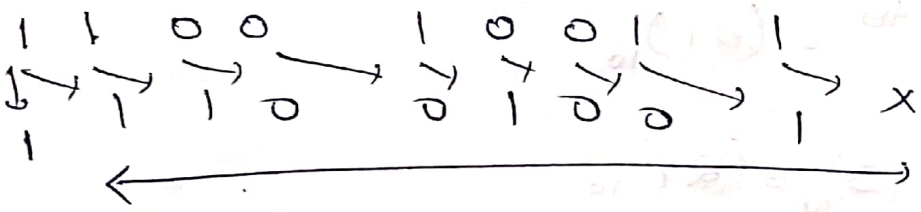
cycle 1 is completed

1 1 0 0 1 0 0 1 1

1 1

↓

Arithmetic shift right



cycle 2 over

1 1 1 0 0 1 0 0 1

0 1

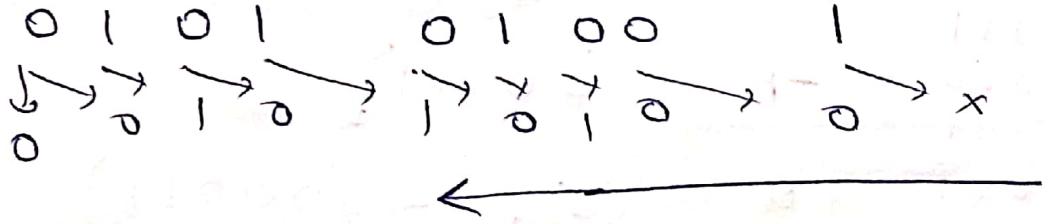
AC = AC + M

Discard Carry

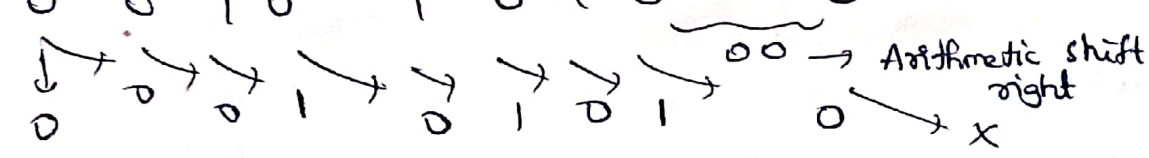
AC = 1 1 1 0

M = 0 1 1 1 +

new AC = 0 1 0 1 = AC + M



Cycle 3 over



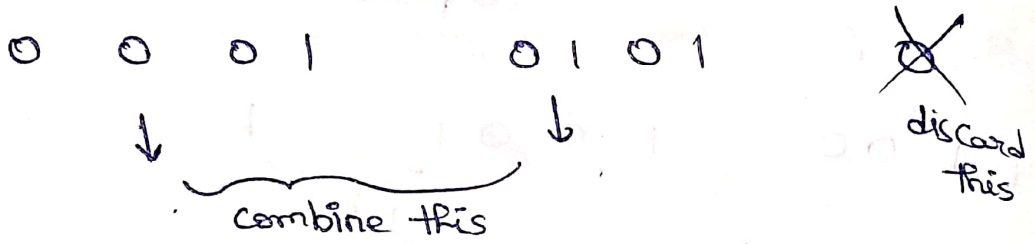
Arithmetic shift right

0 0 0 1 0 1 0 1 0

AC	Q	Q_{-1}
0 0 0 1	0 1 0 1	0

Now register size = 4 = no. of cycles

So, After 4th cycle,



$(0\ 0\ 0\ 1\ 0\ 1\ 0\ 1)_2$

1+4+16

is equal to $= (21)_{10}$

$(7)_{10} \times (3)_{10} = (21)_{10}$

Example – 2 :

Multiply −7 & +3 using Booth's Algorithm.

$M \longrightarrow (-7)_{10} \longrightarrow$ 2's complement of $(7)_{10}$

$M \rightarrow (-7)_{10} \rightarrow (1\ 1\ 0\ 0\ 1)_2$

$-M \rightarrow (0\ 0\ 1\ 1\ 1)_2$

$Q \rightarrow (3)_{10} \rightarrow (0\ 0\ 0\ 1\ 1)_2$

```
  0 0 1 1 1
  1 1 0 0 0
+         1
  1 1 0 0 1
```

00011

AC Q Q_{-1}

0 0 0 0 0 0 0 0 1 1 0

10

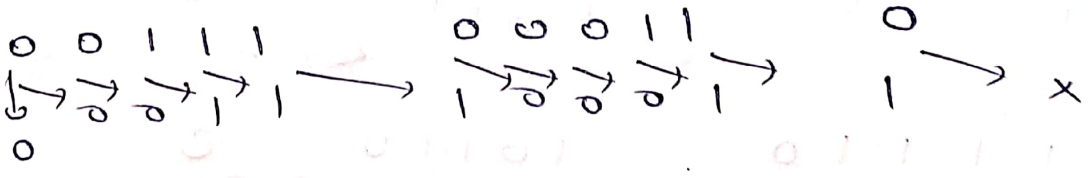
↓

AC = AC − M

AC = 0 0 0 0 0

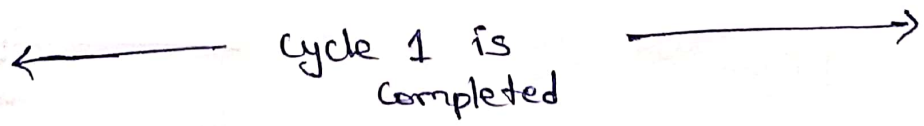
−M = + 0 0 1 1 1

AC = 0 0 1 1 1



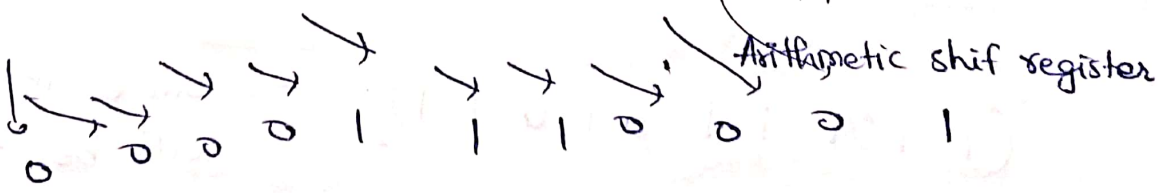
0 0 1 1 1 0 0 0 1 1 0 1

0 0 0 1 1 1 0 0 0 1



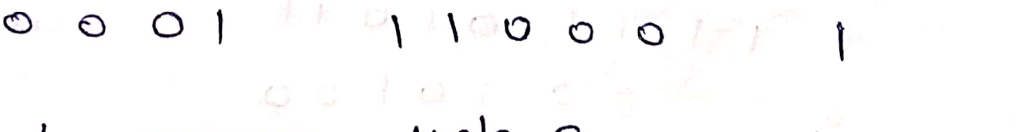
0 0 0 1 1 1 0 0 0 1 1

11



0 0 0 0 1 1 1 0 0 0 1

0 0 0 0 1 1 1 0 0 0 1



0 0 0 0 1 1 1 0 0 0 1

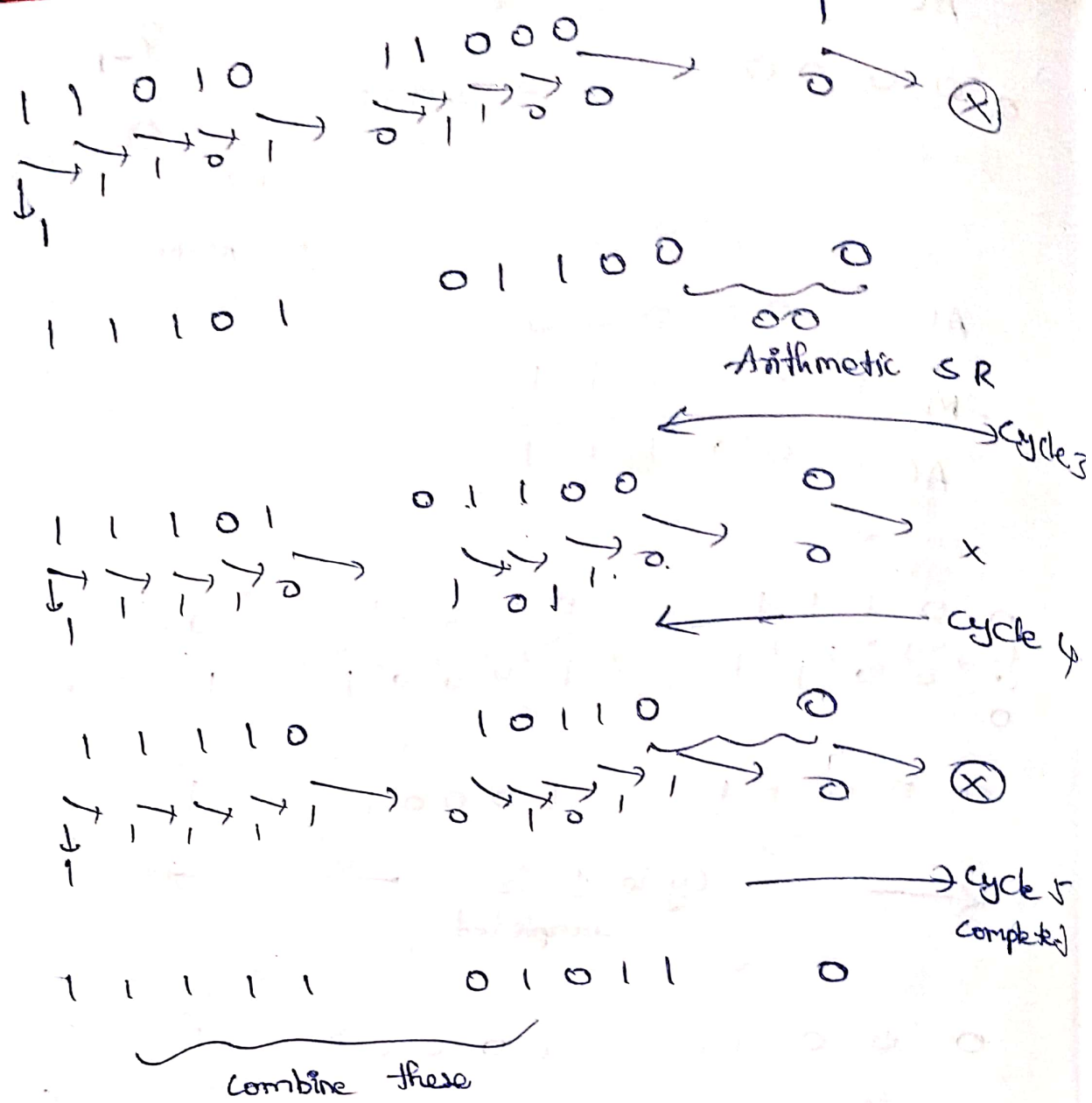
01

AC = AC + M

AC = 0 0 0 0 1

\+ M = 1 1 0 0 1

AC = 1 1 0 1 0 = 1 1 0 1 0 1 1 0 0 0 1



$(11111010 11)_2$ → Since result will be -ve so find 2's complement

```
  1 1 1 1 1 . 0 1 0 1 1
  0 0 0 0 0   1 0 1 0 0
+                     1
-----------------------
  0 0 0 0 0 . 1 0 1 0 1
-----------------------
```

$(0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1)_2 = (-21)_{10}$

- sign because of negative numbers

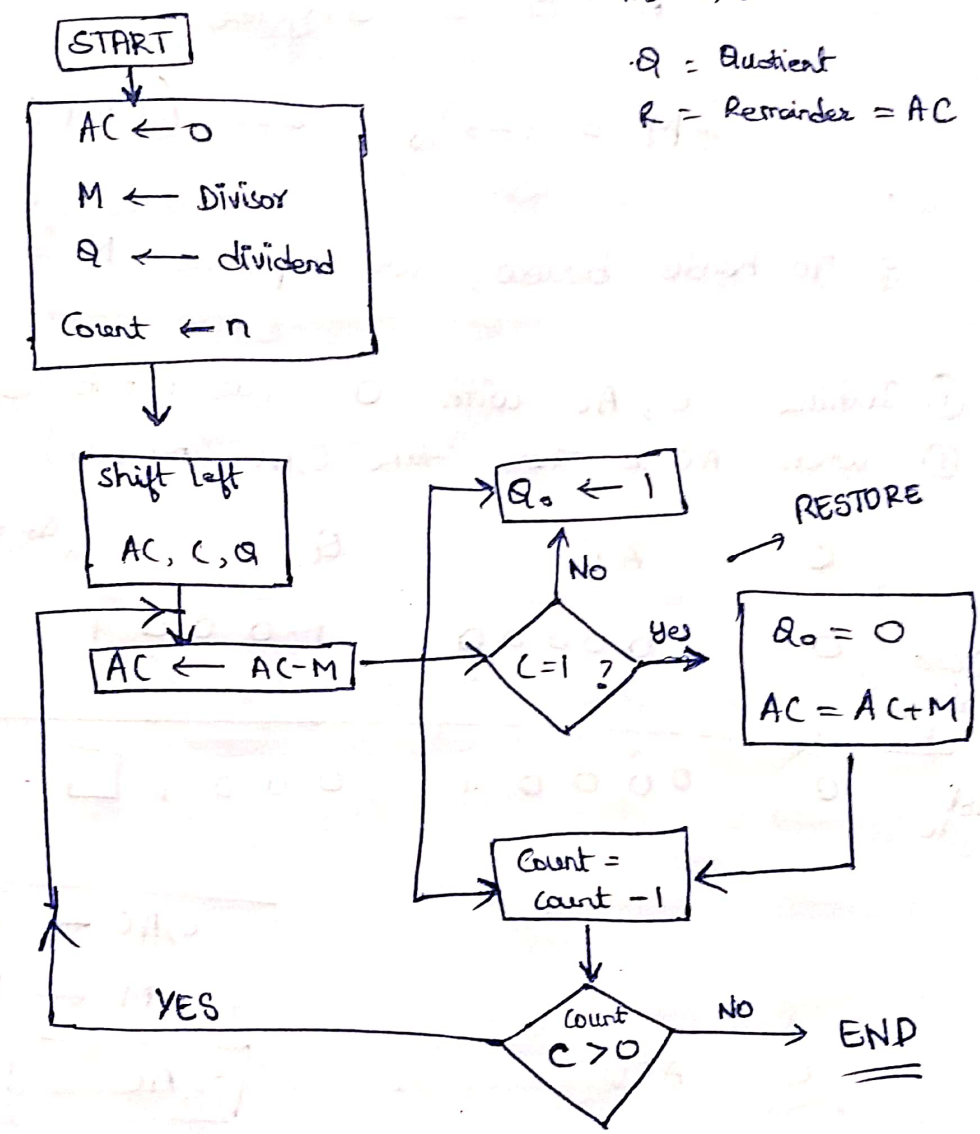
* RESTORING DIVISION ALGORITHM :

→ left shift

→ subtraction

→ carry control bit (using NOT gate)

* Flow chart

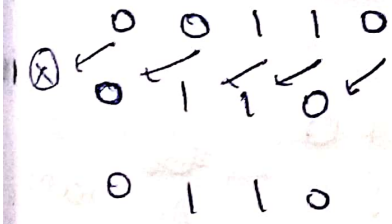


At last :

Q = Quotient

R = Remainder = AC

shift left means



Carry control bit :

1 → 0

0 → 1

(NOT gate)

Example: Perform division using restoring division

Dividend = 17 $\qquad$ $17 \div 3$

Divisor = 03

$Q \leftarrow (17)_{10} \leftarrow (10001)_2$

$M \leftarrow (3)_{10} \leftarrow (00011)_2$ (leading 0 to handle borrow)

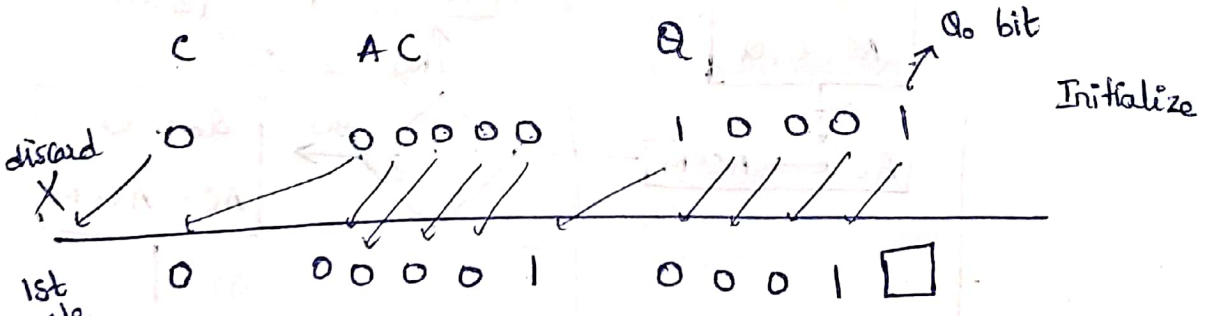
$n = 5 =$ 5 cycles

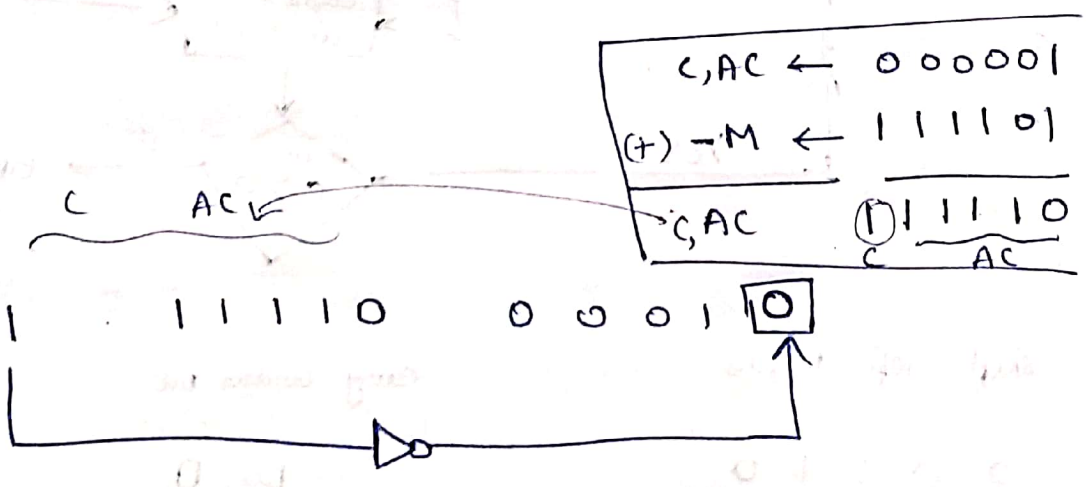
$-M = (-3)_{10} \leftarrow (11101)_2$

& To handle borrow, we represent M in (n+1) bits,

① Initilize C, AC with 0 (as n bits needed)

② when AC is there take C,AC (n+1 bits)

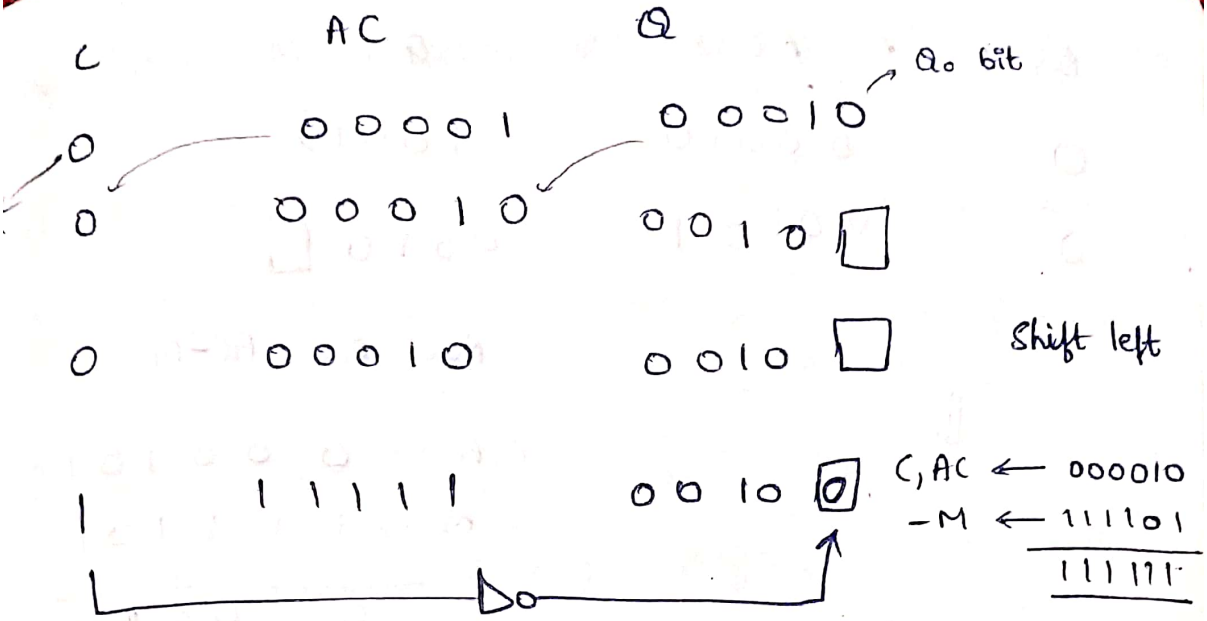




1 11110 00010 → here AC = AC+M

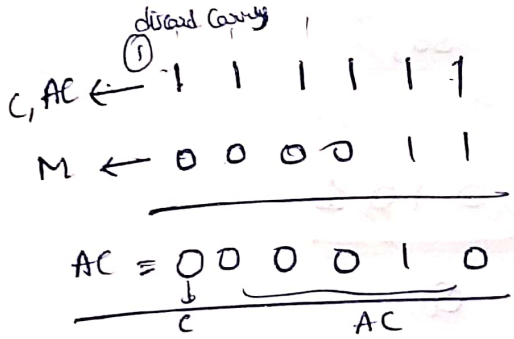
0 00001 00010

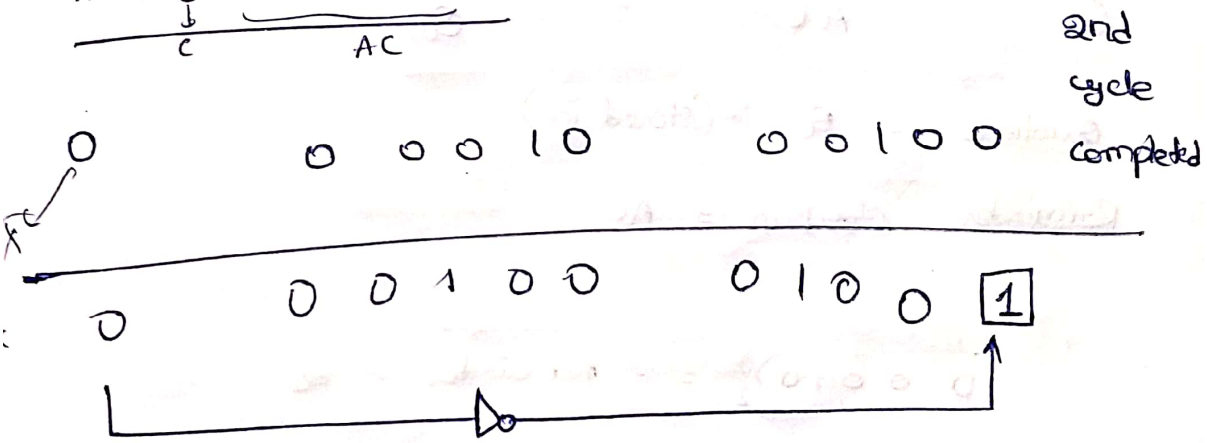
⟷ cycle 1 is completed



1 1 1 1 1 1 0 0 1 0 0

Now since C=1 → Perform restore → AC = AC+M





3rd cycle Completed

* After 4th cycle,

0 0 0 0 1 0 1 0 0 1 0

C	AC	Q
0	0 0010	10010
0	00101	0010 ☐

Now, AC = AC − M

⇓

C, AC → 0 00101
−M → 1 1110 1
0 00000

0	0 0010	0010	1

Since C = 0 ⇒ no restore So,

0	0 0010	00101
C	AC	Q

Quotient = Q (stored in)

Remainder stored in = AC

$(0\ 0010)_2$ = remainder = 2

$(00101)_{10}$ = Quotient = 5

*.

*. Things to remember in Restoring division Algorithm

① Initialize C, AC with 0 (as n bits needed)

② when AC is there take C, AC & perform operations

③ After every cycle check 'C' value & perform restore (or) and perform left shift after every cycle.

④ Perform left shift always after every cycle.

⑤ when there is n register bits = n total cycles to complete

⑥ output after nth cycle is stored,

Quotient is stored in Q

Remainder is stored in AC

⑦ Whenever during AC = AC+M (or) AC = AC-M Carry or borrow is generated at last, discard it & take first most bit as 'C' & remaining as AC. (C, AC)

⑧. Sign R = sign(D)

Sign Q = sign (D) * sign (V)

Quotient

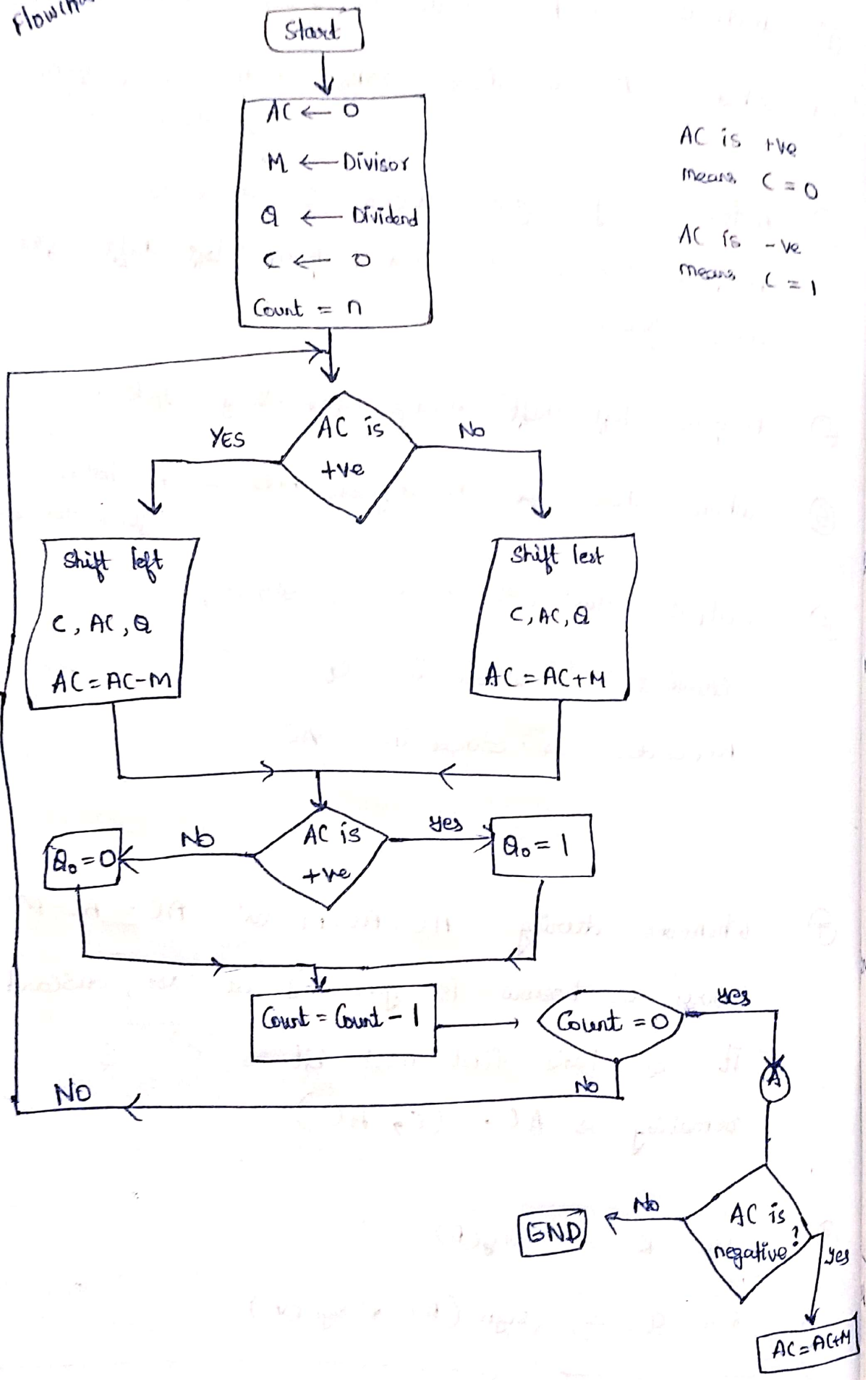
$Q * V + R = D$

Divisor Dividend

Dividend = Quotient × Divisor + Remainder

* Non - Restoring Division :

Flowchart :



AC is +ve means C = 0

AC is -ve means C = 1

Example : Perform division using non-restoring division,

$$(10)_{10} \div (3)_{10}$$

$$Q \leftarrow (10)_{10} \leftarrow (1010)_2$$

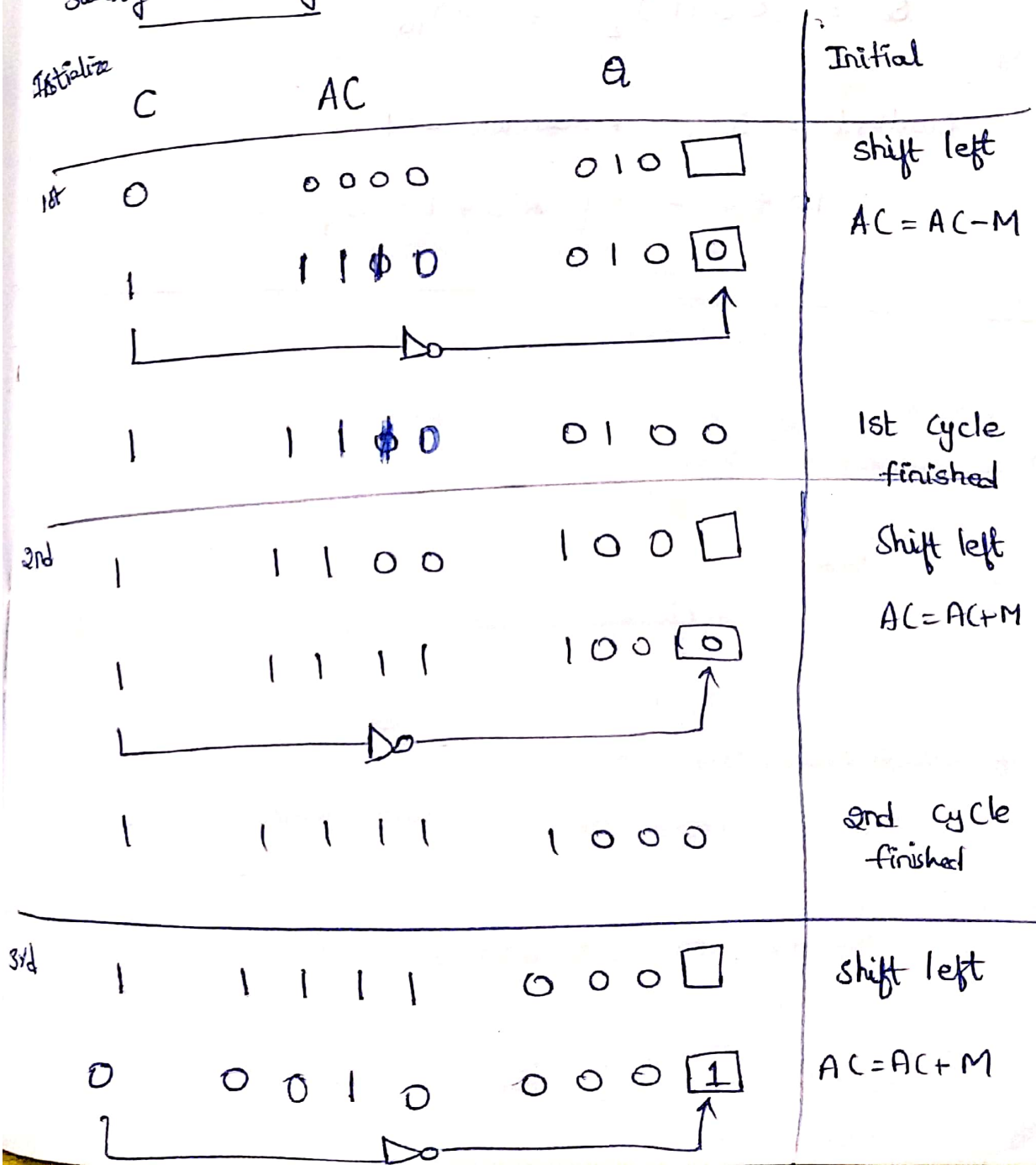
$$M \leftarrow (3)_{10} \leftarrow (00011)_2$$

(0 : handle borrow)

$$-M \leftarrow (-3)_{10} \leftarrow (11101)_2$$

$n = 4$ (no. of cycles)

Starting the Algorithm :



0	0010	000	1	3rd cycle finished
4th				
0	0100	001	□	Shift left
0	0001	001	1	AC = AC − M
0	0001	0011		4th cycle finished

Conclusion

$$R = (0001)_2 = (1)_{10}$$

$$Q = (0011)_2 = (3)_{10}$$

Quotient = 3, Remainder = 1

$10 \div 3 \Rightarrow$ $\boxed{3 * 3 + 1 = 10}$

MODULE 5

MICROPROCESSOR

Examples :

Intel	4004	Intel Core i7
Intel	8085	AMD Athlon
Intel	8086	
Intel	Pentium 4	

*. programmable, multipurpose silicon chip, clock driven, register based, accepts binary data (IC, chip of CPU)

*. INTEL 8086 :

BIU - Bus Interface Unit

EU - Execution Unit

① Fetching is 1st instruction, done by BIU

② Execution of instruction, done by EU

	Time cycle 1	Time Cycle 2	Time Cycle 3
Instruction 1	Fetch 1	Execution 1	~~Execution 2~~
Instruction 2		Fetch 2	Execution 2

Fetch 1 is done 1st, After that it does Execution of 1st instruction parallely with Fetching 2nd instruction & then it does Execution of 2nd instruction & Fetching 3rd instruction ..... → "PIPELINING"

*. with Pipeling it is advantage to 8086 processor

2 Stage pipeline.

Now (2020) we using 20 Stage pipeling.

* EU & BIU indivisually & independently work.

* 8086 is of 16 bit microprocesser (maximum data width & operations are on 16 bits)

* These Address Registers are also of 16 bits.

AH | AL	= 8+8 = 16 bits	= AX
BH | BL	= 8+8 = 16 bits	= BX
CH | CL	= 8+8 = 16 bits	= CX
DH | DL	= 8+8 = 16 bits	= DX

8086 Registers :

(1.) General purpose Registers : We can store & operate.

e.g., AX, BX, CX, DX (16 bits)

We can operate indivisually the registers.

H means higher order 8 bits.

L means lower order 8 bits.

(2.) Segment Registers :

CS - Code Segment

SS - Stack Segment

IP - Instruction pointer

DS - Data Segment

ES - Extra Segment

* operations are done in ALU but result or output is stored in AX, BX, respectively.

*

* MEMORY ORGANIZATION in 8086 :

→ It has 16 bit address register but has a 20 bit address bus.

→ It processes addresses by using the segment : offset mechanism, wherein it left shifts the segment addres by 4 bits then adding the offset to it.

→ So, 16 bit left shifted by 4 gives 20 bit address, and 2^{20} = 1 MiB

→ That's why 8086 is able to access 1MiB of RAM.

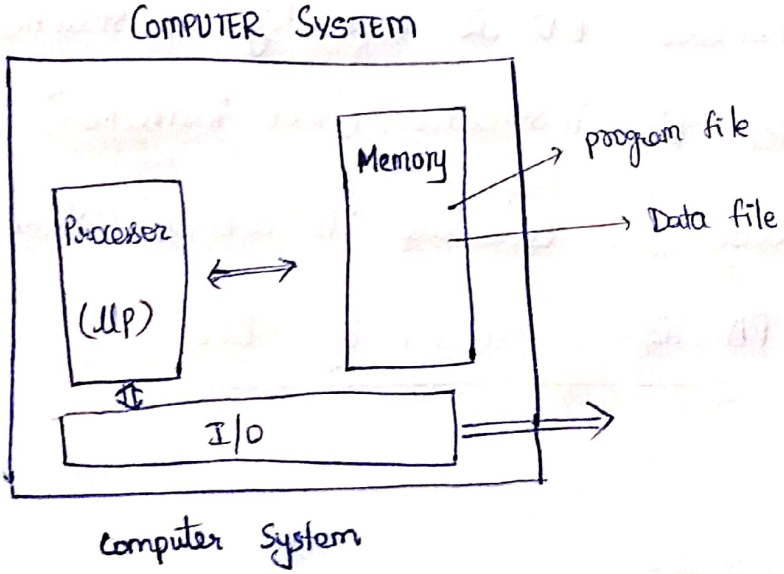
*. Memory is divided into 4 segments.

① Instruction is stored in code segment.

② Data is stored in Data segment.

* Start from BASICS :

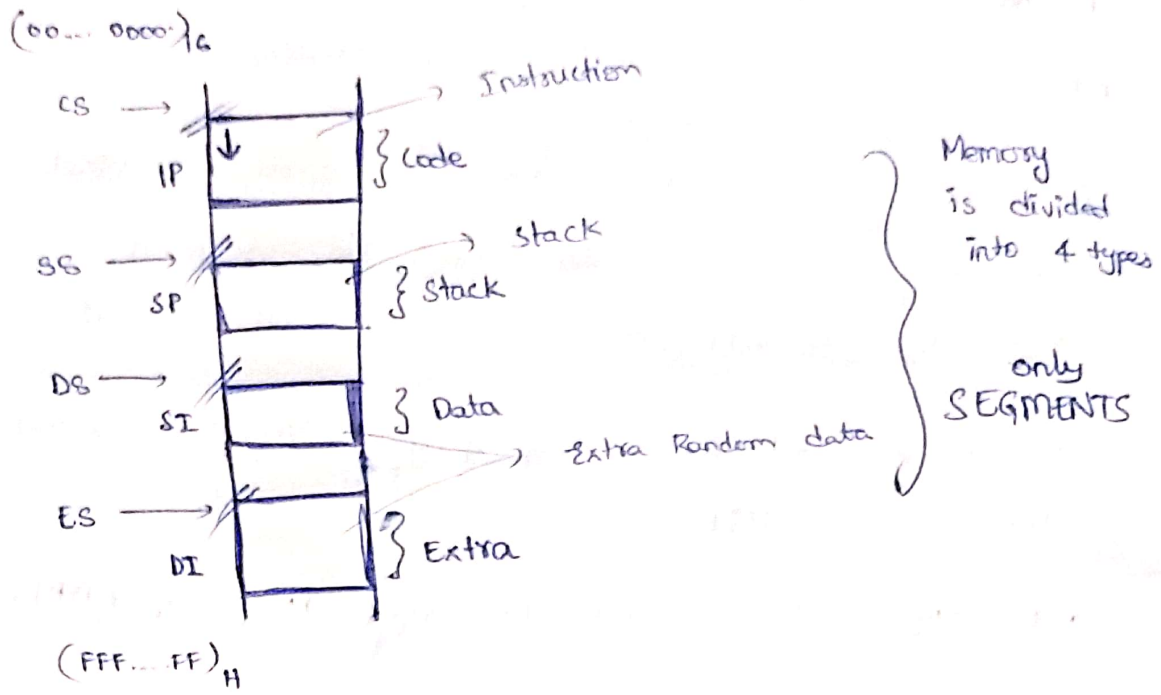
① Processors come into picture wherever programming is involved.



Computer System

Actual Address (or),

* Physical Address = Segment Address × 10_H + Offset Address



IP, SP, SI, DI → offset address

CS, SS, DS, ES → Segment address Registers

* **PA = SA × 10_H + OA**

The real address is Physical Address of file calculated by Computer.

Assume
CS = 1000 H → Segment Addr
IP = 2345 H
PA = 12345 H

* BIU – fetching EU – Execution

* So, whenever EU is executing instruction, BIU wants to fetch the instruction (next instruction) for which BIU needs to calculate its' address, that is done by PA is calculated by BIU.

Importance of BIU : (Functions)

① Fetch next instruction

② Calculate the Physical Address

③ Manage Queue.

The CS, SS, DS, ES, IP are not segments. Segments are present in Memory. But these are segment registers, these contain addresses of all the segment.

imp *. Every instruction, offset address changes for every next, so, Instruction pointer gets on incrementing when every instruction gets on fetching.

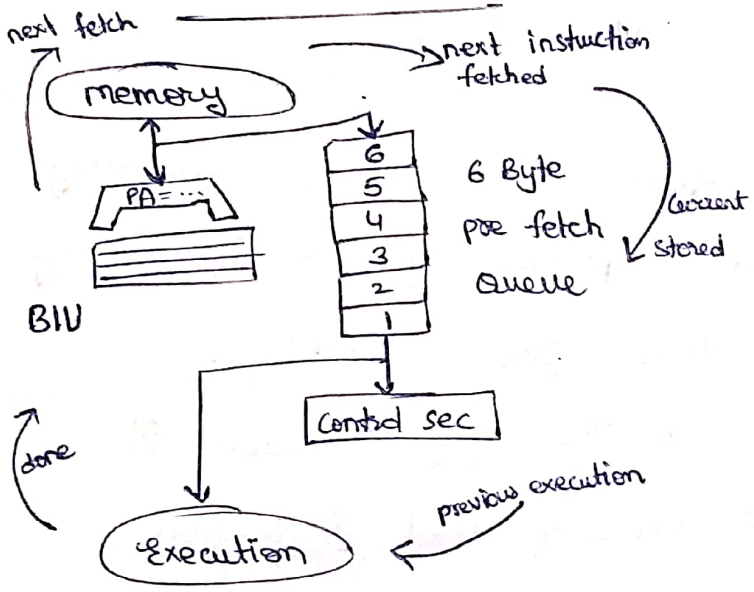
This how IP points to the next instruction, it always gives offset address of next instruction

CS - Segment address
IP - offset address
→ both are used to calculate Physical Address of next instruction

From memory, instruction is fetched inside through Data bus (16 bits)

③ Busses → Address Bus = we gave Address
(16 bits) Data Bus = Instruction comes inside
Control Bus = from this we gave green signal that we want to fetch an instruction

*. Since this is a pipeline process, the instruction is not executed immediately after fetched (since some execution is going on of the previous instruction) but it is fetched in advance and stored in "6-byte Pre-fetch Queue".



BIU fetches & stores it in pre-fetch Que & it is sent to execution after previous instruction got executed & when current instruction is sent from Pre-fetch to EU, the BIU fetches next instruction & sents it to Pre-fetch ..... this process continues.

*. 6-Byte Pre-fetch Queue :

Instruction

1 → Executes

2
3
4
5
6
7
8

next 6 Bytes of the program

When 1st instruction is executed, while 1st instruction execution is going on, BIU fetches next 6 bytes of the program

No. of Bytes in Queue = 6

No. of instruction in Queue = ✗ all are of different bytes.

Example,

opcode

MOV BL, CL ⇒ BL has value of CL

opcode MOV BL, 25H ⇒ BL has value = 25H

opcode MOV BX, 2000H ⇒ BX has value = 2000H

Assembly code → Assembler → Converts to Machine Code

Machine Code → puts this program into memory → Gives it to processor → EXECUTION

Code is called OPCODE

opcode specifies operation, Registers but cannot have number.

MOV BL, 25H → instruction opcode of with 8 bit operand

MOV BL, 2000H → opcode with 16 bit operand

MOV BL, CL → only opcode ....

→ Different instructions are of different bytes.

BIU fetches next 6 bytes of program and stores in Pre-fetch Queue

FIRST IN FIRST OUT

order in which instruction are fetched, should be the order in which they are executed

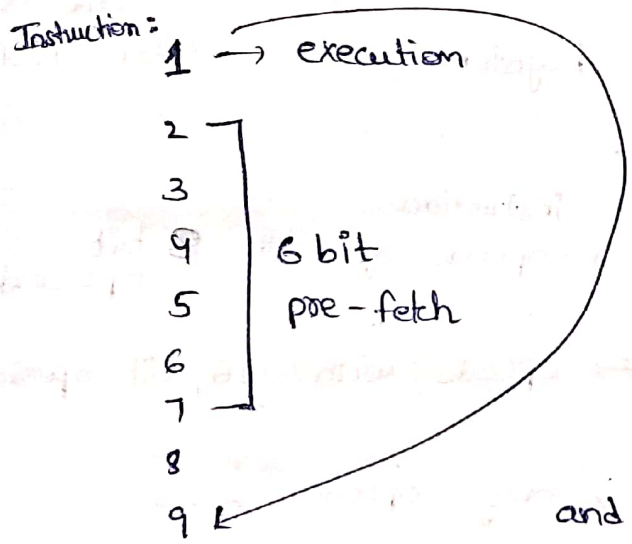
"QUEUE"

* When the 2-bytes of the Pre-fetch Queue are empty, it will fetch the next instruction & store. & the process continues.

2-bytes = 8 + 8 = 16 bits (8086 – 16 bit processor)

* Pipeline saves time,

* Pipelining are assuming the program is going in sequential order. but if there is jump instruction & that may or mayn't be fetched immediately after this instruction



Jump to 9 is written in programme

After 1, 2 to 7 are present in pre-fetch & are ready for execution

and 9th instruction is not present, ✗ so, pipelining fails here

* Pipelining works perfectly fine only for sequential programming (without any JUMP instructions & branch instruction)

*. EU : Execution Unit only for Source & Destination.

SP, BP, SI, DI - offset registers

AX, BX, CX, DX - General purpose registers.

→ General Purpose Registers :

AX - 16 bit = AH AL
BX - 16 bit = BH BL
CX - 16 bit = CH CL
DX - 16 bit = DH DL

} are used by programmers to store the data.

AH contains higher bytes — 12
AL contains lower bytes — 34
→ AX = 1234

Example of an Instructions:

① MOV AL, 34H ; AL = 34
MOV AH, 12H ; AH = 12
MOV AX, 1234H ; AX = 1234

② MOV BL, CL ; BL ← CL
MOV BH, CH ; BH ← CH
MOV BX, CX ; BX ← CX

} Registers

Program to Add :

Question: 04H + 05H = 09H
↓ BL ↓ CL ↓ BL

⇒

Solution : ↓

MOV BL, 04H
MOV CL, 05H
Add BL, CL
~~MOV~~

BL → destination to store

*.

Decode the opcode & operand is to perform operation.

*. operands register -

It is temperory register used by processor to store temperory value. (Similar to ~~a = temp~~ temp) ~~temp = b~~ ~~b =~~

e.g., temp = a
a = b
b = temp

⟹ a is register.
b is register.
temp is operand Register.

*. FLAG Register :

→ It has various flags

→ Each flag gives some states about current result.

→ Ex : Sign flag - tells result is negative or positive
Zero flag - tells results is zero or not
Carry flag - tells Carry is not.

→ Flags keeps on changing based on the status of the result generated by ALU

AX - Accumulator - take parts in ALU
BX - Base Register -
CX - Used as Counter Register - loops, jumps
DX - Used to point to data in I/O operations

*. Pointer & Index Register :

Segment	effset Register	Function
CS	IP	P. Address of next instruction
DS	BX, DI, SI	P. Address of Data
SS	SP, BP	P. Address in stack
ES	BX, DI, SI	P. Address of destination data (for string operations)

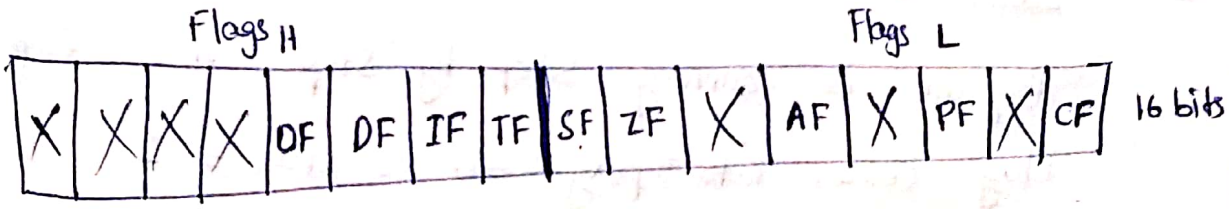
Example : MOV AH, [SI]

Means move the byte stored in memory location whose address is contained in register SI to register AH.

*. FLAG REGISTER :

→ After performing arthemetic operations in ALU, the result is stored in the Destination Register & status of the result is given by Flag Register.

→ It is of 16 bits → 9 bits are in use
→ remaing 7 bits are X



X - bits marked are not defined.

① OF - overflow : (may be status Flag sometimes)

When overflow occurs after ALU operation, then OF changes from 0 to 1

② DF - direction flag :

In 100 different memory locations, 100 different data are there and we are processing the data from 100th location to 1st, or vice-versa & how we are processing the data.

			Status :
higher memory location	to	lower memory location	→ 1
lower memory location	to	higher memory location	→ 0

③ Interrupt Flag : IF

When specific obstacle to program or process is done, then Interrupt Flag status changes.

If Interrupt ⇒ Flag status = 1

④ Trap Flag : TF

Program is running step by step, then Trap Flag comes into play ⇒ Status = 1

CONTROL FLAGS - Flags H (bits)

⑤ SF - Signed Flag : → & compare instructions

When the result is (after ALU operation)

negative → status = 1 (set)

positive → status = 0 (reset)

⑥ ZF - Zero Flag :

When the result is

Zero → status = 1

non-Zero → status = 0

⑦ AF - Auxiliary Carry Flag :

When you are performing operations, when carry is generated from lower bit to higher bit,

Right to left carry → status = 1

⑧ PF - Parity Flag :

No. of 1's in result are even then

Status of parity Flag = 1 ⇒ 1's are even → 1

1's are odd → 0

⑨ CF - Carry Flag :

If Carry is generated from MSB ⇒ status = 1

STATUS FLAGS - Flags L (bits)

EXAMPLES : (FAT Question)

Instruction	C	O	S	Z	P	operation Status
MOV BL, 03H	-	-	-	-	-	MOV doesn't affect
ADD BL, 02H	0	0	0	0	1	5 has even no. of 1's
SUB BL, 05H	0	0	0	1	1	Result is '0'
SUB BL, 02H	0	0	1	0	0	Result is negative
MOV DH, 02H	0	0	1	0	0	MOV doesn't affect
ADD DH, FFH	1	0	0	0	0	Result is 01H with Carry
MOV AL, 04H	1	0	0	0	0	MOV doesn't affect flag
ADD AL, 7FH	1	1	1	0	0	Generates Carry, Sign and overflow.

Given in the Question

Predict all these

* INSTRUCTION SET : (6 types)

1. Data Transfer
2. Arithmetic
3. Logical
4. String Manipulation
5. process control
6. Control transfer

① Data Transfer :

→ Instructions that are used to transfer data/address in to registers, memory locations and I/O ports.

→ The size should be either byte or word

(i) MOV reg2/mem, reg1

MOV reg2, reg1 — (reg2) ← (reg1)

MOV mem, reg1 — (mem) ← (reg1)

MOV reg2, mem — (reg2) ← (mem)

(ii) MOV rg/mem, data

MOV reg, data — (reg) ← data

MOV mem, data — (mem) ← data

data will be transfered to the Physical address, $PA = DS \times 10_H + [offset]$

(iii) XCHG (exchange) : Similar to swapping of two values using temp variable → OPERAND Reg

XCHG reg2/mem, reg1

XCHG reg2, reg1 — (reg2) ⟷ (reg1)

XCHG mem, reg1 — (mem) ⟷ (reg1)

(iv) PUSH reg16/mem

PUSH reg16

$(SP) \leftarrow (SP) - 2$

$MA_S = (SS) \times 16_{10} + SP$

$(MA_S ; MA_S + 1) \leftarrow$ reg16

PUSH mem

$(SP) \leftarrow (SP) - 2$

$MA_S = (SS) \times 16_{10} + SP$

$(MA_S ; MA_S + 1) \leftarrow$ (mem)

v. POP reg16 | mem

POP reg16

$MA_S = (SS) \times 16 + SP$

$(reg16) \leftarrow (MA_S ; MA_{S+1})$

$SP \leftarrow SP + 2$

POP mem

$MA_S = (SS) \times 16_{10} + SP$

$mem \leftarrow (MA_S ; MA_{S+1})$

$(SP) \leftarrow (SP) + 2$

vi IN A, [DX]

IN AL, [DX]

$PORT_{addr} = (DX)$

$(AL) \leftarrow (PORT)$

IN AX, [DX]

$PORT_{addr} = (DX)$

$(AX) \leftarrow (PORT)$

vii. IN A, addr 8

IN AL, addr 8

$(AL) \leftarrow (addr\ 8)$

IN AX, addr 8

$(AX) \leftarrow (addr\ 8)$

viii. OUT [DX], A

OUT [DX], AL

$PORT_{addr} = (DX)$

$PORT \leftarrow AL$

OUT [DX], AX

$(PORT)_{addr} = (DX)$

$PORT \leftarrow AX$

ix. OUT addr 8, A

OUT addr8, AL — addr 8 ← AL

OUT addr8, AX — addr 8 ← AX

⑧ Arithmetic Instructions :

ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP

x. ADD reg2, reg1 — reg2 ← reg2 + reg1

xi. SUB reg2, reg1 — reg2 ← reg2 − reg1

xii. ADC reg2, reg1 — reg2 ← reg2 + reg1 + CF

xiii. SBB reg2, reg1 — reg2 ← reg2 − reg1 − CF

xiv. INC reg8 — (reg8) ← (reg8) + 1

xv. DEC reg8 — (reg8) ← (reg8) − 1

xvi. MUL reg

For byte : AX ← AL × (reg18)

For word : (DX)(AX) ← (AX) × reg16

For byte : (AX) ← (AL) × (mem8)

xvii. IMUL reg

For byte : AX ← AL × (reg8)

For word : (DX)(AX) ← (AX) · (reg16)

xviii. IMUL mem

For byte : (AX) ← (AX) · (mem8)

xix. DIV reg

AL ← AX ÷ (reg8) Quotient

(AH) ← Remainder

XX. CMP reg2/mem , reg1/mem

CMP reg2 , reg1 → result is stored in operand

Modify flags ← reg2 − reg1

If reg2 > reg1 , CF = 0
ZF = 0
SF = 0
} Flags

If reg2 < reg1 , CF = 1
ZF = 0
SF = 1
} Flags

If reg2 = reg1 , CF = 0
ZF = 1
SF = 0

All these will be similar to

CMP reg2 , mem

CMP mem , reg1

reg 8 : 8 bits register

reg 16 : 16 bits register

mem : memory address

reg1 : register 1

reg2 : register 2

*. EXAMPLES, PROGRAMS – 8086 :

(1) Sum of N Consecutive Numbers :

```
        MOV   SI, 2000 H
        MOV   CL, [SI]
        MOV   AL, 00H
        MOV   BL, 01 H      → store
L1 :    ADD   AL, BL        → add BL to AL & store in AL
```

SI - register which stores offset addr

INC BL → increase BL value by 1.

DEC CL → derease value of n by 1. (checks content of CL with zero flag)

JNZ L1 → jump out of loop after n=0

MOV DI, 2002 H

MOV [DI], AX

HLT → halt the program

---

③ Logical Instructions

xxi. AND reg2/mem, reg1/mem :

AND reg2, reg1 (reg2) ← (reg2) & (reg1)

AND reg2, mem (reg2) ← (reg2) & (mem)

AND mem, reg1 (mem) ← (mem) & (reg1)

xxii. Similarly OR, XOR

xxiii. TEST reg2/mem, reg1/mem :

TEST reg2, reg1 modify flags ← (reg2) & (reg1)

TEST reg2, mem modify flags ← (reg2) & (mem)

TEST mem, reg1 modify flags ← (mem) & (reg1)

xxiv. SHR - shift right

SHR reg

a. SHR reg 1

b. SHR reg, CL

SHR mem

a. SHR mem, 1

b. SHR mem, CL

xxv. SHL – Shift left

SHL reg, 1 (or) SAL

xxvi. RCR – Rotate through Carry , (ROR, ROL)

RCR reg

a- RCR reg, 1

b. RCR reg, CL

RCR mem

a. RCR mem, 1

b. RCR mem, CL

xxvii. ROR, ROL

---

④ String Manipulation Instructions :

⇒ DF = 0 → SI & DI are incremented by 1 for byte and 2 for word.

⇒ DF = 1 → SI and DI are decremented by 1 for byte and 2 for word.

It is sequence of bytes (8 bits) & words (16 bits)

- REPZ / REPE

(Repeat string instruction until ZF = 0)

while CX ≠ 0 & ZF = 1

repeat CX ← CX - 1

- REPNZ / REPNE

(Repeat string instruction until ZF = 1)

while CX ≠ 0 & ZF = 0

repeat CX ← CX - 1

Example 1 :

```
MOV   CL , 08H      ⟶  1)
MOV   SI , 1400H    ⟶  2)
MOV   DI , 1450H    ⟶  3)
1: LODSB            ⟶  loads string Byte (6)
   MOV  [DI] , AL   ⟶  shift from AL to DI
   INC  DI          ⟶  4) Increament DI
   DEC  CL          ⟶  Decrement CL & check
   JNZ  L1          ⟶  5) CL is reached 0 , if not loop continues
   HLT              ⟶
```

1) 8 No. of bytes are there is string.
2) Source address = 1400 link to DS
3) Destination address = 1450 link to ES
4) SI auto increaments in string Instructment
5) loop runs for 8 times (CL = 8)
6) AL changes to 11
7) JNE 08h → it is address of string byte.
8) JNZ – Jump if not Zero. : Till it not coming to zero, Loop repeats & checks. Zero flag.

Example 2 :

REPNZ – Repeat till not Zero
MOVSB – move string Byte

Transfer 8 bytes from DS to ES

```
MOV   CL , 08H
MOV   SI , 1400H
MOV   DI , 1450H
REPNZ MOVSB
HLT
```

Example 3 :

```
MOV    CL , 08H
MOV    SI ,  1400H
MOV    DI ,  1450 H
REP    CMPSB        — compare string Byte until
                      and repeat
HLT                 — halt
```

Here, zero flag bit = from 0 to 8 it is 0
and when 08H is reached, Zero flag bit = 1

---

* points to remember :

CMP AX , BX ⟶ result is stored in operand register not AX. We are just modifying flags.

MUL BX ⟶ higher order is stored in DX.
lower order is stored in AX.

Always start with 0 & end with h when you write data in 8086 emulator. (0FDh)

---

* (5) PROCESS CONTROL INSTRUCTION :

STC	Set CF ← 1
CLC	Clear CF ← 0
CMC	Complement Carry CF

STD	Set Direction Flag DF ← 1
CLD	Clear Direction Flag DF ← 0
STI	Set Interrupt enable Flag IF ← 1
CLI	clear Interrupt enable flag IF ← 0
NOP	No operation
HLT	Halt after interupt is set.

---

(6) CONTROL TRANSFER INSTRUCTIONS :

- Transfer the control to a specific destination and or target instruction.
- Donot affect flags.

* 8086 Unconditional transfers.

	Mnemonics	Explanation
1.	CALL reg/mem/disp 16	Call subroutine (functions in c programming)
2.	RET	Return from subroutine
3.	JMP reg/mem/disp8/disp16	Unconditional Jump
4.	JE - Jump if equal	5. JZ - Jump is zero.
6.	JNE → Jump if not equal	Jump if not zero
7.	JNZ ↗	
8.	JGE - Jump if greater than or equal	
9.	JNL - Jump if not less	

(remaining check in laptop)

Example :

```
0000 MOV   AL, 10
0002 MOV   BL, 05
0004 STC                  ---> Set Carry , Carry Flag = 1
0005 CALL  000B           ---> memory location address of Subroutine here IP = 000B
0008 ADD   AL, BL                    (offset address)
000A HLT

000B
     MOV   CL, 10
     L1 :  DEC CL
     JNZ   L1
     RET        - return instruction
```

*. ADDRESSING MODES :

- It is way of locating data or operands, the types of operands used and the way they are accessed for executing an instruction.

- Based on flow instruction : (8086) — 2 basic type of Addressing mode
  1. Sequential control flow instruction → Arithmetic, logical, data transfer, processor control
  2. Control transfer instructions → INT, CALL, RET, JUMP

12 types of Addressing Modes :

1. Register addressing mode
2. Immediate addressing mode

① Register Addressing modes :

- Both operands are specified by registers.

e.g., MOV AX, BX

- A segment to segment movement of data is not allowed. (like MOV SS, DS ~~xxx~~)
- Both source & destination registers should be same size.

② Immediate Addressing mode :

- E.g., : MOV AL, 22H
- Source - data , destination - Register.
- Segment to segment Reg is not allowed

③ ~~Reg~~ Direct addressing Mode :

- E.g., : MOV AL, [1234H] (offset address, IMP *)

MOV BX, [5000H] → BL = [5000H], BH = PA+1

- From Memory location to register direct data will be transferred.

④ Register Indirect addressing mode :

- Eg : MOV AX, [BX]

⇓

MOV BX, 1234
MOV AX, [BX]

is similar to

MOV AX, [1234]

- DS is default segment when the registers BX, DI & SI are used.
- SS is default Segment when register BP is used

⑤ Register Relative addressing mode :

- Eg : MOV AX, 1000H [BX]

⇓

MOV BX, 1234 → Calculate P.A
MOV AX, 1000H [BX]

⇓

Physical Address = $(DS \times 10_H + \overset{1234}{BX}) + (1000)$

↳ AL & (P.A + 1) = AH

⑥ Indexed addressing mode :

- Eg : MOV AX, [SI]

⑦ Based - Indexed addressing mode

- Eg : MOV AX, [BX][DI]

Physical Address = DS × 10_H + DI + BX

↳ in AL & (P.A + 1) = AH

⑧ Relative Based Index

- Eg : MOV AX, 2000H [BX][DI]

Physical Address = DS × 10_H + DI + BX + (2000)

↳ in AL & (P.A + 1) = AH

All these 8 addressing modes belong to Sequential Control flow instruction.

---

⑨ CONTROL TRANSFER ADDRESSING MODES :

Intersegment - If the address locations to which the control is to be transfered lies in different segment other than the current one, the mode is called Intersegment mode

Intrasegment -

If the destination lies in the same segment, the mode is called Intrasegment.

(9) Intrasegment Direct mode :

- The manner in which operand (data) is given in an Instruction. - Addressing mode.
- Eg: JUMP 0002 (IP = 0002)

Example

MOV AL, 05H
MOV BL, 35H
JMP 0002H

↓

Intrasegment Direct mode

(10) Intrasegment Indirect mode :

- Eg:

MOV AL, 05H
MOV BL, 35H
MOV BX, 0002H
JMP BX (indirectly it is written)

↓

Intrasegment Indirect mode

- The value of BX is copied into IP with CS value unchg.

IMP (remember) Whether it is Intrasegment (direct/indirect addressing mode) the Code Segment CS value will not change.

(remember) Whereas the Intersegment, the CS value will change because the current instruction will be present in one code segment and next instruction will be there in different code segment.

↓ Example

e.g.,

MOV AL, 05H
MOV BL, 35H
JMP 2000H : 3000H (no space)

⇒ loads IP ← 3000H
new CS ← 2000H

↓

from 23000 = $PA = CS \times 10_H + IP$

(ii) Intersegment indirect mode :

- Eg : JMP [5000 H] , or JMP [BX or SI or DI]

$IP_L \leftarrow [5000_H]$ and $IP_H \leftarrow [5001_H]$

$CS_L \leftarrow [5002_H]$ and $CS_H \leftarrow [5003_H]$

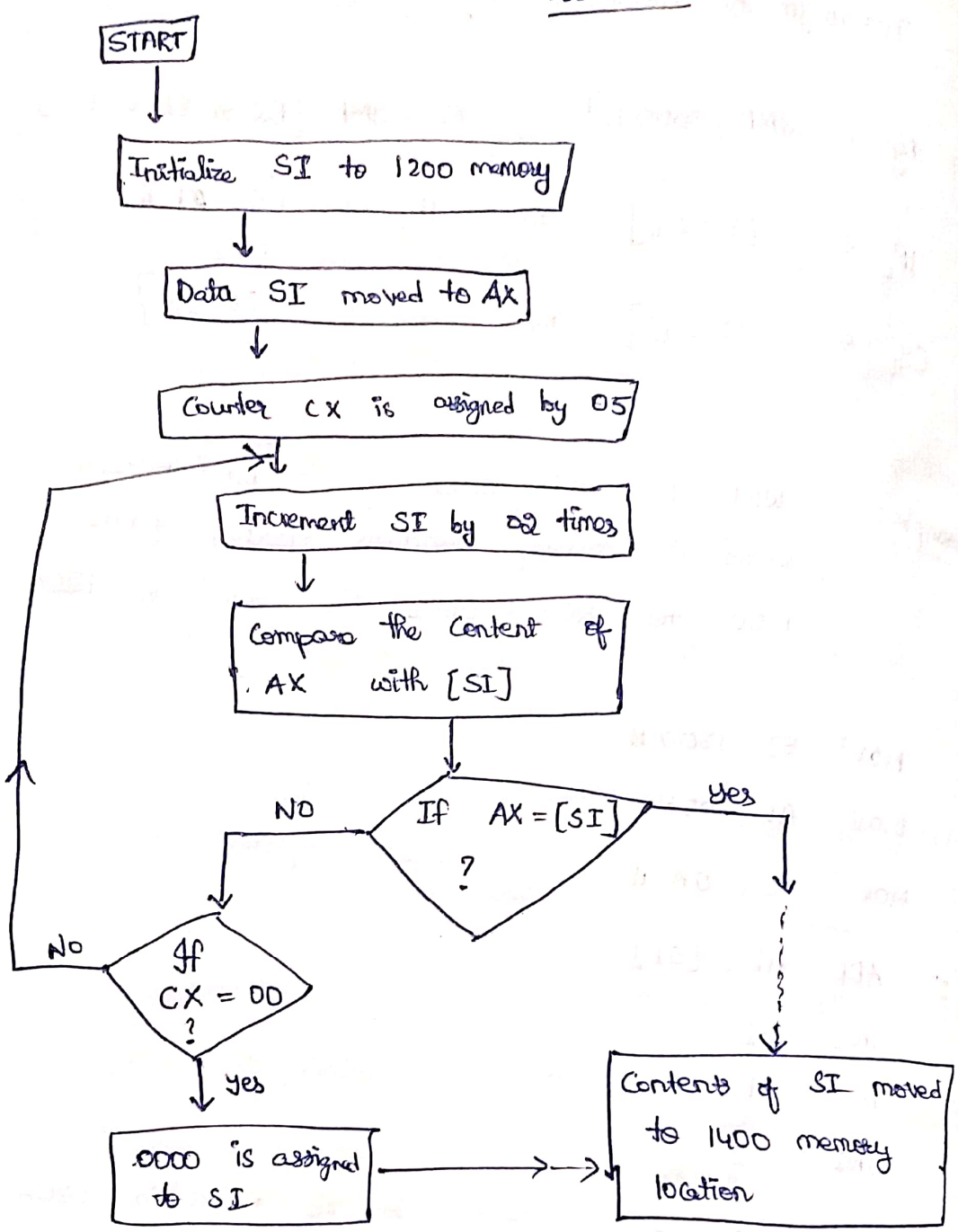
Example : WAP to add series of 8 bit numbers stored in memory locations starting from 1200 to 1209 . Store the result in 120A

```
    MOV   SI, 1200 H
    MOV   AL, 00H.
    MOV   CL, 0A H      → Counter register
L1: ADD   AL, [SI]
    INC   SI
    DEC   CL
    JNZ   L1
    MOV   [SI], AL      → Store the result in 120A
    HLT
```

Example 2 : Searching the existence of certain data in given data Array

10 16 bit numbers = 20 memory locations

FLOWCHART



Memory Address	Data In
1200	AB96
1202	89CD
1204	AB96
1206	4EDF
1208	9197
120A	9600

solution

Mnemonics	Label	Memory Address
MOV SI, 1200H		
MOV AX, [SI]		
MOV CX, 0005H		
ADD SI, 0002 → INC SI INC SI	GG	
CMP AX, [SI]		
JE XX → label		
DEC CX		
JNZ GG → label		
MOV SI, 0000 H		
MOV [1400], SI	XX	
HLT		

*. Example : To seperate odd and even number in a given data array.

Logic :

MSB LSB

0000 0110 → = 6_{10}

MSB LSB

0000 0101 → = 5_{10}

If LSB = 0 → even binary number

MSB = anything

LSB = 1 → odd number binary

MSB = anything

Address	Program		Explanation
	MOV	CL, 06	Set counter in CL register
	MOV	SI , 1600	Set source Index as 1600
	MOV	DI , 1500	Set Destination Index as memory address 1500
Loop1	MOV	AL , [SI]	Load data from source memory
	ROR	AL , 01	Rotate AL once to right
	JNC	Loop1	If bit is one jump to loop1
	ROL	AL , 01	Rotate AL once to left
	MOV	[DI] , AL	Move result to Destination
	INC	SI	SI is increased to 1
	INC	DI	Increament destination index
	DEC	CL	Decrement the count
	JNZ	Loop1	jump if CL not 0 to loop1
	HLT		halt/exit the program

INPUT	1600	2
	1601	5
	1602	7
	1603	6
	1604	12
	1605	15

OUTPUT	1500	5
	1501	7
	1503	15

*. WAP to find the greatest number in given set of data array.

Mnemonics	Comment
MOV SI, 1500H	SI ← 1500
MOV CL, [SI]	CL ← [SI]
INC SI	SI ← SI + 1
MOV AL, [SI]	AL ← [SI]
DEC CL	CL ← CL - 1
INC SI	SI ← SI + 1
L1: CMP AL, [SI]	AL - [SI]
JNC L2	Jump to L2 if CY=0
MOV AL, [SI]	AL ← [SI]
L2: INC SI	SI ← SI + 1
LOOP L1	CX ← CX - 1 & jump to L1 if CX not 0
	AL → [1600]

for smaller number

JC L2

DEC CX MOV [1600], AL
JNZ L1 HLT

Memory address (offset)	Data (Input)
1500	05
1501	25
1502	35
1503	20
1504	30
1505	15

Memory (offset)	output
1600	35

*. WAP to sort 8 bit data array in A.O. The array consists of 5 numbers starting from location 3000H : 4000H

Address	Initial data	Ext loop1	Ext loop2	3	4
4000	55	45	35	25	15
4001	45	35	25	15	25
4002	35	25	15	35	35
4003	25	15	45	45	45
4004	15	55	55	55	55

3000 H ← Data Segment

4000 H ← offset [SI]

```
        MOV   AX , 3000 H
        MOV   DS , AX
        MOV   CH , 04 H
L3:     MOV   CL , 04H
        MOV   SI , 4000H
L2:     MOV   AL , [SI]
        MOV   AH , [SI]+[01 H]
        CMP
        MOV   AL , AH
        JB    L1
        JZ    L1
        MOV   [SI+1] , AL
        MOV   [SI] , AH
L1:     INC   SI
        DEC   CL
        JNZ   L2
        DEC   CH
        JNZ   L3
        HLT
```

1. WAP to reverse a string of 10 bytes using stack. check whether the string is palindrome or not. If it is palindrome display FFH on display unit whose address is 52 H else, display 00 H.

SI		STACK		DI	
2300	11	FFF1	AA	2500	AA
2301	22	FFF2	99	2501	99
2302	33	FFF3	88	2502	88
2303	44	FFF4	77	2503	77
2304	55	FFF5	66	2504	66
2305	66	FFF6	55	2505	55
2306	77	FFF7	44	2506	44
2307	88	FFF8	33	2507	33
2308	99	FFF9	22	2508	22
2309	AA	FFFA	11	2509	11

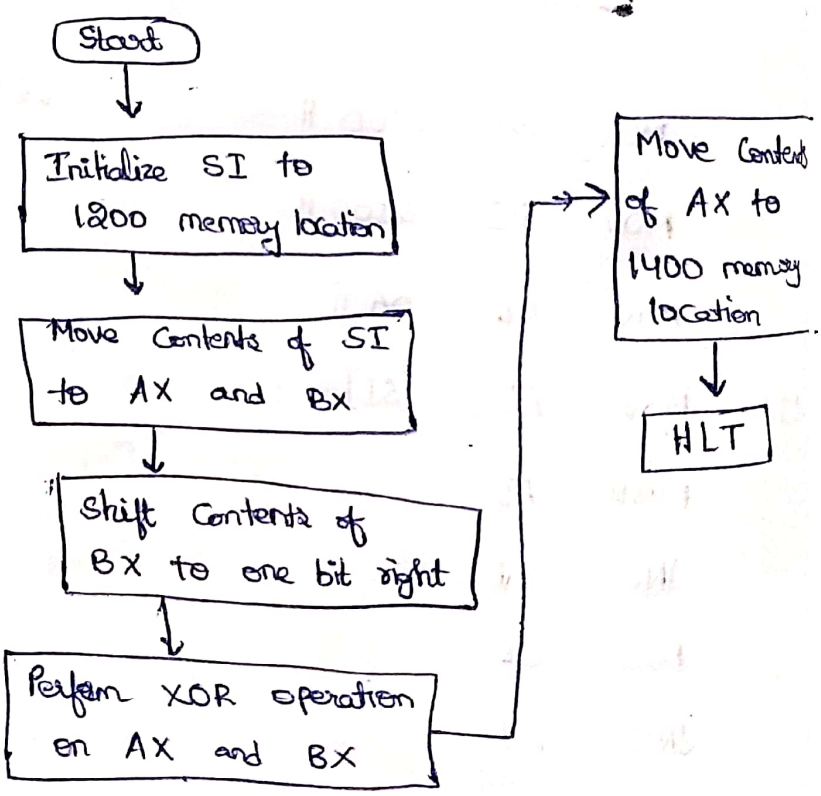
```
    MOV   SI, 2300 H
    MOV   DI, 2500 H
    MOV   CL , 0A H
L1: MOV   AL , [SI]
    PUSH  AL
    INC   SI
    DEC   CL
    JNZ   L1
    MOV   CL, 0AH
L2: POP   AL
    MOV   [DI] ; AL
    INC   DI
    DEC   CL
    JNZ   L2
```

```
        MOV   SI , 2300 H
        MOV   DI , 2500 H
        MOV   CL , 0A H
        CLD   ------> for comparing both the locations and clearing direction flag
REPZ    CMPSB
        JNZ   L3
        MOV   AL , FF H
        OUT   52 H
        JMP   Exit
L8 :    MOV   AL , 00H
        OUT   52 H
```

---

*- WAP (ALP) to Convert Binary number to gray code.

Flowchart :



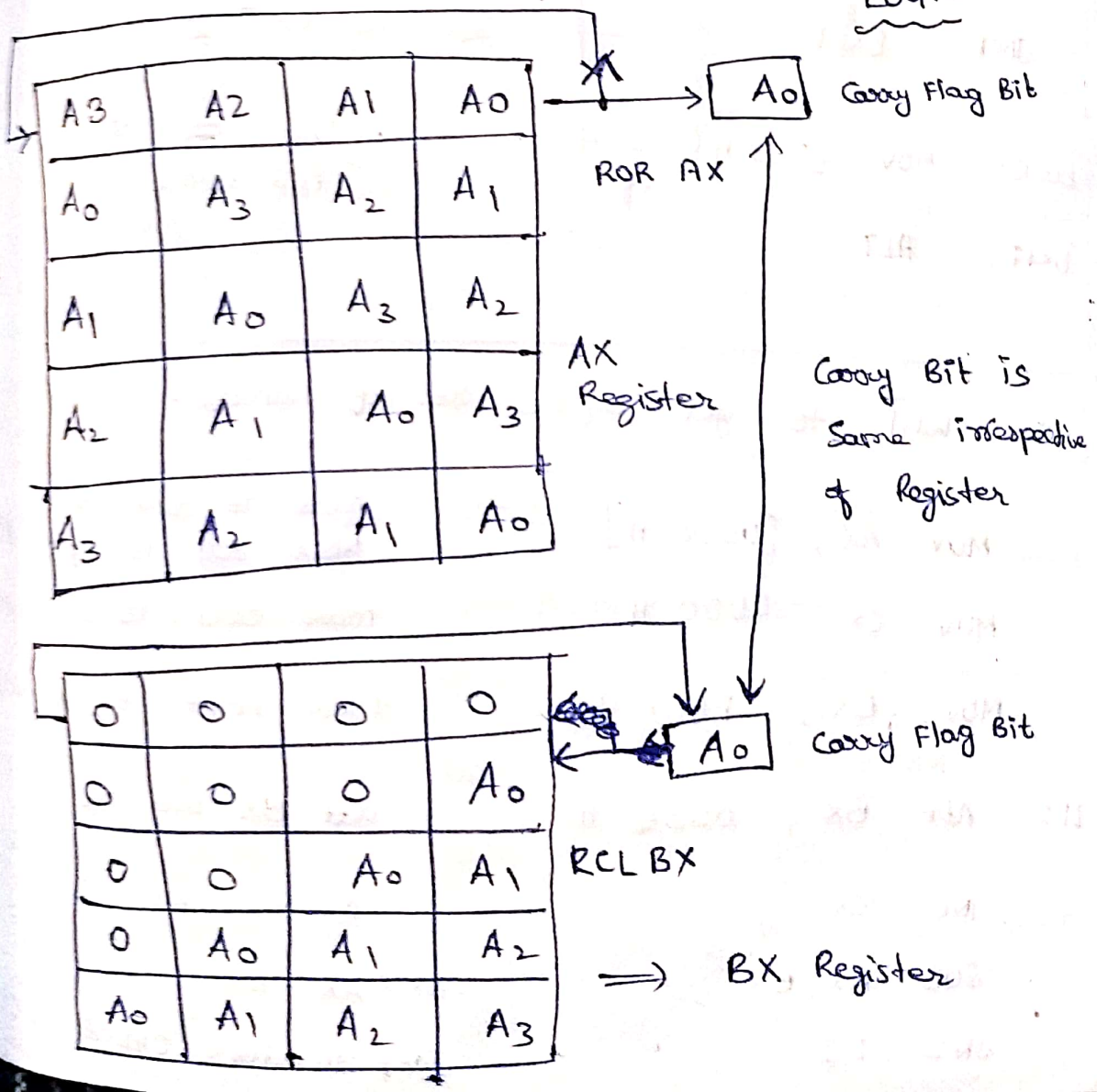
Program :

```
MOV  SI , 1200
MOV  AX , [SI]
MOV  BX , [SI]
SHR  BX , 01        ⟶  check flow chart
XOR  AX , BX
MOV  [1400] , AX
HLT
```

---

* Program to check number for bit wise palindrome. If palindrome place FFH at 2500 H or place 00H at 2500 H

LOGIC :



10 in Hexdecimal = 16 in Binary

$10_H = 16_2$

Program :

```
    MOV  AX, [2300 H]
    MOV  CL, 10H      ⟶ we need to shift 16 bits (16 times)
UP: ROR  AX, 1
    RCL  BX, 1
    DEC  CL
    JNZ  UP           ⟶ 16 times loop process
    CMP  AX, BX
    JNZ  DOWN         ⟶ if no zero go to down label
    MOV  [2500H], FFH ⟶ Declare as Palindrome
    JMP  EXIT         ⟶ Jump to EXIT label
DOWN: MOV [2500H], 00H ⟶ Declare as not a Palindrome
EXIT: HLT
```

---

* WAP to get Square root of number.

```
    MOV  AX, [0500 H]   move to data from offset 500 to register AX
    MOV  CX, 0000 H     move 0000 to register CX
    MOV  BX, FFFF H     move FFFF to register BX
L1: ADD  BX, 0002 H     add BX and 02
    INC  CX             CX = CX + 1
    SUB  AX, BX         AX = AX - BX
    JNZ  L1             Jump to address 040A if ZF≠0
```

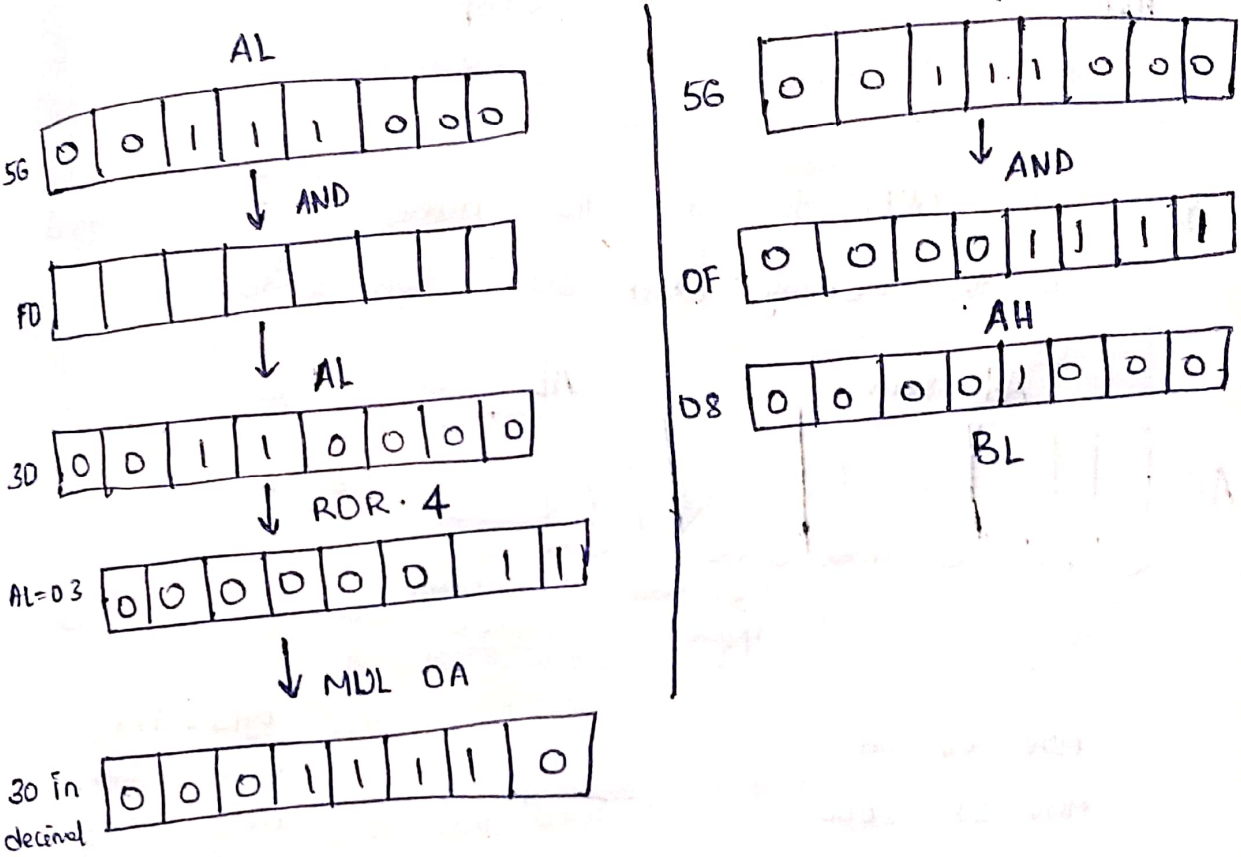
```
FFFF
0010
-----
1 0001
```

```
MOV [0600], CX      Store the contents of CX to offset 600
HLT
```

---

*. Conversion of BCD to Hexadecimal : (ALP)

$(56)_{10} = \underbrace{0011}_{3_H}\,\underbrace{1000}_{8_H} \rightarrow (38)_H$



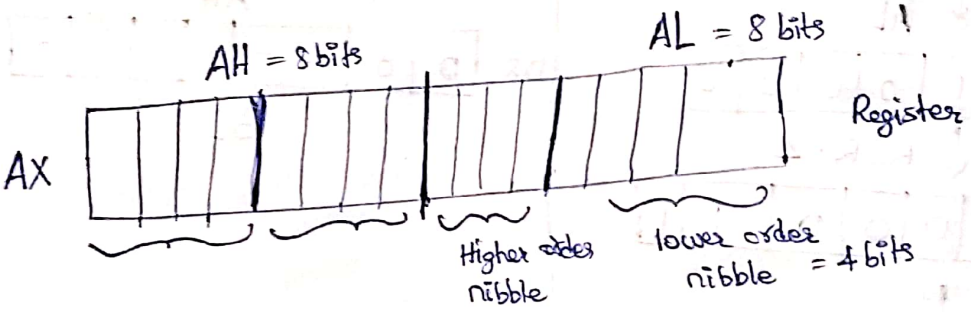
Program :

```
MOV SI, 1600    ⟶  Set Source index as 1600
MOV DI, 1500    ⟶  Set Destination index as 1500 memory address
MOV AL, [SI]    ⟶  Load BCD number into AL register
MOV AH, AL      ⟶  Move content of AL to AH
AND AH, OF      ⟶  Mask MSD of BCD number
MOV BL, AH      ⟶  Save LSD in BL register
AND AL, FO      ⟶  Mask LSD of BCD number
```

```
MOV   CL , 04   ⟶  Load 04 to Counter
ROR   AL , CL   ⟶  rotate AL by counter time
MOV   BH , 0A   ⟶  move 0A to BH register
MUL   BH        ⟶  Multiply BH register
ADD   AL , BL   ⟶  Add AL, BL
MOV   [DI] , AL ⟶  Move result to Destination
HLT             ⟶  Stop
```

---

* WAP (ALP) to swap the nibbles of 10 data stored in the memory which starts from 2000.



bytes = 8 bits
nibble = 4 bits
bit = 0 or 1
word = 16 bits

```
      MOV  CL, 0A
      MOV  SI , 2000
      MOV  DI , 3000
L1:   MOV  AL , [SI]
      ROR  AL , 04
      MOV  [DI] , AL
      INC  SI
      INC  DI
      JNZ  L1
```

* Consider all the 8 bits of an output port having address 54H are connected to 8 LEDs. Write a program to blink all LEDs ON and OFF with time gap of 5 seconds. Consider the time taken by any instruction to be executed is 1 second.

Port Address = PA: 54H

data = 1 → ON
data = 0 → OFF

$(FF)_{16} = (1111\ 1111)_2$ → all LED's are ON

$(00)_{16} = (0000\ 0000)_2$ → all LED's are OFF

```
L1 :    MOV  AL , FF H
        OUT  54 H          ---> output port
        CALL DELAY         ---> DELAY is subroutine which takes 5 seconds to execute
        MOV  AL , 00H
        CALL DELAY
        OUT  54 H
        CALL DELAY
        JMP  L1

DELAY : MOV  CL , 02 H     ---> 1 second
L2 :    DEC  CL            ---> 2 second
        JNZ  L2            ---> 2 second
        RET
```

$(AA)_{16} = (1010\ 1010)_2$ → alternative LED's

$(55)_{16} = (0101\ 0101)_2$ → alternative LED's

*. Write a program to add the contents of memory location 0500 H register BX & CX. Add immediate byte 05 H to the data residing in memory location, whose address is computed using offset = 0600H. store the result of the addition in 0700H. Assume data segment's starting physical address is 20000H

```
MOV   AX, 2000 H
MOV   DS, AX
ADD   BX, [0500 H]
ADD   CX, BX
MOV   DL, 05 H
ADD   DL, [0600]
MOV   [0700], DL
HLT
```

~~ADD [0600], 05H~~ wrong way of writing

---

*. WAP (ALP) to get Factorial of 10 numbers stored from the starting location 4000 H : 1000 H The results should be stored in 4000 H : 2000 H

```
MOV   AX, 4000 H
MOV   DS, AX
MOV   SI, 1000 H
MOV   DI, 2000 H
MOV   CL, 0A H
MOV   AL, 01 H
Next: MOV   BL, [SI]
Look: MUL   BL
      DEC   BL
      JNZ   LOOK
      MOV   [DI], AX
```

```
MOV  [DI], AX
INC  SI
INC  DI
LOOP NEXT
```

→ Decrement counter register by 1 and check it reached zero

↓

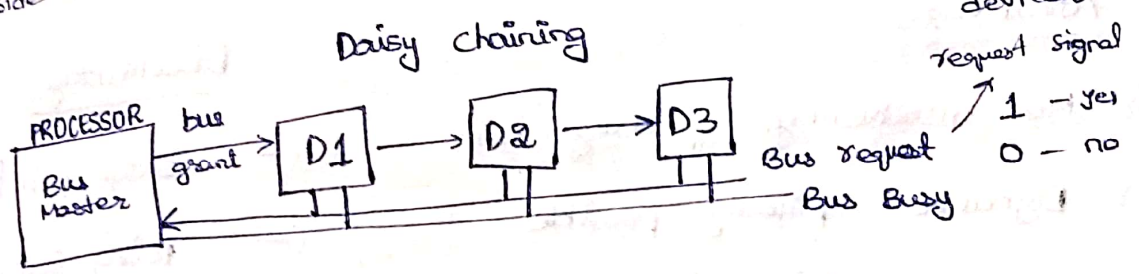
LOOP ← CX = CX − 1 & JNZ NEXT

---

*. INTERRUPT : (Data Transfer Techniques) → ① ② ③

- The devices which interrupt are pheripheral devices, Interrupt devices
- Interrupt based by priority → Interup is nested : Interrupt inside a Interrupt by pheripheral devices.

① Daisy chaining



D_1 D_2 D_3 are devices

request signal

1 – yes

0 – no

Steps :

1. If bus not busy, makes bus request by either D_1 D_2 D_3
2. Master (processor) activates bus grant to which sent request
3. If device gets bus grant, mark bus busy.

(+) Simple

(+) Only three extra bus lines

(−) Hardwired priority

(−) Poor fault tolerance

Advantages → applicable when LESS no. of devices

Advantages

Disadvantages

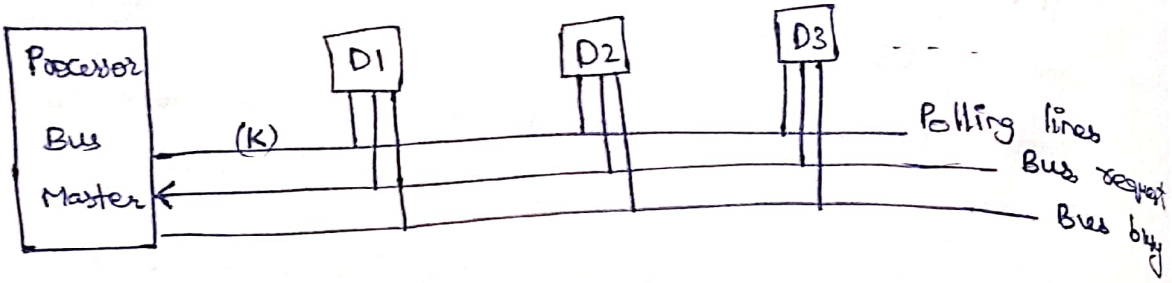
Disadvantages

$D_1 > D_2 > D_3$ (priority of bus request)

↓

If either of $D_1 D_2 D_3$ are fault then the next device can never get ~~fault~~ signal / grant.

(2) POLLING



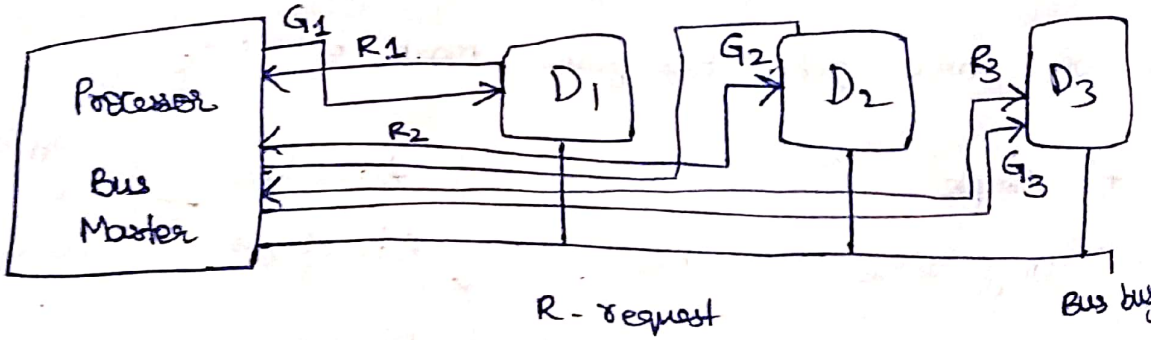
Steps :

1. If bus not busy, make bus request
2. Master polls by placing device ID on polling lines (K) (master decides priority)
3. If device gets bus grant, mark bus busy.

Advantages	Disadvantages
(+) No disadvantage of daisy chain	(−) Extra poll lines
(+) Dyramic priority possible	(−) Polling Delay
(+) Extra poll lines / More control lines (−)	

(3) INDEPENDENT REQUEST

n divices
2n lines



R - request
G - grant

Steps :

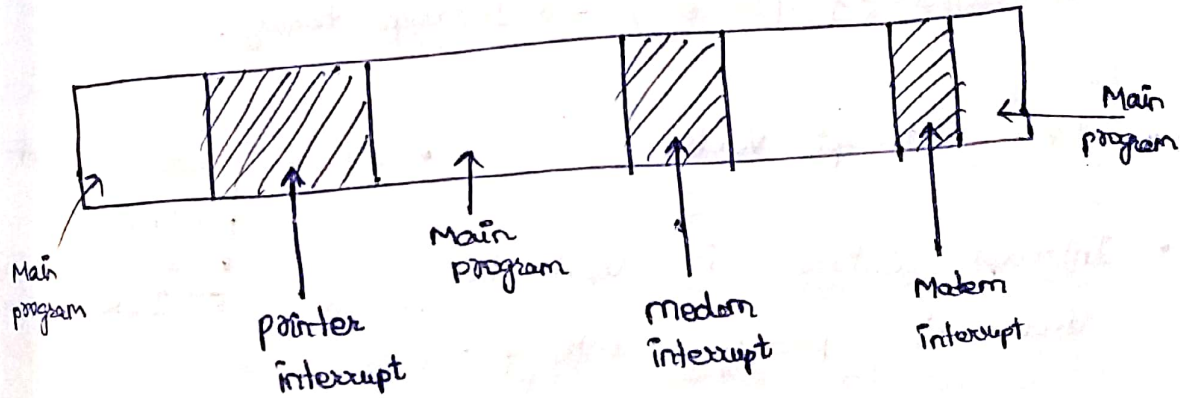
1. If bus not busy, make bus request
2. Master decides who to grant access and indicates through grant line
3. If device gets bus grant, mark bus busy.

The purpose of Interrupts :

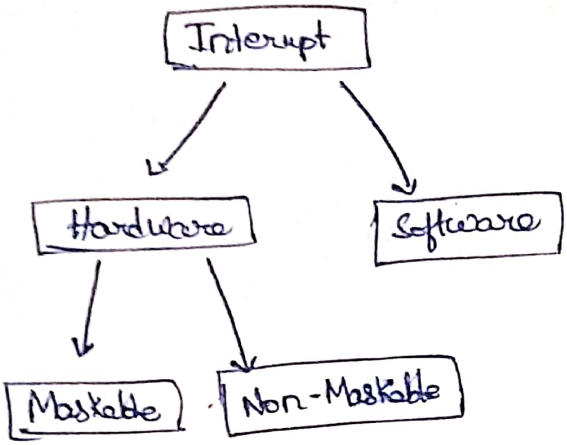
* I/O organizations be 3 types

1. Programmed I/O (Polling or Daisy chaining)
2. Interrupt driven I/O
3. DMA

- Interrupts are useful when interfacing I/O devices with low data-transfer rates, like a keyboard or a mouse, in which case polling the devices wastes valuable processing time.
- The peripheral interrupts the normal application execution, requesting to send or receive data.
- The processor jumps to special program called Interrupt Service Routine to service the peripheral.
- After the processor services the peripheral, the execution of the interrupted program continues.



*. TYPES OF INTERRUPT



- Hardware – I/O devices
  Software – programmer
- Interrupt instructions can be written as INT (0 – 255)

- All software interrupts are Maskable.
- Maskable – The interrupts that can be disabled by the processor/programmer when occured.
- NMI (non-maskable) needed to be serviced. Power Failure is non-maskable.
- Vectored interrupt – When Interrupt Service Routine is executed (from memory address PA = DS * 10 + SI). This vector table memory address information is stored.
- IVT – Interrupt Vector Table.
- More than 1 Interrupts, nested Interrupts (priority has to be obeyed by processor) → Interrupt Priority.
- IV – Interrupt Vector
- IH – Interrupt Handler
  - → Interrupt Service Routine
  - → ISR.
- Interrupt Controller is used to decide the priority of Interrupt.

* BASIC TERMINOLIGIES :

1. Interrupt pins – 40 pins (basic 3 pins)
   - INTR – Interrupt request – which requests ⇄ 0 ✗ / 1 ✓
   - INTA – Interrupt Acknowledgement – when 0 signal ✓ / when 1 signal ✗ (↕ transfer of signal)
   - NMI – Non-maskable Interrupt – Power Failure
   - Hardware Interrupts (To store memory by registers)

2. ISR – Interrupt Service Routine is also called
   Interrupt handler : Code used for handling specific interrupt

3. IM – Interrupt Masking – Ignoring (disabling an interrupt)

4. NMI – non-maskable Interrupt – need to be serviced.

* INTERRUPT PROCESS FLOW :

- HARDWARE INTERRUPT :

→ INTR : Interrupt Request

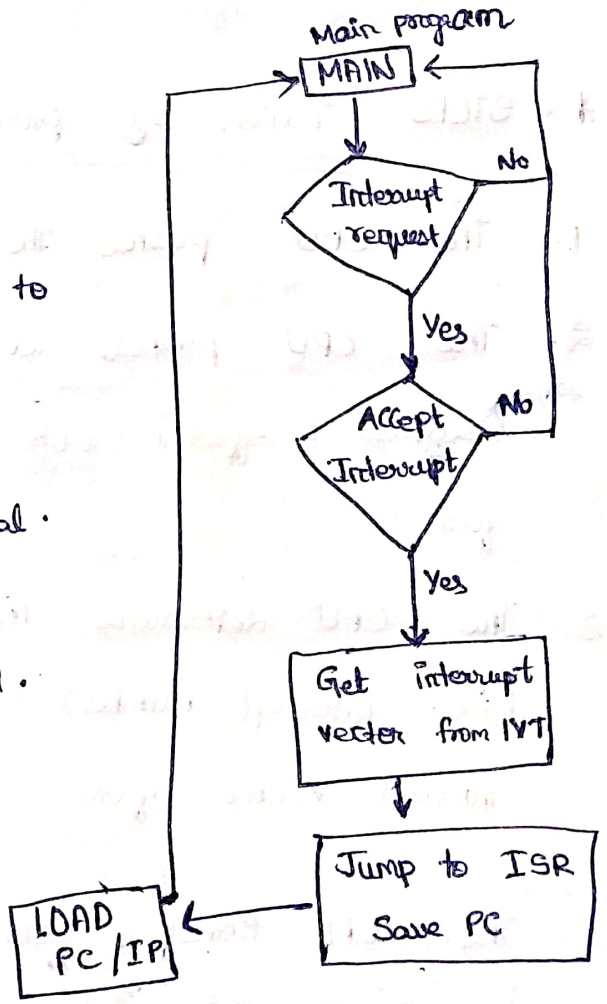
→ Activated by peripheral device to interrupt processor.

→ INTA – Level triggered.
– Active with 0 signal.

→ INTR – level triggered
– Active with 1 signal.

→ NMI – Edge triggered
– Active with 0 to 1 transition

–



both are same { PC – Program Counter
IP – Instruction Pointer

③ NMI : Edge Triggered. Active with 0 to 1 transition

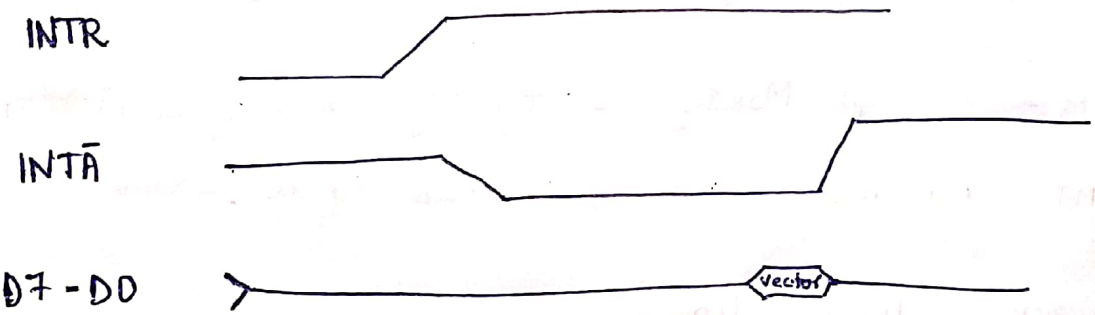
: Doesn't need any Acknowledgement signal.

: Non-maskable (need to be serviced)

: Must remain at logic 1, until it is accepted by the processor.

: Before the 0 to 1 transition, NMI must be at logic 0 for at least 2 clock cycles.

: No need for interrupt acknowledgement.



*. STEPS Taken by processor in case of Interrupt :

1. The CPU pushes the flag register into stack
2. The CPU pushes a far return address (segment : offset) onto the stack, segment value first.
3. The CPU determines the cause of interrupt (i.e. interrupt number) and fetches the four byte interrupt vector from IVT.
4. The CPU transfers control to ISR whose address is specified by interrupt vector table entry.

5. When the ISR want to return control, it must execute an IRET (interrupt return)

6. The interrupt return pops four return address and the flags off the stack.

* INTERRUPT VECTORS :

- Interrupt Vector table occupies the address range from 00000H to 003FF H (the first 1024 bytes in the memory map)

- IVT = 256 * 4 = 1024 memory bytes are used to store vector table.

- Each interrupt has 4 bytes in ISR.
  - CS = 8 bit 8 bit = 2 bytes
  - IP = 8 bit 8 bit = 2 bytes

1 byte = 8 bits
2 byte = 16 bits
4 byte = 32 bits

Code segment = 16 bits = 2 bytes
Instruction Pointer = 16 bits = 2 bytes
(or) Program Counter

Total Each interrupt = 32 bits = 4 bytes

∴ There are total 256 interrupts. Each interrupt 4 bytes = 256 * 4 = 1024 total
(from 00000 H to 003FF H)

- 256 interrupt has 256 different ISR. Each ISR has set of CS, IP to calculate Physical Address of ISR.

- 256 Interrupts :
  - * 0 to 4 → dedicated = 5 Interrupts
  - * 5 to 31 → for system use = 27 interrupts
    - 08H ~ 0FH : 8259 A
    - 10H ~ 1FH : BIOS
  - * 32 to 255 → for users = 224 interrupts
    - 20H ~ 35H : DOS
    - 40H ~ FFH : open

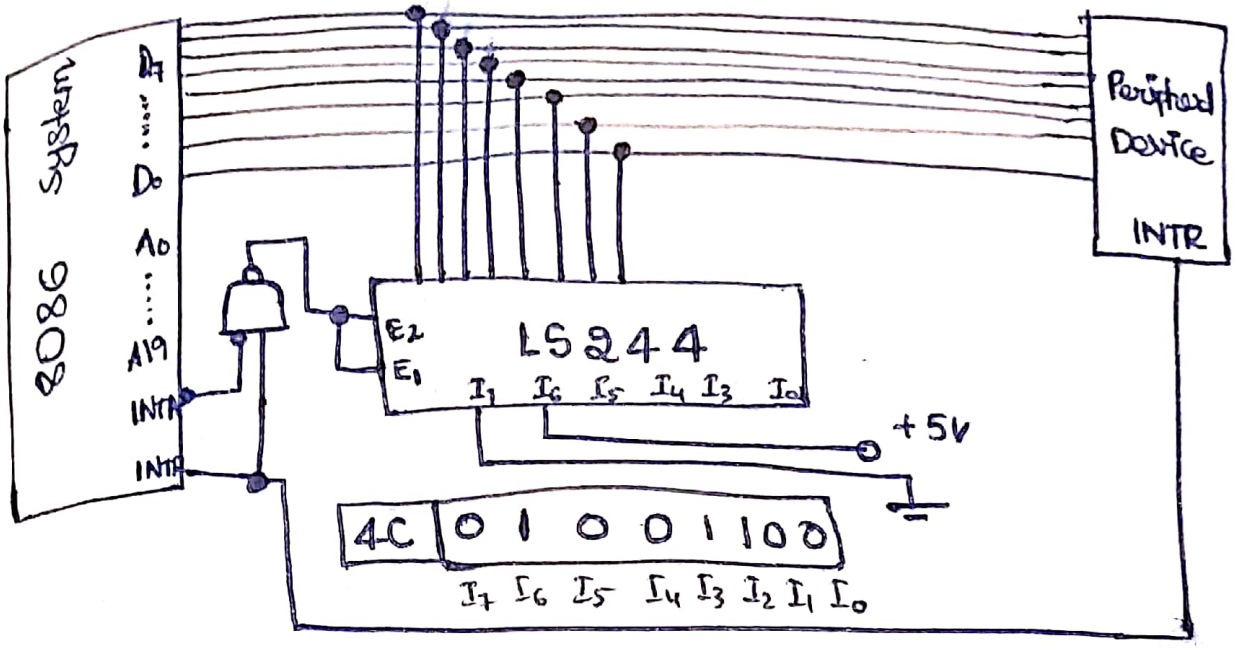
IVT - memory (000 to 3FF) 20 bit Physical Address

*. Example : Draw a circuit diagram to show how a device with interrupt vector 4CH can be connected on 8086 microprocessor system

Answer :

- The peripheral device activates the INTR
- The processor responds by activating the INTA
- The NAND gate enables the 74LS244 octal buffer. (Here 4CH appears on the data bus)
- The processor reads the data bus to get the interrupt vector.

Peripheral device wants to communicate through the Processor



*. Interrupt Vector Table – Example

Interrupt Vector Table – Real Mode (16-bit) Example

- Using the Interrupt Vector Table shown below, determine the address of the ISR of a device with interrupt vector 42H.
- Answer: Address in table = 4 X 42H = 108H
- (Multiply by 4 since each entry is 4 bytes)
- Offset Low = [108] = 2A, Offset High = [109] = 33
- Segment Low = [10A] = 3C, Segment High = [10B] = 4A
- Address = 4A3C:332A = 4A3C0 + 332A = 4D6EAH

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000	3C	22	10	38	6F	13	2C	2A	33	22	21	67	EE	F1	32	25
00010	11	3C	32	88	90	16	44	32	14	30	42	58	30	36	34	66
.....	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
00100	4A	33	3C	4A	AA	1A	1B	A2	2A	33	3C	4A	AA	1A	3E	77
00110	C1	58	4E	C1	4F	11	66	F4	C5	58	4E	20	4F	11	F0	F4
.....	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
00250	00	10	10	20	3F	26	33	3C	20	26	20	C1	3F	10	28	32
00260	20	4E	00	10	50	88	22	38	10	5A	38	10	4C	55	14	54
.....	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
003E0	3A	10	45	2F	4E	33	6F	90	3A	44	37	43	3A	54	54	7F
003F0	22	3C	80	01	3C	4F	4E	88	22	3C	50	21	49	3F	F4	65

Interrupt Vector Table – Real Mode (16-bit) Ex

- Write a sequence of instructions that initialize vector 40H to point to the IS
- Answer: Address in table = 4 X 40 = 100H
- Set ds to 0 since the Interrupt Vector Table begins at 00000H
- Get the offset address of the ISR using the Offset directive
- and store it in the addresses 100H and 101H
- Get the segment address of the ISR using the Segment directive
- and store it in the addresses 102H and 103H

```
push  ax          } Save registers in the stack
push  ds
mov   ax,0        } set ds to 0 to point to the interrupt vector table
mov   ds,ax
mov   ax offset isr40 } Get the offset address of the ISR & store
mov   [0100h],ax      } it in the address 0100h (4x40 = 100h)
mov   ax,segment isr40 } Get the segment address of the ISR
mov   [0102h] ax       } and store it in the address
pop   ds          } Restore registers from the stack.
pop   ax
```

---

*. INTERRUPT MASKING :

- The processor can inhibit when request is sent.

→ Software Interrupt Processing & Hardware Interrupt processing

*. INTERRUPT Programs Instructions :

INT 0 = reserved for Divide Error.

INT 1 = reserved for Single step operation.

INT 2 = reserved for NMI pin. (Power failure)

INT 3 = reserved for Break point setting. (for Debugging)

INT 4 = reserved for overflow (INTO instruction)

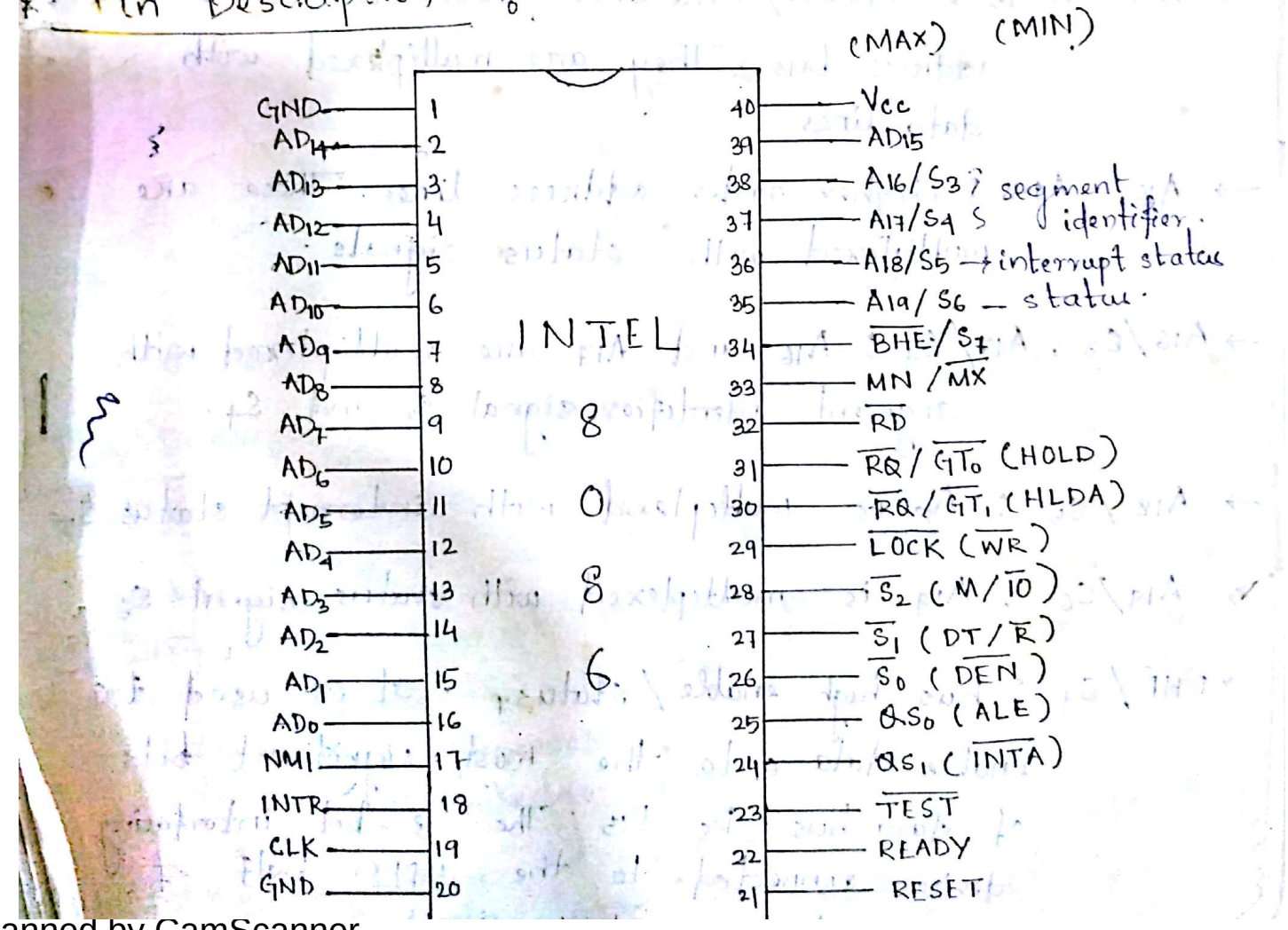
INT nn = Interrupt. Run ISR pointed by vector nn.

Q (Read) on / Features of 8086 microprocessor :— M-(4) (3)

8086 is a 16-bit, N-channel, HMOS microprocessor (HMOS - High speed MOS). 8086-1

→ It is a 40 pin IC package or DIP.
→ It has 16-bit databus and 20-bit wide address bus. It requires +5V. dc for its operation.
→ Using 20-bit address lines, its memory addressing capacity is 2^{20} = 1 MB of memory.
→ It can operate on 5, 8 and 10 MHz frequency.
→ It has introduced the pipelining operation technology.
→ Memory segmentation is first introduced in this microprocessor.

* Pin Description :—



1. Minimum mode
2. Maximum mode.

→ When only one 8086 CPU is to be used in a microcomputer system, 8086 up is used in the minimum mode of operation. In this mode the CPU issues the control signal required by memory and I/O devices.

→ In multiprocessor system it operates in the maximum mode. In this mode the control signals are issued by 8288 bus controller which is used with 8086 for this purpose.

→ The MN/$\overline{MX}$ decides the operating modes of 8086.

→ 8086 up pin description is as follows.

→ $AD_0 - AD_{15}$: Address/Data lines. These are low-order address bus. They are multiplexed with data lines.

→ $A_{16} - A_{19}$: Higher order address lines. These are multiplexed with status signals.

→ A_{16}/S_3, A_{17}/S_4 : A_{16} and A_{17} are multiplexed with segment identifier signal S_3 and S_4.

→ A_{18}/S_5 : A_{18} is multiplexed with interrupt status S_5

→ A_{19}/S_6 : A_{19} is multiplexed with status signal S_6

→ $\overline{BHE}/S_7$: Bus high enable/status. It is used to enable data onto the most significant bits of data bus $D_8 - D_{15}$. The 8-bit interfacing devices connected to the upper half of the data bus use $\overline{BHE}$ signal.

$\overline{RD}$ (Read) : This signal is used for read operation. It is active low.

READY : This signal indicates that the peripheral is ready to transfer data. The I/O or memory sends acknowledgement through this pin. This pin is active high.

RESET : This signal resets the system. This signal is active high.

→ CLK : It requires clock frequency of 5, 8 or 10 MHz.

→ INTR : Interrupt request.

→ NMI : Non-maskable interrupt pin.

→ $\overline{TEST}$: When this signal is active the microprocessor continues execution or else it waits. It includes an additional test control during wait states.

→ Vcc : Power supply of +5 v dc is used.

→ GND : Ground connection (0v).

Description for Minimum mode :—

For minimum mode operation $MN/\overline{MX}$ pin is made HIGH (1) or +5v power supply.

$\overline{INTA}$: Interrupt acknowledgement signal. It is active low. On receiving interrupt signal the processor issues an interrupt acknowledge signal.

ALE : Address latch enable. The up sends this signal to latch the address.

→ DEN : Data Enable :
→ This signal activates the external data bus buffers
→ It acts as an output enable signal, when the 8286 octal bus transceiver is used.

→ DT/$\overline{R}$: — Data Transmit / Receive
→ This signal controls the direction of dataflow through the transceiver i.e Intel 8286/8287
→ When it is HIGH, data are sent out,
when it is LOW, data are received.

→ M/$\overline{IO}$: — Memory / Input output.
→ When it is HIGH, CPU wants to access mem
and when it is LOW, CPU wants to access input/output device.

→ $\overline{WR}$: — Write control signal.
→ When this signal is LOW, the processor performs write operation.

→ HLDA : — Hold acknowledge signal.
→ It is issued by the processor, when the external device passes a hold request.

→ HOLD : —
→ When any external device wants to use the address and databus, it sends a HOLD request to the processor through this pin.

Pin Description for Maximum Mode :-

In this mode MN/$\overline{MX}$ is LOW.

QS_1, QS_0 :- Queue status

→ These signals show the instruction queue status.

QS_1	QS_0	
0	0	→ No operation
0	1	→ 1st byte of opcode from queue
1	0	→ Empty the queue
1	1	→ Next byte from the queue.

$\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$:- status signals

→ These signals are connected to the bus controller i.e Intel 8288.

→ The bus controller generates memory and I/O access control signals.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0	0	0	→ Interrupt acknowledge
0	0	1	→ Read data from I/O port
0	1	0	→ Write data into I/O port
0	1	1	→ Halt
1	0	0	→ Opcode fetch
1	0	1	→ Memory Read
1	1	0	→ Memory write
1	1	1	→ Passive state.

$\overline{LOCK}$:-

→ This signal is used to lock peripherals from the system.

→ $\overline{RQ}/\overline{GT_1}$, $\overline{RQ}/\overline{GT_0}$:- Request / Grant.

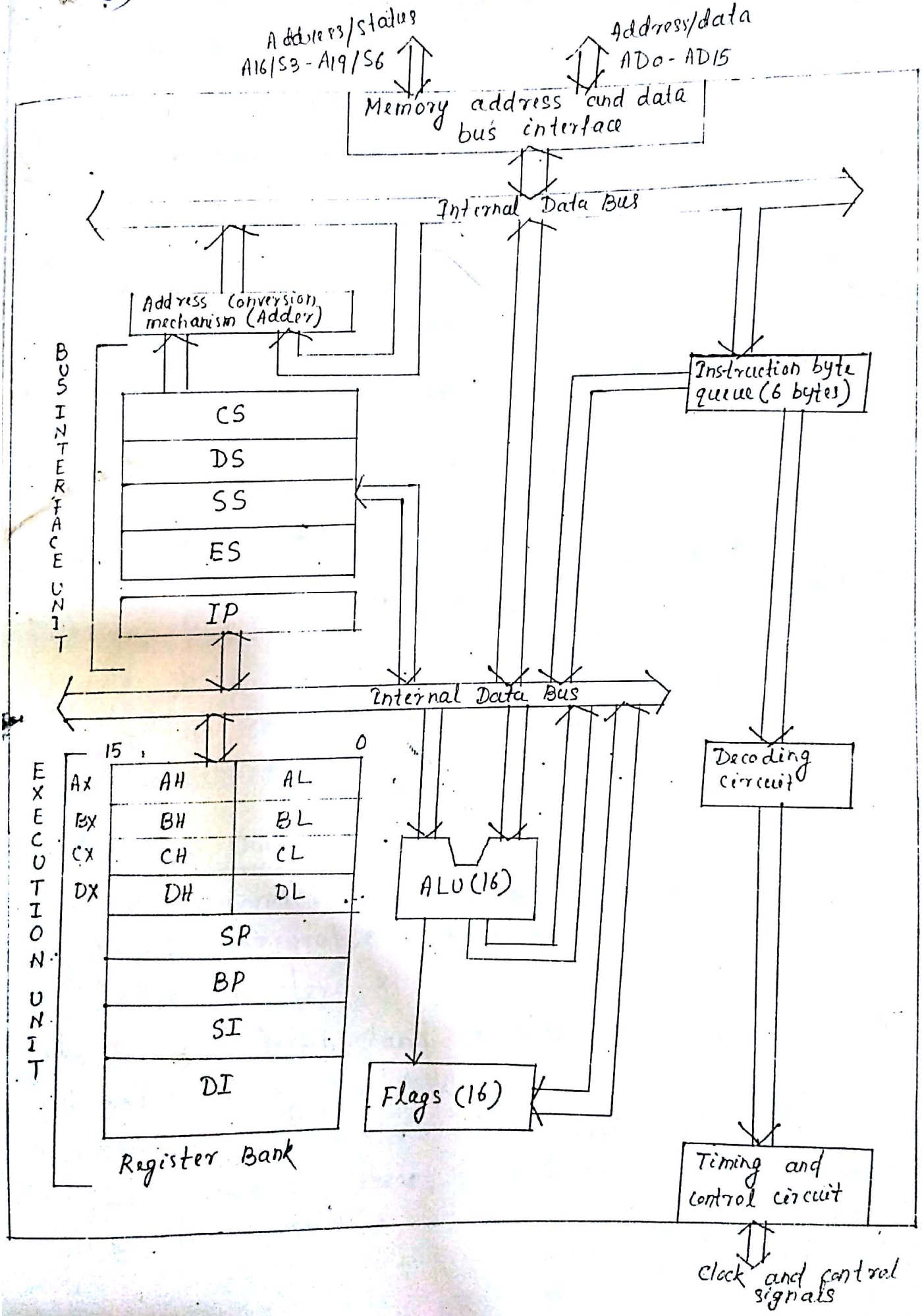
→ These signals are used for local bus priority control.

→ Other processors ask the CPU through these lines to release the local bus.

→ $\overline{RQ}/GT_0$ has higher priority than $\overline{RQ}/GT_1$.

8086 ARCHITECTURE

(2)



8086 Microprocessor Architecture :—

→ The 8086 Microprocessor Architecture is divided into two parts.

1. Bus Interface unit (BIU)
2. Execution unit (EU)

Bus Interface Unit :—

→ The BIU consists of the following units, like Instruction Queue, Segment Registers, Instruction Pointer etc.

→ It interfaces the ~~outside~~ processor to the outside i.e it is responsible for performing all external bus operations like fetch, read, write, input and output of data.

→ The BIU uses instruction queue for pipelined operation. The instruction queue is a 6-byte first-in-first-out register which can be understood as a cache memory. The queue permits the prefetch of upto 6 bytes of instruction code.

Execution Unit :—

→ The EU consists of the following units, like, ALU, Flag register, General purpose registers, Pointer and Index registers etc.

→ The EU decodes and executes the instructions prefetched by the BIU.

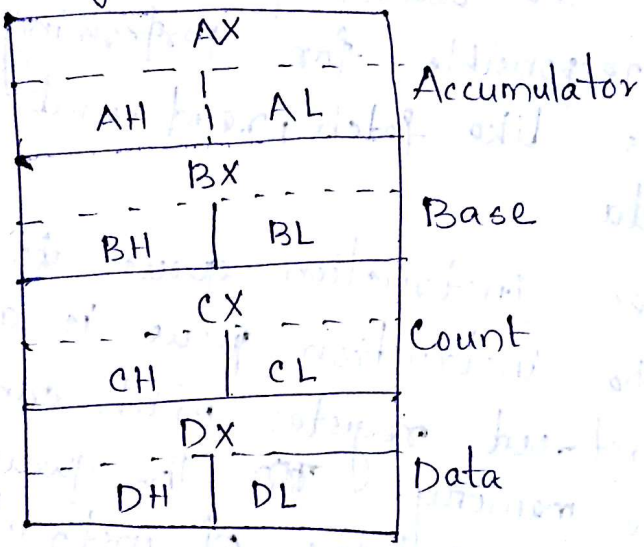
→ It also tests the status and control flags and updates these flags based on the results of the instruction.

Register Organisation :-

→ 8086 microprousssor has a powerful set of registers of 16-bit each. The registers are categorized into four groups.

1. General Data Registers
2. Segment Registers
3. Pointers and Index Registers
4. Flag Register.

General Data Registers —



→ 8086 µp contains 4 general data registers like AX, BX, CX and DX.

→ They are used to hold data, variables, results etc temporarily for faster operation.

Accumulator (AX) :-

→ AX is used as 16-bit accumulator, with the lower 8-bits designated as AL and higher 8-bits as AH for 8-bit operations.

→ It performs all the arithmetic and logic operations and the result is also stored in accumulator.

Base Register (BX) :-

→ BX register is used as a general purpose register as well as to store the offset for forming physical address in certain addressing modes.

Count Register (CX) :-

→ CX register is used as a default counter in case of string and loop instructions.

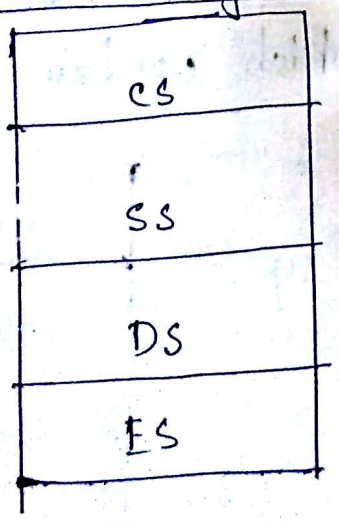
→ CX register is also used for the count of the no. of bits by which the contents of an operand must be shifted or rotated during the execution of the multibit shift or rotate instructions.

Data Register (DX) :-

→ DX register is used in I/O operations to hold the address of I/O port.

→ DX register also holds the remainder after a word division and holds the high-order bits (MSB) of the result after a word multiplication (32-bit).

Segment Registers :-



→ There are 4 segment registers in 8086 up.
They are code segment (cs), Data segment (DS), stack segment (ss) and Extra segment register (ES).
→ Each of them contains a 16-bit base address that points to the corresponding segment in memory.

→ Code segment (cs) :—

→ It is used for addressing a memory location in the code segment of memory where the executable program or instructions are stored.

stack segment (ss) :—

→ It is used for addressing stack segment of memory which is used to store stack data.

Data segment (Ds) :—

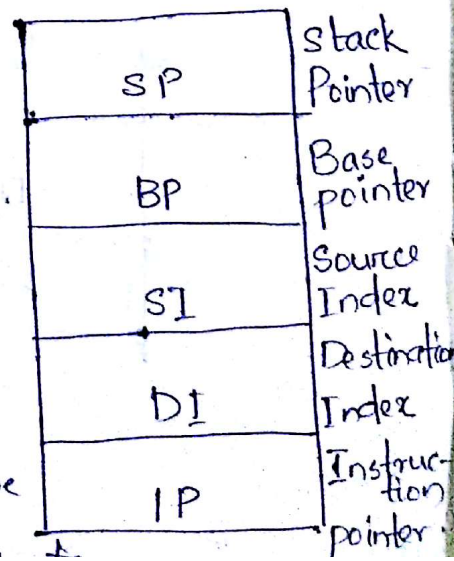
→ The data segment register points to the data segment of the memory where the data is stored.

Extra segment (ES) :—

→ This register points to the extra segment of the memory. The ES is used as another data segment of memory, which contains data.

Pointers and Index Registers :—

→ There are 3 pointers in 8086 up.
They are,
Instruction pointer (IP)
Base pointer (BP)
Stack Pointer (SP)



→ The pointers and index registers contain the offset addresses of memory locations relative

the segment registers.

→ Instruction Pointer (IP) -

→ The function of IP is similar to a program counter, but it contains the offset address instead of the actual address of the next instruction.

→ IP contains the offset address within the codesegment.

→ IP is combined with the CS to generate the address of the next instruction to be executed.

→ Stack Pointer (SP) —

→ The contents of SP are used as offset from the current value of stack segment (SS) during the execution of instructions that involve the stack segment

Base Pointer (BP) —

→ BP also contains the offset within the stack segment (SS). BP contains the offset in the based addressing mode.

→ There are two index registers in 8086 microprocessor. They are.- Source Index register (SI)
Destination Index register (DI)

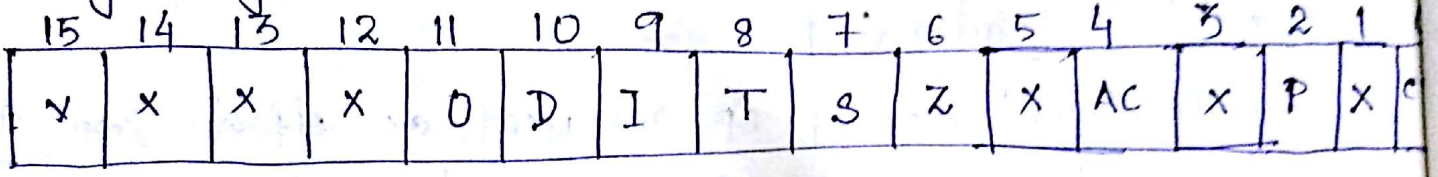
→ The index registers are used as general purpose registers as well as for offset storage purpose.

→ The SI register is used to store the offset of source data in data segment and DI register is used to store the offset of destination data in data or extra segment.

→ Default segment and offset registers :—

Segment	Offset
CS	IP
SS	SP or BP
DS	BX, SI, DI
ES	DI

Flag Register :—



→ 8086 microprocessor has a 16-bit flag register which contains 9 active flags. The flag register can be divided into two parts.

1. Conditional flags
2. Machine control flags.

→ The conditional or status flags are set or reset on the basis of the result generated from the ALU.

→ The lowerbyte of the flag register along with the overflow flag (O) contains the conditional flags.

→ The control flags contain 3 flags like direction flag (D), Interrupt flag (I) and trap flag (T).

→ These flags are used to control the program flow and for controlling the microprocessor.

CY - Flag —

→ This flag is set when there is a carry from the MSB in case of addition or a borrow in case of subtraction.

i.e carry from D_{15} bit for 16-bit operation.
" D_7 bit for 8-bit operation.

P - Flag

→ Parity flag is set when the ALU output has even parity and is reset when the ALU output has odd parity.

→ Parity is the count of 1's is a number.

AC - Flag

→ The AC flag is set when there is a carry after addition and borrow after subtraction between D_3 and D_4 bit positions (for 8-bit data) and carry from D_7 - D_8 (for 16-bit data)

Zero (Z) - Flag

→ The Z flag shows that the result of an ALU operation is zero or ~~any other no~~ non zero.

→ If Z = 1, the result is zero.
Z = 0, the result is nonzero.

S - Flag

→ This flag is set, when the result of any ALU operation is positive or negative.

→ The MSB of the result shows the sign bit.

→ If the sign bit = 0, the no. is positive
= 1, the no. is negative.

for 8-bit operation, D_7 bit represents the sign bit
16-bit operation, D_{15} bit " "

Overflow Flag –

→ It is based on the (n-1) bit carry of the result.

→ Over flow occurs when signed nos. are added or subtracted.

→ If the result of a signed operation is large enough to be accomodated in a destination register, then overflow occurs.

→ i.e for 8-bit signed operation,

if there is a carry from D_6 – D_7 bit

OF is set (1).

for 16-bit signed operation,

if there is a carry from D_{14} – D_{15} bit of the result

OF is set (1).

e.g. +127 = 7F = 01111111

+01 +01 = 00000001

10000000 (80H)

OF = 1 (set)

(as there is a carry from D_6 – D_7 bit).

Trap Flag –

→ The TF = 1 (set), when the 8086 processor enters into the single step mode.

or else it is reset.

→ In single step mode the processor executes one instruction at a time and it is useful for debugging the programs.

Interrupt Flag :- (IF)

- This flag (IF) is set (1), when the maskable interrupt or INTR is received by the processor.

→ IF = 1 (set), if the INTR pin is enabled.

= 0 (Reset), if the INTR pin is disabled.

Direction Flag :- (DF)

→ DF is used for string manipulation instructions. i.e the direction flag selects the increment or decrement mode for DI and SI registers in string instructions.

→ If, DF = 1 (set), the registers are automatically decremented.

= 0 (Reset), the registers are automatically incremented.

BUS OPERATION :—

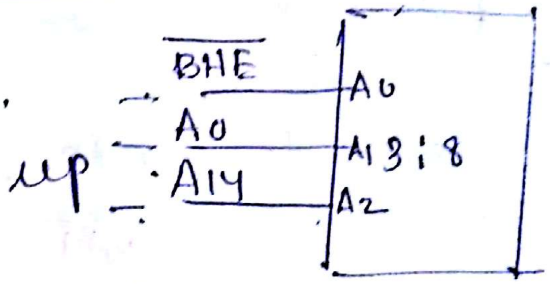
→ The 3 buses of 8086 µp i.e address, data and control function in the same manner as that of 8085 µp.

→ If data is to be written in the memory, the microprocessor outputs the memory address on the address bus, outputs the data to be written into memory on the data bus and issues a write ($\overline{WR}$) to memory and $M/\overline{IO} = 1$.

→ If data is to be read from memory, the microprocessor outputs the memory address on the address bus, issues a read ($\overline{RD}$) memory signal and accepts the data via the data bus.

Read Machine cycle for Minimum mode :—

2 × 8 K = 16 KB ROM
16 KB RAM.



(A14)

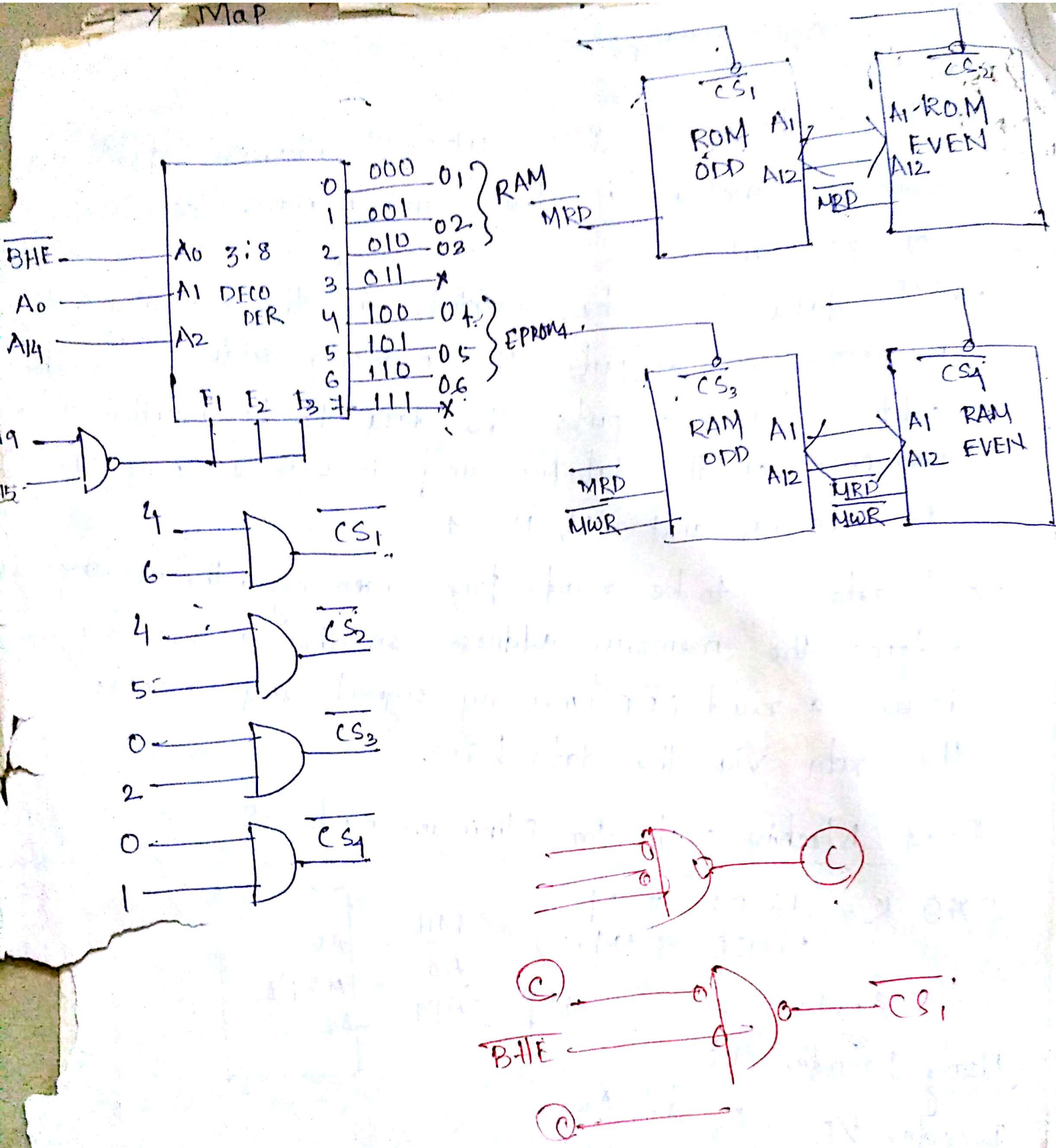
Using Decoder ckt

Decoder I/P → A_2 A_1 A_0

µp I/P → A_{14} A_0 $\overline{BHE}$

01	0	0	0	— both odd and even RAM
02	0	0	1	— only even RAM
03	0	1	0	— only odd RAM.
	0	1	1	X
04	1	0	0	— both odd and even ROM
05	1	0	1	— even ROM
06	1	1	0	— odd ROM.
	1	1	1	X

→ Map



Interfacing

Interface two 8K×8 EPROMs and two 8K×8 RAM chips with 8086 microprocessor?

Soln: For a good and efficient interfacing, the two conditions are

1. Memory map should be continuous.
2. Minimum hardware should be used.

→ In this circuit,

2 × 8K × 8 = 16 KB EPROMs

and 2 × 8K × 8 = 16 KB RAMs are used.

→ For 16 KB capacity,

$16\ KB = 2^4 \times 2^{10} = 2^{14} = 2^n$

i.e $n = 14 = A_0 - A_{13}$ lines are used for Register selection

$A_{14} - A_{19}$ lines are used for chip selection purpose.

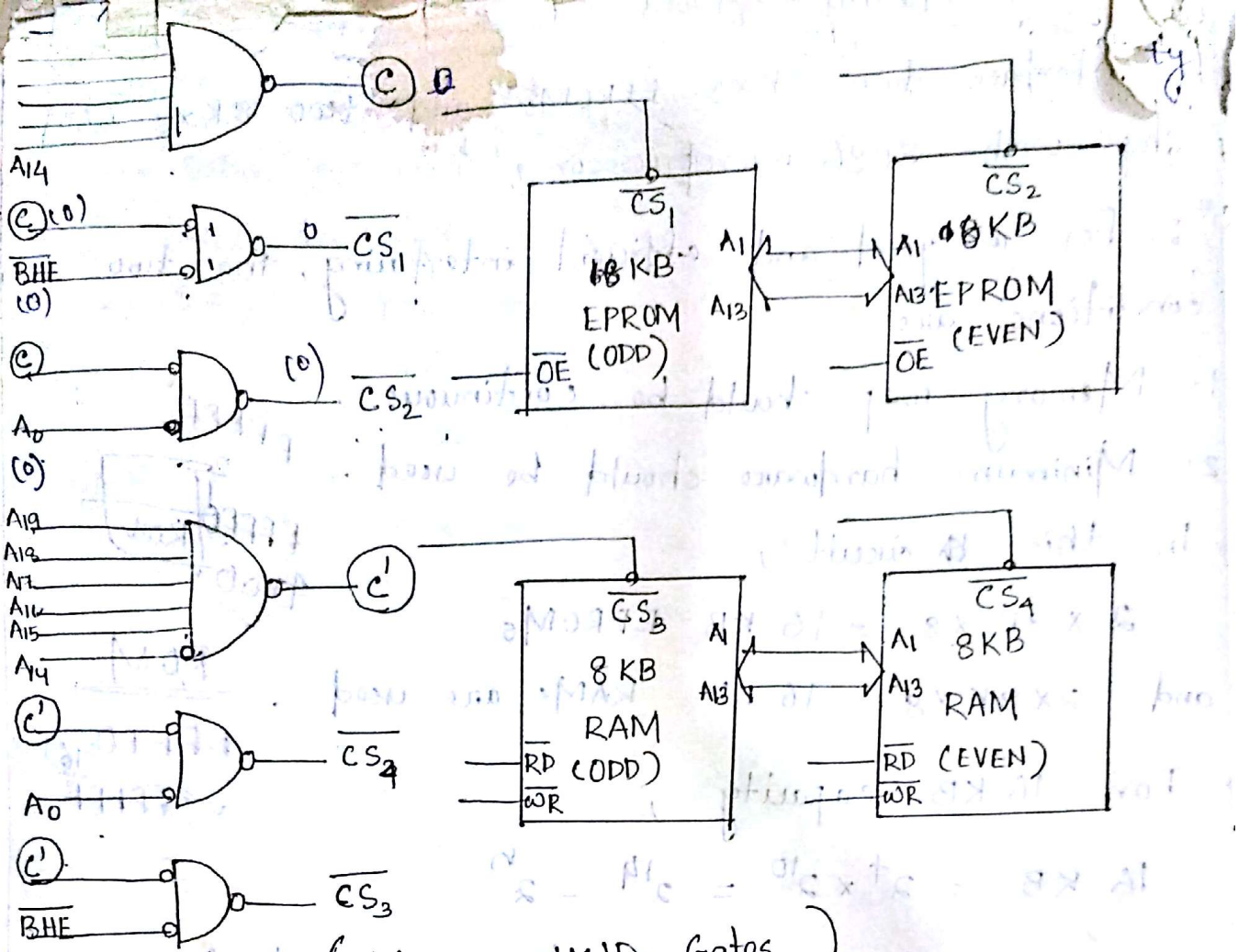
for 16 KB RAM also,

$A_0 - A_{13}$ lines are used for register selection.

and $A_{14} - A_{19}$ lines are used for chip selection.

Memory Map:

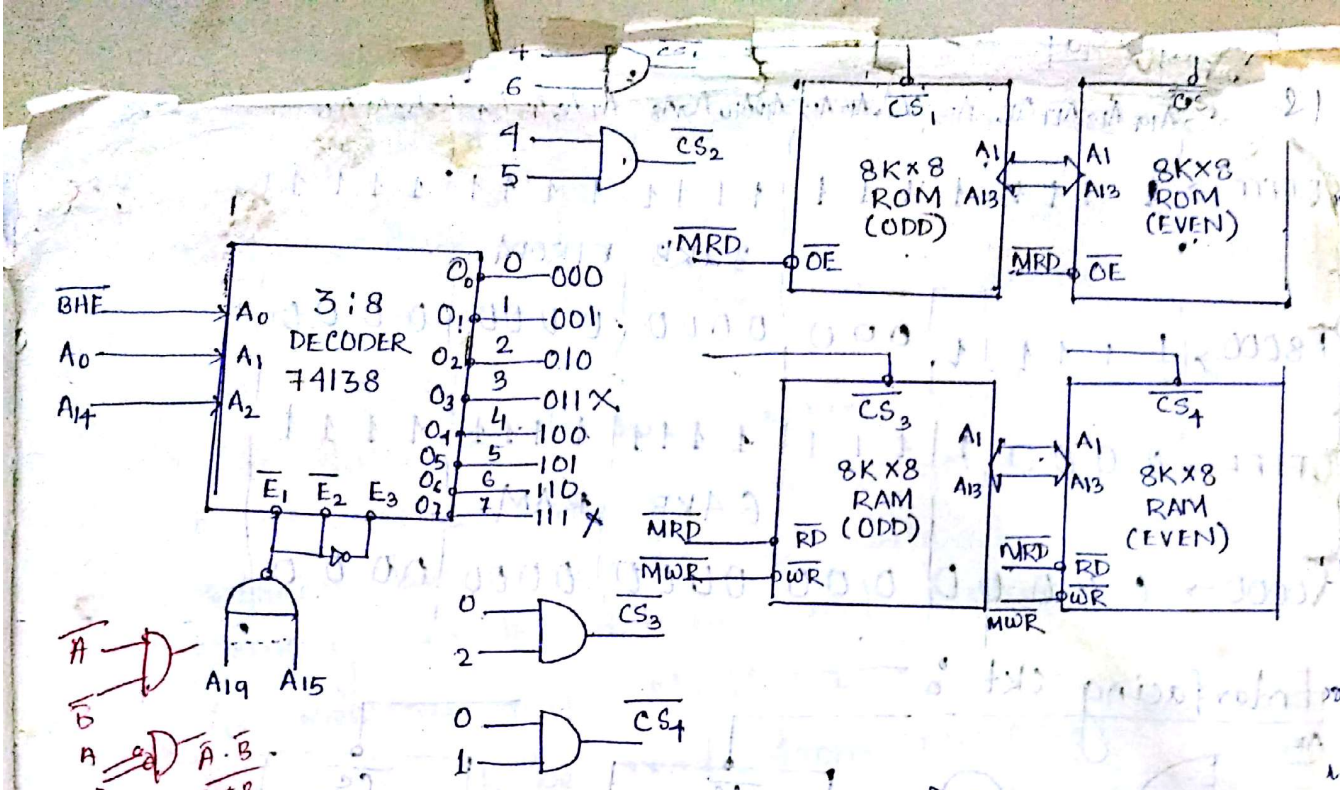
	A_{19} A_{18} A_{17} A_{16}	A_{15} A_{14} A_{13} A_{12}	A_{11} A_{10} A_9 A_8	A_7 A_6 A_5 A_4	A_3 A_2 A_1 A_0
$FFFF_H$ →	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1
		16 × 8 EPROM			
0000_H →	1 1 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
$1FFF_H$	1 1 1 1	1 0 1 1	1 1 1 1	1 1 1 1	1 1 1 1
		16 × 8 RAM			
0000_H	1 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0



(Using NAND Gates)

→ Memory Chip selection for Decoder circuit — (3:8)

Decoder I/P →	A_2	A_1	A_0	Function
	A_{14}	A_0	$\overline{BHE}$	
Word transfer →	0	0	0	→ Both odd & even banks of RAM are selected
Byte transfer (D_0-D_7) →	0	0	1	→ Only even bank of RAM is selected.
Byte transfer (D_8-D_{15}) →	0	1	0	→ Only odd bank of RAM is selected
Word transfer →	1	0	0	→ Both even and odd EPROMs
Byte transfer (D_0-D_7) →	1	0	1	→ Only even EPROM
Byte transfer (D_8-D_{15}) →	1	1	0	→ Only odd EPROM



(Using DECODER circuit)

2.2. Interface two 16K×8 EPROM and two 32K×8 RAMs with 8086 microprocessor. The starting address of RAM is 00000H.

2×16 KB

$= 32$ KB $= 2^n$ (for EPROM)

$\Rightarrow 2^5 \times 2^{10} = 2^n$

$\Rightarrow 2^{15} = 2^n$

$\Rightarrow n = 15$

15 lines are required for register selection

i.e $A_0 - A_{14}$

$A_{15} - A_{19}$ lines are used for chipselection.

For RAM,

2×32KB

$= 64$KB $= 2^n$

$= 2^6 \times 2^{10} = 2^n$

$\Rightarrow n = 16$

16 lines are used for register selection

i.e $A_0 - A_{15}$.

$A_{16} - A_{19}$ lines are used for chi...

8086 up ADDRESSING MODES :—

→ Addressing mode indicates the way of locating data or operands, the types of operands used and the way they are accessed for executing instruction.

→ According to the flow of instruction execution, the 8086 instructions can be categorized as,

(i) Sequential control flow instructions

(ii) Control transfer instructions.

→ Sequential control flow instructions are the instructions in which after execution, the control is transferred to the next instruction appearing immediately after it in the program.

e.g: Arithmetic instructions, Logical, data transfer, and processor control instructions are under this group.

→ The control transfer instructions, transfer their control to some predefined address after their execution.

e.g. INT, CALL, RET and JUMP instructions are under this category.

→ 8086 up addressing modes are -

1. Register Addressing Mode
2. Immediate " "
3. Direct " "
4. Register Indirect " "
5. Register Relative " "
6. Indexed " "
7. Based Indexed Addressing Mode
8. Relative Based Indexed " "
9. Intrasegment Direct " "
10. Intrasegment Indirect " "
11. Intersegment Direct " "
12. Intersegment Indirect " "

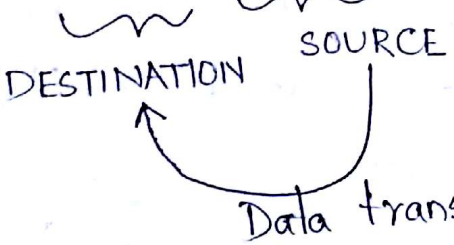
→ Addressing modes (1 – 8) comes under the sequential control flow instructions group and from (9 - 12) come under the control transfe group.

(1) Register Addressing Mode –

→ It transfers or copies a byte or word from source register or memory location to the destina register or memory location.

e.g. MOV AX , BX

DESTINATION SOURCE



Data transfer flow.

→ All the registers of 8086 exp except IP can be used in this mode.

→ Here both the source and destination registers should be of same size.

→ A segment → to segment data movement i not allowed.

(2) Immediate Addressing Mode :–

→ It transfers the source data (immediate byte or word) to the destination register or memory location.

e.g. MOV AL , 22H

Destination Register. Source data (byte)

MOV AX , 1234H.

(3) Direct Addressing Mode :—

→ It moves a byte or word between a memory location and a register.

→ Here a 16-bit memory address i.e the offset address is directly specified in the instruction as a part of it.

e.g. MOV AL, [1234H]
(D, S)

MOV AX, [5000H]
(D, S)

Here [1234H] and [5000H] act as the offset address.

→ The contents of the physical address which is formed from the offset address is transferred or copied to the destination register.

→ So the segment associated with this mode can be DS. Let DS = 1000H.

MOV AL, [1234H]

i.e offset address is 1234H.

→ So the physical address

= DS × 10 + offset value

= 1000H × 10 + 1234H

= 11234H

→ Let 11234H memory location content is = 50H.

So 50H is transferred to AL Reg.

⇒ AL = 50H.

(4) Register Indirect Addressing Mode -

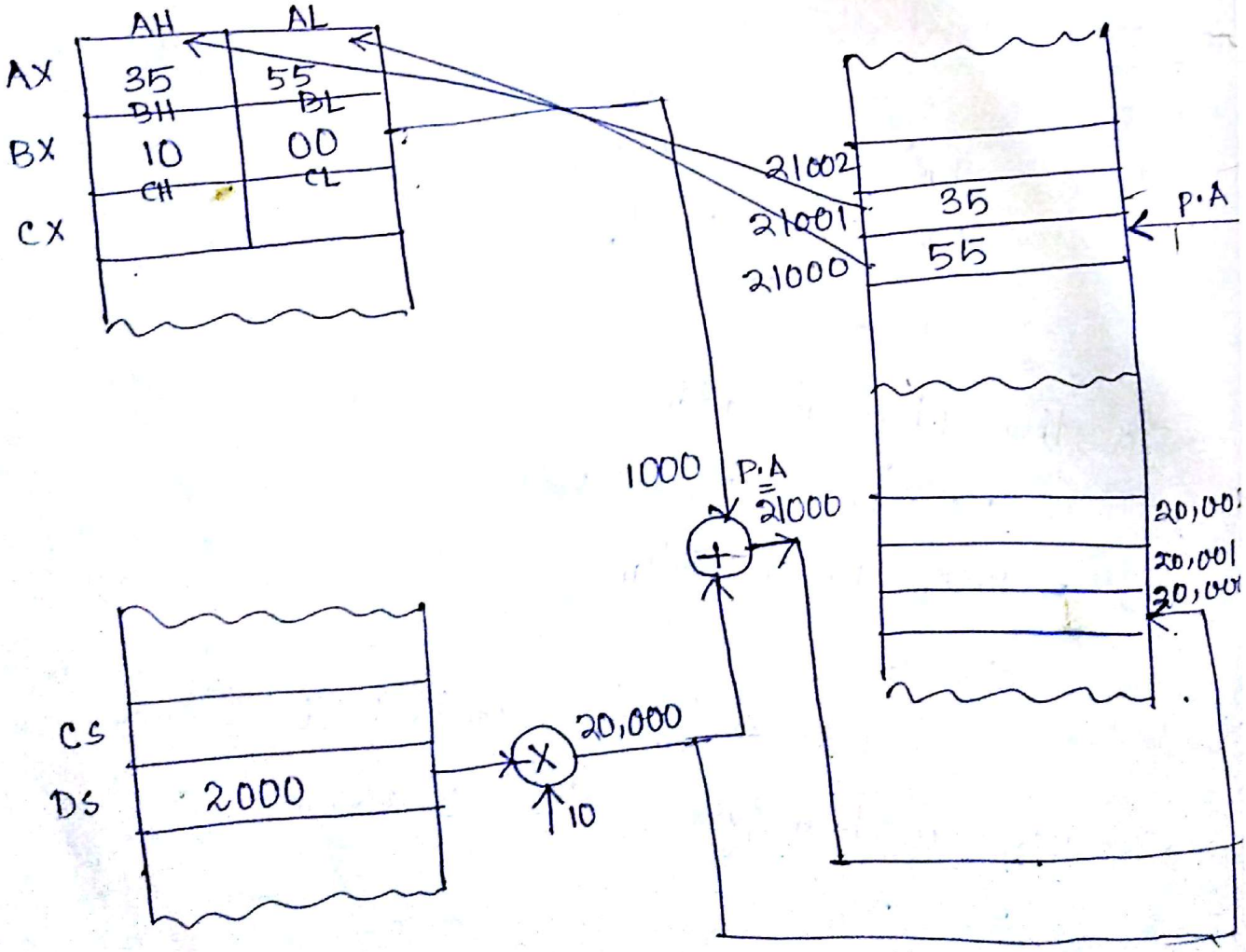
→ Register indirect addressing allows data to be addressed at any memory location using the offset registers : BP , BX , DI and SI.

→ The Data segment (DS) is used as the default segment in this mode , when it uses BX , DI or SI to address memory.

and stack segment (SS) is used as the default segment , when BP is used to address memory.

e.g. MOV AX , [BX]

Let BX = 1000_H , DS = 2000_H



→ After execution the contents of 21000_H and 21001_H i.e 55_H and 35_H are transferred to AX i.e AL and AH respectively.

(5) Register Relative Addressing Mode —

→ In this mode, the data in a segment of memory are addressed by adding an 8-bit or 16-bit displacement to the contents of a base register (BX or BP) or an index register (SI or DI).

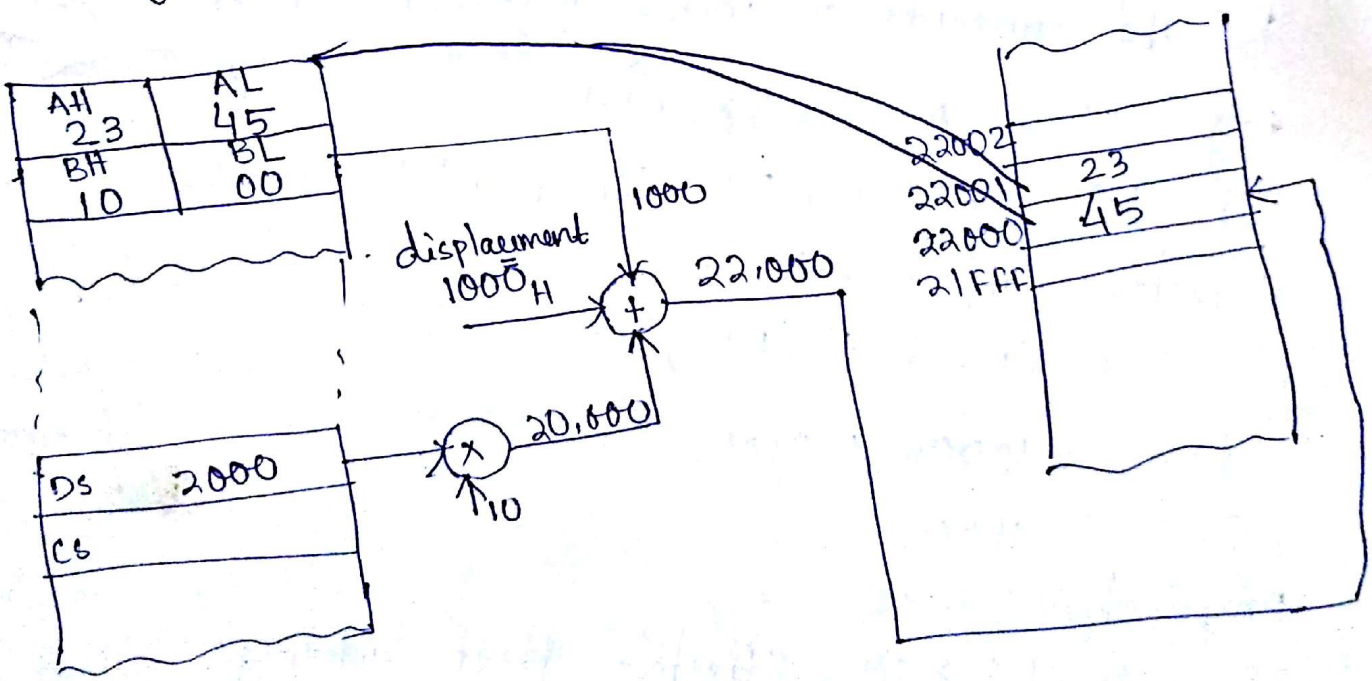
→ Here DS or ES are the default segments for BX, DI or SI registers and SS is the segment for BP.

e.g. MOV AX, 1000_H [BX]

et, BX = 1000_H, DS = 2000_H, Displacement = 1000_H.

So effective offset address
= BX + displacement
= 1000 + 1000 = 2000_H.

Physical address = DS × 10 + Effective offset address
= 2000 × 10 + 2000 = 22000H.



AX = 2345
AH = 22001 = 23
AL = 22000 = 45.

(6) Indexed Addressing :—

→ In this mode, the offset address of the operand is stored in one of the index registers like (SI or DI). DS and ES are the default segments for index registers SI and DI respectively.

e.g. MOV AX, [SI].

(7) Based - Indexed Addressing :—

→ In this addressing mode one base register (BX and one index register (SI or DI) are used to indirectly address memory.

→ In this mode the effective address is formed by adding contents of a base register (BX or BP) to the contents of index register (SI or DI).

e.g. MOV AX, [BX][DI]

Let DS = 2000H, BX = 1000H, DI = 0100H.

Effective offset address

= [BX] + [DI]

= 1000H + 0100H

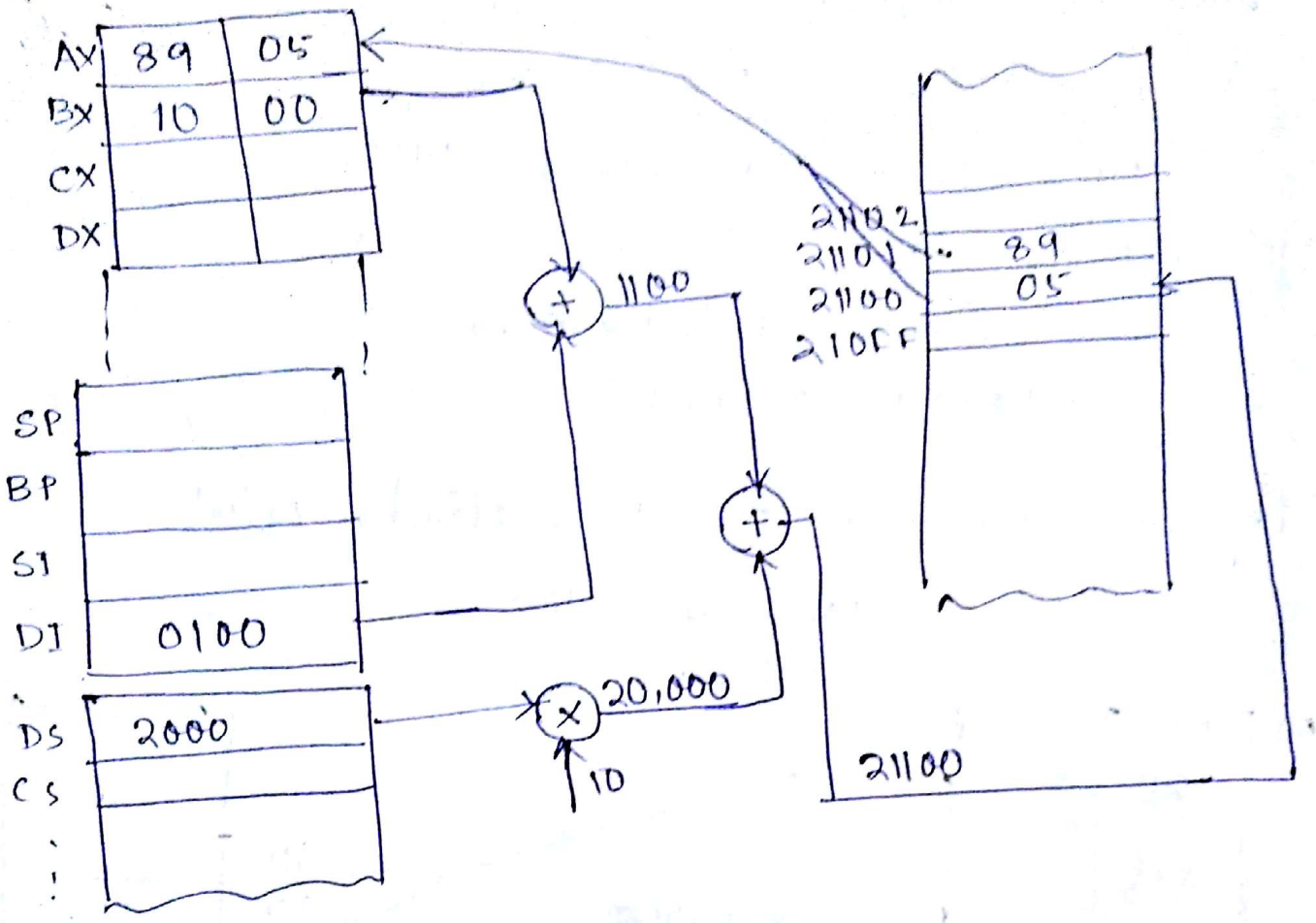
= 1100H.

Physical address

= DS × 10 + Effective offset address

= 2000 × 10 + 1100H

= 21100H.



AX = 8905

AH = 89 = [21101]

AL = 05 = [21100].

8) Relative Based Indexed Addressing :—

→ The Relative Based Indexed Addressing mode is similar to the Based Indexed mode but it adds a displacement along with the base register and index register to form the memory address. Here the effective offset address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the base registers (BX or BP) and any one of the index registers (SI or DI).

eg. MOV AX, 0100_H [BX][DI]

Let DS = 1000_H , BX = 2000_H

DI = 3000_H , displacement = 0100_H.

The effective offset address

= BX + DI + displacement

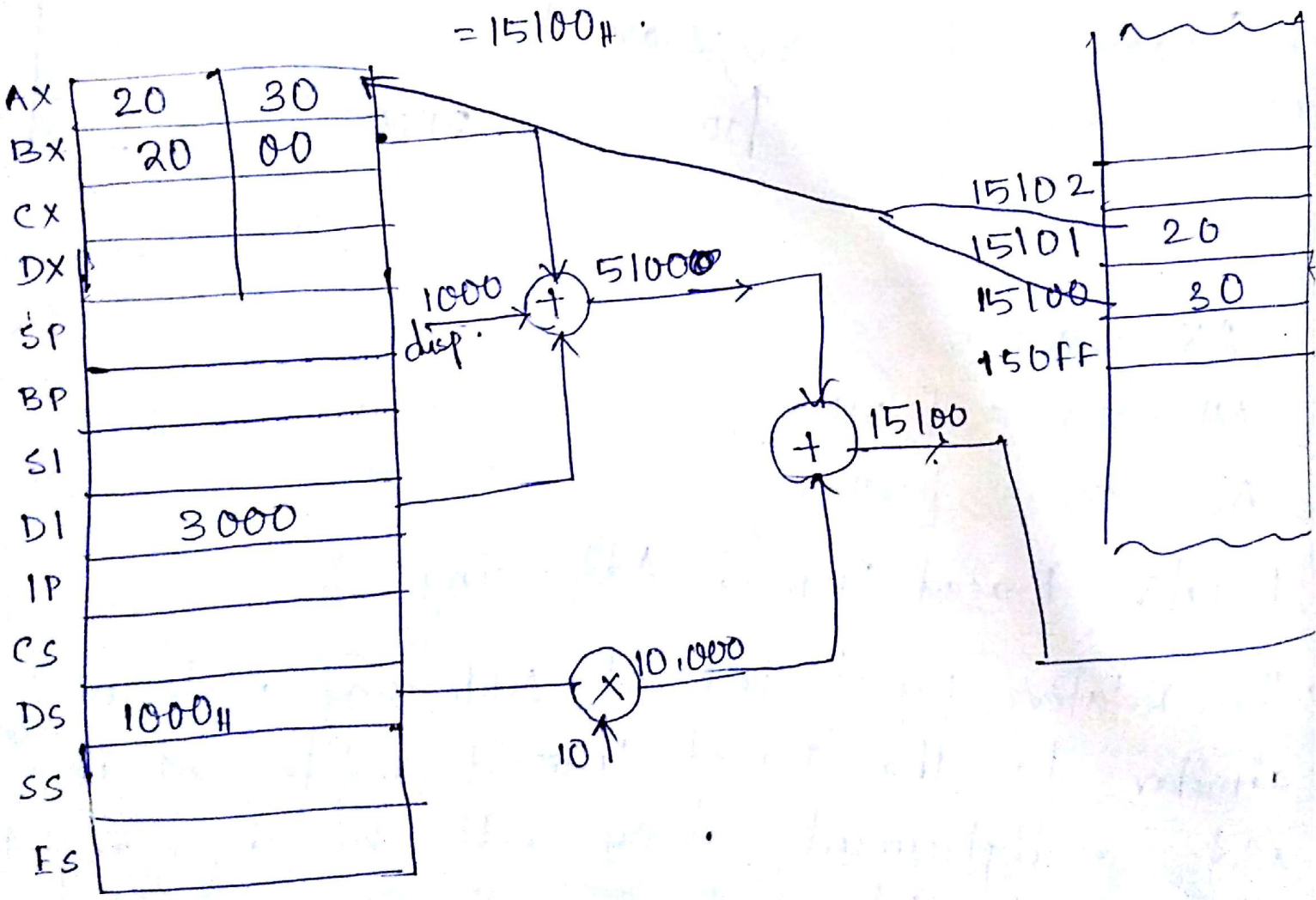
= 2000 + 3000 + 0100

= 5100_H.

Physical address = DS x10 + effective offset

= 1000 x 10 + 5100

= 15100_H.



AX = 2030

AH = 20 = [15101]

AL = 30 = [15100]

* Addressing modes for control transfer instructions or Program Memory Addressing modes –

→ If the address location to which the control is to be transferred lies in a different segment other than the current one, the mode is called Intersegment mode. If the destination lies in the same segment, the mode is called intrasegment mode.

9. Intrasegment Direct mode :–

→ In this mode the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value.

→ The effective branch address is the sum of an 8-bit or 16-bit displacement in the current contents of IP.

e.g. JMP [02]

E.A = [IP] + [02]

Physical address location = CS × 10 + (IP + 02)

10. Intrasegment Indirect Addressing mode :–

→ In this mode the effective branch address is the contents of a register or memory location that is accessed using any one of the data addressing modes.

→ The contents of IP will be replaced by the effective branch address.

e.g. JMP BX

→ In JMP BX instruction, the control is jumped to an address specified by the 16-bit register. The value of BX is copied into IP with CS value unchanged. Then the physical address of the next instruction is obtained using the current contents of CS and new value of IP.

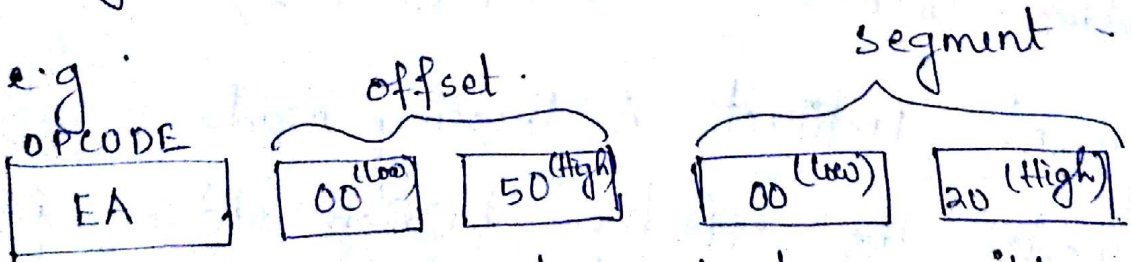
11. Intersegment Direct memory Addressing :—

→ This addressing mode is used to provide means of branching from one segment to another.

→ Here the instructions are of 5 bytes. 1st byte is the OPCODE followed by 4 bytes 32-bit immediate data.

Bytes 2 and 3 are loaded into IP and bytes 4 and 5 are loaded into CS.

e.g.



→ The JMP instruction loads CS with 2000H and IP with 5000H; the jump destination is 25000H

12: Inter segment Indirect Addressing mode -

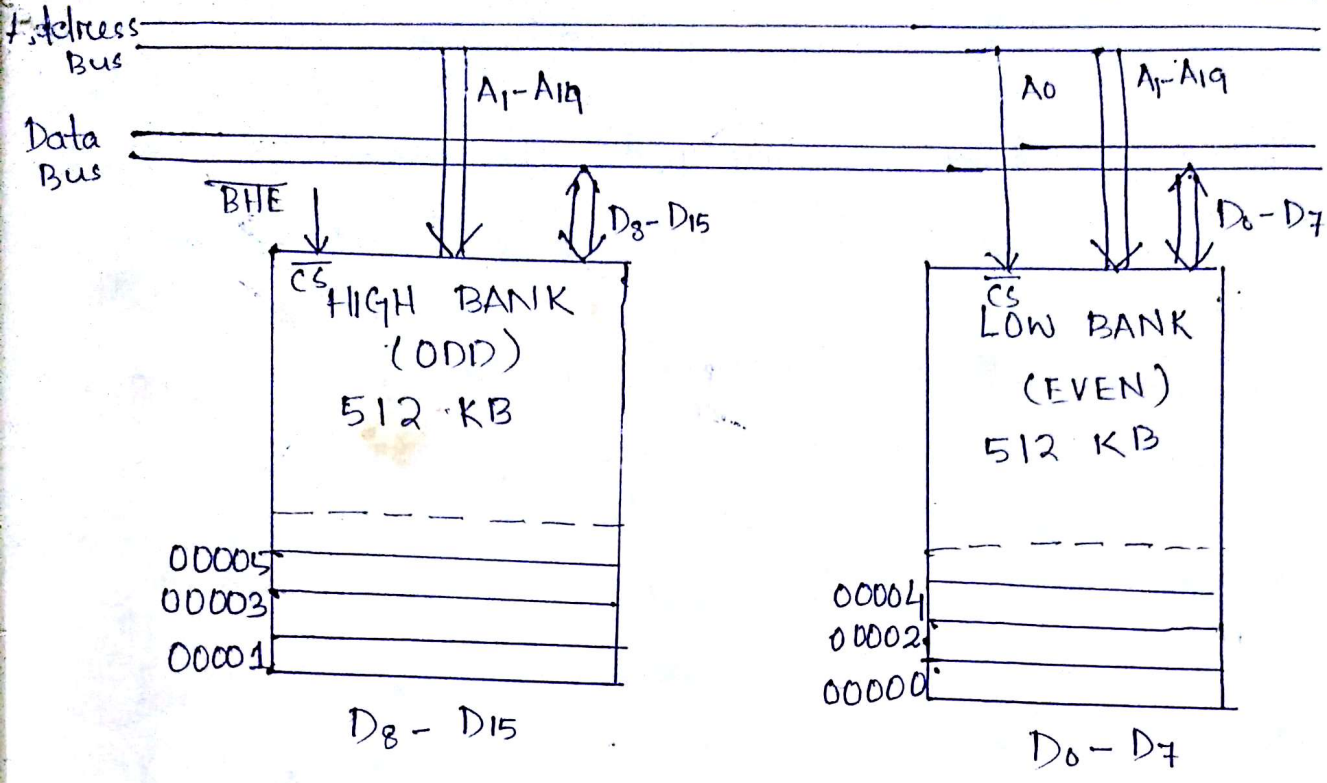
→ This addressing mode replaces the contents of IP and cs with the contents of two consecutive words in memory that are addressed using any one data addressing mode.

→ Here the instruction loads the contents of memory locations addressed by [reg 16 bit reg] and [16 bit reg + 1] into Ds into IP and then loads the contents of memory locations addressed by [reg 16 + 2] and [reg 16 + 3] in Ds into cs.

→ The registers used for reg 16 are BX, SI and DI.

Physical Memory Organisation :—

8086-5



→ In 8086 based system, the 1M bytes memory is physically organised as odd bank and even bank each of 512 kbytes, addressed in parallel by the processor.

→ The lower bank (even bank) contains bytes with only even addresses like 0, 2, 4 etc. where as the higher bank (odd bank) contains bytes with only odd addresses like 1, 3, 5 etc. i.e location 0 will have the address 1, location 1 will have the address 3, location 2 will have the address 5 etc.

→ The data lines of the lower bank is connected to $D_0 - D_7$ of 8086 where as the data lines of upper bank is connected to $D_8 - D_{15}$ of 8086.

→ The lowerbank is selected by A_0, where as the upper bank is selected by $\overline{BHE}$.

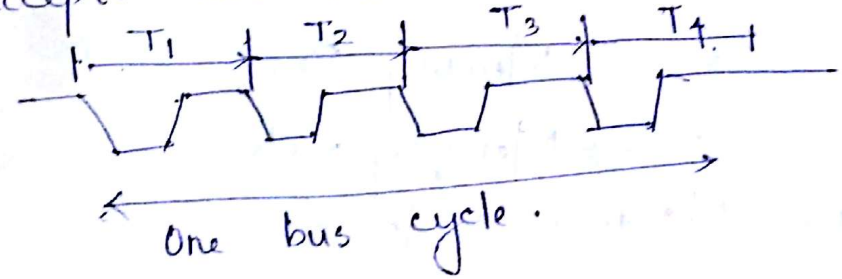
BUS TIMING :—

General Bus operation —

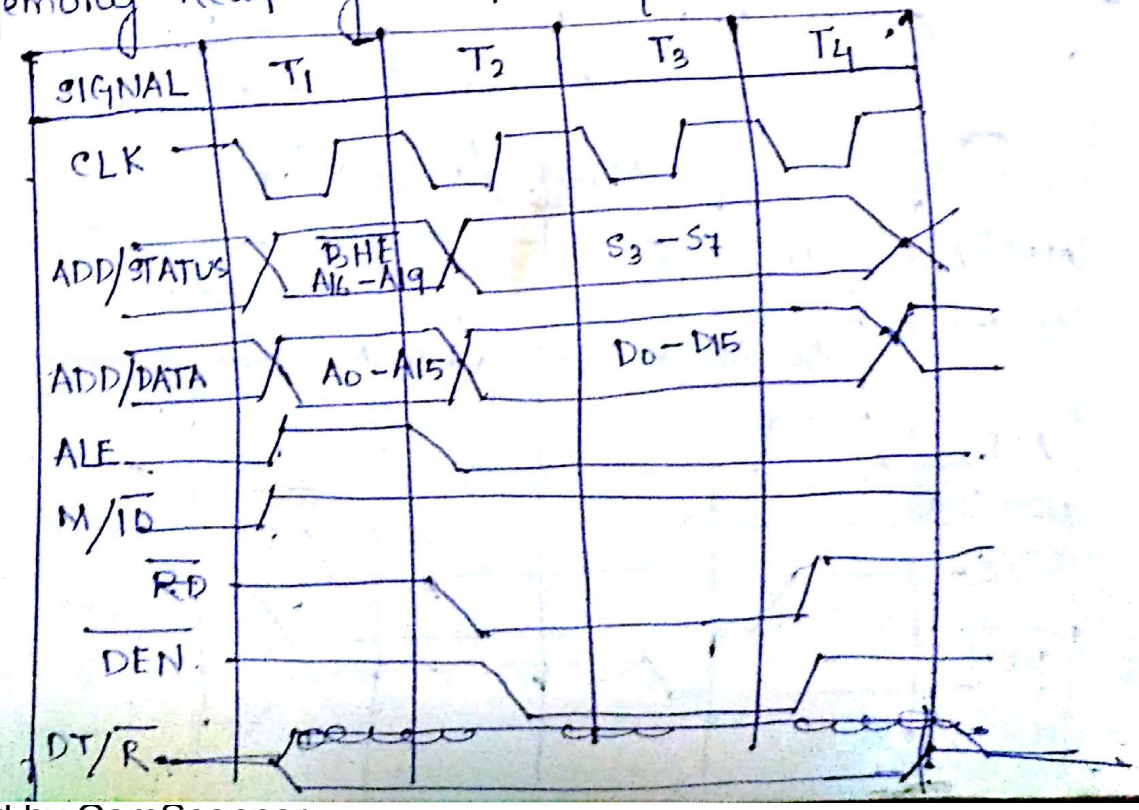
→ The 3 buses of 8086 up i.e address, data and control function in the same manner as that of 8085 up.

→ If data is to be written in the memory, the microprocessor outputs the memory address on the address bus, outputs the data to be written into memory on the data bus and issues a write ($\overline{WR}$) to memory and $M/\overline{IO} = 1$.

→ If data is to be read from memory, the microprocessor outputs the memory address on the address bus, issues a read ($\overline{RD}$) signal and accepts the data via the data bus.



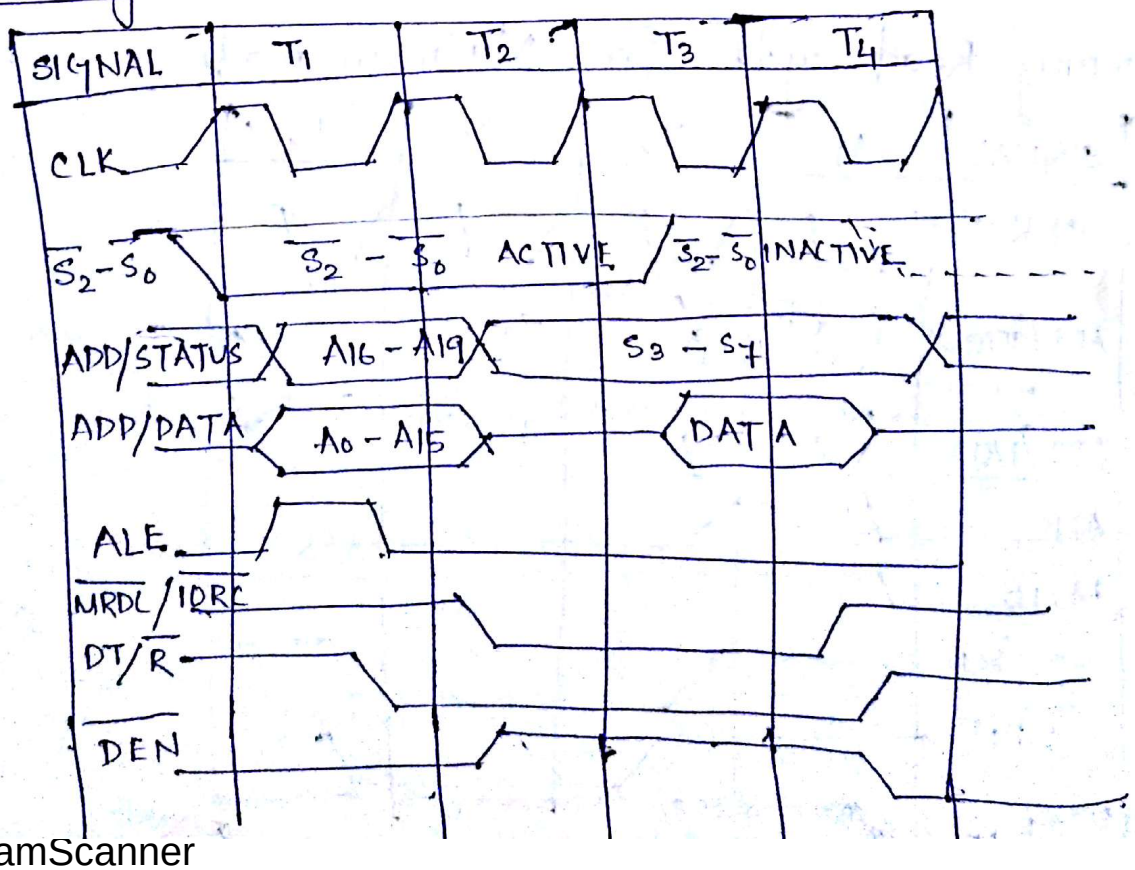
Memory Read cycle for Minimum mode —



→ The Read cycle begins in T_1. During T_1, the 8
outputs the 20-bit address of memory locat
to be accessed on $AD_0 - AD_{15}$ and $A_{16} - A_1$
→ ALE signal is used to separate $A_0 - A_{15}$
from $D_0 - D_{15}$.
→ In T_2, the status bits S_3 through S_7 are
output. This status information is maintained
through T_3 and T_4.
→ Input data are read by 8086, during T_3
and in T_4, $\overline{RD}$ and $\overline{DEN}$ return to logic 1.

M/$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Operation type
0	0	1	→ I/O Read
0	1	0	→ I/O write
1	0	1	→ Memory Read
1	1	0	→ Memory write.

Read cycle for Maximum mode :—



→ The maximum mode read cycle is similar as that of minimum mode read cycle.

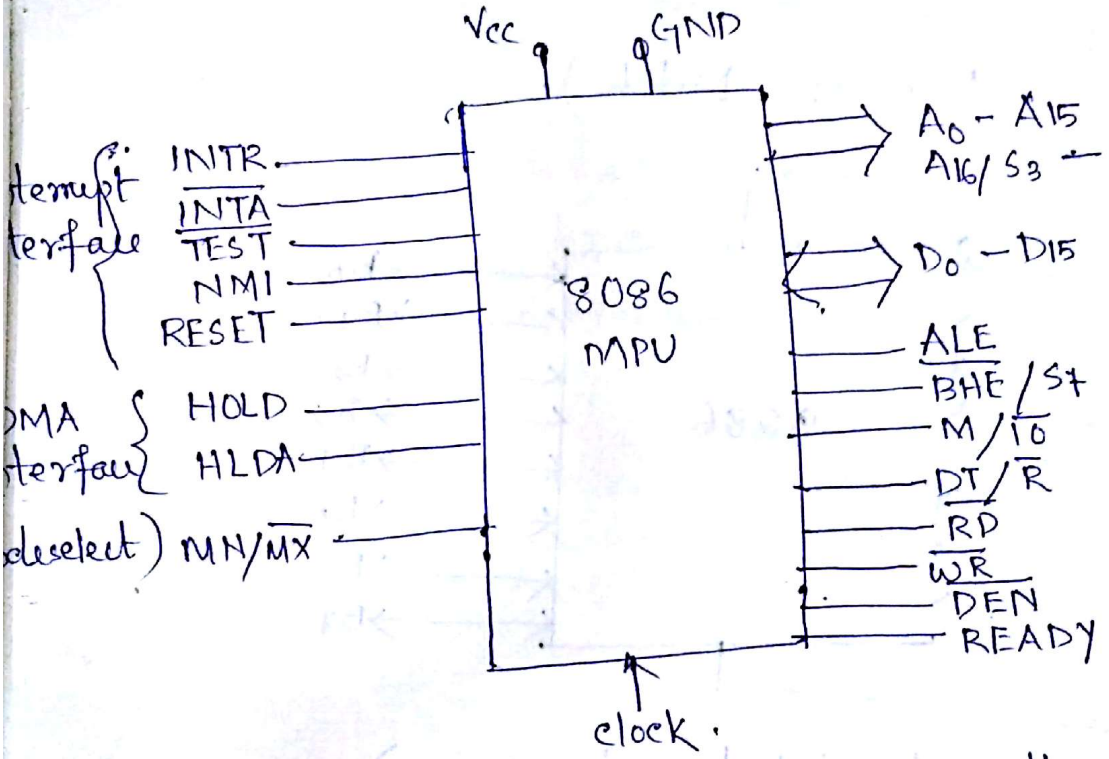
→ Only the difference is that, the status signals $\overline{S_2}\,\overline{S_1}\,\overline{S_0}$ output just prior to the beginning of the bus cycle. This status information is decoded by the 8288 bus controller to produce control signals ALE, $\overline{MRDC}$, DT/$\overline{R}$ and $\overline{DEN}$.

Minimum - mode 8086 system :-

→ In Minimum mode, 8086 microprocessor, MN/$\overline{MX}$ pin maintained at a logic 1.

→ In this mode, all the control signals are given out by the processor it self to implement the memory an I/O interfaces.

→ A single processor is used in this system.

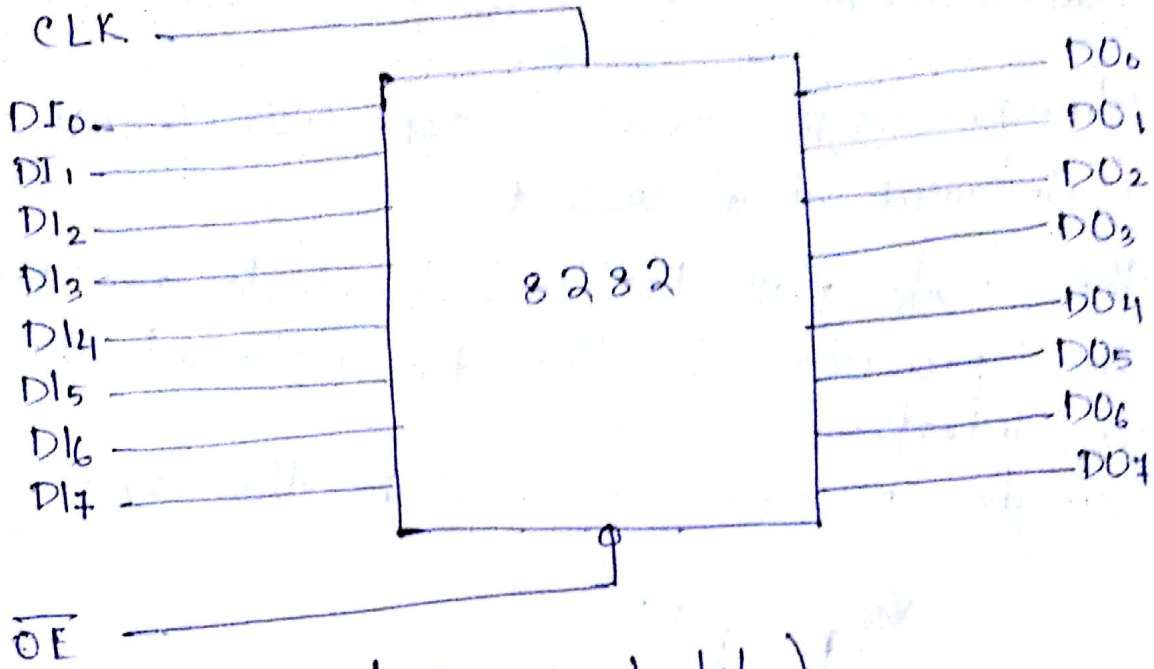


→ Along with the processor, the other components that are used in the system are latches, (8282) transceivers, clock generator, (8284) memory and I/O devices.

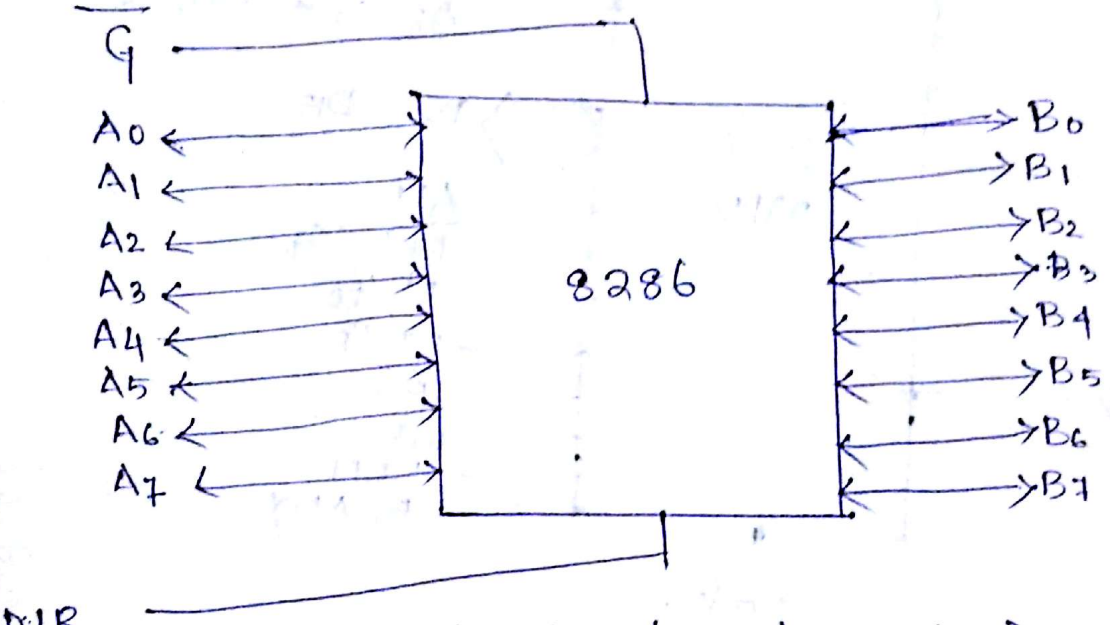
→ The Latch ckt is used to latch the address lines from the multiplexed address/data bus.

→ The Latches used are the 8-bit latches and generally buffered output of D-type FFs like IC 8282. It is controlled by the ALE signal.

→ 3 - Latches are used to generate a 20-bit address.



(8282 Latch)



(Octal bus transceiver)

→ The transceivers are bidirectional buffers and sometimes they are called as data amplifiers. They are used to separate the valid data from the multiplexed address/data signal.

→ To decode the 16-bit data lines (D0 - D15) two transceivers are required.

→ The bidirectional input/output lines are through A0 - A7 and B0 - B7. The $\overline{G}$ input is used to enable the buffer for operation

→ Again, the logic level of the DIR (direction) input selects the direction in which data are to be transferred.

e.g. If it is at logic 1, transmission is from A to B.

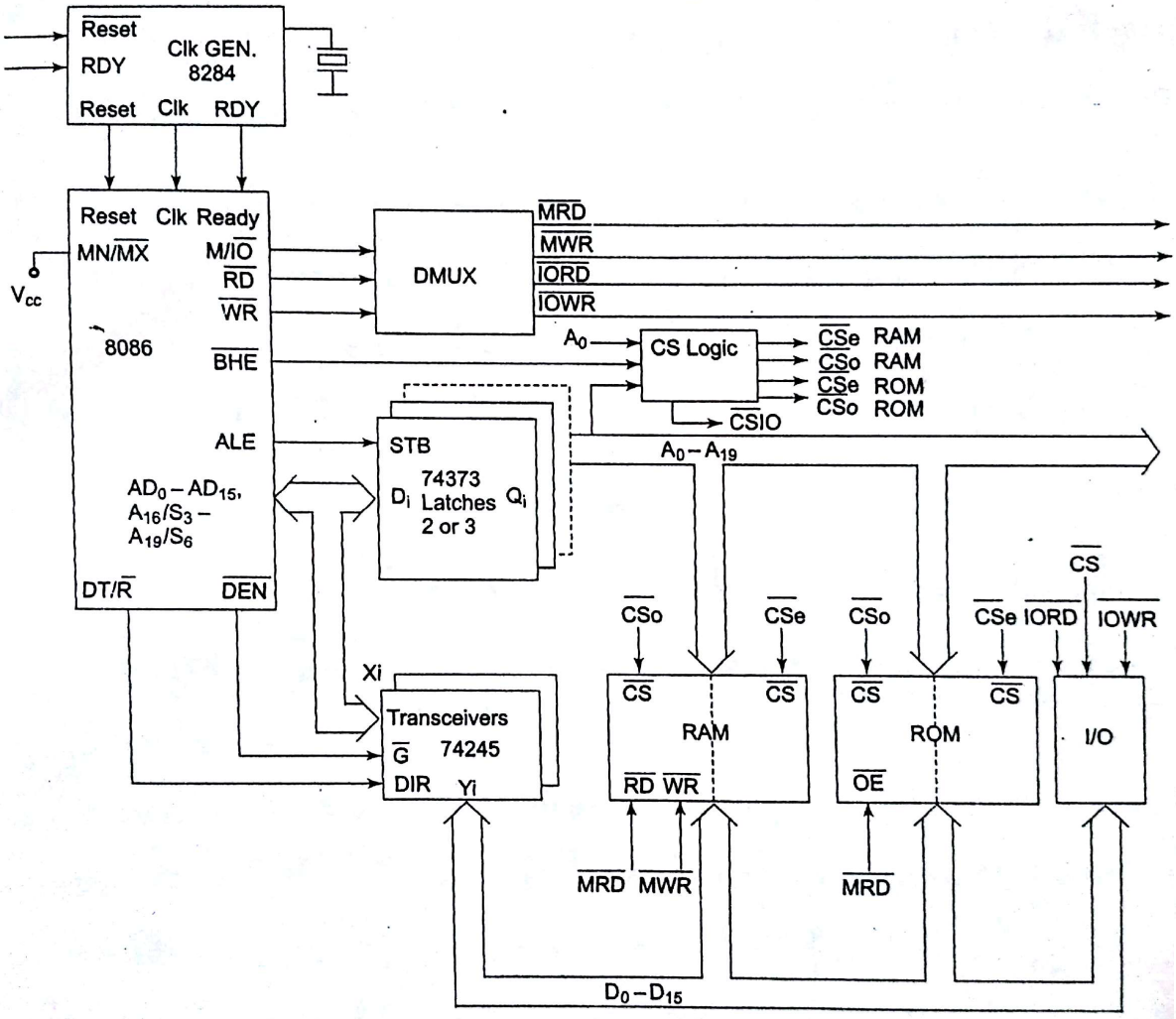
" 0, " B to A.

→ The 8284 clock generator connected with 8086 microprocessor provides some basic functions like

clock generation

RESET synchronization

READY synchronization.



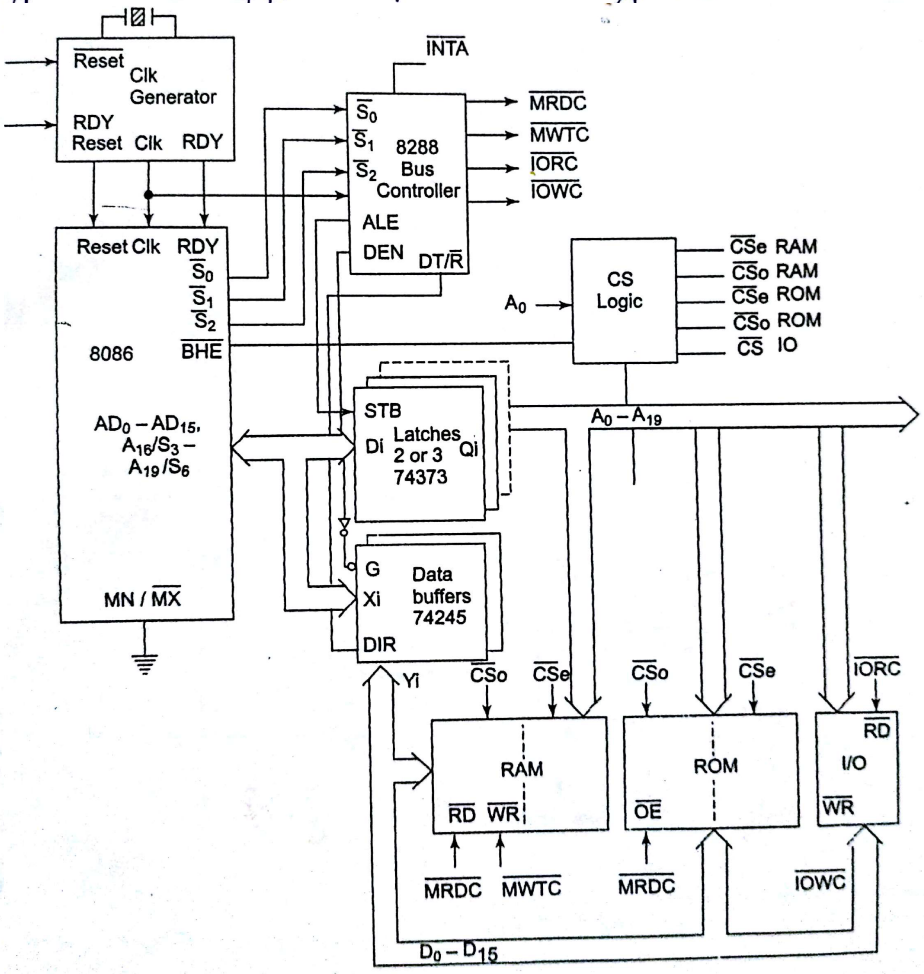
Maximum mode 8086 system :-

→ In this mode MN/$\overline{MX}$ pin is grounded.

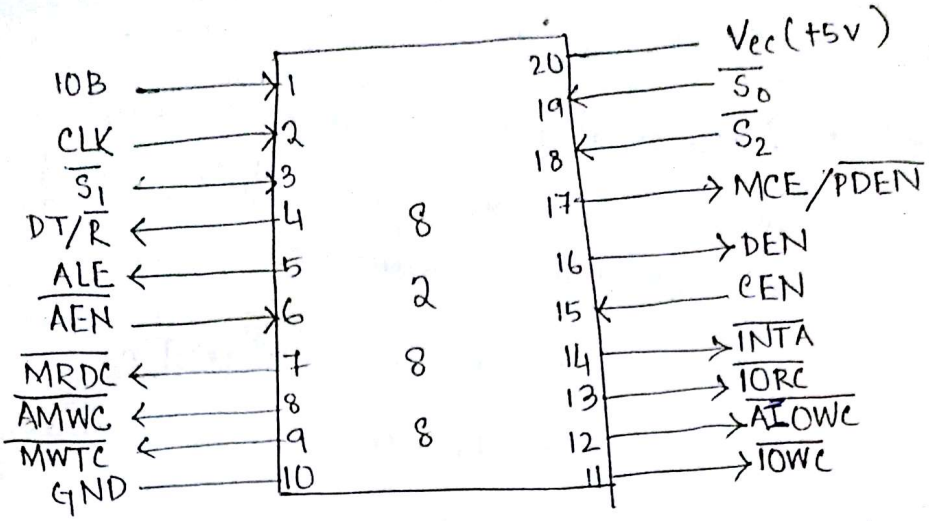
→ Maximum mode differs from minimum mode is that some of the control signals must be externally generated.

→ So in this mode, the processor produces signa for multiprocessor system environment.

→ In maximum mode the $\overline{WR}$, $\overline{RD}$, DT/$\overline{R}$, $\overline{DEN}$, ALE and $\overline{INTA}$ signals are no longer produced b 8086 processor. So it requires an external bus controller IC-8288 bus controller, which decodes the $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ status lines to identify the type of bus cycle.



→ 8288 Bus Controller :-



(Pin diagram of Bus controller Intel 8288)

→ The bus controller Intel 8288 is used in the maximum mode configuration of 8086 microprocessor.

→ It is used to decode the status line signals $\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$ to create system bus control signals, and to connect more than one Intel 8086 in one system bus or to generate more than one system bus.

→ The functions of different pins are

1) $\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$:- Bus cycle status signals. These are decoded and control signals are produced.

2) CLK :- TTL clock signal. It is connected to CLK output from clock generator 8284.

3) $\overline{AEN}$, CEN, IOB :- Bus priority and mode control signals.

$\overline{AEN}$ - Bus priority enable

CEN - Command enable

IOB - Mode control

These are used to control the generation of various control signals.

4) $\overline{MRDC}$: Memory Ready control signal.

5) $\overline{MWTC}$: Memory write control signal.

6) $\overline{AMWC}$: Advance memory write control signal. It function is same as $\overline{MWTC}$ but is active one clockpulse earlier.

7) $\overline{IORC}$: I/O Device Read control signal.

8) $\overline{IOWC}$: I/O Device Write control signal.

9) $\overline{AIOWC}$: Advanced I/O Device Write control signal. It activated one clockpulse earlier than $\overline{IOWC}$ otherwise its function is same as $\overline{IOWC}$

10) $\overline{INTA}$: Interrupt Acknowledge. It is output during two interrupt acknowledge bus cycles and is used as memory read control sig

11) MCE/$\overline{PDEN}$: Cascade / Peripheral Data Enable. It used in configuration using Priority Int controller IC8259.

12) ALE : Address Latch Enable signal.

13) DT/$\overline{R}$: Data Direction control signal.

14) DEN : Data Buffer control signal.

15) V_{cc} : +5v, power supply.

16) GND : Ground connection pin.

5

8086-7

→ 8086 Microprocessor Interrupts :—

→ 8086 microprocessor has 256 types of interrupts. INTEL has assigned a type no. to each interrupt. The type numbers are in the range of 0 to 255.

→ The interrupts can be initiated either by executing "INT n" instruction where n is the type no. of the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

→ Out of the 256 interrupts,

→ type 0 – type 4 are dedicated for specific functions by INTEL and they are called INTEL predefined interrupts.

→ next 27 i.e from type 5 – type 31 are reserved by INTEL for use in future or for system calls/services.

→ The upper 224 interrupts i.e from type 32 to type 225 are available to the user as hardware or software interrupts.

INTEL Predefined Interrupts :—

→ The INTEL predefined interrupts are :

Divide by zero (type 0)
Single step (type 1)
Non-maskable interrupt - NMI (type-2)
Break point " (type-3)
Interrupt on overflow (type-4)

Divide by zero interrupt :—

→ This interrupt is used by INTEL on a part of execution of the divide instruction.

→ 8086 microprocessor will automatically do a type-0

interrupt if the result of a division operation is too large to fit in the destination register. It is also non-maskable.

Single-step interrupt :-

→ When the TF = 1, the 8086 processor will automatically generate a type-1 interrupt after execution of each instruction.
→ The user can halt the processor temporarily and return the control to user so that after execution of each instruction the processor status can be verified.
→ If they are correct then we can go to execute next instruction.
→ Execution of one instruction by another is known as single step and this feature will be useful to debug a program.

Non-maskable interrupt :-

→ This interrupt is produced when the processor receives a high signal on the NMI pin.
→ It is used to save program data or processor status in case of system AC power failure.

Break-Point interrupt :-

→ It is used to implement a break point function which executes a program partly or upto the desired point and then returns the control to user.
→ It is useful to debug a program by executing the program part by part.
→ The user can insert "INT 3" instruction at the desired locations and execute the program.

Overflow interrupt :-

→ During ALU operation, if OF flag is active, this interrupt is used to point out an error condition.
→ The type-4 interrupt is initiated by "INT O" instruction.

Software Interrupts of 8086 :-

→ The 'INTn' instructions are called software interrupts.
→ The 'INTn' instructions will initiate type-n interrupt and the value of n is in the range of 0 to 255.
→ Therefore all the 256 type interrupts including the INTEL predefined and reserved interrupts can be initiated through INTn instruction.
→ The software interrupts are non-maskable and with higher priority than hardware interrupts.

Hardware Interrupts of 8086 :-

→ The interrupts initiated by applying appropriate signals to INTR and NMI pins of 8086 are called hardware-interrupts.
→ All the 256 type interrupts including INTEL predefined and reserved interrupts can be initiated by applying a high signal to INTR pin of 8086.
→ When a high signal is applied to INTR pin and the hardware interrupt is enabled/unmasked then the processor runs on interrupt acknowledge cycle to achieve type number of the interrupt from the device which sends the interrupt signal.
→ The interrupting device can send a type no. in the range of 0 to 255. Therefore all the 256 type of interrupts can be initiated through INTR pin.

→ The hardware interrupts initiated through INTR are maskable by removing the interrupt flag (IF) i.e the hardware interrupts are masked / disabled when IF = 0 and they are unmasked when IF = 1.

→ The interrupts are initiated through INTR pin.

8086 Interrupt priority :—

→ The priority structure for various interrupts are

Divide by zero — Highest
NMI :
INTR :
Single-step — Lowest

Direct Memory Access —

→ In direct memory access, the external Logic is permitted to access the memory directly and to execute the read/write operation.

DMA in Minimum mode (MN/$\overline{MX}$ = +5v)

→ The direct memory access in minimum mode of 8086 up through HOLD and HLDA signal as follows :
→ When the external Logic requests to access the memory directly, it makes a request through high signal at HOLD pin.
→ The processor acknowledges throug HLDA pin at the end of the current bus cycle.
→ Processor ~~tran~~ shares the address ; data and control lines simultaneously.

DMA in Maximum mode (MN/$\overline{MX}$ = GND)

→ In maximum mode $\overline{RQ}/\overline{GT_0}$ and $\overline{RQ}/\overline{GT_1}$ are used for the direct memory access operation.
→ So $\overline{RQ}/\overline{GT_0}$ and $\overline{RQ}/\overline{GT_1}$ are the two channels for DMA, with which two separate external Logics may access the memory directly.

→ The DMA operation is like.
→ The external logic makes a request for DMA by sending a low pulse at $\overline{RQ}/\overline{GT}$. The 8086 samples $\overline{RQ}/\overline{GT}$ at low to high transition of CLK.
→ 8086 receives the DMA request at the end of the current bus cycle or an idle clock period by sending a low pulse on the same $\overline{RQ}/\overline{GT}$ pin. The 8086 shares the address/data and control lines at the same time.
→ When memory operation is completed, the external logic conveys the processor by sending a low pulse on the same $\overline{RQ}/\overline{GT}$ pin.

Halt state :—

→ When HLT operation is performed the processor enters into Halt state.
→ In this state, the 8086 µp halts/stops fetching and performing instructions.
→ In Halt state no signals as used. The Halt state can be terminated by an Interrupt request or a Reset operation.
→ During Halt state, DMA request on HOLD pin (min) or $\overline{RQ}/\overline{GT}$ (max) pin gets identified and the DMA operation occurs.
→ Again, on the completion of DMA, the 8086 returns to Halt state.

Wait for Test state :—

→ When the 8086 µp performs the WAIT instruction, it enters WAIT FOR TEST state.
→ In this state the 8086 produces an endless sequence

of idle clock periods.

→ During these clock periods the address/data and control lines are not floated and processor may perform memory read bus cycles to fill up the instruction object code queue.

→ A Low signal on $\overline{TEST}$ input pin terminates this wait state. If a valid int. req.. is received in this wait state, the interrupt will be processed.

→ But after the processing of interrupt the 8086 will come to wait stat.

→ The main utility is to synchronize the 8086 program with respect to an external signal, coming from a timer ckt or an external logic which mark an event.

Comparison between 8086 and 8088 :—

→ The comparison between 8086 and 8088 is as follows.

* The 8088 has the same ALU, the same set of registers and the same instruction set as 8086.
* The 8088 has a 4-byte instruction queue in BIU as against the 6-byte queue of 8086.
* The 8088 BIU fetches a new instruction byte to enter into the queue, every time there is a one byte place in the queue. On the other hand 8086 BIU fills the queue when its empty space is of 2 bytes.
* Like 8086, 8088 has a 20-bit address bus. Memory capacity is also 1MB.
* 8088 has an 8-bit data bus.
* $\overline{BHE}$ signal is not present in 8088. so it does not have even and odd address banks.

8086 Microprocessor Instructions :- (M) (3) (1) (2)

→ Instructions are classified on the basis of function they perform.

M-(3)

8086-2

→ The instruction types are

1. Data copy / transfer instructions.
2. Arithmetic and Logical instructions.
3. Branch instructions
4. Loop instructions
5. Machine control instructions
6. Flag Manipulation instructions
7. Shift and Rotate "
8. String "

1. Data copy / Transfer instructions —

→ All the instructions which perform data movement comes under this group.

→ They are move, load, store, exchange, input, output, push and pop instructions.

→ The source of data may be a register, memory location, port etc. & the destination may be a register, memory location, or port.

e.g. MOV, XCHG, PUSH, POP, LDS, LEA, IN, OUT etc.

```
MOV  AX, 1234H
MOV  AX, BX
MOV  AX, [SI]
MOV  AX, [1234]
MOV  AX, 1234 [BX] [DI]
```

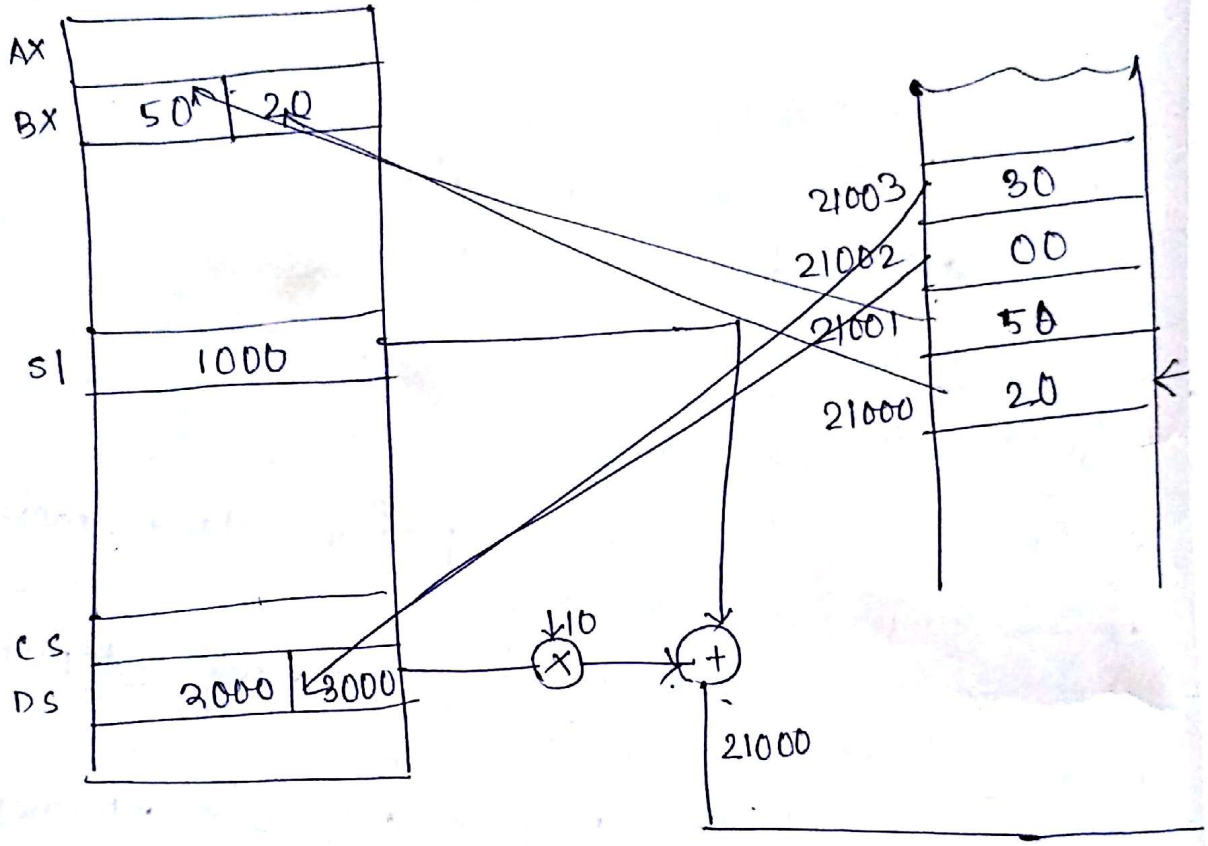
Load / store / Exchange instructions —

→ LDS reg, mem.

→ it loads a word (16-bit) from the sp memory into specified reg.

e.g. ~~LDS SI, [2000]~~

~~SI = 1~~ LDS BX, [SI] seg - DS



→ Again it also loads a word from the next to memory locations into DS register.

e.g. LDS SI, [2000].

→ LES reg, mem :—

→ it loads a word (16-bit) from the specified m locations into the specified register.

→ Again, it also loads a word from the next t memory locations into ES register.

LES DI, [BX].

LEA reg, mem :

→ it loads offset address into the specified register. It can also determine the offset address of a variable memory location specified in the instruction to load into the specified register.

```
LEA   BX , [DI]
LEA   BX , 6000 [SI]
LEA   SI , 44H [BP]
LEA   BX , [SI]
LEA   CX , [BX + SI + 5000H]
```

→ LAHF :-

→ it loads the low-order 8-bits of the flag register into AH register.
i.e SF, ZF, AF, PF & CF.

→ SAHF :-

→ it stores the contents of AH register into the lower order bits of Flag register.

→ XCHG, reg :-

→ it exchanges the contents of the 16-bit specified register with the contents of AX.

```
XCHG  [5000] , AX
XCHG  BX .    (AX ⟷ BX)
```

* The exchange instruction can't perform exchange operation between segment or memory to memory.

XLAT :-

→ Translat operation in code conversion problems.

→ This instruction reads a byte from the lookup tab
it first adds the content of AL with BX to perfor
a memory address within DS. Then it copies the
contents of this address into AL

i.e (AL) + (BX) + (DS x 10) → AL.

DS = 0300

BX = 0100

AL = 41_H — ASCII 'A'

P.A = DS x 10 + (BX + AL)

= 03000 + (0100 + 41)

= 03141

(ASCII → EBCDIC conversi

A Z

→ after executing XLAT instruction,

03141 → AL

i.e AL = 41

i.e letter 'A' in EBCDIC

→ IN instruction

IN ac, port :-

→ it transfers data from a port to accumulator
for 8-bit, AL register & for 16-bit, AX register.

e.g IN AL, 8-bit port address

IN AX, 16 bit "

→ IN ac, DX,

→ it reads 8-bit or 16-bit data from a port
whose address is in DX register.

→ for 8-bit data, DX → AL

16-bit " , DX → AX.

→ OUT port, acc :—

→ It transfers data from accumulator to an I/O port specified.

→ AL → 8-bit port

AX → 16-bit port.

e.g. OUT P8, AL

OUT P8, AX .

Stack Related Instructions —

PUSH —

→ it pushes the contents of the specified register/memory location on to the stack.

→ The stack pointer is decremented by 2, after each execution of the instruction. The higher byte is pushed first and then the lower byte.

e.g. PUSH AX
PUSH DS
PUSH [2000]

PUSH AX → it pushes the contents of AX /specified register on to the top of the stack.

PUSH F → pushes the content of the flag register onto the top of the stack.

POP —

→ when this instruction is executed, it loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer. The stack pointer is incremented by 2. e.g. POP AX
POP DS
POP [2000]
POP F.

e.g. SS = 1000_H , SP = max FFFF

PA_{BOS} = 1000 x 10 + FFFF

= $1FFFF_H$

Let SP = 0020 ,

PA_{TOS} = 1000 x 10 + 0020

= 10020_H .

Arithmetic Instructions :—

→ The instructions of this group perform addition, sub multiplication, division, increment, decrement, comparis Ascii and decimal adjustment etc.

e.g. ADD, SUB, INC, MUL, DIV, CMP, DAA etc.

→ These instructions and their operations can be perfor on numbers expressed in a variety of numeric dat formats. They include unsigned or signed binary bytes or words, unpacked or packed BCD bytes or Ascii numbers.

ADD

ADD DL, 33_H (DL = DL + 33_H)
ADD BX, 2456 (BX = BX + 2456)
ADD AX, 0100 (AX = AX + 0100)
} Immediate addition

ADD [BX], AL [(DS x 10 + BX) + AL] = AL
ADD CL, [BP]
ADD BX, [SI + 2]
ADD AX, [5000_H]
} Reg to memory or memory to reg. addition.

~~ADD~~ [5000], 4000_H [(CS x 10 + 5000) + 4000] =

A

→ ADC → Addition with carry.

→ It adds two specified operands and the carry status (i.e the carry of the previous stage). The result is placed in the specified destination.

e.g. ADC ac, data —

→ adds 8-bit immediate data and carry status to the contents of AL and the result is placed in AL.

→ for 16-bit data, AX reg is used.

ADC AL, AH (AL = AL + AH + carry)

ADC CX, BX (CX = CX + BX + carry)

ADC DH, [BX]

↳ (the byte contents of the data segment memory location addressed by BX is added with DH with carry and the result is stored in DH.)

~~e.g. ADC AX, CX~~

SUB :- Subtract operation.

→ SUB ac, data

(A - data = A)

→ SUB mem/reg, data

(mem/reg - data = mem/reg)

→ SUB mem/reg, reg/mem.

SBB :- Subtraction with borrow.

→ DEC → Decrement by one

DEC D (D = D - 1)

→ NEG → Negation, NEG D

(D = 0 - D)

→ Multiply and Divide instructions —

→ MUL

→ it is used to multiply two unsigned nos. of the contents of a specified register or memory. 8-bit, it is multiplied with AL, for 16-bit it is multiplied with AX.

→ For 8-bit result is stored in AL
16-bit " " in AX & DX.

IMUL

→ it is an instruction for multiplication of two signed nos. The result is also a signed no.

→ for 8-bit, AL.
16-bit, AX & DX (MSB)

→ DIV

→ division of two unsigned nos.

Q - AX
R - DX.

→ IDIV

→ division of two signed nos.

→ INC (Increment instruction)

→ INC reg (increment 16-bit register)

→ It adds one (0001) to the contents of the specified register.

e.g. INC BX

→ INC mem/reg (increment register/memory)

→ the contents of the specified 8-bit or 16-bit register or memory location is incremented by one.

→ segment registers are not incremented.

→ DEC (Decrement instruction)

→ DEC reg (decrement 16-bit register)

→ It decrements the specified 16-bit reg. by one

DEC mem (decrement memory)

2.
68.
22
01
39
36.

→ Decimal Adjust instructions

→ DAA (Decimal adjust after addition)

→ It is used to convert the result of the addition of two BCD numbers to a valid BCD number.

→ The operation is done only on the contents of AL.

→ The DAA instruction should be used always after an ADD or ADC instruction immediately.

→ If the lower nibble of the result is greater than 9 or if AF = 1, add 06 to lower nibble in AL. After adding 06 in the lower nibble, if the upper nibble of AL is greater than 9 or if carry flag (CF = 1) DAA instruction adds 60H to AL.

e.g. AL = 73 , BL = 29 .

ADD AL , BL , AL = AL + BL

AL = 73 + 29 = 9C

⇒ AL = 9C

$$\begin{array}{r} 12 \\ +06 \\ \hline 18 \end{array}$$ AL = 02

CF = 01 .

→ DAS (Decimal adjust after BCD subtraction) —

→ It is used to get correct result in BCD, when a BCD no. is subtracted from another BCD no .

→ It is used after SUB or SBB instruction .

It operates only on the contents of AL register

→ AAA (ASCII adjust for addition)

→ Numeric data from an input device into a microproc are usually in ASCII code .

→ The AAA instruction is used after the addition to get correct result in unpacked BCD no . This instruction operates only on AL register .

→ AAD (Adjust AX Register for division) — (ASCII adjust before division)

→ It converts two unpacked BCD digits into AX to the equivalent binary no. The equivalent binary no. is placed in AL register .

→ This adjustment is done before dividing two unpacked BCD digits in AX by an unpacked BCD byte .

→ After division AL contains quotient

AH contain remainder .

→ AAM - (Adjust result of BCD multiplication)

→ It is used after multiplication of two unpacked BCD nos. to get correct result in unpacked BCD.

→ The result is placed in AX.

AAS (Ascii adjust for subtraction)

→ It is used to get the result in unpacked BCD after the subtraction of the Ascii code of a no. from the Ascii code of another number.

→ It works on AL register.

→ CBW (Convert byte to word)

→ It copies the sign bit of the operand in AL register to all the bits of AH register.

→ If sign bit of AL is 1, FF is placed in AH.

" AL is 0, 00 is placed in AH.

→ ~~CWD (Convert signed word to signed double word)~~

→ ~~It extends the sign~~

→ CMP ac, data —

→ It compares 8-bit immediate data with the contents of AL register or 16-bit data with AX.

e.g. CMP AL, 58H
CMP AX, 9230H.

CMP AL, CL
CMP AL, [BX].

CMP CL, 23H.
CMP [BX], 23H.

Logical Group

→ Instructions of this group perform logical AND, OR, XOR, NOT, operate, rotate, shift etc.

→ **AND**

→ bit by bit logical AND operation of two operands.

→ AND AL, 05H

AND AX, 1234H

AND AL, BL

AND AX, BX

OR

→ bit by bit logical OR operation

XOR

→ EX-OR operation.

NOT

→ 1's complement of specified mem/reg.

TEST

→ it performs a bit by bit logical AND operation on the two operands.

→ But the difference is that the AND instruction changes the destination operand, while the TEST instruction does not.

→ A TEST affects only the condition of the flag register, which indicates the result of the TEST.

Shift and Rotate instructions :—

→ Shift and Rotate instructions manipulate binary nos. at the binary bit level.

→ It has 4 different shift instructions.

SHL (Shift Logical Left)
SAL (Shift Arithmetic left)
SHR (Shift Logical right)
SAR (Shift Arithmetic right)

→ Logical shift operations function with unsigned numbers, and arithmetic shift function with signed nos.

e.g.
```
SHL AX, 1
SAL BX, CL
SHR CX, CL
SAR SI, 1
```

Rotate instruction —

→ Rotate instructions position binary data by rotating the information in a register or memory location from one end to another or through the carry flag.

e.g. ROL → Rotate left

ROR → Rotate right
RCL → Rotate Left through carry
RCR → Rotate right through carry.

String Manipulation Instructions —

→ A series of data bytes or words available in memory at consecutive locations are called byte strings or word strings.

→ The length of the string is always stored in CX r

→ For referring to a string, two parameters are requi
(a) starting or end address of the string
(b) Length of the string.

→ The starting address of source string is locate the memory location by combining SI and DS cor

→ The starting address of destination string is loca in memory location combining DI and ES.

→ **LODS** : Load string byte or string word.

→ it loads AL/AX register by the contents of a string pointed by DS:SI register pair.

LODSB — Loads a byte into AL

AL = DS:[SI] , SI = SI ± 1

LODSW — Loads a word into AL

AL = DS:[SI] , SI = SI + 2

SI = 1000 , DS = 1000 (DS×10 + 1000)

= 11000_H, 11001_H

the instruction loads the contents of memory locations 11000_H and 11001_H into AX.

→ STOS :- Store string byte or string word

→ The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES:DI register pair.

STOSB → Store string byte

ES:[DI] = AL ; DI = DI + 1

STOSW → store string word.

ES:[DI] = AL , DI = DI + 2 .

→ MOVS : Move string.

→ it transfers data from one memory location to another.

MOVSB : Move string byte , ES:[DI] = DS:[SI].
DI = DI + 1 , SI = SI + 1

MOVSW : Move string word.

→ CMPS : compare string.

→ it compares two sections of memory data as bytes or words.

CMPSB :- Compare string bytes

CMPSW :- Compare string word.

→ SCAS : Scan string

→ it scans a string of bytes (SCASB) or words (SCASW) for an operand byte or word specified in the register AL or AX.

→ Control transfer or Branching Instructions —

→ Unconditional JMP

→ short jump, near jump & and far jump.

→ Intrasegment jump
→ Intersegment jump.

JMP 1200

→ Conditional Jump

→ It test the following flag bits
S, Z, CY, P and O.

LOOP :—

→ The LOOP instruction is a combination of a decrement & and conditional jump.

→ Subroutine instructions

→ Subroutine is a special segment of program that can be called for execution from any p in a program.

→ The two basic instructions for subroutine handling is CALL and RET.

e.g. CALL 2200

RET 1008 etc.

→ Intrasegment & Intersegment.

Flag Manipulation instructions

CLC → clear ca CF
CMC → complement CF
STC → set CF
CLD → Clear DF
STD → set DF
CLI → clear IF
STI → set IF
LAHF → Load AH from flage.
SAHF → store AH into flage.

Machine Control Instructions —

→ The machine control instructions donot require any operand.

WAIT — wait for TEST pin to go low.
to test current status of the processor.

HLT — Halt the processor

NOP → No operation

LOCK → Bus lock instruction

ESC → Escape to external device like numeric co-processor.

REQUIREMENT OF ASSEMBLER DIRECTIVE:

- Assembler is a program used to convert assembly language program to it's equivalent machine code modules which may further be converted to executable codes.
- It decides the address of each label and substitutes the values for each of the constants and variables.
- It forms the machine code for the mnemonics and data in the assembly language program.

For completing all these tasks, an assembler needs some hints from the programmer ie. the required storage for a particular constant or variable, logical names of the segments, types of different routines and modules, end of file etc. These type of hints given to the assembler using some predefined alphabetical strings are called as Assembler directive.

These directives are commonly used in programming using the Microsoft Macro Assembler or Turbo Assebler software.

Assembler Directives :— (4) 8086-4

1. Assume :—

→ This directive is used to inform the assembler the names of the logical segments, which are to be assigned to the different segments used in an assembly language program.

2. DB (Define Byte) :—

→ This directive DB defines a byte-type variable, a variable which occupies one byte of memory space.

3. DW (Define Word) :—

→ The directive DW defines a word type variable i.e a variable which occupies two bytes of memory space.

4. DD (Define Double word) :—

→ The directive DD defines a double word type variable a variable which occupies four bytes of memory.

→ The defined variable may have one or more values in the statement.

5. DQ (Define Quadword) :—

→ The directive DQ defines a quad-word type variable (a variable which occupies 8 bytes of consecutive memory locations).

6. DT (Define Tenbytes) :—

→ It defines a variable of ten bytes.

→ END (End of program) :—
— it informs assembler the end of a program module.

→ ENDP (End Procedure) :- —

→ it informs assembler the end of a procedure.

→ In assembly language programming, subroutines are called procedure. A procedure may be an independent program module to give certain result or the required value to the calling program.

→ END M (End Macro) :- —

→ It is used to inform assembler that this is end of the macro. The directive ENDM is used with the directive MACRO to enclose macro instructions.

→ END S (End of segment) :- —

→ It informs assembler that it is the end of a segment.

→ EQU (Equate) :- —

→ It is used to give a name to certain value / symbol. If any value of symbol is used many times in an assembly language program, a name given to the value or symbol to make programming easy.

→ Even (Align or Even memory address) :- —

→ It informs assembler to increment the content of the location counter to the next memory address, if it is already not at an even address.

→ The content of the location counter holds the address of the memory location assigned to an instruction or variable or constant.

→ EXTRN (External) :-

→ It informs that the names, procedures and labels following this directive have already been defined in some other program modules.

→ when the programmer informs assembler that the declared item is an external one, the assembler puts this information in the object code file so that the linker can connect the concerned two program modules together.

→ DD (Define Double word) :-

→ It defines a double word type variable (a variable which occupies 4 bytes of memory). The defined variable may have one or more values in the statement.

→ DQ (Define Quadword) :-

→ It defines a quad-word type variable (a variable which occupies 8 bytes of consecutive memory locations). The defined variable may have one or more values in the statment.

→ GLOBAL :-

→ It can be used in place of public or extrn directive. In the program module where a name or symbol is defined, it can be declared public using GLOBAL directive.

→ Group :-

→ It informs assembler to form a logical group of the segments named after this directive. The segments which are grouped into one logical group will have

the same segment group base.

→ INCLUDE (Include source code from file) :- —

→ It informs assembler to insert a block of codes from a named file into the current program module. It is convenient to keep all data and macros in a file called header file or include file.

→ LABEL (Label) :- —

→ In assembly language program, labels are used to give names to memory addresses.
→ when assembler begins assembly process, it initializes a location counter to keep the trac of memory location.
→ The content of the location counter holds the address of a memory location assigned to an instruction during assembly proces.

→ LENGTH (Length) —

→ It is an operator to determine the no. of elements in a data item such as an array a streing.

→ LOCAL —

→ when certain variables, constants, procedures labels are to be used in only one progra module, they are declared local using LOCAL directive.

→ MACRO :-

→ A sequence of instructions to which a name is assigned is called a macro.
→ Macros are same as subroutines.
→ The MACRO directive informs assembler the begining of a macro. It follows the name which is given to the macro.

→ NAME :-

→ A Large program may contain several program modules. Specific name can be given to each program module using NAME directive.

→ OFFSET :-

→ It is an operator to determine the offset (displacement) of a variable or procedure with respect to the base of the segment which contains the named variable or procedure.
→ The operator can be used to load a register with the offset of a variable.

→ ORG (Originate) :-

→ The ORG directive directs assembler to set the location counter at memory address specified after the directive.
→ If ORG directive is used, the location counter is set to the memory address specified with ORG directive.

PROC (Procedure) :- —

→ The directive PROC indicate the start of a procedure. The type of the procedure FAR or NEAR is to be specified after the directive.

→ NEAR is used to call a procedure which is within the program module and FAR is used to call from some other program module.

PTR (Pointer) :- —

→ It indicates the type (BYTE, WORD or DWORD of a variable or label.

PUBLIC (Public) :- —

→ A large program usually consist of several prog modules. The variables, constants, procedures and labels defined in a program module may be used in some other program module. So such item are to be declared public using PUBLIC directive

RECORD (Record) :- —

→ It defines the bit pattern within a byte o word.

Segment (Logical segment) —

→ It indicates the beginning of a logical segment

→ The SEGMENT and ENDS directives are used to enclose a logical segment containing code or data.

SEG (Segment) —

→ It is an operator. It is used before a variable, procedure and label.

SHORT (Short) :—

→ It is an operator. It informs assembler that only one byte for the displacement is required to code a jump instruction.

SIZE (Size) :—

→ It is an operator to determine the number of bytes allocated to a data item. It differs from the LENGTH operator as the LENGTH operator determines the no. of elements in a data item.

Struck or STRUC (structure Declaration) :—

→ It defines the begining of a data structure.
→ Within the named data structure the data type which is a collection of primary data types such as DB, DW, DD etc is declared.

Type :—

→ It is an operator. It determines the type of a variable. It actually determines the no. of bytes allocated to the variable and substitutes the no. in place of the variable before which TYPE appears.

Branch Program Structure :—

IF - THEN - ELSE :—

→ The high-level language IF - THEN - ELSE is expressed in general by the program structure

```
IF <condition>
THEN
ELSE statement ;
statement ;
```

→ If the condition tested for is satisfied, the state associated with THEN is executed.
Otherwise, the statement corresponding to ELSE pa

The Loop Program Structure - REPEAT - UNTIL and WHI

→ In many practical applications a group of instruc may need to be executed over and over again until a condition is achieved. This type of cont flow program structure is known as a loop.

→ The high-level Language REPEAT - UNTIL construct an example of a typical loop operation and is expressed in general by the program structure

```
REPEAT
    statement 1 ;
    statement 2 ;
        :
    statement N ;
UNTIL < condition >
```

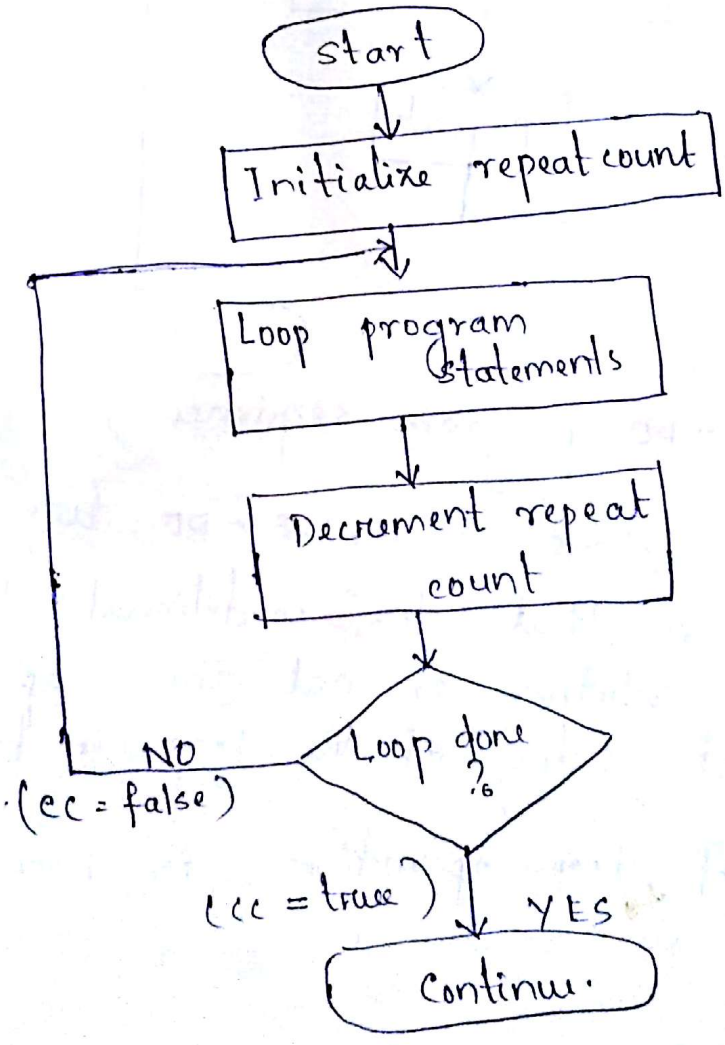
→ After statements 1 through N are performed a conditional test is performed by the UNTIL statem
→ If this condition is not satisfied, control flow

is returned to REPEAT and the operations performed by the statements are repeated.

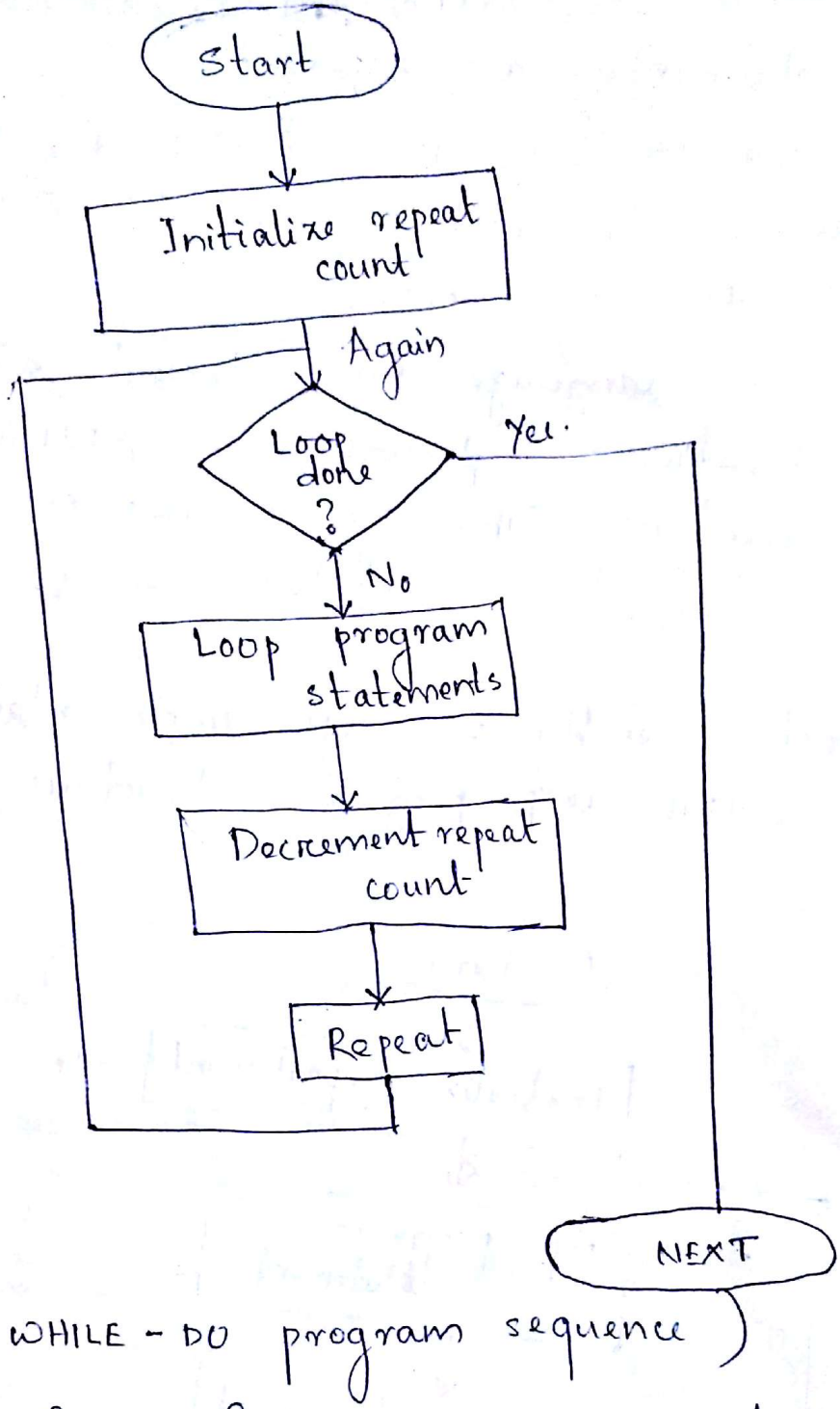
→ Again, if the condition is satisfied the loop operation is complete and control is handed over to the statement following UNTIL.

→ A high-level language DO statement with a fixed no. of iterations implements a REPEAT UNTIL program structure. This type of loop operation may be used to repeat a computation a fixed no. of times.

→ A typical assembly language implementation of a REPEAT - UNTIL loop program structure is given below.



(REPEAT - UNTIL program sequence)



(WHILE - DO program sequence)

→ The figure shows a WHILE - DO loop structure.

→ It differs in that the conditional test used to decide whether or not the loop will repeat is made at entry of the loop instruction sequence.

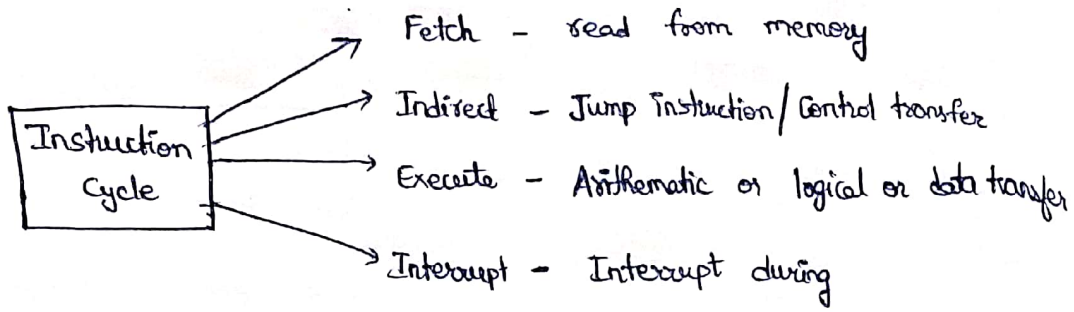
→ This type of loop operation is known as a WHILE loop or sometimes as a WHILE - DO loop

→ Its program structure is expressed in general as

```
WHILE <expression>
  statement 1 ;
  statement 2 ;
     :
     :
  statement N ;
```

→ The statements 1 through N are executed only if the conditional test executed by the WHILE statment is satisfied.

CENTRAL PROCESSING UNIT

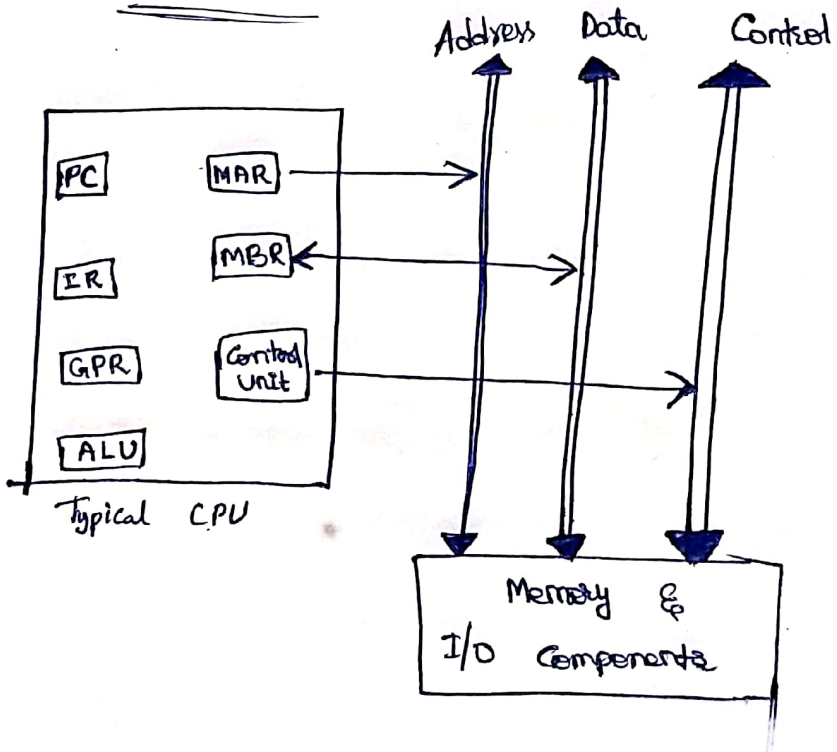
- Each instruction = 1 instruction cycle = Fetch cycle + Execute cycle
- Each cycle has no. of steps called Micro-operations.
- 

MOP - micro operations : we need to generate Control transfer

I. FETCH : (Refer IAS Architecture)

- Read opcode of Instruction in memory - FETCH
- Read operand of Instruction in memory - READ

II. FETCH SEQUENCE :



t1, t2 are time period of cycles.

(....) - Content in

... - directly

t1 : MAR ← (PC)	t1 : MAR ← (PC)
t2 : MBR ← (memory) PC ← (PC) + 1	t2 : MBR ← (memory)
t3 : IR ← (MBR)	t3 : PC ← (PC) + 1 IR ← (MBR)

- Rules for Clock Cycle Grouping :

1. Proper sequence must be followed
2. Conflicts must be avoided
   - Must not read & write same register at same time cycle.
   - MBR ← (memory) & IR ← (MBR) must not be done in same time
3. Also : PC ← (PC) + 1 involves addition
   - Use ALU
   - May need additional micro operations :)

III. INDIRECT CYCLE :

MAR ← [IR (address)] - address field of IR

MBR ← (memory)

IR (address) ← [MBR (address)]

- MBR contains an address
- IR is now in same state as if direct addressing had been used.

14. INTERRUPT CYCLE :

t1 : MBR ← (PC) (old Program counter)

t2 : MAR ← save address

t : PC' ← interrupt service routine address (new program counter)

t3 : memory ← (MBR) : Content of MBR ie. previous PC will be stored in memory

* . MICRO - OPERATIONS For Immediate Addressing Mode :

Example : MOV R1, 25H : R_1 register gets the immediate value 25H

Fetch
- T1 : MAR ← PC — PC puts address of the next instruction into MAR
- T2 : MBR ← Memory — MBR gets instruction from memory through data bus.
- T3 : IR ← MBR; PC ← PC + 1 — IR gets instruction "MOV R1, 25H" from MBR; PC increaments

Execution
- T4 : R1 ← 25H (IR) — R1 register gets value 25H from IR.

Example : MOV R1, R2 : R_1 gets data from R2.

T1 :
T2 : } Fetch same
T3 :

T4 : R1 ← R2 : R1 register gets the value from R2 register.

*. MICRO - OPERATIONS for Direct Addressing Mode :

MOV. R1, [2000 H]

Ex:

T1 :
T2 :
T3 : } Fetch is same for any — Micro-operation

T4 : MAR ← IR (2000 H) — MAR gets address 2000 H from IR

3 MOP { T5 : MBR ← memory ([2000 H]) — MBR gets contents of 2000 H from memory

T6 : R1 ← MBR ([2000 H])

*. MICRO - OPERATIONS for Implied Addressing Mode :

Ex: STC , Set the Carry Flag (CF ← 1)

T1 :
T2 :
T3 : } Fetch is same for any Micro processor operation in any Addressing mode.

T4 : CF ← 1 — Carry Flag in Flag Register becomes 1.

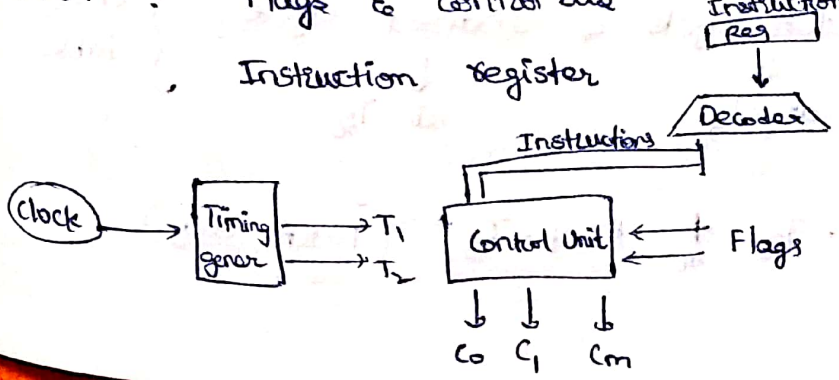
*. CONTROL UNIT IMPLEMENTATION :

Implementation of Control Unit is broadly of two types :

1. Hard wired implementation (RISC)
2. Microprogrammed implementation

*. INPUT : Flags & Control bus
, Instruction register

T : Time
I : Instruction
F : Flags
C : Control signal



(i) State Stable method :

T-States	I1	I2	.....	In
T_1	C_{11}	C_{12}	...	C_{1n}
T2	C_{21}	C_{23}	...	C_{2n}
⋮	⋮	⋮	...	⋮
Tm	C_{m1}	C_{m3}	.....	C_{mn}

C_{mn} : Control signal generated at mth Time for nth Instruction

Advantage : It is simplest method for small instruction set

Disadvantage : Flowchart method is efficient than this method

- For each C_{mn} (Control signal), to derive a Boolean expression of that signal as function of inputs.

Ex : C_5 : activates data path between external bus into MBR

PQ = 00 Fetch Cycle

PQ = 11 Interrupt Cycle

PQ = 10 Execute Cycle

PQ = 01 Indirect Cycle

AND = .
OR = +

C_5 → Fetch cycle & Indirect cycle } check table
↓ at T_2 ↓ at T_2

$C_5 = P \cdot \bar{Q} \cdot T_2 + P \cdot Q \cdot T_2$ (Fetch only)
(or)

control signal = Fetch + Execution

Example : LDA, ADD, AND are to be executed instructions

$C_5 = \bar{P}\bar{Q}T_2 + \bar{P}QT_2 + P\bar{Q}(LDA + ADD + AND)T_2$

(ii) Flow-chart / Delay

*. Example :

LOAD

		check table & diagram
t1:	MAR ← (IR address)	C_8
t2:	MBR ← (Memory)	C_5, C_R
t3:	AC ← (MBR)	C_{10}

STOR

t1	MAR ← (IR address)	C_8
t2	MBR ← AC	C_{11}
t3	Memory ← MBR	C_{12}, C_W

Suppose LOAD → A & STOR → B

$C_8 = At_1 + Bt_1$

$C_5 = At_2$

$C_{10} = At_3$

$C_R = At_2$

$C_{11} = Bt_2$

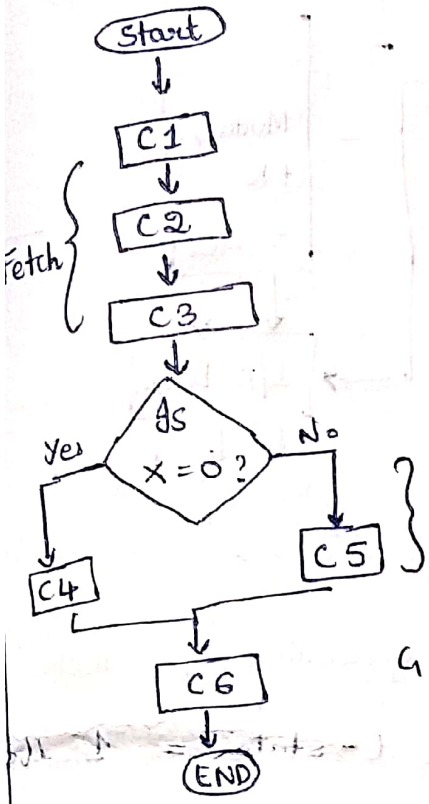
$C_{12} = Bt_3$

$C_W = Bt_3$

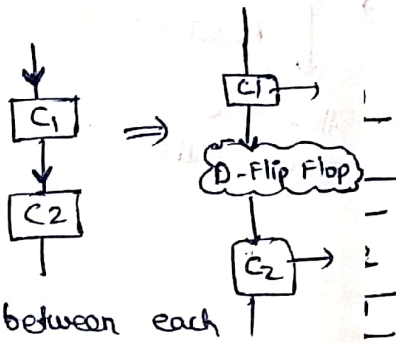
(Check next two pages for clear solution)

...

FLOW CHART (or) DELAY ELEMENT METHOD :



- Supports for maximum 2 Instructions in processor
- Between C_1, C_2 there should be time delay (done using D-Flip Flop.



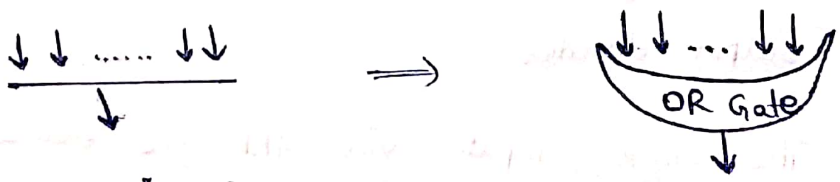
between each control signal

C_1, C_2, C_3

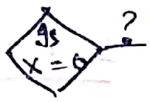
- These D-Flip Flops are inserted between every two consecutive control signals.

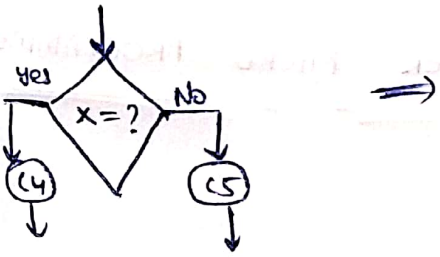
C_4, C_5

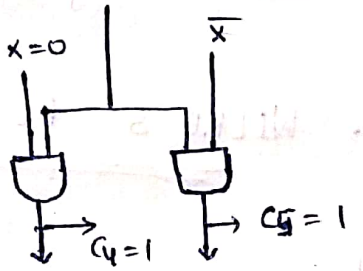
- Of all D Flip Flops only one will be active at a time. So the method is also called "One Hot Method"
- In a multiple entry point, to combine two or more paths, we use "OR GATE"



Decision box

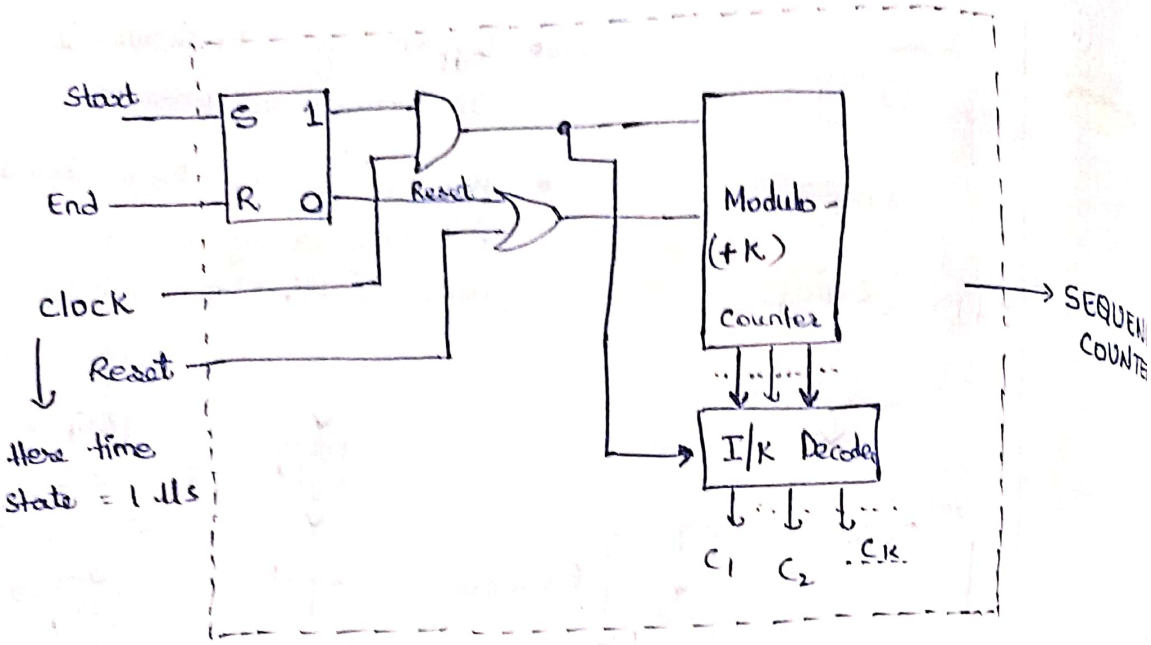






- A decision box is replaced by a set of two complementing "AND gate".

(iii) SEQUENCE COUNTER METHOD :



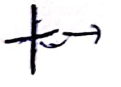
IMP • The time gap between generation of clock signals (suppose C_1 & C_2) = 1 t-state = 1 μs

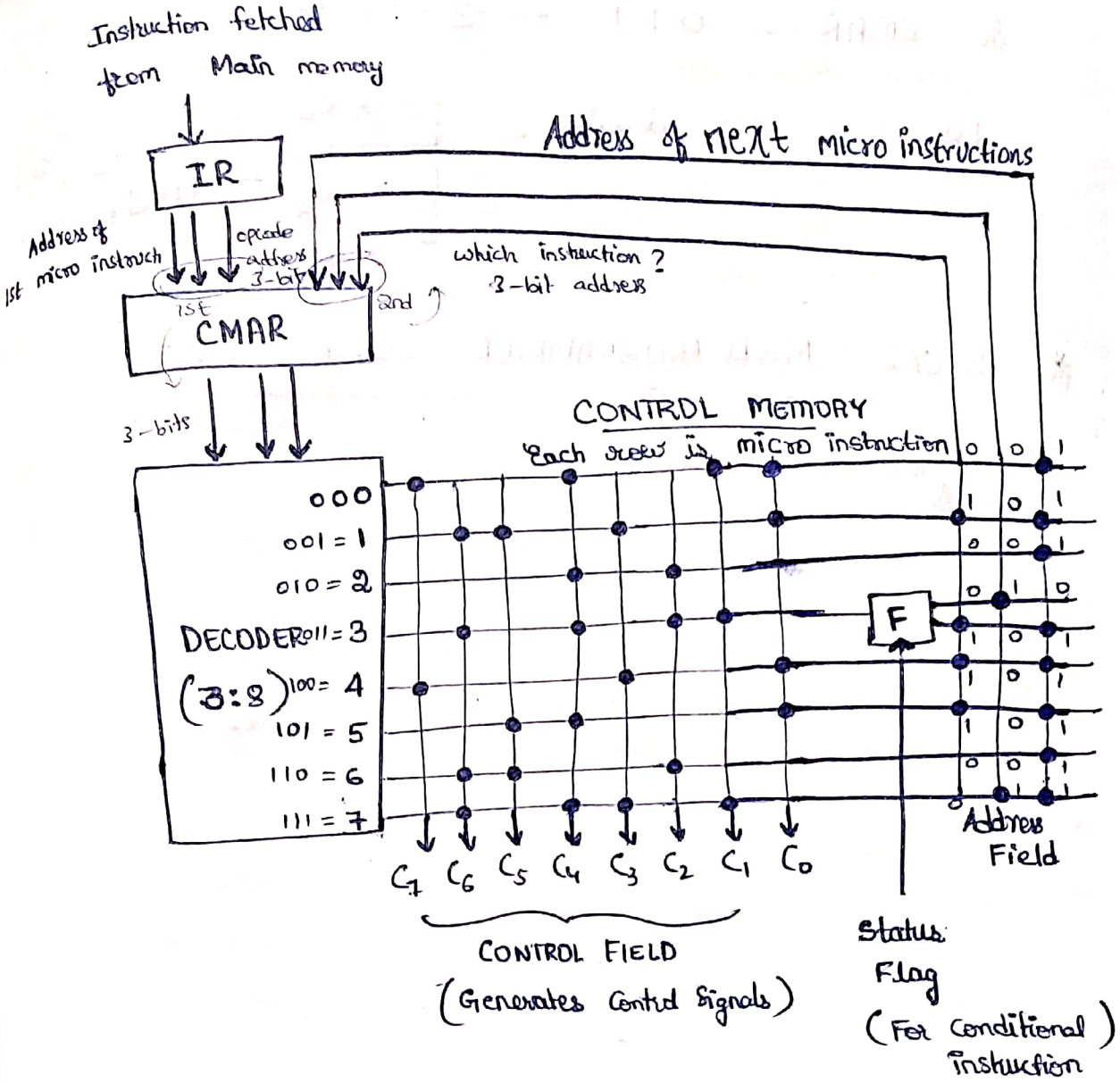
- This is the most popular form of hardwired control unit.
- Instead of D-Flip Flops, S R Flip Flop is used

IMP • If there are "K no. of distinct steps" producing control signals, we employ a "mod K" and K output decoder

IMP • The 'clock' input via AND gate ensures each count will be generated after "1 T-state"

*. WILKE'S DESIGN FOR MICRO PROGRAMMED CU :





- Each instruction = no. of micro instructions.
- Micro instructions are stored in Control Memory.
- Control Memory is present inside Control Unit.
- CMAR (Control Memory Address Register) stores the address field of IR (3 bit address field)

Example If CMAR → 010 ⇒ = 2

For 2 in 3:8 Decoder → C_4, C_2 } Generated

- Next Micro instruction (3 bit field) is given by Address field

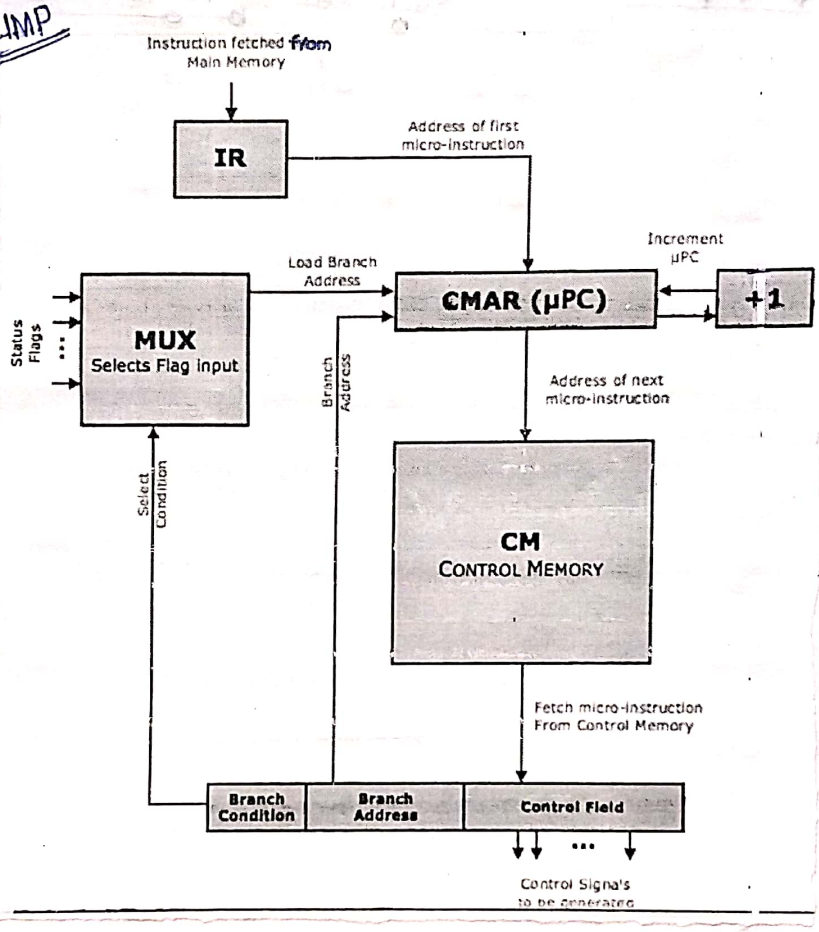
$a_0 a_1$ → next instruction = 001

If CMAR = 011 → 3

For 3 in 3:8 decoder → C_6, C_4, C_2, C_1 → Flag

*. ACTUAL MICRO PROGRAMMED CONTROL UNIT :

*IMP



μPC - micro Program Counter means the register which stores address of next micro instruction.

CMAR - Control Memory Address Register = Stores address field of IR

* TYPICAL MICROPROGRAMMED CONTROL UNIT :

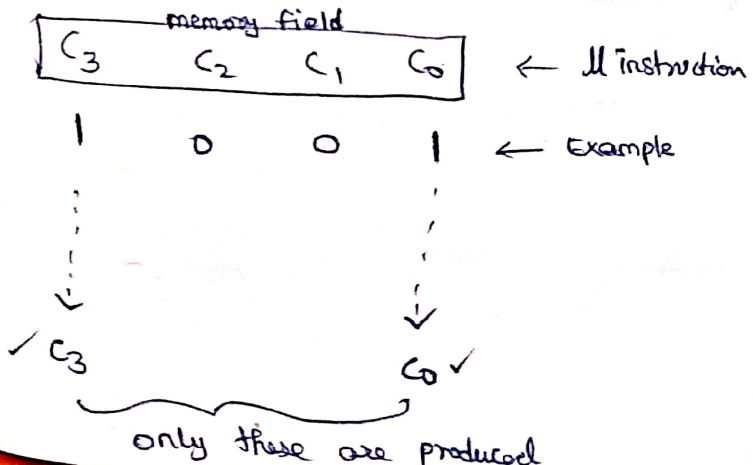
1) Microprogrammed Control Unit produces control signals by **software**, using **micro-instructions**.
2) A program is a set of instructions.
3) An instruction requires a set of Micro-Operations.
4) **Micro-Operations are performed by control signals.**
5) Here, these control signals are generated using **micro-instructions.**
6) This means **every instruction requires a set of micro-instructions**
7) **This is called its micro-program.**
8) Microprograms for all instructions are **stored in a small memory called "Control Memory".**
9) The Control memory is **present inside the processor.**
10) Consider an **Instruction** that is **fetched from the main memory** into the Instruction Register (**IR**).
11) The processor uses its unique **"opcode"** to identify the **address of the first micro-instruction.**
12) That address is loaded into **CMAR** (Control Memory Address Register) also called **μIR.**
13) This address is **decoded to identify the corresponding μ-instruction** from the Control Memory.
14) There is a big **improvement over Wilkes' design**, to reduce the size of micro-instructions.
15) Most micro-instructions will **only have a Control field.**
16) The **Control field** Indicates the **control signals to be generated.**
17) Most micro-instructions **will not have an address field.**
18) Instead, **μPC will simply get incremented** after every micro-instruction.
19) This is as long as the μ-program is executed **sequentially.**
20) If there is a **branch μ-instruction** only then there will be an **address filed.**
21) If the branch is **unconditional**, the **branch address will be directly loaded into CMAR.**
22) For **Conditional branches**, the Branch condition will **check the appropriate flag.**
23) This is done using a **MUX** which has all flag inputs.
24) If the **condition is true**, then the **MUX will inform CMAR to load the branch address.**
25) If the **condition is false** CMAR will simply get **incremented.**
26) The control memory is usually implemented using **FLASH ROM** as it is **writable** yet **non volatile.**

* MICRO INSTRUCTION FORMAT :

- The main part of micro instruction is Control Field.
- It determines control signal to be produced.
- It can be of two different formats : Horizontal, Vertical.

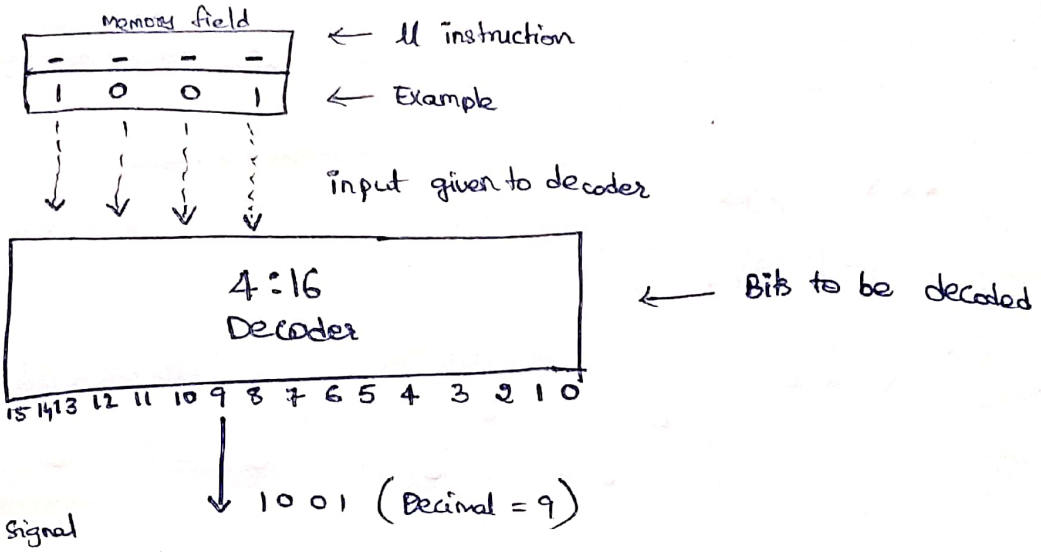
1) Horizontal Micro-Instruction :

Here every bit of micro instruction corresponded to control signal whichever bit is "1", that particular control signal will be produced by micro instruction.



- maximum Control signals produced = 4
- maximum bits in μ instr = 4

2) Vertical Micro instruction :



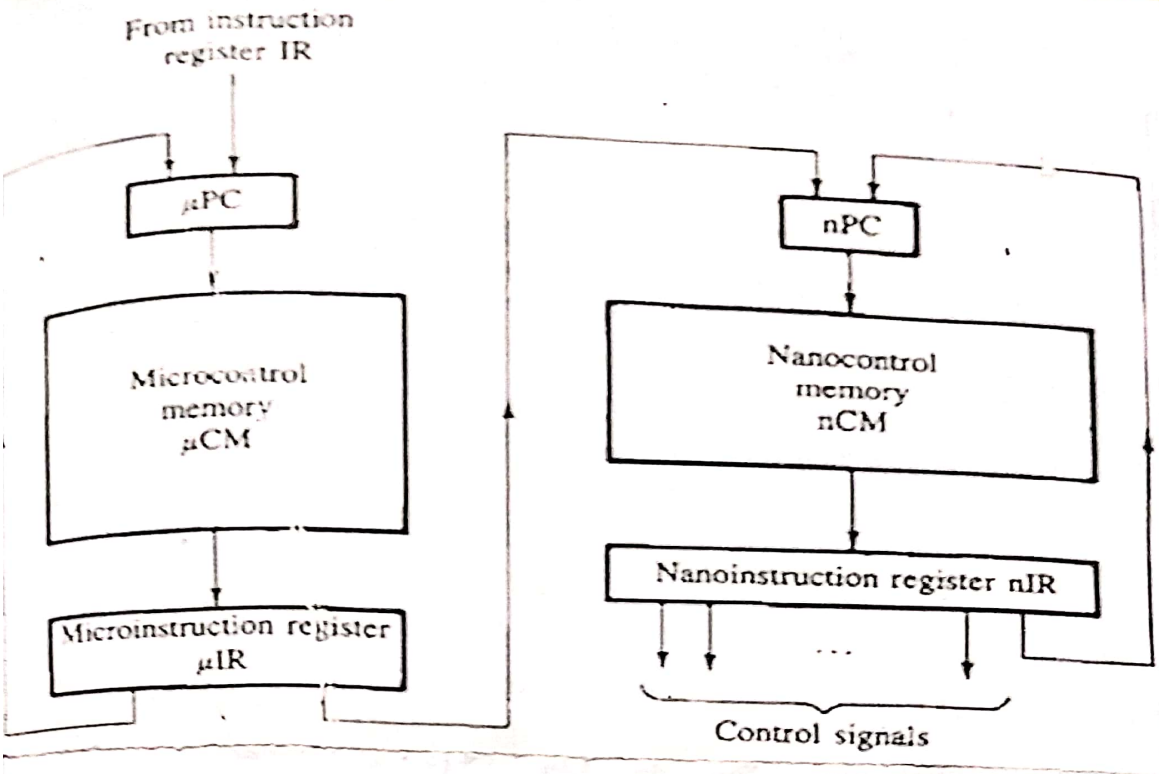
Control Signal produced → C_9 control signal generated (since decimal = 9)

(1001) → 9

that value's control field is active & generates.

- Maximum Control signals generated = 16 .
- Maximum bits of Micro Instruction = 4 .

	HORIZONTAL MICRO-INSTRUCTION	VERTICAL MICRO-INSTRUCTION
1	Every bit of the micro-instruction corresponds to a control signal.	Bits of the micro-instruction have to be decoded to produce control signals.
2	Does not require a decoder.	Needs a decoder.
3	N bits in the micro-instruction will totally produce N control signals.	N bits in the micro-instruction will totally produce 2^N control signals.
4	Multiple control signals can be produced by one micro-instruction.	Only one control signal can be produced by one micro-instruction.
5	As the control signals increase, the micro-instruction grows wider. Hence the Control Memory grows Horizontally.	To produce more control signals, more number of micro-instructions are needed. Hence the Control Memory grows Vertically.
6	Executes faster as no decoding needed.	Executes slower as decoding is needed.
7	Micro-instruction are very wide. Hence Control memory is large.	Micro-instruction are much narrower. Hence Control memory is small.
8	Circuit is simpler as a decoder is not needed.	Circuit is more complex as a decoder is needed.



NOT in Syllabus

NANO-PROGRAMMING

1) **Horizontal** μ-instructions can produce **multiple control signals** simultaneously, but are **very wide**.
2) This makes the Control Memory **very large in size.**
3) **Vertical** micro-instructions are **narrow, but on decoding** can **produce** only **one control signal.**
4) This makes the Control Memory **small** but the execution is **slow.**
5) Hence a **combination of both techniques** is needed called **Nano-Programming.**
6) Here we have a **two level control memory.**
7) The instruction is fetched from the **main memory into IR.**
8) Using its **opcode** we load **address of its first micro-instruction** into **μPC,**
9) Using this address we **fetch the micro-instruction** from μ-Control Memory (**μCM**) **into μIR.**
10) This is in **vertical form** and has to be decoded.
11) The decoded output **loads a new address** in a Nano program counter (**nPC**).
12) Using this address we **fetch the Nano-instruction** from Nano-Control Memory (**nCM**) into **nIR.**
13) This is in **horizontal form** and can **directly generate control signals.**
14) Such a combination **gives advantage of both techniques.**
15) The size of the Control Memory is **small** as μ-instructions are **Vertical.**
16) Multiple control signals can be **produced simultaneously** as Nano-instructions are **Horizontal.**

MEMORY

*. Specifications :

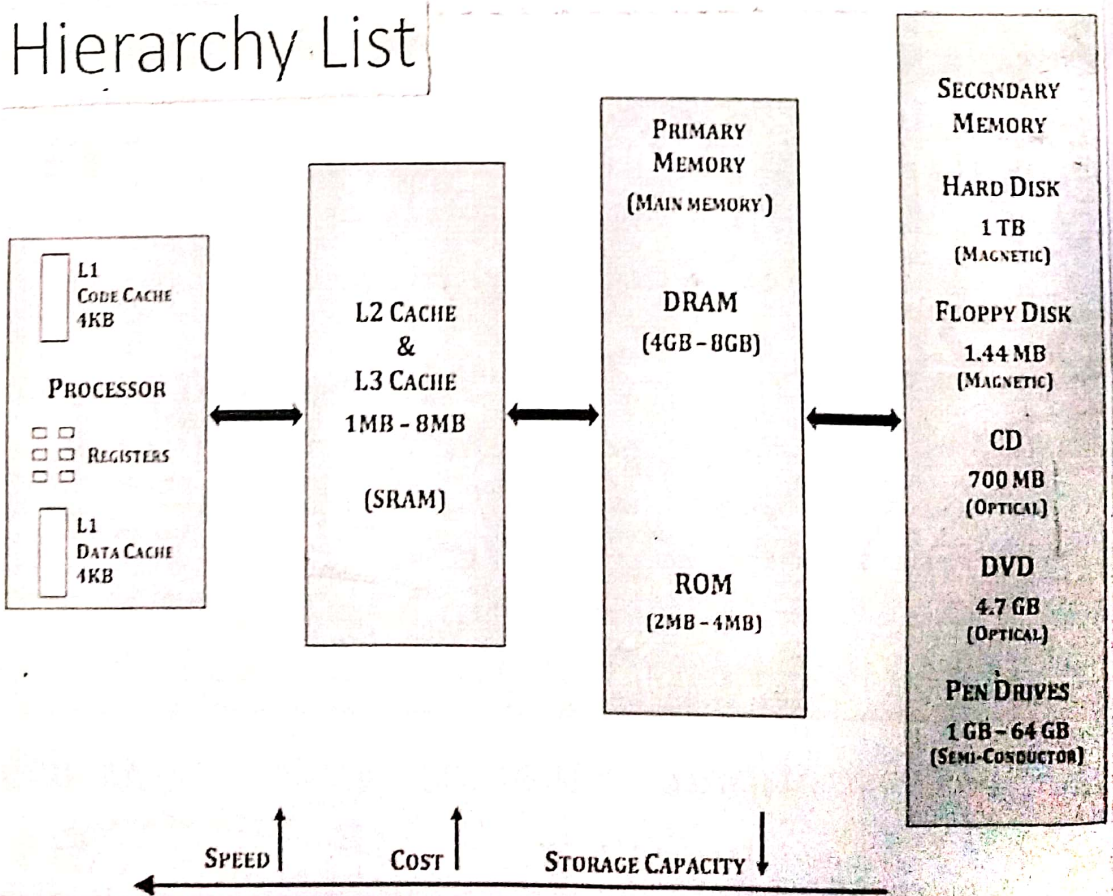
- How much ? Capacity
- How fast ? Time is Money
- How expensive ? Cost

*. REGISTERS :

- Registers are present inside processor.
- They are basically set of Flip-Flops. (MOSFETS)
- They store data & address and can directly take part in arithmetic and logic operations.
- They are very small in size typically just a few bytes.

IMP • Maximum data by 1 register = 64 bit data

IMP • We cannot afford more size (Smallest), Fastest memory device.



*. PRIMARY MEMORY :

- It is the original form of memory, also called Main Memory
- It comprises of RAM (Random Access Memory) and ROM (Read only Memory), both are Semi-Conductor memories. (MOSFETS)

IMP • ROM is non-volatile. It is used in storing permanent information like the BIOS program. It is typically of 2MB – 4MB in size. (Basic Input Output System)

IMP • RAM is writable & hence is used for daily operations. Every file that we access from secondary memory, is first loaded into RAM. To provide large amount of working space RAM is typically 4 GB – 8 GB.

*. SECONDARY MEMORY :

- The main purpose of secondary memory is to increase the storage capacity, at low cost.
- It's biggest component is Hard Disk, where all files inside a Computer are stored.
- It is writable as well as non-volatile.
- Typical size of HD is 1 TB.
- Disk memories are much slower than chip memory. But also much cheaper.

*. PORTABLE SECONDARY MEMORY :

- These are required to physically transfer files between computers.
- Floppy Disk : It is magnetic form of storage. Typical size is 1.44 MB.

- CD : It is an optical form of storage.
  Typical size = 700 MB.

- DVD : It is also optical form of storage.
  Typical size = 4-7 MB

- Pen drives & Memory Cards : It is semi conductor form of storage. It is composed of FLASH ROM.

→ It's special type of ROM that's writable as well as non-volatile.

→ Typical size = 1 GB to 64 GB depending on Cost.

IMP
* CACHE MEMORY :

- It is the fastest form of Memory SRAM (static RAM)
- The main memory uses DRAM (Dynamic RAM)

IMP
- SRAM uses flip-flops and hence is much faster than DRAM which uses capacitors.
- But SRAM is also very expensive compared to DRAM.

IMP
- Hence, only the current portion of the file we need to access is copied from Main memory (DRAM) to Cache memory (SRAM), to be directly accessed by the processor.

- This gives maximum performance and yet keeps the cost low.
- Typical size of Cache is around 2MB – 8MB.

IMP
- Depending upon the location of Cache, it 3 types
  L1, L2, L3.

MEMORY CHARACTERISTICS

1) Location

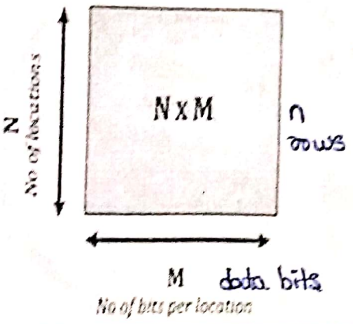
Based on its physical location, memory is classified into three types.

- **On-Chip:** This memory is present **inside the CPU**. E.g.:: Internal Registers and **L1 Cache**.
- **Internal:** This memory is present **on the motherboard**. E.g.:: **RAM**.
- **External:** This memory is **connected to the motherboard**. E.g.:: **Hard disk**.

2) Storage Capacity

This indicates the **amount of data stored** in the memory.
Obviously it should be **as large as possible**.
It is represented as N x M.
Here,
N = **Number of memory locations** (no of words)
M = **Number of pits per memory location** (word size)
E.g.:: (4K x 8) means there are 4K locations of 8-bits each.

Maximum = Hard Disk
Minimum = Cache Memory
memory
1 byte



3) Transfer Modes

Data can be accessed from memory in two different ways.

- **Word Transfer:** Here, if CPU needs some data, it will transfer only that amount of data.
  E.g.:: Data accessed from **L1 Cache**.
- **Block Transfer:** Here, if CPU needs some data, it will transfer an entire block containing that data. This makes further access to remaining data of this block much faster. This is based on Principle of Spatial Locality. A processor is most likely to access data near the current location being accessed.
  E.g.:: On a **cache miss**, processor goes to **main memory** and copies a **block** containing that data.

4) Access Modes

Memories can allow data to be accessed in two different ways.

- **Serial Access:** Here locations are accessed one by one in a **sequential manner**.
  The access time depends on how far the target location is, from the current location.
  Farther the location, **more** will be its **access time**.
  E.g.:: Magnetic tapes.
- **Random Access:** Here **all locations** can be directly accessed in any **random order**.
  This means **all locations** have the **same access time** irrespective of their address.
  E.g.:: Most modern memories like RAM.

5) Physical Properties

There are various Physical attributes to memory.

- **Writeable: Contents** of the memory **can be altered**. E.g.:: **RAM**
- **Non-Writeable: Contents** of the memory **cannot be altered**. E.g.:: **ROM**
- **Volatile: Contents** of the memory are **lost** when power is **switched off**. E.g.:: **RAM**
- **Non-Volatile: Contents** of the memory are **retained** when power is **switched off**. E.g.:: **ROM**
- *Most secondary memories like Hard disk are Writable as well as non-volatile.*

6) **Access Time (t_A)**

It is the time taken between **placing the request** and **completing the data transfer.**
It should be as **less as possible.**
It is also known as **latency.**

7) **Reliability**

It is the **time** for which the memory is expected to **hold the data without any errors.**
It is measured as **MTTF: Mean Time To Failure.**
It should be as **high as possible.**

8) **Cost**

This indicates the **cost of storing data** in the memory.
It is expressed as **Cost/bit.**
It must be **as low as possible.**

9) **Average Cost**

It is the total cost per bit, for the entire memory storage.
Consider a system having **two memories M_1 (RAM) & M_2 (ROM)**
If C_1 is the cost of memory M_1 of size S_1
& C_2 is the cost of memory M_2 of size S_2

Then the average cost of the memory is be calculated as:
$C_{AVG} = (C_1 S_1 + C_2 S_2)/ (S_1 + S_2)$

Small sizes of **expensive** memory and **large** size of **cheaper** memory **lowers** the **average cost.**

10) **Hit Ratio (H)**

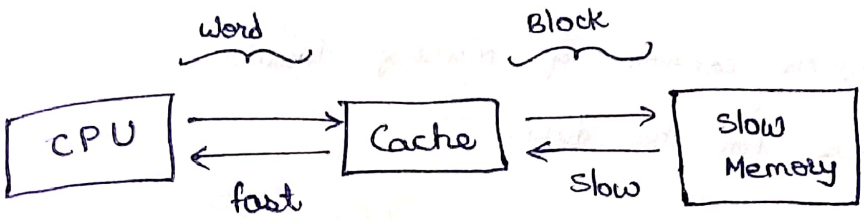
Consider two memories M_1 and M_2.
M_1 is **closer** to the processor E.g.:: **RAM**, than M_2 E.g.:: **Hard disk.**
If the **desired data is found in M_1**, then it is called a **Hit**, else it is a **Miss.**
Let N_1 be the number of **Hits** and N_2 the number of **Misses.**
The **Hit Ratio** H is defined as **number of hits divided by total attempts.**
$H = (N_1) / (N_1 + N_2)$
It is expressed s a percentage.
H can never be 100%. In most computers it is maintained around 98%.

From the above discussion it is clear that no single memory can satisfy all the characteristics, **hence we need a hierarchy of memories.**
Cache memories are the **fastest** but also the **most costly.**
Hard disk is **writeable** as well as **non volatile** and is also very **inexpensive**, but is much **slower.**
CD/DVD etc. are needed for **portability.**
ROM is **nonvolatile**, and is used for **storing BIOS.**
DRAM is **writable, faster than hard disk** and **cheaper than SRAM** hence forms **most part of Main Memory.**

*. The data will be read & written faster in cache.

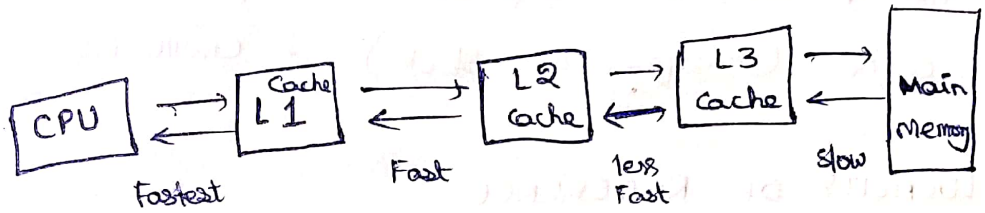
* CACHE AND MAIN MEMORY :

1)



SINGLE CACHE

2)

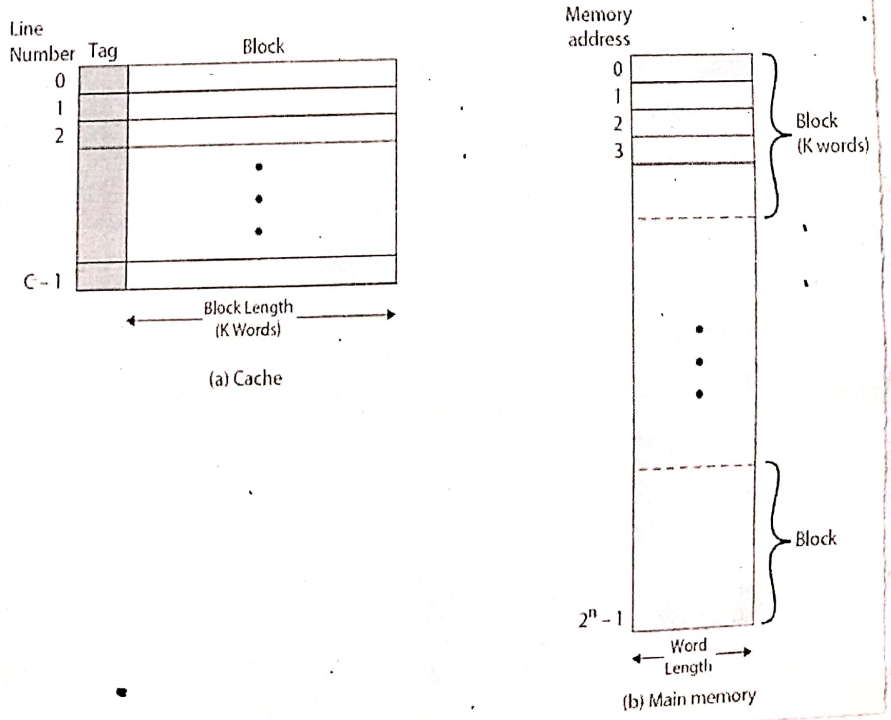


THREE LEVEL CACHE

- Data found in Cache – Cache Hit.
  Data not found in Cache – Cache Miss.

IMP
* No. of blocks in Main memory = $\frac{\text{No. of words in Main Mem}}{\text{No. of words in Cache}}$

Cache/Main Memory Structure



*. CACHE OPERATION :

- CPU requests contents of memory location.
- check cache for this data.
- If present, get from cache (fast)
- If not present, read required block from main memory to cache (transfred as block) – CACHE MISS

IMP *. LOCALITY OF REFERENCE :

To avoid multiple cache miss. To continue with cache hit.

- Then deliver from cache to CPU.
- Cache includes tags to identify which block of main memory is in each cache slot.

IMP *. TAG in cache blocks tells us, when there is cache miss, where the whole block is found in Main memory. (data is in which block number ?)

IMP * CACHE MEMORY MAPPING

- Cache Mapping decides which block of Main memory comes into which block of Cache memory.

IMP
- Needed to balance between Hit Ratio (good ↑), Search time (very less), Tag size (very less)

- Each tag → which block of Main Memory.
- Cache directory – All list of these tags.

IMP
- Design Cache Mapping such that Tag must be minimum size.

*. TYPES OF POPULAR CACHE MAPPING TECHNIQUES :

1. Associative Mapping (or) Fully Associative Mapping.
2. Direct Mapping (or) One-way set Associative Mapping.
3. Set Associative Mapping. (Two-way set Associative Mapp) or n-way.

---

① ASSOCIATIVE MAPPING :

- During memory operations, Blocks are loaded from Main memory to Cache Memory.
- Cache Mapping decides which block of Main memory comes into which block of cache memory.
- Any block of Main memory can be mapped at any available block of Cache Memory.
- There are no rules restricting the mapping at all.
- This means the Full cache is available for mapping hence the name Fully Associative.

(IMP)
Example :

Consider Pentium Processor Cache :

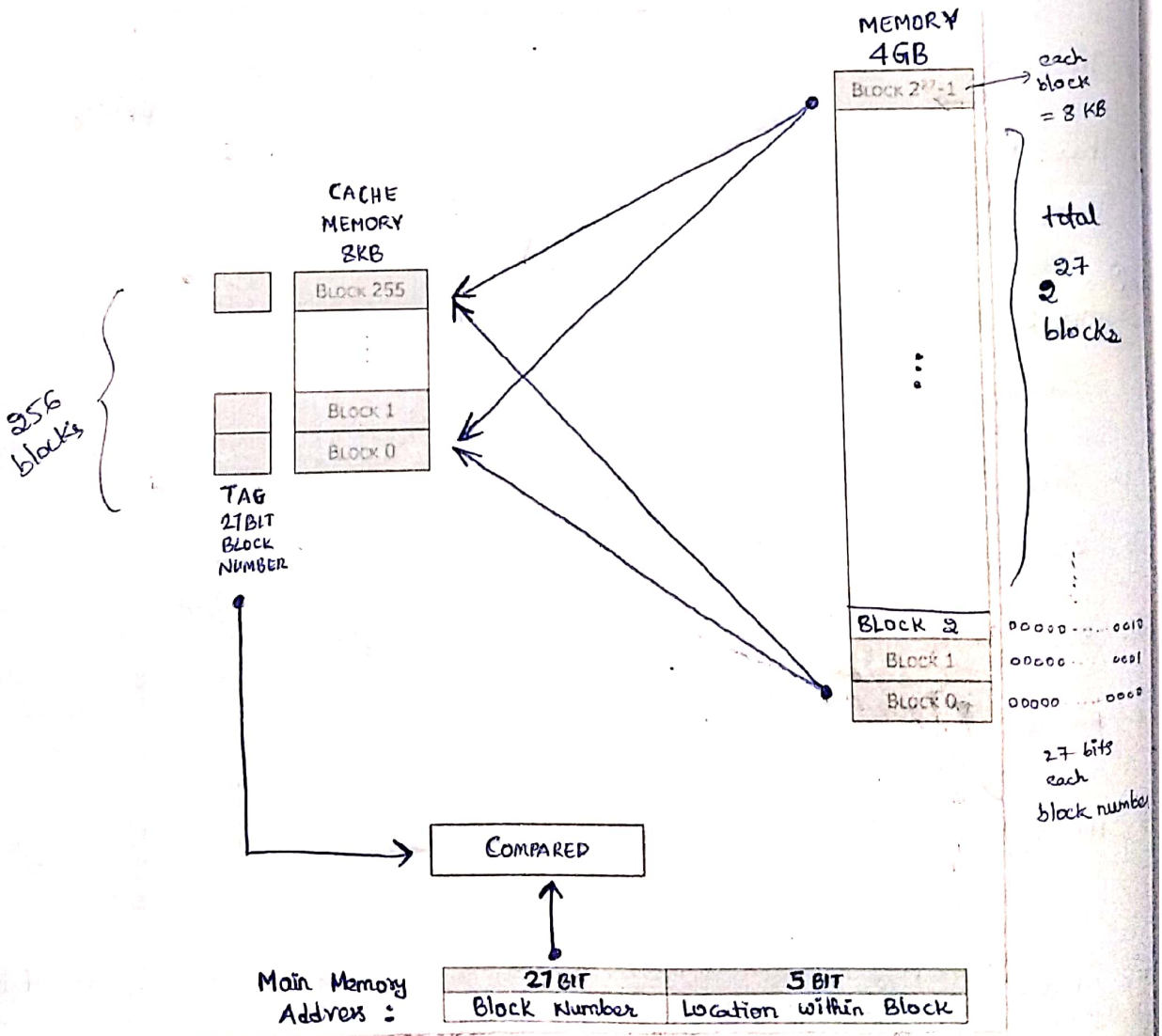
1. Size of Main Memory : 4 GB = 2^{32}
2. Size of Cache Memory : 8 KB = 2^{13}
3. Size of Cache Block (Line) : 32 bytes (word) = 2^5
4. No. of Blocks in Main Memory : $\frac{\text{Size of Main memory}}{\text{Size of block}} = \frac{2^{32}}{2^5} = 2^{27}$
5. No. of Blocks in Cache Memory : $\frac{\text{Size of Cache memory}}{\text{Size of Block}} = \frac{2^{13}}{2^5} = 2^8$
6. Main Memory address : 32 bits (because main Memory is 4GB) $= 2^{32}$

Tag size :

- A block of cache memory can contain any block of main memory out of a possible 2^{27} blocks.
- Hence, the Tag next to next every block in cache memory must be of 27 bits.

Searches :

- A block of main Memory can be mapped into any block of Cache Memory out of 256 Blocks. Hence, we need to do 256 searches in Cache Memory.
- Cache miss - If even after 256 searches, it is not able to find the data in cache block, then it goes to main memory block number to search (by Tag)



Tag Size:
A block of Cache Memory can **contain any block** of main memory out of a possible 2^{27} **blocks.**
Hence, the **Tag** next to every block in Cache Memory must be of **27 bits.**

Searches:
A block of main Memory can be **mapped into any block** of Cache Memory out of **256 Blocks.**
Hence, we need to do **256 searches** in Cache Memory.

Method of Searching:
The Processor issues a 32 bit Main Memory address. It can be divided as:

Main Memory Address:	27 BIT	5 BIT
	Block No.	Location Within Block

This 27 bit Block number is the block number we need to search.
The Tag of each cache block also contains a 27-bit block number.
This is the block number that's present in that respective cache block.
These two block numbers are compared. **If they are equal, it's a HIT.** ✓
If not equal, the search is repeated with the Tag of the next cache block.
This is done a total of 256 times as there are 256 blocks in Cache Memory.
If none of them match with the block number we are searching, then it's a Miss. ✓

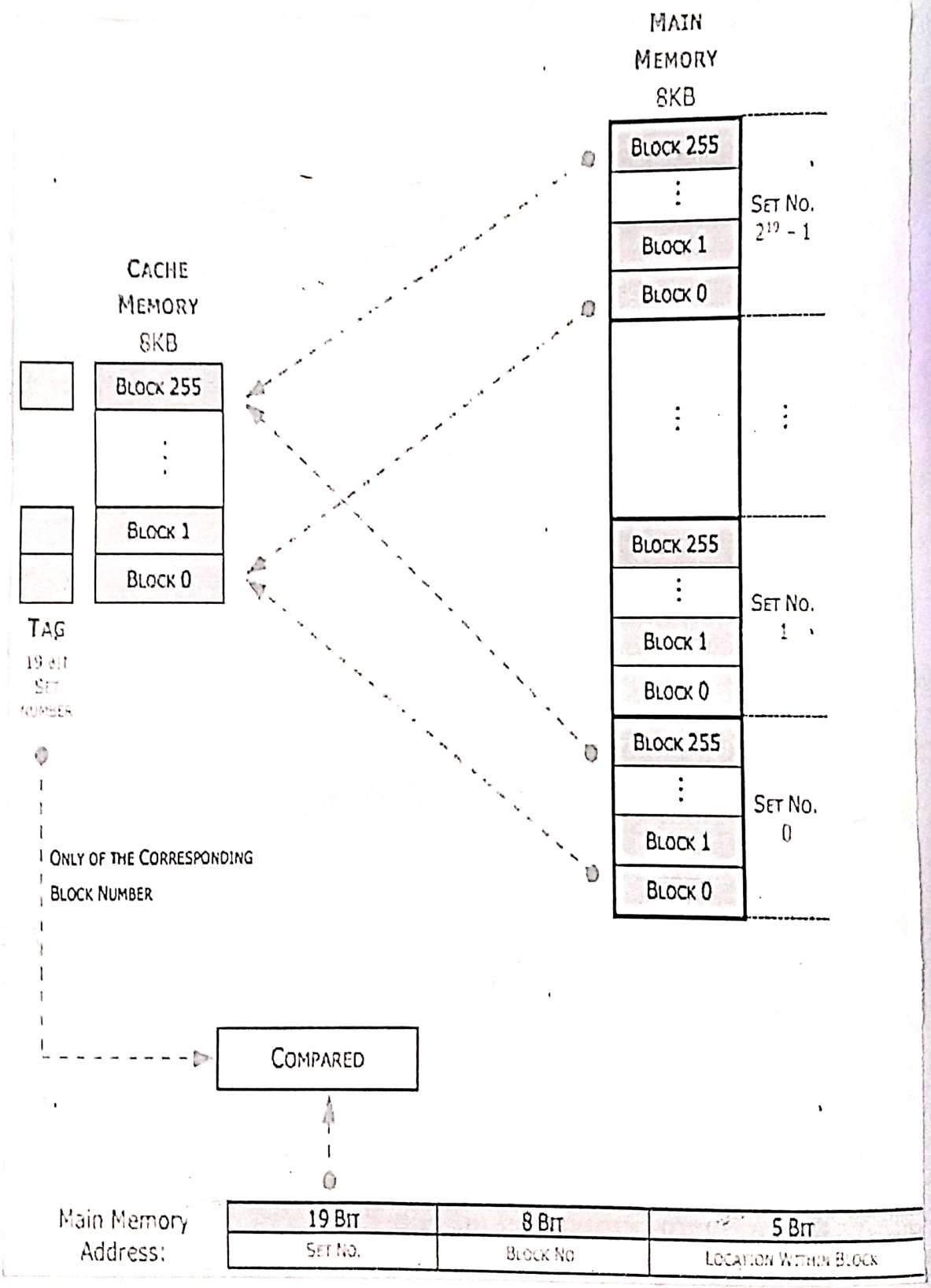
Advantage
Since the full cache is available for mapping, it causes maximum utilization of Cache Memory hence gives the **Best Hit Ratio.** *

Drawback
Tag Size too big: **27bits.** – Cache Memory device is costly.
Searches are too many: **256.** – delay will be more.

Abbr.	Prefix name	Decimal size	Size in thousands	Binary approximation	Address variable size
K	kilo-	10^3	1,000	$1{,}024 = 2^{10}$	10
M	mega-	10^6	$1{,}000^2$	$1{,}024^2 = 2^{20}$	20
G	giga-	10^9	$1{,}000^3$	$1{,}024^3 = 2^{30}$	30
T	tera-	10^{12}	$1{,}000^4$	$1{,}024^4 = 2^{40}$	40
P	peta-	10^{15}	$1{,}000^5$	$1{,}024^5 = 2^{50}$	50
E	exa-	10^{18}	$1{,}000^6$	$1{,}024^6 = 2^{60}$	60

2) DIRECT MAPPING :

- 2^{13} locations are treated as 1 set. Size = 2^{13} = 8 kB.
- No. of sets in cache memory = 1.



Direct Mapping (One Way Set Associative Mapping)

Direct Mapping technique states:
Any block of Main Memory can only be mapped at ONE block of Cache Memory.
Since there is only one way of Mapping, its also called One Way Set Associative Mapping.
We treat the entire Cache as One Set.
The Main Memory is divided into Sets which are then subdivided into Blocks.
A Block of Main Mem. (of any set), can only be mapped into the same Block No. in Cache Mem.
This means, Block 0 of Main Memory (of any set), can only be mapped into Block 0 of Cache Memory.
In other words, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set.

Consider Pentium Processor Cache

Size of Main Mem:	4GB = 2^{32}
Size of Cache Mem:	8KB = 2^{13} ... this is treated as One Set
Hence Size of Set:	8KB = 2^{13}
Size of Cache Block (Line):	32 bytes (words) = 2^5
No. of Blocks in a set:	Size of Set (2^{13}) ÷ Size of Block (2^5) = 2^8 = 256
No. of Sets in Main Mem:	Size of Main Mem (2^{32}) ÷ Size of Set (2^{13}) = 2^{19}
No. of Sets in Cache Mem:	1
Main Mem address:	32 bits (because main Mem is of 4GB = 2^{32})

Tag Size:
Since, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set, the Tag has to only indicate the Set No. of Main Memory, from which the block is present.
As Main Memory has 2^{19} sets, the Tag size is 19 bits.

Searches:
If we need Block 0 if Main Memory, we only need to search Block 0 of Cache Memory.
Hence, we need to do only 1 search in Cache Memory to know if it is a Hit or a Miss.

3) Set Associative Mapping :

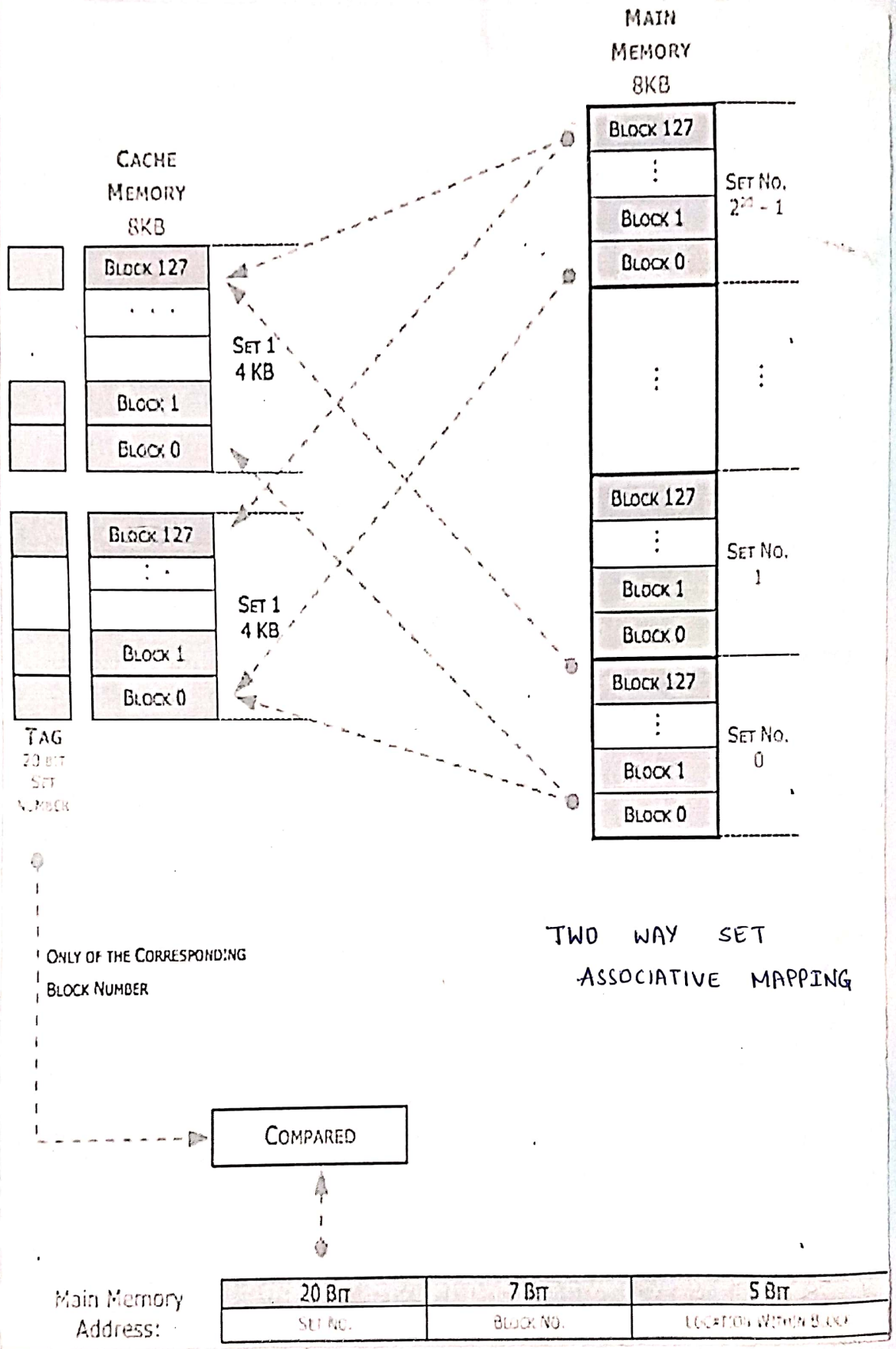
Set Associative Mapping (Two Way Set Associative Mapping)

Two way Set Associative Mapping technique states:
A block of Main Memory can only be mapped into the same corresponding Block No. of Cache Memory, in any of the two sets.
Since there are two ways of Mapping, its called Two Way Set Associative Mapping.
We treat the entire Cache as **Two Sets**. The Main Memory is divided into Sets, subdivided into Blocks.
A Block of Main Mem. (of any set), can only be mapped into the same Block No. in Cache Memory again of any set. This means, Block 0 of Main Memory (of any set), can only be mapped into Block 0 of Cache Memory, into one of its two sets.
In other words, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set.

Consider Pentium Processor Cache (This is Actually how Pentium's Cache is implemented)

* IMP

Size of Main Mem:	4GB = 2^{32}
Size of Cache Mem:	8KB = 2^{13} ... this is treated as Two Sets
Hence Size of Set:	4KB = 2^{12}
Size of Cache Block (Line):	32 bytes (words) = 2^5
No. of Blocks in a set:	Size of Set (2^{12}) ÷ Size of Block (2^5) = 2^7 = 128
No. of Sets in Main Mem:	Size of Main Mem (2^{32}) ÷ Size of Set (2^{12}) = 2^{20}
No. of Sets in Cache Mem:	2
Main Mem address:	32 bits (because main Mem is of 4GB = 2^{32})



TWO WAY SET ASSOCIATIVE MAPPING

Tag Size:
Since, Block 0 of Cache Memory can only contain Block 0 of Main Memory but of any Set, the Tag has to only indicate the Set No. of Main Memory, from which the block is present.
As Main Memory has 2^{20} sets, the **Tag size is 20 bits.**

Searches:
If we need Block 0 if Main Memory, we only need to search Block 0 of Cache Memory, but in any of the two sets. Hence, we need **2 searches** in Cache Memory to know if it is a Hit or a Miss.

Method of Searching:
The Processor issues a 32 bit Main Memory address. It can be divided as:

Main Memory Address:	20 BIT	7 BIT	5 BIT
	Set No.	Block No.	Location Within Block

First we look at the block no. we need, to know where we need to search.
We then look at the Set No. that we need and compare it with the Tags of the corresponding Block no in the Cache Memory, in any of the two sets.
Assume the Main Memory address is 5:0:6. This means we need location 6 of Block 0 of set 5. *IMP
We go to Block 0 of Cache Memory, in both sets.
It has a Tag, which gives the Set No of Main Memory whose Block 0 is present in Cache Memory.
These required Set no. is compared with the two Tags. **If found in any of the 2, it's a HIT, else Miss.**

Advantage
In 2 Searches we know if it is a Hit or a Miss. **Tag Size = 20 bits.**

Drawback
Since the method is **flexible**, it significantly **increases** the **Hit Ratio.**

n-WAY SET ASSOCIATIVE MAPPING :

Expanding the logic of set associative cache further, we can derive the following conclusion:
*IMP

NO OF WAYS	NO OF SEARCHES	TAG SIZE	NO OF BLOCKS IN A SET
2 Way	2	20 bits	128
4 Way	4	21 bits	64
8 Way	8	22 bits	32
16 Way	16	23 bits	16
32 Way	32	24 bits	8
64 Way	64	25 bits	4
128 Way	128	26 bits	2
256 Way	256	27 bits	1

This becomes exactly the same as Fully Associative: 256 Searches, 27-bit Tag

No. of comparators needed = no. of bits (n).