*Hexadecimal addition:-

*1.
```
  134H
+ 762H
------
  896H
```

*2.
```
  FFh
+  1h
------
  100H
```

*3.
```
  772H
+ 329H
------
  A9BH
```

*4.
```
  111H
+ FABH
------
 10BCH
```

*Subtraction:-

*1.
```
  13H
- 7H
----
```

*2.
```
  343H
- 147H
------
```

*3
```
  00h
- 1h
----
```

*4.
```
  872H
- 298H
------
```

*Features of M.P :->

*1. Data bus :->

*2. Address bus :->

*3. Control bus :->

*4. Speed :-> clock frequency.

*5. Instruction set :-> Set of instructions that are supported by the Processor.

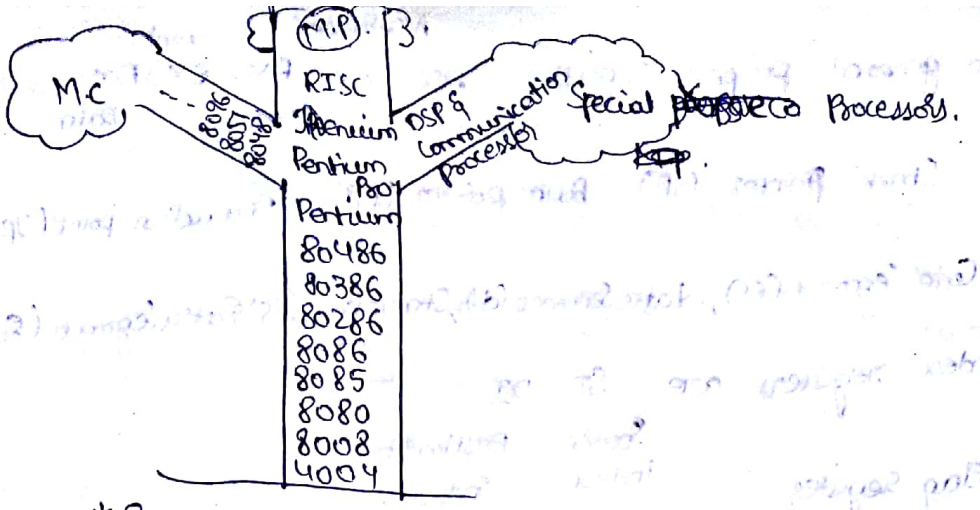
*6. Registers array :-> Set of registers " " " " " M.P & width of the register.

*7. Power :-> +5V.

*8. package :-> * Single Inline package {SIP} 
* Dual " " {DIP}. 
* Size, no. of pins, shape

*9. Design Technology :-> RISC {Reduced Instruction Set Computer}
CISC {Complex Instruction Set Computer}

*10. manufacturing Technology :-> Whether it is an NMOS, PMOS, CMOS.



Evolution tree of M.P

30/11/17

UNIT-I

*Features of 8086:-

* 16 bit M.P. i.e.. ALU, Registers, Instruction size are designed to work on 16 bit only.

* Data bus = 16 bits.

* Address bus = 20 bits.

*Memory capacity = $2^{20} = 2^{10} \times 2^{10} = 1KB \times 1KB = 1MB$.

* These 20 bits are represented using Hexa decimal bits. i.e.. 00000 H to FFFFF H

* It Contains 40 pins, 20 pins on each side i.e.. DIP.

* Clock freq is 5MHz or 8MHz or 10MHz, but in most of cases 5MHz is used

* Design technology is CISC & nmos technology.

* Power that is required is +5V.

* We have fourteen registers each of 16 bits. & the registers are.

* The general purpose registers are AX, BX, CX, DX
  (Accumulator, Base, Count, Data)

* Stack Pointer (SP), Base pointer (BP), Instruction pointer (IP)
* Code Segment (CS), data Segment (DS), Stack Segment (SS) Extra Segment (ES)
* Index registers are SI, DI
  (Source index, Destination index)
* Flag registers

} Special Purpose registers

→ *) It operates in two mode i.e.. minimum mode & maximum mode
  configuration {System uses only one 8086} {uses cascading 8086}

*) Perform Bit, Byte operations as well as Block operations

* Register organisation of 8086 :→

① General purpose registers :→

* AX - Accumulator :→ *used as default register for IO, ALU operations & result obtained is stored into it

* BX - Base register :→ * It stores 16 bit offset Address

* CX - Count register :→

  * used as a Counter for Rotate, Shift operations

  * Loop instructions are stored in CX

* Data Register - DX :- {extended accumulator}.

* ~~Stored~~ used for storing IO port addresses.

* used for storing extra bits during 16 bit Mul & Div operations.

* These general purpose registers are of 16 bits but they can also be used as 8 bit registers.

(16) AX	AH (8)	AL (8)
(16) BX	BH (8)	BL (8)
(16) CX	CH (8)	CL (8)
(16) DX	DH (8)	DL (8)

Solution

* Flag Register :- {16 bit}.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	O	D	I	T	S	Z	X	A	X	P	X	C

③ @ *) CF → Carry flag = Set (when there a carry from MSB)

= 0 O.W.

①* Total 9 flags in that 6 are Conditional / Status flags they are affected during arithmetic operations.

* they are Carry (CF), parity (PF), Auxiliary (AF), Zero (ZF), Sign Flag (SF), overflow flag.

②* 3 are Control flags they are

Direction (DF), Interrupt (IF), Trap (TF).

→*) (b) PF ⇒ Parity flag = 1 – Even parity

0 – odd parity.

→*) (c) AF ⇒ Auxiliary flag = 1 – In the addition of BCD, if the sum is >9 we add 6

∴ when there's a carry from 4th bit in BCD, then AF is set.

Ex:

```
  39          1111
+ 48          0001  1001
----   =>     0100  0000
  87          ----------
              1000  0001
                   +0110
              ----------
              1000  0111
              ----------
                8     7
```

= 0 → if no carry is generated.

→*) (d) Zero flag ⇒ ZF = 1 ⇒ if the result is zero

= 0 ⇒ if the result is not zero.

→*) (e) Sign flag ⇒ SF = 1 ⇒ if the number is Negative

= 0 ⇒ if the number is positive

{for signed numbers}

→*) (f) Overflow flag ⇒ OF = 1 → overflow

= 0 → No overflow.

* Ex:- -109 -108.

Sign bit

2|109
2|54 → 1
2|17 → 0
2|8 → 1
2|4 → 0
2|2 → 0
1 → 0

109 ⇒ $01000101
108 ⇒ $01000100

1's of 109 ⇒ 10111010
+1
2's ← 10111011

1's of 108 ⇒ 10111011
+1
2's. ← 10111100

```
  111
  10111011
① + 10111100
  ---------
  01110111
```

→ we are adding two -ve numbers but we get +ve number

⇒ overflow occurred.

for Vice versa also overflow is said to be occurred.

* Even if the carry into from MSB ≠ Carry out of MSB.

* ~~Status~~ Control flag:- we can control them but we can't control status flags.

*(a) Direction flag (DF):- =1 ⇒ SI & DI are decremented automatically.
=0 ⇒ SI & DI are incremented automatically.

*(b) Interrupt flag (IF):- =1 ⇒ If we want to allow Interrupts.
=0 ⇒ Interrupts are disabled.

→ *(c) Trap flag - (TF) :- = 1 ⇒ Single stepping mode is Enabled.
We can see results at End of Each Instruction
0 ⇒ Results will be seen at the End of the Program.

* Give the status of Conditional flags for the following Ex's

*1. 39H + 42H

*2. FFH + 1H

*3. 00H − 1H

*4. 39ACH + FCD3H

*5. A3H − F4H

*1. 39H + 42H

```
  0011   1001
+ 0100   0010
  0111   1011
    7      B.
```

CF = 0
PF = 1
AF = 0
ZF = 0
SF = 0
OF = 0

*2. FF + 1H

```
  1 1111 1111
+ 0000 0001
  0000 0000
```

CF = 1
PF = 1
AF = 0
ZF = 1
SF = 0
OF = 0

*3. 00 − 1

1 → 0000 0001

1's of 1 → 1111 1110

2's of 1, + 1

1111 1111

0000 0000

+ 1111 1111

FFFF

CF = 0; PF = 1; AF = 0; ZF = 0; OF = 0

*4. 39AC + FCD3 = 367F

0011 1001 1010 1000 → 10, 13, 13

1111 1100 1101 0011 → 10, 12, 22

0011 0110 0111 1111

+0110 +0110 +0110 +0110

1001 1100 1101 0101

CF = 1; PF = 0; AF = 1; ZF = 0; OF = 0; SF = 0

*5

CF = 0; PF = 1; AF = 1; ZF = 0; OF = 0; SF = 0

04/12/17

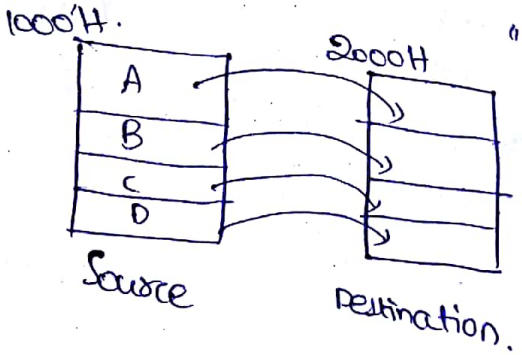
* Segment Registers :- {16 bit}

  * CSR → Code Segment Register → Stores Starting address of Code/program
  * DSR → Data " " → data & variables related to Code/Prog
  * ESR → Extra " " → String data
  * SSR → Stack " " → stack data i.e. return addresses {during subroutines}

* Code segment stores the code/program but code segment register stores the starting address of code/program
* Similarly all other.

* Index Registers :- {16 bit}

  * SI → Source Index – Stores starting address of source string
  * DI → Destination Index – " " " " " destination "



* Pointer Registers :- {16 bit}

  * IP – Instruction pointer – Stores address of next Instruction
  * SP – Stack pointer – Stores address of top of the stack
  * BP – Base pointer

Similar to PC

* Physical address of 8086 :- (20 bit)

Physical address = 10H * Base Address + offset Address
(20) (Segment ") (16) (Effective ") (16)

Ex:- B.A = 2000 H = 16 bit
EA = 3000 H = 16 bit.

PA = 20,000 H + 3,000 H = 23,000 H ⇒ 20bit

* B.A or S.A are provided by Segment registers i.e.. CSR, DSR, ESR, SSR

* OA or EA are provided by IP, SP, BP, SI, DI, BX.

* B.A is Starting address of the segment

* O.A is the address with refeence to starting address

* For Code segment CS the offset is provided by IP
SA : OA
i.e.. CS : IP
DS : SI / BX / BP / DI
ES : DI
SS : SP / BP.
} Default SA & OA Combination

* Calculate the physical address if CS = 2000H, IP = 3000H, SP = 5000H, BP = 9000H, DS = 6000H, SS = 1000H, ES = A000H, SI = C000H, DI = D000H.

*Sol) PA = 10 * CS + IP.

= 20,000H + 3000H

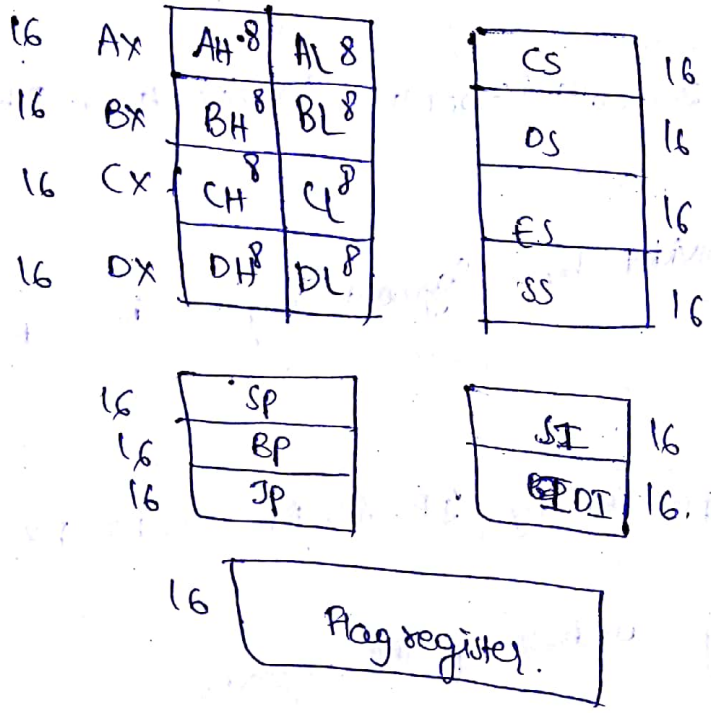
= 23,000H.

PA = 10 * DS + BP

= 60,000H + 9000H

= 69,000H.

* ARCHITECTURE OF 8086 :-



PA = 10 * ES + DI.

= 10 * A000H + D000H.

= AD000H.

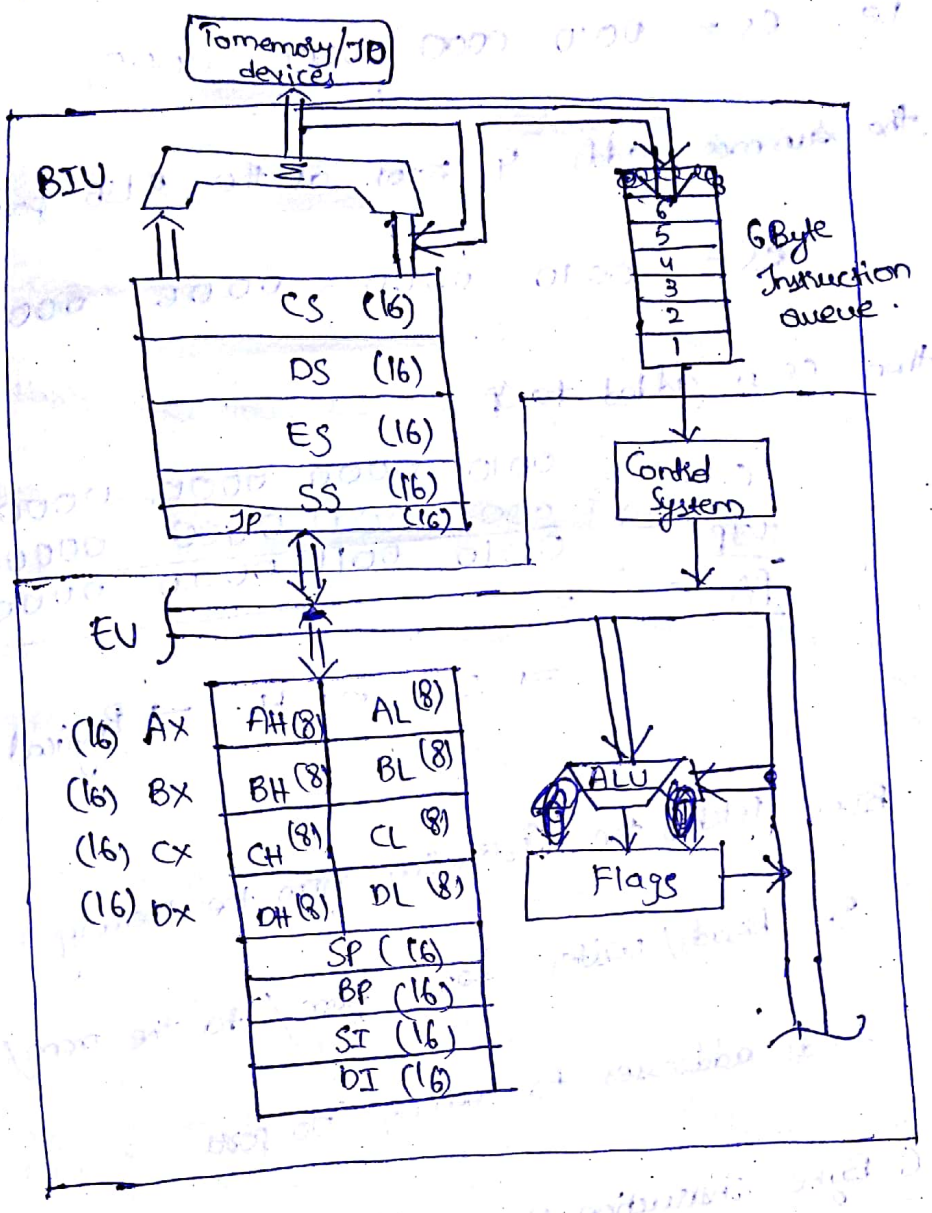
PA = 10 * SS + SP.

= 10,000H + 5,000H

= 15,000H.

06/12/17

*ARCHITECTURE OF 8086:-



*① Bus Interface unit ② Execution unit (BIU & EU)

*① BIU:-

* Consists of segment registers & IP i.e. 5 registers & a summer & 6 byte Instruction queue.

* It generates 20 bit physical address (PA)

*For Ex:-

CS = 2000H, IP = 3000H.

i.e.. CS = 0010 0000 0000 0000

the summer adds 4 zeros at the LSB position.

CS = 0010 0000 0000 0000 0000

then CS is added to IP.

```
CS  ⟶  0010 0000 0000 0000 0000
+IP ⟹+ 0000 0011 0000 0000 0000
PA  ⟶  0010 0011 0000 0000 0000
```

⇒ 23000H. → Physical address (20 bit

* BIU Fetches the instruction from the memory.
* It Reads/writes data from/to the mem/IO ports.
* Sends addresses to memory/IO ports
* 6 Byte instruction queue fetches the next instruction while the current instruction has been executed.
* EU decodes & executes the instruction
* The 6 Bytes of next instruction are decoded by CS byte by byte. (↓ Control System)
* ALU is used to perform arithmetic, logical, shift, Rotate operations & it updates the flag register.

*EU informs the BIU from where the next instruction has to be fetched.

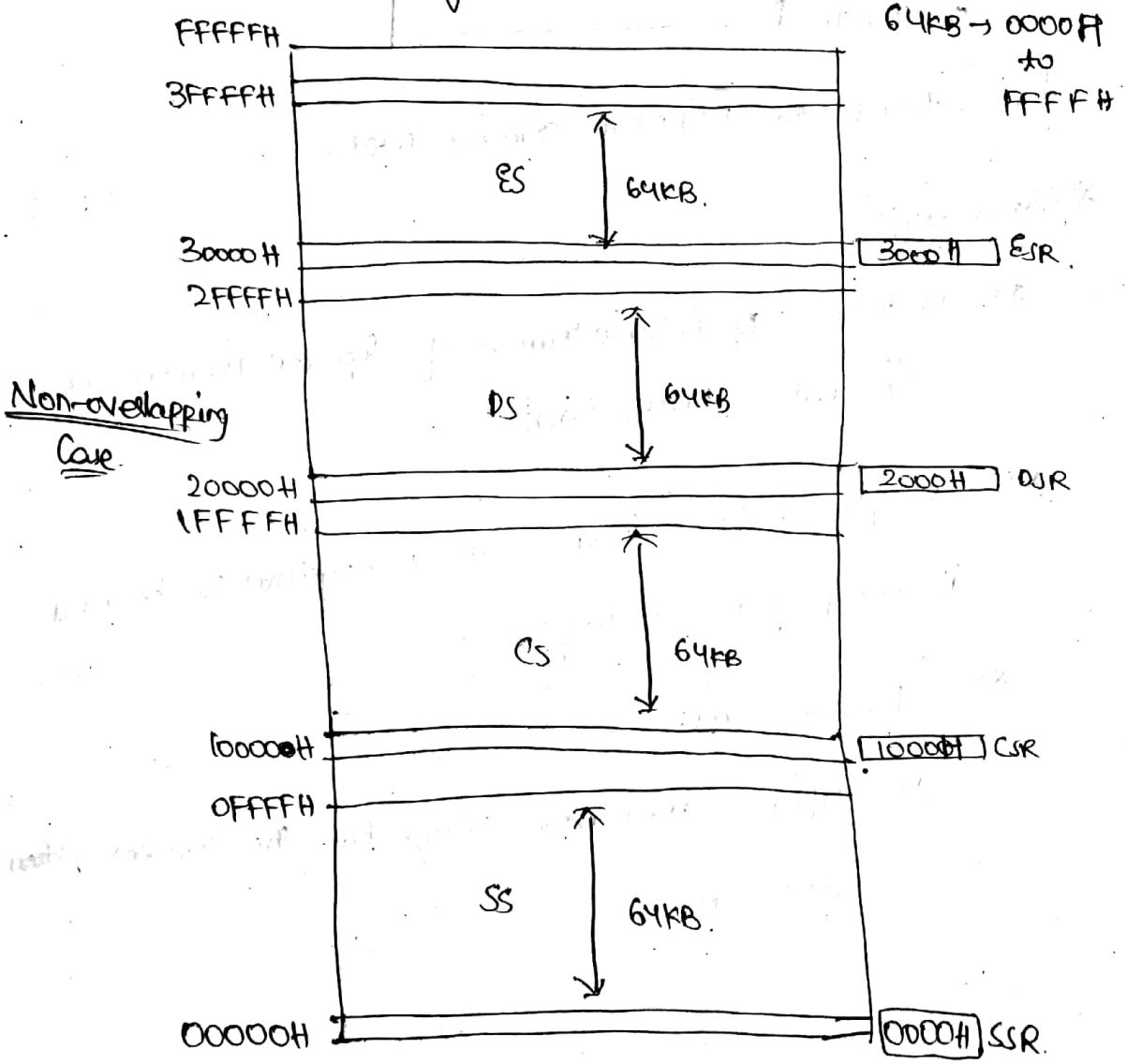
* write abt the registers (what they contain).

* 07/12/17 *

*Memory Segmentation:→

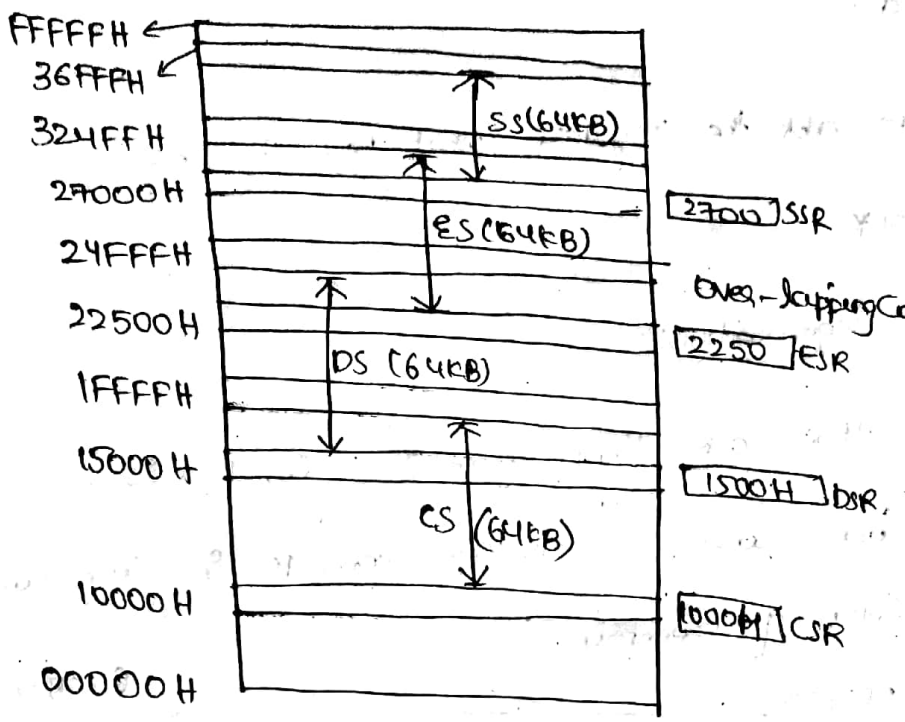
* there are four segments CS, DS, ES, SS

Each one of 16 code segments & each can store 64KB of memory.



*There is no need to follow the same order & the memory from 40000H to FFFFFH are not used, but if we count

Count we can add more Segments of Code, data, Extra,



for 37000H to FFFFFH are not used.

*Advantages:-

*1. Protection ~~of~~ ~~possible~~ because of seperate memory areas for Code, Data, Stack.

*2. Multiple code or data or stack segments can be used if the program is large.

*3: Program Relocation is possible.
i.e.. Offset address wont change but the segment address is changed.

* ****

* Addressing modes of 8086 :-

* Depending upon instructions the Address data modes are classified.

* (I) Sequential Control flow Instructions.

* (II) Control transfer Instructions.

* There are 8 types of modes :- Based on (I).

* (1) Immediate Addressing mode * (2) Direct addressing mode.

* (3) Register " " * (4) Register Indirect " "

* (5) Index " " * (6) Base Index " "

* (7) Register Relative " " * (8) Relative Based Index " "

* (1) Immediate Addressing mode :-

* Data is available in the Instruction ~~set~~ itself

* Eg :-
```
MOV AL, 07H
MOV AX, 1234H  {Here A 12 in AH & 34 in AL}
MOV CX, 0093H
MOV DL, 89H
```
(DL → Register, 89H → data)

(2) Register Addressing mode :-

* If the source & destination both are registers

* Eg :-
```
MOV AL, CL
MOV AX, DX
MOV BX, CX.
MOV AX, BX.
```
(AX, BX → Registers)

07/12/17

*③ Direct addressing mode :-

* If the address of the memory location is specified in the Instruction.

*Eg:- ⊛ MOV AX, [1234H]

↓

not a data it is address because of [ ]

EA = 10H * DS + 1234H.

* it means Contents of 1235H are stored in A_H & Contents of 1234H are stored in AL.

⊛ MOV AL, [1234H]

*it means Content of 1234H are stored in AL

{Always the address location must be even. if odd then invalid Instruction}

*④ Register Indirect mode :- Sometimes the memory which contains the data is determined Indirectly using offset registers.

*Eg * MOV BX, 1000H

MOV AX, [BX]

↓

We should use only BX, SI, DI → offset registers

But the default segment register is DS

i.e. EA = 10H * DS + [BX] / [SI] / [DI]

*(5) Index mode :- Off set address is stored in Index registers.

* The source registers are always SI // DI

* for that default segment register is DS.

⇒ EA = 10H * DS + [SI]/[DI]

*(6) Based Index registers mode :-

* off set address is stored in Base & Index registers.

EA = 10H * DS + [BX] + [SI]
or
[BP] [DI]

* MOV AX, [BX][SI]
or
MOV AX, [BX][DI]
or
MOV AX, [BP][SI]
or
MOV AX, [BP][DI]

*(7) Relative register mode :- displacement is present

MOV AX, 50H [BX]
↳ 8bit displacement
or
MOV BX, 1000H [SI]
↳ 16bit displacement.

EA = 10H * DS + [BX] + 50H
or
[SI] + 1000H

* (8) Relative Based Index :-

* MOV AX, 50H[BX][SI]

MOV AX, 2000H[BX][DI]

EA = 10 * DS + [BX] + [SI] + 50H

or

EA = 10H * DS + [BX] + [DI] + 2000H

* Calculate the Effective address [E.A] for the following addressing modes if.

[AX] = 1000H ; [BX] = 2000H ; [SI] = 3000H ; [DI] = 4000H ;
[CS] = 5000H ; [DS] = 6000H ; [BP] = 7000H ; [CX] = 8000H.

* (i) MOV AX, [BX]
* (ii) MOV BX, [SI]
* (iii) MOV AL, 25H [BX][DI]
* (iv) MOV BX, [DI]
* (v) MOV BL, 7700H [BP][DI]
* (vi) MOV AX, [BP][SI]
* (vii) MOV AX, BX

Sol * (i) EA = 10H * 6000H + 2000H = 62000H.

(ii) EA = 10H * 6000H + 3000H = 63000H.

(iii) EA = 10H * 6000H + 25H + 2000H + 4000H
= 66025H.

* (iv) EA = 10 * 6000H + 4000H = 64000H

* (v) EA = 10 * 6000H + 7700H + 7000H + 4000H

= 60000H + 7700H + 7000H + 4000H.

= ~~78700H~~ 72700H.

11000
7700
18700
60000
78700

* (vi) EA = 10 * 6000H + 7000H + 3000H

= ~~6A000H~~ 6A000H

7000 H
+3000 H
A000H

8000
7700
2700
60000
72700

* (vii) ~~EA = 2000H i.e. BX~~ No, EA Calculation just moving the Contents of Bx to Ax.

(Imp.) * 8086 Architecture, memory segmentation of 8086, addressing modes of 8086.

Slip test

18/12/19

* II) Control transfer Instruction Addressing mode :-

- Intra segment (Destination is within the segment)
  - Direct
  - Indirect
- Inter segment. (Dest in different segment of outside the segment)
  - Direct
  - Indirect.

*1. Intra segment Direct addressing mode :-

* In this the address to which the control is to be transfer lies in a same segment & it is passed to the Instruction directly.

*Ex:- JMP SHORT LABEL

*2. Intra segment Indirect :-

*

* Ex:-

*3

*4.

* The address to which the control is to be transfered lies in different segment & it is passed to the Instruction Indirectly.

* The contents of memory block contains four bytes like

LSB ← IP
MSB ← IP+1
LSB ← CS
MSB ← CS+1.

* The segment address of memory block may be replaced using any addressing mode except Immediate.

* Instruction set of 8086 :-

* I. Data transfer / Copy Instructions.
* II Arithematic Instruction.
* III. logical "
* IV Flag manipulation "
* V. Loop "
* VI Branch "
* VII Machine Control "
* VIII String manipulation "

* II Arithematic Instructions *

1* ADD
2* ADC
3* INC
4* SUB
5* SBB
6* DEC
7* CMP
8* NEG
9* MUL
10* IMUL
11* DIV
12* IDIV
13* CBW
14* AAA
15* AAS
16* AAM
17* AAD
18* DAA
19* ~~DAD~~ DAS.

*ADD:→

*Syn:→ ADD destination, source.

Eg:→ ADD AX, 2300H

ADD AX, BX

ADD AX, [2300H]

ADD [2300H], BX

ADD AX, [BX]

ADD [1000H] [2300H] →Not valid.

*ADC:→ {Add with Carry}

*Syn:→ ADC destination, source. {used to add two 32 bit numbers}

(d) ← (d) + (S) + CF

Eg:→ ADC AX, 2300H {Immediate AM}

ADC AX, BX {Register AM}

ADC AX, [2300H] {Direct AM}

ADC [1000H], BX

ADC AX, [BX] {Register, Indirect AM}

* Inc :- {Increment}

* Syn :- INC destination. {(d) ← (d) + 1}

Eg :- INC AX

INC AL

INC BL

INC [1000H]

* SUB :- {Subraction}

* Syn :- SUB destination, Source {(d) ← (d) − (s)}

* Eg :- Same as ADD

* SBB :- {Subraction with borrow}

* Syn :- SBB destination, Source ; {(d) ← (d) − (s) − (CF)}

* Eg :- Same as ADD

* DEC :- {decrement}

* Syn :- DEC Destination ; {(d) ← (d) − 1}

* Eg :- Same as Increment.

*CMP:- {compare}

*Syn:- CMP destination, source

* Performs subtraction between d & s but result is not updated, only flags are affected.

*Eg:- MOV AL 07H
MOV BL 05H
CMP AL, BL

*Result:- CF = 0 i.e.. [AL] > [BL]

or CMP AL, [SI]

MOV AL 05H
MOV BL 07H
CMP AL, BL

*Result:-
* CF = 1 i.e.. [AL] < [BL]

*NEG:- {2's compliment} of destination

*Syn:-
NEG destination. {d ← 2's of (d)}

*MUL:- {unsigned Multiplication}

*Syn:- MUL source

(D) ← (S) * (D)

(AX) ← (BL) * (AL) → 8 bit

(DL)(AX) ← (BX) * (AX) → 16 bit.

* IMUL :- {Signed multiplication}

*Syn:- IMUL source.

(D) ← (D) * (S)

Same as MUL {8 bit & 16 bit}

* DIV:- {unsigned Division}.

*Syn:- DIV source.

$D \leftarrow \frac{D}{S}$

*Eg:- * DIV BL ; $AX \leftarrow \frac{AL \text{ or } AX}{BL}$ {8 bit}

{8bit) 8bit (Q → AL
R → AH

AH → Remainder {R}
AL → Quotient {Q}

* DIV BX ; (DX)(AX) ← $\frac{AX}{BX}$

DX → R
~~AH → R~~
~~AL → Q~~
AX → Q.

{16 bit}.

{16) 16bit (Q → AX
bit R → DX

* IDIV:- {Signed division}

*Syn:- IDIV source.

*Same as DIV {Everything}.

* CBW :- {Convert Signed byte into Signed word}

* Syn :- CBW {works only on Accumulator}

* Eg :-

MOV AL, F2H. {1111 0010}

CBW

Sign bit. ↓ mag = $2^7 - (2^6+2^5+2^4+2)$

Now AX = 1111 1111 1111 0010

Sign bit

F F F 2.

mag = $2^{15} - (2^{14}+2^{13}+2^{12}+2^{11}+2^{10}+2^9+2^8+2^7+2^6+2^5+2^4+2)$

* MOV AL, 02H {0000 0010}

CBW

Sign bit.

Now AX = 0000 0000 0000 0010.

0002H

* CWD :- {Convert Signed word into Signed double word}

* Syn :- CWD.

* Eg :- MOV AX, 1234H

CWD.

Result is in (DX) (AX)

↓ ↓

0000H - 1234H

20/12/17

*AAA:→ {ASCI adjust after addition}

```
MOV  AH, 00H
MOV  AL, 33H

MOV  BL, 37H

ADD  AL, BL

AAA
```

Hexa	ASCI Value
0	→ 30H
1	→ 31H
2	→ 32H
3	→ 33H
.	.
.	.
9	.
A	→ 41H
B	→ 42H
C	.
D	.
E	.
F	→ 46H.

*Execution:→

AL ← 33H

BL ← 37H

AL ← AL + BL

i.e.. AL ← 6AH

** Now, Execution of AAA is as follows.

(i) It masks the ~~MSB~~ upper four bits in AL

i.e.. AL ← 0AH.

(ii) If AL > 9 ⇒ +6 {if lower four bits of AL are greater than 9}

AL = 0000 1010

masked ← (1) 0000 0110

0000 0000

i.e.. AL ← 00

* as we are adding 6 then the AH is incremented, If we are not adding then AH is as same as before. ie.. 0.

AH ← AH + 1

AH ← 01

i.e.. AX ← 0100H. → unpacked BCD form of '10'

*AAA gives the result in the form of unpacked BCD as shown above

Packed BCD {4bit}	unpacked BCD {8bit}
4 → 0010	4 → 0000 0010 i.e.. 04.
53 → 0101 0011	53 → 0000 0101 0000 0011 i.e.. 0503

Ex: 33 & 35 using AAA.

⇒ 33
35
68.

i.e.. AL ← 68

⇒ AL ← 08 masking upper four bits.

* lower four bits of AL = 8 b.i.e.. less than 9.

So, no correction required. & also AH no increment of AH

⇒ AH = 0.

i.e.. AX = 0008 H.

* DAA (Decimal adjust after addition):-

```
MOV AL, 73H   // Decimal value of 73
MOV BL, 27H   // Decimal value of 27.
ADD AL, BL    // Result is 9AH.
DAA.
```

*Now DAA Execution

AL ← 9A H.

AL ← 1001 1010

+6 → 1 1 + 0 1 1 0
1010 0000

+60 → 0110
Carry flag ← (1) 1 1
AL ← 0000 0000

i.e.. AL ← 00 H.

CF = 1

i.e.. Ans = 100H.

```
 1
 58
+29
----
 81
```

* 58 + 29.

*Sol) AL ← 81 H. But we should get 87.

Both are less than 9. So, check for auxiliary flag.

```
  0101  1000
  0010  1001
  ----  ----
  1000  0001
(+6) +  0110
  ----  ----
  1000  0111
```

87 is decimal value.

i.e.. AL ← 87.

* 49 + 52

*Sol) AL ← 9B

```
AL ← 1001 1011
+6 → +    0110
  ---------------
     1010 0001
+60 → 0110 0000
  ---------------
     0000 0001
```

CF = 1 AL ← 01H

i.e.. Ans = 101

* AAS:- {ASCI adjust after subtraction}

```
MOV AH 00H
MOV AL, 36H
MOV BL, 33H
SUB AL, BL
AAS.
```

AL ← (36-33) H = 3H So no adjustment

Ex:-

```
MOV AH 00H
MOV AL 36H
MOV BL 37H
SUB AL, BL    // AL = 36H - 37H = FFH
AAS.
```

Execution of AAS:-

(i) mask the upper four bits.

AL ← 0F H.

(ii) if lower four bits are >9 then Sub 6. of Else leave it.

⇒ AL ← 0FH - 06H = 09H.

(iii) if there is a carry then, AH is decremented by 1.

AH ← 00H - 01H = FFH.

i.e.. AH ← FFH; AL ← 09H.

⇒ AX ← FF09H is answer

* AAS gives 10's of answer

{i.e.. 36 - 37 = -1 ← 10's of 1 is 9}

* DAS:-

* AAM:-

* DAS :- {Decimal adjust after Sub}

* Same procedure as that of DAA But Sub 6 instead of adding 6.

```
MOV  AL, 63H
MOV  BL, 17H
SUB  AL, BL
DAS
```

AL ← 63H − 17H = 4CH.

i.e. 0100 1100

− 0110

0101 0000

4

0100 1100

− 0110

0100 0110

4 6

= 46.

* AAM :- {ASCII Adjust After Multiplication}

* Syn :- AAM

* Operation :-

```
MOV  AL, 07H  // Unpacked BCD of 7
MOV  BL, 06H  // " of 6
MUL  BL
AAM
```

0111 × 0110

0000

0111x

0111xx

0000xxx

00 1 0 1 0

2 A

After AAM ⇒ It is stored as AX ← 0402H

⇒ 07H × 06H = 02AH

i.e. (7×6=42) Any = Unpacked BCD

*AAD:→ (ASCII Adjust Before Division)

*Syn:→ AAD.

```
MOV AX, 0407H // unpacked BCD of 47.
MOV BL, 07H   // "   "   "   7.
AAD           // Converts 0407 into Hexadecimal
              ie.. 47 into H => 002FH.
DIV BL
// AX = 05 06H
       ↓  ↓
       AH AL
```

7)47(6→Q
42
(5) → R

*Logical Instructions:→

1*AND
2*OR
3*NOT
4*XOR
5*TEST
6*SHL/SAL
7*SHR
8*SAR
9*ROL
10*ROR
11*RCL
12*RCR.

*1. AND:→

*Syn:→ AND destination, source.

*Eg:→
```
MOV AX, 1234H
MOV BX, 5787H
AND AX, BX.
```

AX = 0001 0010 0011 0100
BX = 0101 0111 1000 0111
AX ∧ BX = 0001 0010 0000 0100

1 2 0 4H

⇒ AX ← 1204H.

AX ∨ BX = 0101 0111 1011 0111

5 7 B 7 H

⇒ 57B7H.

*2. OR:→

*Syn:→ OR Destination, Source.

*Eg:→ MOV AX, 1234H

MOV BX, 5787H

OR AX, BX.

AX ← 5787H.

*3. NOT:→

*Syn:→ NOT Destination

*Eg:→ MOV AX, 1010H

NOT AX

AX ← EFEFH.

0001 0000 0001 0000

1110 1111 1110 1111

E F E F

*4. XOR:→

*Syn:→ XOR Destination, Source.

MOV AX, 1234H

MOV BX, 3787H

XOR AX, BX

AX ← 45B3H

AX ⊕ BX ← 0100 0101 1011 0011

4 5 B 3

*5. TEST :→

* Syn :→ TEST Dest, Source.

* Eg :→ MOV AX, 1234H

MOV BX, 5787H.

TEST AX, BX

* Operation :→

*Performs logical AND but result isn't updated only flags are affected i.e.. Zero & parity flags

*6. SHL/SAL :→

Syntax :→ SHL Register, Count of 1

SHL Destination, Count (Depending upon the Count value, it shifts the contents of register towards left)

*Eg :→

MOV AX, 4539H

MOV CL, 05H.

SHL/SAL AX, CL

AX → 0100 0101 0011 1001

1st → 1000 1010 0111 0010

2nd → 0001 0100 1110 0100

3rd → 0010 1001 1100 1000

4th → 0101 0011 1001 0000

5th → 1010 0111 0010 0000

A 7 2 0

= A720H

AX ← A720H.

AND But
ated only
affected
& parity flags.

on the
it shifts
of register

*7. SHR:-

*Syn:- SHR Register, Count
or
SHR Destination, Count.

*Eg:-
MOV AX, 4539H
MOV CL, 05H.
SHR AX, CL.

AX → 0100 0101 0011 1001

1st → 0010 0010 1001 1100

2nd → 0001 0001 0100 1110

3rd → 0000 1000 1010 0111

4th → 0000 0100 0101 0011

5th → 0000 0010 0010 1001

0 2 2 9

AX ← 0229H

⑧ *SAR:-

Syn:- SAR Register, count.

(or)

SAR Destination, count.

*ⓐ MOV AX, 4539H.

MOV CL, 05H

SHR AX, CL

* AX → 0100 0101 0011 1001

DH → 0010 0010 1001 1100

2nd → 0001 0001 0100 1110

3rd → 0000 1000 1010 0111

4th → 0000 0100 0101 0011

5th → 0000 0010 0010 1001

↓ ↓ ↓ ↓

0 2 2 9

AX ← 0229H

MSB = 0 So SAR = SLR.

(b) *MOV AX, 9539H

MOV CL, 05H.

SAR AX, CL.

AX = 1001 0101 0011 1001

⇒ 1100 1010 1001 1100

→ 1110 0101 0100 1110

→ 1111 0010 1010 0111

→ 1111 1001 0101 0011

→ 1111 1100 1010 1001

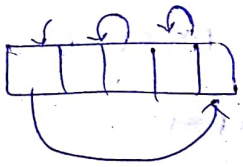
F C A 9

AX ← FCA9H {MSB=1 so SAR ≠ SLR}

*22-12-17

*ROL:→ {Rotate left}

*Syn:→ ROL Register Count.



Eg:- MOV AX, 1234H

MOV CL, 04H

ROL AX, CL.

AX→ 0001 0010 0011 0100

1st → 0010 0100 0110 1000

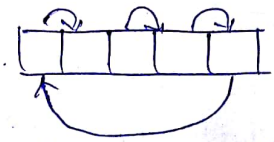
2nd → 0100 1000 1101 0000

3rd → 1001 0001 1010 0000 AX ← (2341)H.

4th → 0010 0010 0100 0001 →

*ROR:- {Rotate Right}

*Syn:- ROR register, Count.

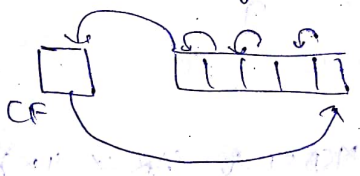


MOV AX, 1234H MOV CL, 04H.
ROR AX, CL

∴ AX ← 4123H. Ans

*RCL:- {Rotate left through Carry}

Syn:- ROR register, Count.



MOV AX, 1234
MOV CL, 04H
ROR AX, CL

assume CF=1

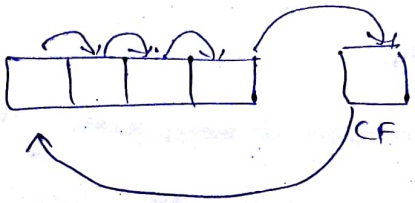
1 0001 0010 0011 0100.

1st → 0 0010 0100 0110 1001
2nd → 0 0100 1000 1101 0010
3rd → 0 1001 0001 1010 0100
4th → 1 0010 0011 0100 1000

CF=1 & AX= 2348H

*RCR :- {Rotate Right through Carry}.

*Syn :- RCR Register, Count.



*Same Ex :-

CF = 1

AX = 0001 0010 0011 0100 CF = 1

1st → 1001 0100 0110 1001 0

2nd → 0100 0100 1000 1101 0

3rd → 0010 0010 0100 0110 1

4th → 1001 0001 0010 0011 0

AX = 9123 H ; CF = 0.

9123 H

/* Program for unpacked BCD to unpacked BCD */

0023H → 0203H.

Steps :-

(i) 0230 H {0023 H}

(ii) 0203 H. {0230 H}

Program :-

```
MOV   AX, 0023H
MOV   CL, 04H
ROL   AX, CL      ; {AX = 0230H}
ROR/ROL  AL, CL   {AX = 0203H}
```

*Unpacked to packed.

0203H to

*Program:→

```
MOV  AX, 0203H
MOV  CL, 04H
ROR/ROL  AL, CL      {0230H} ↔ {0203H}
ROR  AX, CL          {0023H} ↔ {0230H}
```

*III Data Transfer Instructions:→

*MOV
* PUSH
* Pop
* XCHG
* IN
* OUT
* XLAT
* LEA.
* LDS/LES
* LAHF.
* SAHF
* PUSHF
* POPF.

Port used in the Case of Interface of 8086 with External devices

*MOV {Move}:→

Syn:→ MOV Dest, source.

Eg:→

```
MOV  AX, 1234H
MOV  AH, 12H
MOV  AL, 34H
MOV  AX, [3000H]
     {[3000] → AH}
     {[3001H] → AH}
MOV  [1300H], AX
MOV  [2000H], BL
MOV  AX, BX
```

3H to 0023H

*PUSH :- {Push into stack}

*Syn:- PUSH AX.

operation:- {SP ← SP-1, Stack ← AH}

1.* The stack pointer is decremented by 1 & AH contents are moved into stack.

{SP ← SP-1, Stack ← AL}

2.* Again the SP is decremented by 1. & AL contents are moved into stack.

*Eg:- MOV AX, 1234H
PUSH AX

*POP :- {Pop from stack}

*Syn:- POP AX.

Oper:- * The contents of top of the stack {SP} are moved into "AL" then SP is incremented by 1.

* The contents of top of the stack {SP} are moved into "AH" then SP is incremented by 1

* XCHG :- {Exchange}

Syn:- XCHG, destination, source. {Both cannot be a mem. location}

{XCHG [2000H], [3000H] not valid}

Operation:- D ⟷ S.

Eg:- XCHG AX, BX
XCHG [2000H], AX
XCHG BX, [4000H]

* IN:- {Input a port}

*Syn:- IN Accumulator, Data register.
AL/AX DX.

*operation:- Reading port.

*Ex:- MOV DX, 8000H
IN AL, DX.

IN AX, DX

{in the case of interface of processor with External devices to Read & write we use IN & OUT Respectively}

{in this case of interface Mov is not valid}.

{DX → ~~Port~~ Stores port address}

{8bit data in 8000H is moved to AL}

{16bit " " " " " " AX}

*OUT:-

*Syn:- OUT Data register, Accumulator.

*operation:- writing into port.

Ex:- OUT DX, AL

OUT DX, AX.

04-01-18 Take xerox

05-01-17

(V) * Branch Instruction *

- Unconditional
- Conditional

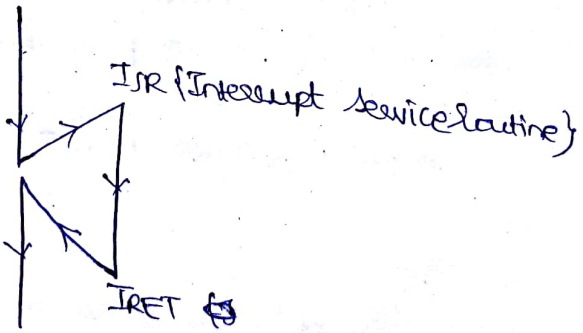
* JMP
* CALL → NEAR → IP
  → FAR → (CS) IP
* RET
* LOOP
* INT N
* INT O
* IRET

*INT N :→ {Interrupt type Number}

Eg :- INT 3. → used to terminate the program.

@ Each Interrupt takes 4 locations i.e. total 256 ISR are there ⇒ Total 256 × 4 = 1024 locations.

Eg 3 → 3×4 = 12
→ 000 CH



*INT 0;→ { Interrupt on overflow }.

Type 4 interrupt.

*Conditional

Syntax		Conditional		
Zero	* JZ / JE	label → ZF=1	* JBE / JNA (Below Equal / Not above)	label ↓ CF=1 (or) ZF
No Zero	* JNZ / JNE	" → ZF=0	* JNBE / JA (Not Below Equal / Above)	" CF=0 (or) ZF
Sign	* JS	" → SF=1	* JL / JNGE (less / Not greater Equal)	" Neither SF=1 nor OF
No Sign	* JNS	" → SF=0	* JNL / JGE	" Neither SF=0 nor OF
overflow	* JO	" → OF=1	* JLE / JNG (less than Equal / No carry)	" ZF=1 (or) neither SF=1 nor OF
no overflow	* JNO	" → OF=0	* JNLE / JE (Not less than Equal / Equal)	" ZF=0 (or) atleast any SF & OF=1
Parity	* JP / JPE	" → PF=1		
No parity	* JNP	" → PF=0		
Below, not above Equal carry,	* JB / JNAE / JC	" → CF=1		
	* JNB / JAE / JNC	" → CF=0.		

* loopE / loop Z → ZF=1
Equal / Zero.
loopNE / loopNZ → ZF=0
not Equal / not Zero

*INT 0 :→ { Interrupt on overflow }.

Type 4 interrupt.

*Conditional	Syntax	Conditional	
Zero	* JZ / JE	label → ZF=1	* JBE / JNA label ↓ CF=1 (OR) ZF=1
No Zero	* JNZ / JNE	" → ZF=0	Below Equal Not above " CF=0 (OR) ZF=0
Sign	* JS	" → SF=1	* JNBE / JA
No Sign	* JNS	" → SF=0	Not Below Equal Above.
overflow	* JO	" → OF=1	* JL / JNGE " Neither SF=1 nor OF=1
no overflow	* JNO	" → OF=0	less Not greater Equal
Parity	* JP / JPE	" → PF=1	* JNL / JGE " Neither SF=0 nor OF=0
No parity	* JNP	" → PF=0	* JLE / JNG " ZF=1 (OR) neither SF=1 nor OF=1
Below, not above Equal carry,	* JB / JNAE / JC	" → CF=1	(less than Equal) No carry
	* JNB / JAE / JNC	" → CF=0.	* JNLE / JE. " ZF=0 OR atleast anyone of SF & OF=1.
			Not less than Equal Equal.

* loop E / loop Z → ZF=1
Equal / Zero.
loop NE / loop NZ → ZF=0
not Equal / not Zero

CF=1 (OR) ZF=1

CF=0 (OR) ZF=0

SF=1 NOT OF=1

SF=0 NOT OF=0

SF=1 XOR OF=1

atleast anyone SF & OF=1

(VI) Machine Control Instructions :- / Processor Control Instructions :-

* NOP {No operation} → performs nothing till 4 Clock cycles

* HLT {Halt} → stops the Execution/terminates (Syntax is same as written)

* WAIT → Processor will be in present state until TEST = 0.

* ESC → {Escape} → frees the system bus for External Controllers like DMA or Peripherals devices

* LOCK → {Bus lock} → used as prefix with another Instruction

Prefix

* ~~Control~~

~~* Control~~

VII * String manipulation instructions :-

* REP. {Repeat}

* MOVSB / MOVSW

* CMPSB / CMPWB

* SCASB / SCASW

* LODSB / LODSW

* STOSB / STOSW

*REP :- {Repeat} → REPE / REPZ {E→Equal ; Z=zero ; N=Not}

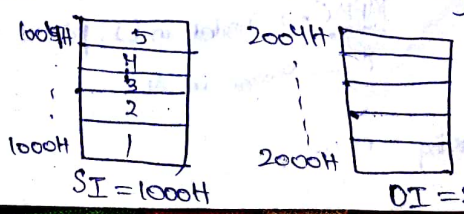
→ REPNE / REPNZ

* used as prefix to other Instructions.

* it Repeats the other instruction until CX=0. {Count=0}

* MOVSB :- Move string byte / Move string word.

/MOVSW



P.A = 10* DS + {SI} ; PI = 10H * ES + DI.

Ex:-

```
MOV SI, 1000H
MOV DI, 2000H
MOV CL, 05H
MOV AL, [SI]      ⇒ AL = 1.
up: MOV [DI], AL
    INC SI
    INC DI
    LOOP up
    INT 03H.
```

(05) →

```
CLD
REP MOVSB / MOVSW
INT 03H.
```

*CMPSB/CMPSW {Compare String Byte / String word} :-

* (i) DS : SI Str1
ES : DI Str2.

* (ii) length of String = CL/CX.

* (iii) Str1 = Str2 ⇒ ZF = 1
Str1 ≠ Str2 ⇒ ZF = 0.

Syntax :-

```
REPE CMPSB./CMPSW.
```

*SCASB /SCASW {Scan String byte / Scan String word} :-

*(i) ES:DI
AL/AX

MPMC → Stored in ES pointed to DI
→ Stored in AL.

*(ii) length of string = CL/CX.

*(iii) If there is any match ⇒ stop the execution.

ZF = Continue.

Syntax: REPNE SCASB / SCASW.

* LODSB/LODSW :⇒ (load string byte / load string word)

* Loads a byte or word ~~stored~~ to AL or AX ~~is~~ stored in DS:SI

* AL/AX ⟵ SI

*(i) CLD ⇒ SI is incremented

* (ii) SI is ~~updated~~ Incremented by 1 if LODSB

" " " " 2 " LODSW.

* STOSB/STOSW (store string byte / store string word) :⇒

* Stores AL/AX to string ES:DI.

AL/AX ⟶ DI.

*(i). CLD ⇒ DI incremented.

* (ii) DI is ~~upda~~ incremented by 1 if STOSB

" " " 2 " STOSW.

08/01/18

ASSEMBLER DIRECTIVES :-

1. DB {define byte} :- {1 byte}

* format :- Variable name DB Value1, Value2, ... ValueN

* operation :-

Eg :- Num DB 05H

ARR DB 01H, 02H, 03H, 04H, 05H.

2. DW {Define word} :- {2 bytes}

* format :- Variable name DW Value1, Value2, ... ValueN

* Eg :- Num DW 1234H

ARR DW 1234H, 5678H, CA09H.

3. DQ {Define Quad word} :- {4 words → 8 bytes}

* format :- Variablename DQ 12345678H Value1, Value2, ...

* Eg :- Num DQ 12345678H

4. DT {Define Ten Bytes} :- {10 bytes}

* format :- Variable name DT Value1

* Eg :- Num DT 123456789AH.

5. ASSUME {Assume logical name to segment} :-

* format :- Assume Segment Reguleg : Segment name, Segment Regiteg : segment name

* operation :- Gives names to segment registers

*Example:- Assume CS:CODE, DS:DATA.

6. *END:- {end of the program}

*format:- (i) END
(ii) END LABEL

*operation:- End of the program.

7*ENDS:- {End of the Segment}.

*format:- Segment name, ENDS.

8* PROC:- {Start of the procedure}

*format:- Procedure name PROC NEAR/FAR

*operation:-

*Eg:- FACT PROC NEAR → with is the segment is NEAR.

RET {Return to main program}

FACT ENDP. {End of the Procedure}

*9. ENDP:-

Procedure name ENDP

10* Dup :- {duplicate}

* format :- Variable name datatype num Dup(Value)

* operation :- To initialise memory location & assign values to the memory location.

* Eg :- Num DB 10 Dup (00H).

Initialise 10 memory locations with 10 zeros.

Num DB 5 Dup(01H), 90H.

01H 01H 01H 01H 01H 90H

*11. EQU :- {equate}

* format :- * (i) Variablename EQU value

* (ii) Assignname. EQU 'string'

* Eg :-

* (i) Expr EQU $\left(\frac{A_1 - A_2}{3}\right)/3$

* (ii) Pi EQU 22/7

*12. PTR {Pointer} :-

* format :- Datatype PTR.

* Eg :- MUL BYTE PTR [SI],

Ex:-

```
LEA SI, array
MOV AL, [SI]
MUL AL
MUL BYTE PTR [SI]   → MUL [SI] ; wrong. Cause the assembler
ADD DX, AX            doesn't know whether it is
                      a byte or word.
```

Directive

*13 ORG (originate) :-

*i) ORG Numeric value

*ii) ORG $+numeric value

Operation:- To give or assign value to the location Counter.

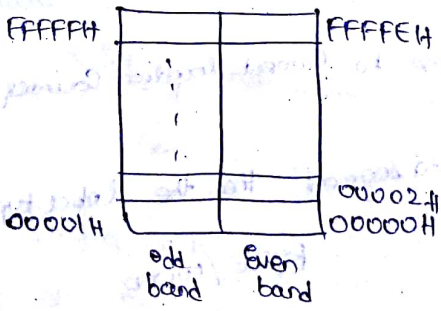
*Ex:-

```
ORG   3000H
Numl  dB  01H, 05H, 09H
```

* i.e. 01 is stored in 3000H
05 is stored in 3001H
09 is stored in 3002H

*14 Even {align even mem.address} :-

* Format :- Even



*To access data at Even address it takes only one bus but for to access data at odd address it takes 2 buses.

*Operation:- Data segment

num db 05H.

Even

num dw 1234H, 5678H, 1122H, 3388H.

Data Ends

15

*OFFset :-

*format :- OFFset Variablename

*Eg:- Data segment

List dw 1234H, 5678H, 1122H, 9999H.

Data Ends

Code segment.

MOV AX, Data

MOV DS, AX

XOR AX, AX

LEA SI, list (or) MOV SI, OFFSET List.

*16. *label :-

* format :- labelname label label type.

*operation :- gives name to Current location Counter.

If we use it in Data segment then the label type will be BYTE / WORD

" " " Code segment " " " FAR "

*17. EXTRN :→ {external}

*Format :→

*(i) for DS :→ EXTRN Variable1 : Datatype1 , Var2 : Datatype2, ...

*(ii) for CS :→ EXTRN label1 : Near/far , label2 : Near/far, ...

*Operation :→ Can be used in other modules if declared as Extern in one module. but the variables must be specified as public. in other module.

*18. Public :→

*format :→ (i) PUBLIC Var1, Var2, ... for DS

(ii) PUBLIC label1, label2, ... for CS.

*19. MACRO :→ {Start of a Macro}

*format :→ * (i) Macroname MACRO (argument1, argument2, ...)

arguments are optional.

*operation :→

* Set of instructions which are used repeatedly Can be put in Macro.

*20. ENDM :→ {end of the macro}

*format :→ * DISP MACRO MSG1

ENDM

*PROCEDURE & MACRO:-

Procedure	MACRO
*1. No. of repeated instructions are less	*1. less
*2. Execution time is less	*2. more
*3. it takes less memory	*3 more memory
*4. CALL & Return instructions are used	*4. Accessed by its name
*5. Assembler directives used are PROC, ENDP	*5 MACRO & ENDM
*6. it uses stack	*6 it doesn't.
*7. Parameters are passed in the form of register, memory, pointers, stack	*8. Arguments in the form of

12/01/18

UNIT-II

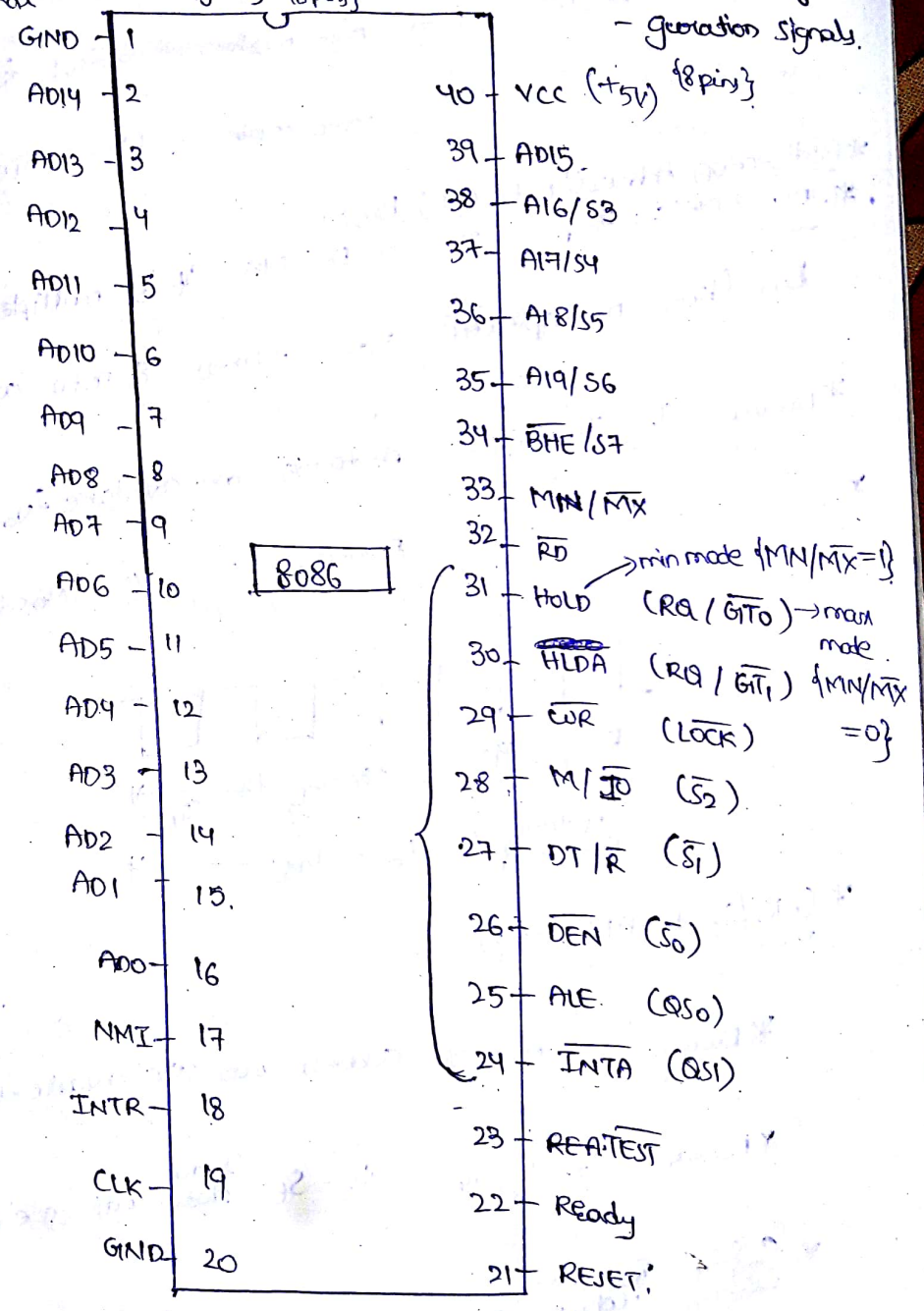
ASSEMBLING LEVL PROGRAMMING OF 8086 AND INTERFACING

Signal descriptions of 8086 :-

(I) Common signal for both min or max modes {32 pins}

(II) min mode configuration Signals {8 pins}

(III) max mode configuration signals. {8 pins}



* We have total 20 address bits & 16 data bits for 8086.

* AD0 – AD15 → 16 Address & 16 data busses.

* ~~A16/S3 – A19/S6~~ A16/S3 – A19/S6 → 4 Address bus. & 4 Status bus

Pin no 33 :- MN/$\overline{MX}$ = 1 ⇒ min mode is selected {ie.. V_{CC}}

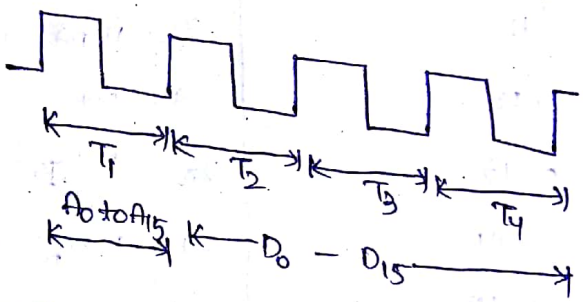
= 0 ⇒ max mode is selected {ie.. GND}

*Bidirection Address/Data (AD) Bus :-

*(AD0 – AD15) :- * A0 – A15 or D0 – D15, it is multiplexed data bus {time multiplexed} ie.. Address & Data are sent.

*During T_1 clock cycle A0 to A15 are available {No data}

* " T_2, T_3, T_4 " " D0 to D15 " " {No address}



* (A16/S3 to A19/S6) :-

*During "T_1" A16 – A19 address bits are available.

*During "T_2, T_3, T_4" $S_3 - S_6$ Status ~~data~~ bits are available.

* S_3 & S_4 :- Indicate which segment is used to generate the physical address {20 bit}.

S_4	S_3	Segment
0	0	ES
0	1	SS
1	0	CS or none
1	1	DS

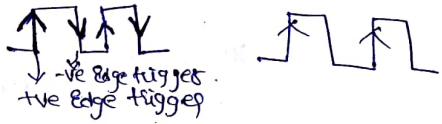
"none" is for IO operations.

* $\underline{S_5}$:→ Status of Interrupt Enable flag {whether IF is Enabled or disabled}

* $\underline{S_6}$:→ always zero & not used.

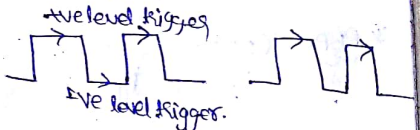
*NMI :→ {non maskable Interrupt}

* Edge triggered Interrupt (+ve)



*INTR :→ * Interrupt Request. {maskable interrupt}

* (+ve) level triggered interrupt.



*CLK :→ * Clock

* $\overline{BHE}/S_7$:→ {Pin 34}

for 34=0 ⇒ * Bus high Enable {BHE} during T_1 it is activate.

for 34=1 ⇒ *, S7 status bus is Enable during T_2, T_3, T_4.

* BHE Enables higher data bus i.e.. D_8 to D_{15} during read or write operations.

* S7 is always 1 .{not used}.

* $\overline{RD}$:→ (Read) ($\overline{in}$ 32)

* Active low & an o/p signal

* If 32=0 → * If $\overline{RD}$=0 it reads data from the External peripheral devices.

* RESET (21) :→

* If 21=1 then all the registers are intialized to zero except CS.

* CS is initialised to FFFFH.

i.e.. 8086 starts its Execution from "FFFF0H".

* READY (22) :→

* I/p to 8086

* To know whether the i/p Io devices are ready to perform Io operations.

* $\overline{TEST}$ (23) :→

* To Syrchronise the External devices.

* ~~$\overline{TEST}$ = 0 then~~

Processor is in wait state untill $\overline{test}$ = 0

i.e.. it performs nothing in wait state.

* $\overline{RD}$:→ {Read} $\{^{R_{in}}_{32}\}$.

* Active low & an o/p signal

* If 32=0 ⇒ * If $\overline{RD}$=0 it reads data from the external peripheral devices.

* RESET{21} :→

* If 21=1 then all the registers are initialized to zero except CS.

* CS is initialised to FFFFH.

i.e.. 8086 starts its execution from "FFFF0H".

* READY (22) :→

* I/p to 8086

* To know whether the i/p I/o devices are ready to perform I/o operations

* $\overline{TEST}$ (23) :→

* To synchronise the external devices.

* ~~$\overline{TEST}$ =0 then~~

Processor is in wait state until $\overline{test}$ = 0

i.e.. it performs nothing in wait state.

* II Minimum mode pins :→

* HOLD (31)
* HLDA (30)
* $\overline{WR}$ (29)
* M/$\overline{IO}$ (28)
* DT/$\overline{R}$ (27)
* $\overline{DEN}$ (26)
* ALE (25)
* $\overline{INTA}$ (24)

*HOLD & HLDA are used when an external device requesting the 8086.

* $\overline{WR}$ {write} :→
  * Perform write operation to the peripherical devices.

* M/$\overline{IO}$:→ {28}
  If 28=1 → memory operation {write/read}
  If 28=0 → IO operations {write/read}

* DT/$\overline{R}$:→ {27}
  * If 27=1 → Data is transmitted.
  * If 27=0 → Data is received.

* $\overline{DEN}$ {26} :→
* If 26=0 → Enables the transceivers. {it receive & transmit the data}

* ALE {25} :→ {Address latch Enable}
  * If 25=1 → It latches the address during T_1

i.e.. ALE = 1 ; Address lines are seperated from Address-data bus & address-status bus. i.e.. demultiplexing is done.

If ALE = 0 ⇒ No demultiplexing is done.

* $\overline{INTA}$ (24) :- Interrupt Acknowledge.

* If 24=0 * sends Interrupt acknowledge when there is an interrupt from peripherical devices.

18/01/18

*III Maximum mode pin * These pins are active when "33=0"

* RQ/$\overline{GT0}$; RQ/$\overline{GT1}$; $\overline{LOCK}$; $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$, QS0, QS1

(31) (30) (29) (28) (27) (26) (25) (24)

Request/Grant — Status of instruction queue — Bus cycle.

* $\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$ *	* Type of machine cycle *
0	0	0	Interrupt acknowledge
0	0	1	I/o Read
0	1	0	I/o write
0	1	1	Halt
1	0	0	Instruction fetch
1	0	1	memory read
1	1	0	memory write
1	1	1	Passive (inactive)

QS1	QS0	*Function*
0	0	Queue is idle
0	1	I byte of opcode
1	0	queue is Empty.
1	1	Subsequent byte of opcode

* write an ALP to perform addition of two 3x3 matrices

/* Program */

```
Assume  CS:code, DS:data.
Data segment.
num1 db  01H, 02H, 03H, 04H, 05H, 06H, 07H, 08H, 09H.
num2 db  01H, 02H, 03H, 04H, 05H, 06H, 07H, 08H, 09H.
num3 db  ? or 04
data Ends
code
~~Data~~ segment
Start:
        mov, AX, data
        MOV DS, AX
        XOR AX, AX.
        LEA DI, num2
        ~~LEA SI, A~~ MOV CL, 09H
        MOV CL, 09H
        LEA ~~BX~~, SI. SI, num1
Up:     mov AL, [SI]
        ~~LEA BX, num2~~
Wrong   mov BL, [DI]
        ~~MOV CL, 09H~~
        Add AL, BL
        mov CL, AL
        INC SI
        Inc DI
        DEC CL
        JNZ up
```

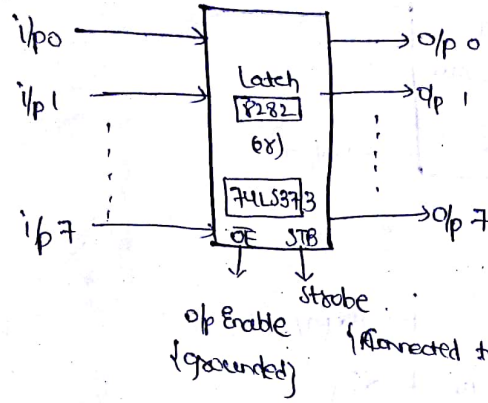
```
        LEA SI, num1
        LEA DI, num2
        LEA BX, num3
up:     MOV AL, [SI]
        ADD AL, [DI]
        MOV [BX], AL
        INC SI
        INC DI
        INC BX
        Loop up
```

```
UP:  mov, AL, [SI]
     Add  AL, [DI]
```

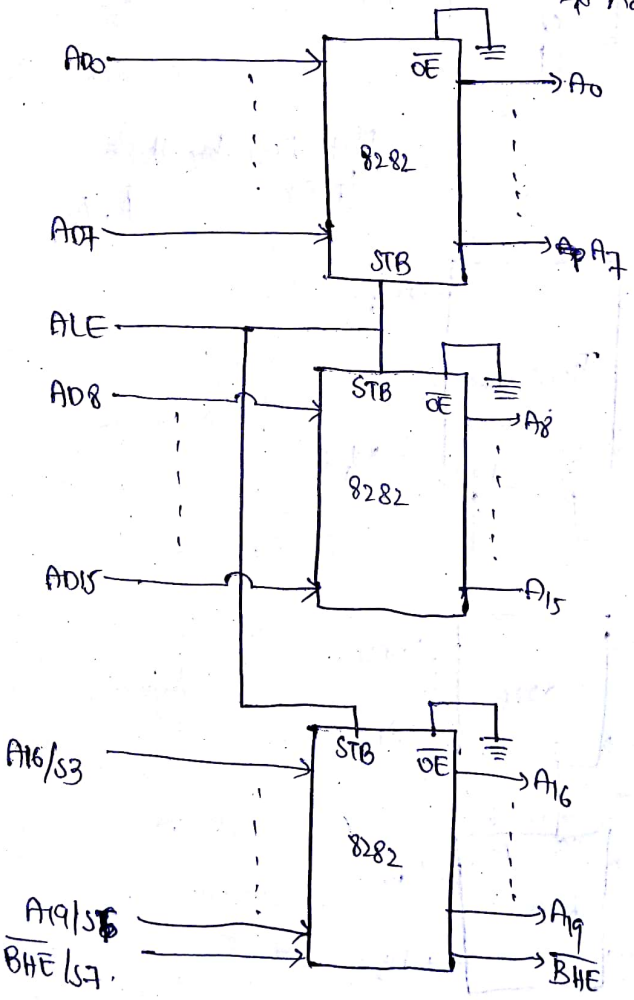
/* Perfect square root of a given number */

```
Data segment
Start:
        MOV  AX, Data
        MOV  DS, AX
        XOR  AX, AX
        MOV  DL, 25H
        MOV  BL, 01H.
        MOV  AL, 00H
UP:     SUB  DL, BL
        INC  BL,
        INC  BL
        INC  AL
        JZ   L4
        .1
```

19/01/18 *Demultiplexing of 8086 bytes :-

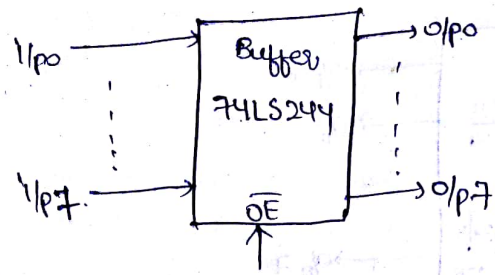


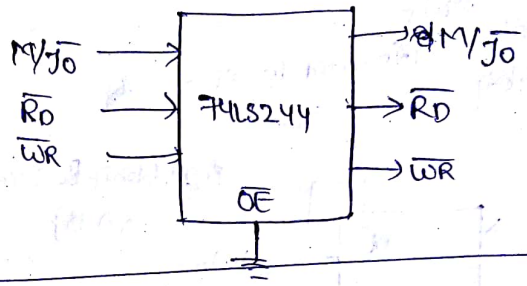
* Address Bus demultiplexing.
{A-B}



* Buffering of 8086 buses :-

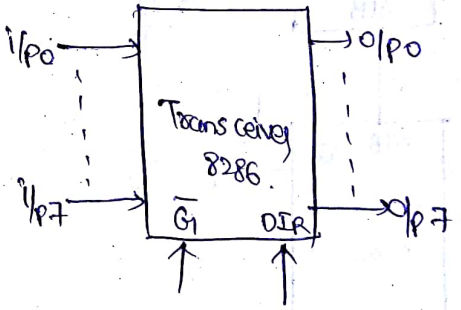
Control Bus demultiplexing (CB)

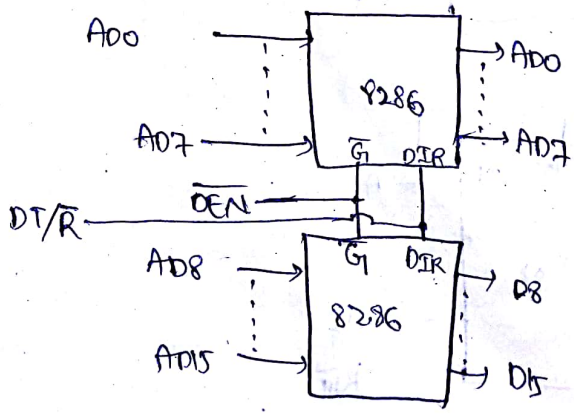




*

Data Bus demultiplexing (DB) D.B.



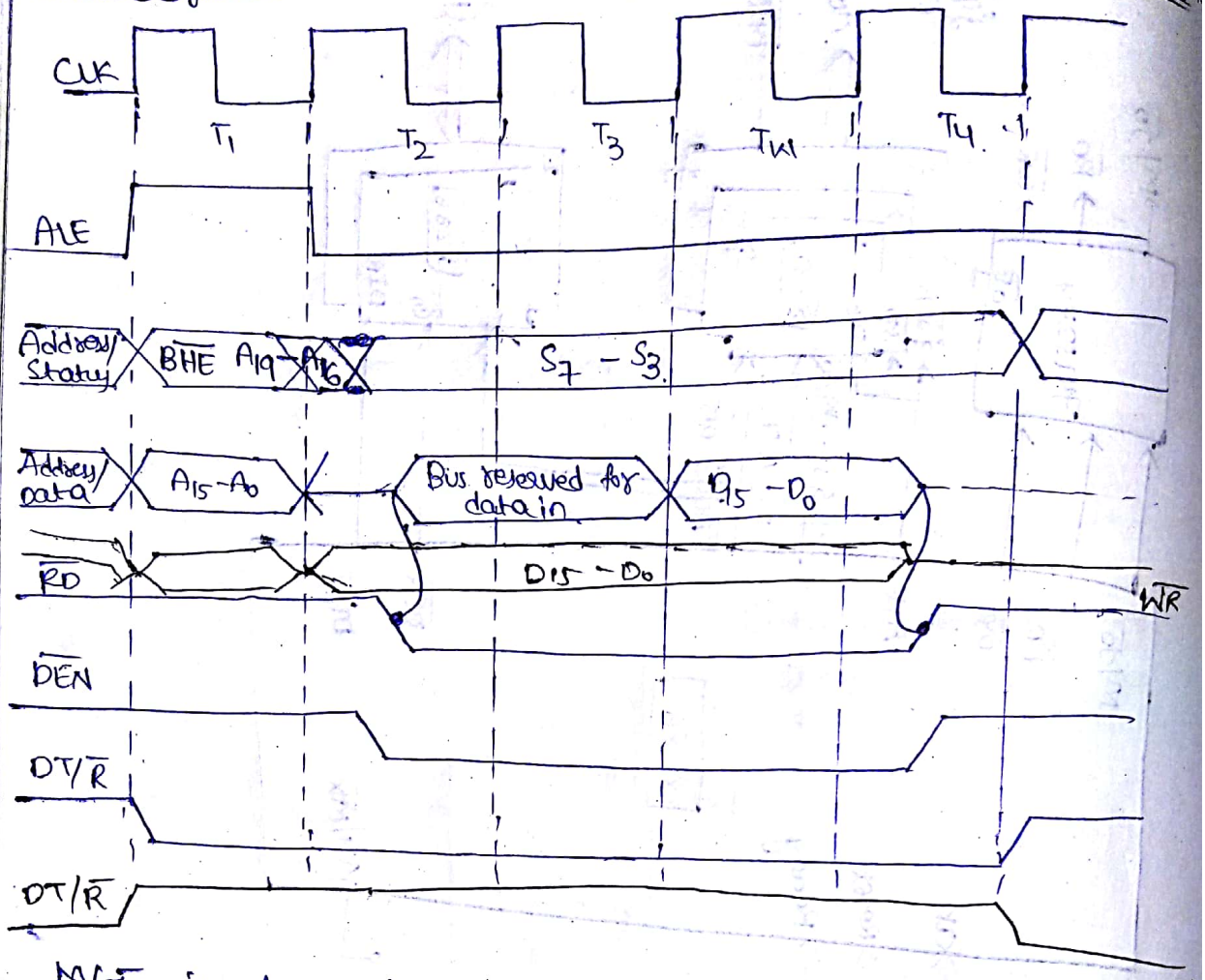


19/10/08

* Minimum mode Configuration :- (33 (MN/$\overline{MX}$) = 1)

* minimum mode Confi- Pinning diagram:→

* Read cycle:→



M/IO is also available during T_1 to T_4 & it is high if M/IO = 1 & low if M/IO = 0.

* T_1 :→

* During T_1 $A_{19} - A_{16}$ are seperated from $A_{16}/S_3 - A_{19}/S_6$.
& also BHE is seperated from BHE/S7 &
$A_{15} - A_0$ are seperated from $AD_{15} - AD_0$.

* Read operation is not Enabled during T_1.

* DEN is also not Enabled during T_1.

* During T_1 $DT/\overline{R} = 0$ = data is received. & it continues through out the machine cycle.

* T_2 :-

* ALE is disabled.

* S_3-S_7 are seperated from A_{19}/S_8 - A_{16}/S_3 & $\overline{BHE}/S_7$.

* Address/data bus is tristated. {until IO device puts the data on data bus}

* $\overline{Read}$ is Enabled. {initiated}

* DEN is active {Enables the transceiver}.

* $DT/\overline{R}$ $\Rightarrow 0 \Rightarrow$ data is recieved.

* T_3 :->

* Tristate. Continues here also, until it gets the data from IO device on data bus.

* ∴ until it get the valid data, it is in wait state.

* T_4 :-

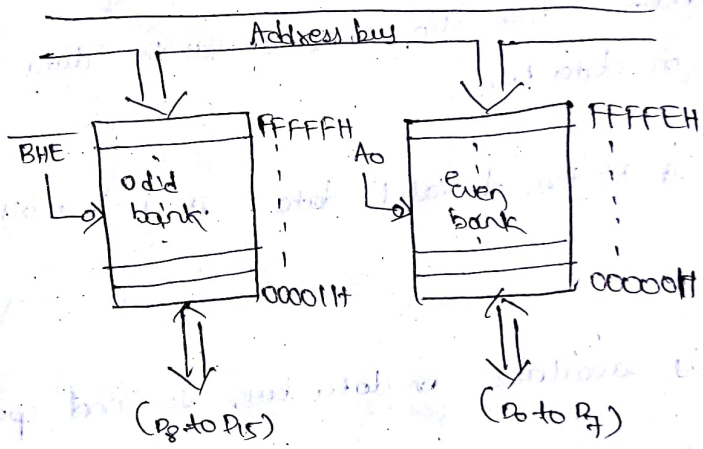
* Data is available on data bus, so Read operation takes place.

* After Read operation is Completed & again address/data bus will be tristated. & after that $DT/\overline{R} = 1 \Rightarrow$ Data will be transmitted.

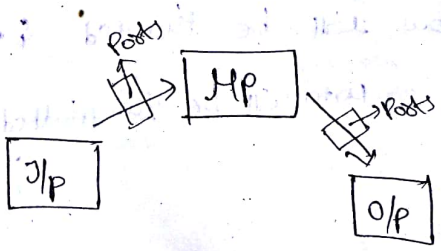
*Physical memory organisation of 8086:-

* 1MB → Even bank (512KB) ← A_0
  → Odd bank (512KB) ← $\overline{BHE}$

$\overline{BHE}$	A_0	Bank selection & Type of data transfer
0	0	Two banks & 16 bit
0	1	Odd bank & 8 bit
1	0	Even bank & 8 bit
1	1	none & no data transfer



*I/O Interfacing:-



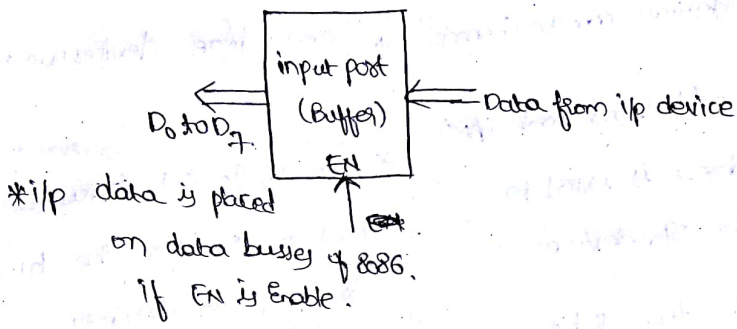
* I/O data transfer is done by "PORTS"

* Input port & Output port

* Input port :-

* To ready the data from input devices.

* "Tristate-buffer" is an ex of i/p port.

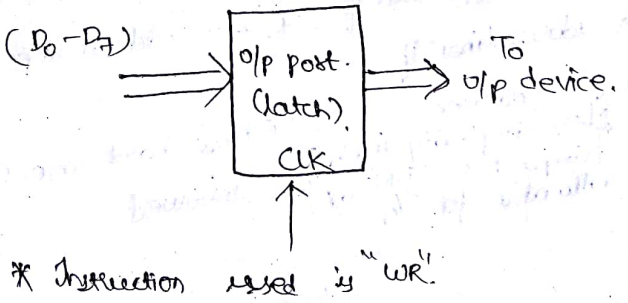


* Instruction used to read the data is "$\overline{RD}$".

*output port :-

*To send data from 8086 to o/p devices.

* "latch" is an ex of o/p port.



* Instruction used is "WR".

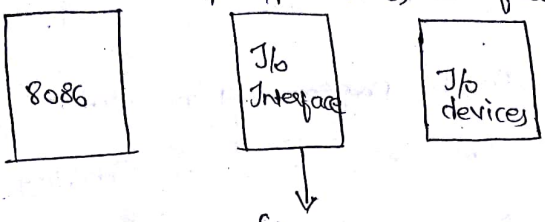
I/o Interfacing Techniques :- *(i) I/o mapped I/o (isolated I/o)

*(ii) memory mapped I/o.

* I/O mapped I/O *	*Memory mapped I/O *
*8 bit / 16 bit address is used to address the I/O devices.	* 20-bit address is used address the I/O device.
* 2^8 / 2^{16} devices can be Connected	* 2^{20} = 1MB devices can be
*If direct addressing mode (AM) 8bit address is used to address the I/O devices.	* memory Related Instructions used to transfer the data 8086 & I/O devices Ex: MOV, LEA etc.
*If indirect then 16 bit	
* IN & OUT Instruction are used to transfer data b/w 8086 & I/O devices.	
* $M/\overline{IO} = 0$.	* $M/\overline{IO} = 1$
* 16-address lines are decoded if indirect & 8-address lines if direct A.M.	* 20-address lines are decode
* 64 Kbytes are allocated for I/O mapped I/O. (hardware less Complexity involved Compared to M.M I/O)	* More hardware Complexity involved.

* 8255 - PPI :- {Programmable peripheral Interface}.

→ Parallel I/p o/p (PIO) Interface.



I/O ports { latches, Buffers

* Features of 8255 :-

→ (1) * 40-pin DIP {Dual inline package}.

→ (2) * 4-ports

3-I/O ports {Port A, B, C}
Each port contain 8 I/O lines
Port A → PA_0 to PA_7
Port B → PB_0 to PB_7
Control port is used to program I/O ports.

1 control port → {control word register}

Port C → PB_0 to PB_7.

→ (3) ~~***~~

∴ there are total 24 I/O lines.

24 I/O lines

Group A {12 lines}	Group B {12 lines}
– Port A	– Port B
– Port C (upper) {PC_4 to PC_7}	– Port C (lower) {PC_0 to PC_3}

* modes of operation of 8255 :- mode {can be selected using control port}

*(i) BSR mode {Bit set Reset} *(ii) IO mode.

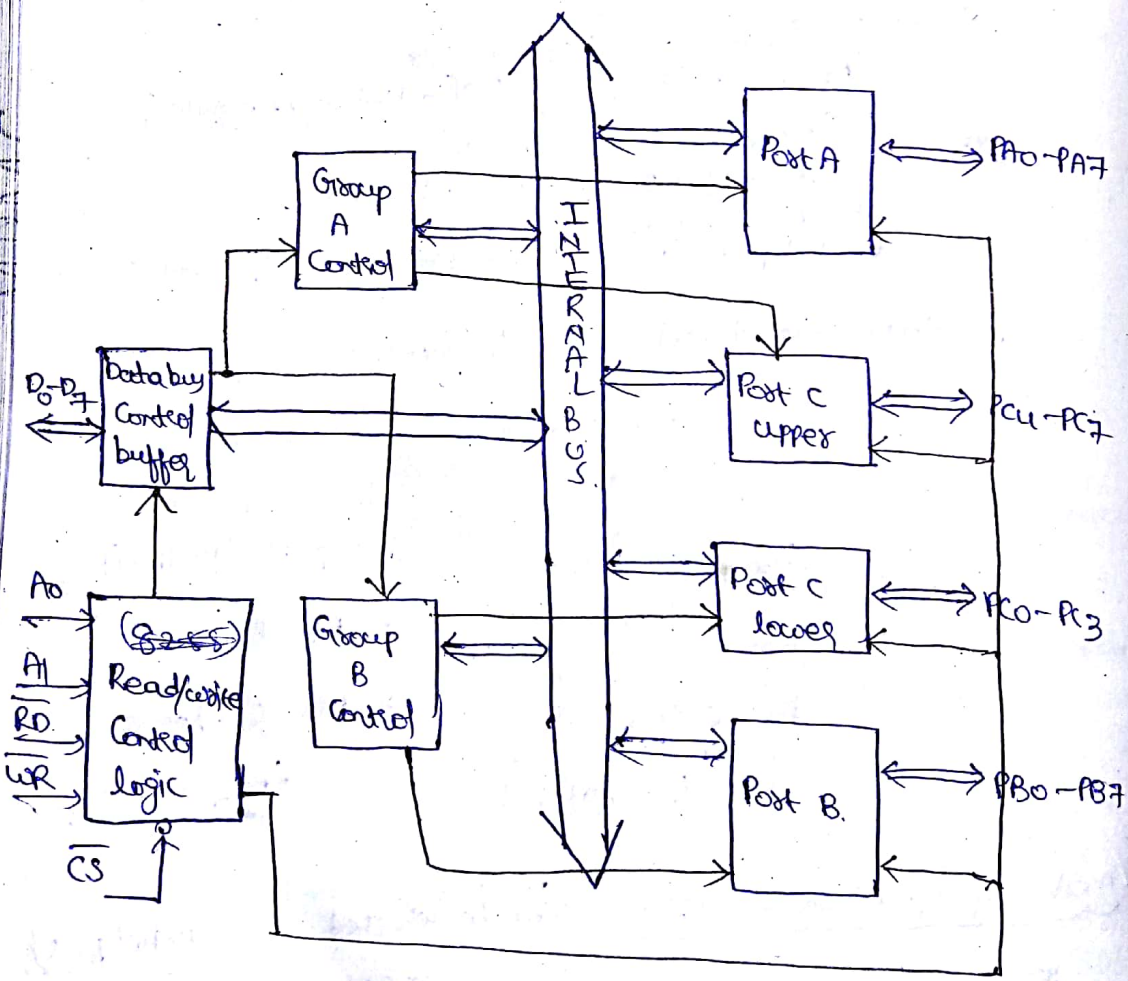
*(i) used to Enable or disable port C pins.

*(ii) used to program Port A, B, C as I/O or o/p.

To modes → mode 0 {Simple I/O mode} all 3 ports are i/p or o/p mode
→ mode 1 {Hand shaked I/O mode}
→ mode 2 {Bidirectional I/O mode}

* PA, PB, PC can be programmed in mode 0
* PA, PB " " " mode 1 & PC
* is used as handshake {Synchronizing}
* PA can be " " " mode 2 & PB, PC are " " " "

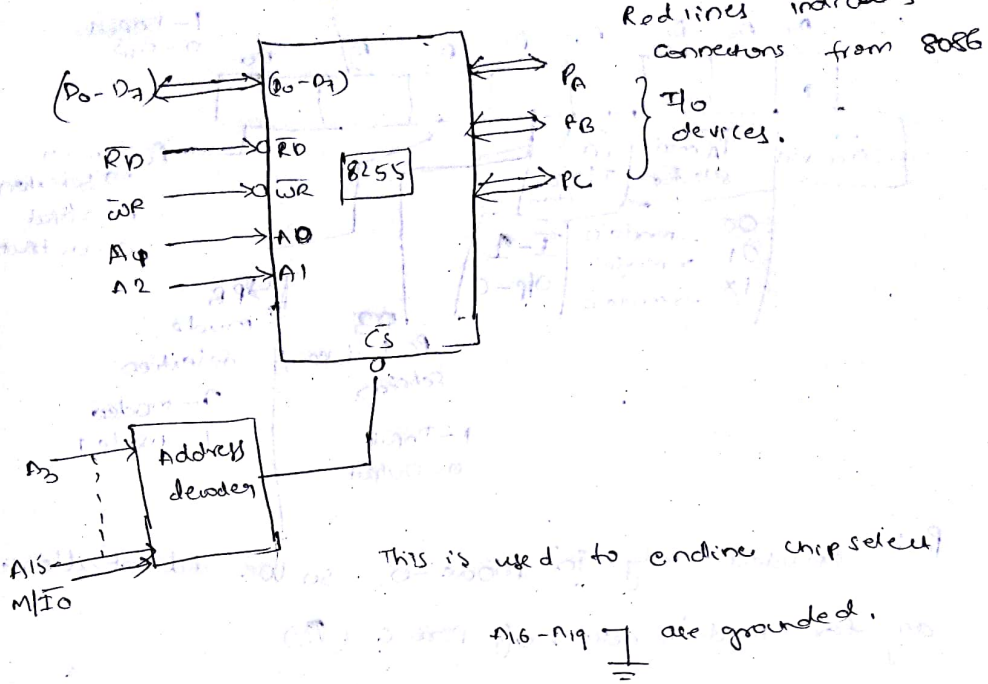
* 8255 PPI ARCHITECTURE:-



all 3 ports are ... i/p or o/p modes.
)
o mode)
21 & Pc
- & PB, Pc
PA0-PA7
PC4-PC7
PC0-PC3
PB0-PB7

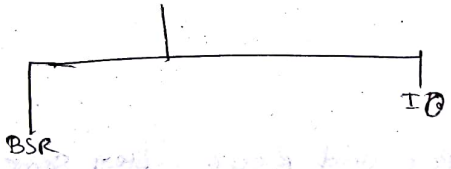
24/1/18

8255 Interfacing to 8086

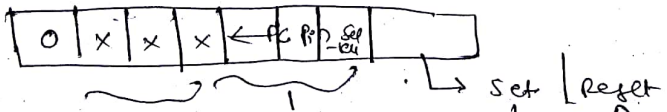


This is used to endline chip select

** Control word formats of 8255:-

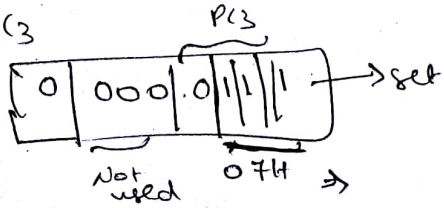


i) BSR : Bit set Reset of Pc



000 PC0
001 PC1
⋮ ⋮ ⋮
111 PC7

Ex:- Set PC3



1 0 1 — PC5

1 1 0 — PC6

1 1 1 — PC7

For ex: To set PC3

0 x x x 0 1 1 1

0 0 0 0 0 1 1 1

0 7

07h

***

(ii) IO mode: