MODULE - 2

* Addressing Modes of 8086

Addressing mode indicates a way of locating data or operands.

The instructions in 8086 is of two types:

(i) Sequential Control flow instructions.

(ii) Control transfer instructions.

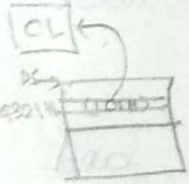
(i) Sequential Control flow instructions:

(1) Immediate addressing mode: In this type of addressing immediate data is a part of instruction and appears in the form of successive byte or bytes.

eg:- MOV AX, (0005H) → value

Here 0005H is the immediate data and is moved to AX.

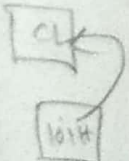
(2) Direct addressing mode: In this addressing mode, only the offset address is specified in the instruction.



eg:- MOV CL, [4321H]; Move the data in the offset address 4321H into CL

Effective address, EA = DS * 10H + 4321H

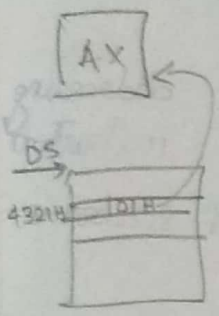
(3) Register Addressing mode: In this mode, operands are specified using registers. All registers, except IP, may be used in this mode.



eg:- MOV CL, DL ; Move the data in DL to CL

MOV AX, BX : move data in BX to AX

(4) Register Indirect : The address of the memory location which contains data or operands is determined in an indirect way, using the offset registers. The offset address of the data is either in BX or SI or DI registers. The default segment is either DS or ES.



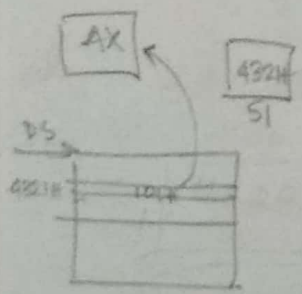
eg:- MOV AX, [BX] ; Here, data is present in the memory location in DS, whose offset address is in BX.

$$EA = DS * 10H + [BX]$$

(5) Indexed : In this addressing mode, offset of the operand is stored in one of the index registers.

DS is the default segment for index registers SI and DI.

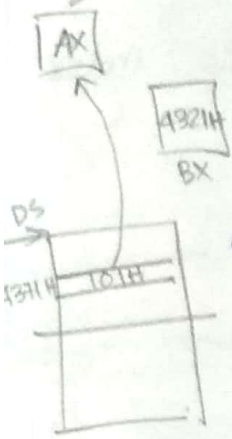
In case of string instruction DS and ES are default segments for SI and DI respectively.



eg:- MOV AX, [SI] ; Here, the data is available at an offset addr stored in SI in DS.

$$EA = DS * 10H + [SI]$$

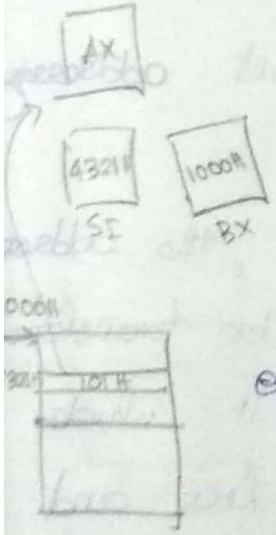
(6) Registes Relative : In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default DS or ES segment.



eg:- MOV AX, 50H [BX] ; Here the data is stored in the memory location which is (50H + the offset address in BX of DS

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

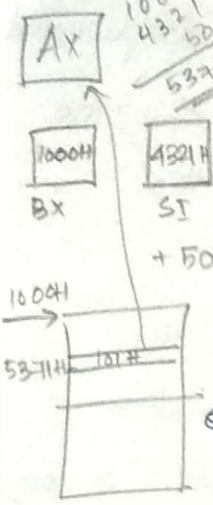
(7) Base Indexed : The effective address of data is formed in this addressing mode, by adding content of a base register (anyone of BX or BP) to the content of an index register (anyone of SI or DI). The default segment registes may be ES or DS.



eg: MOV AX, [BX], [SI]. Here, the data is stored in the offset address given in SI, which belongs to the segment whose base address is given in BX.

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

(8) Relative Base Indexed : The effective address is formed by adding an 8 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 in a default segment.



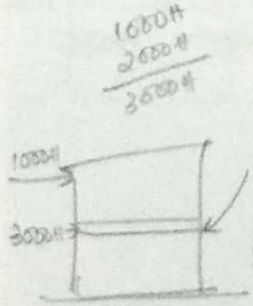
eg:- MOV AX, 50H [BX] [SI] ; Here 50H is the immediate displ, BX is the base register and SI is an index register.

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

(ii) Control Transfer Instructions

Intersegment and Intrasegment Mode: There are two addressing modes for the control transfer instructions; intersegment and intrasegment addressing modes.

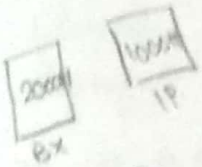
(a) Intrasegment Direct Mode : In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value.



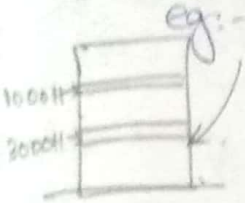
In this addressing mode, the displ. is computed relative to the content of the instruction pointer, IP.

eg:- JMP short label. JMP 2000H.

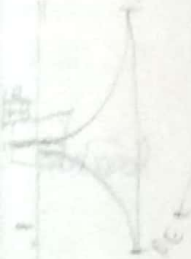
(10) Intrasegment Indirect Mode: In this mode, the displ. to which the control is to be transferred is in the same segment in which the current transfer instruction lies, but it is passed to the instruction indirectly.



eg:- JMP [BX] ; Jump to effective addr stored in BX ⟹ JMP [BX + IP]

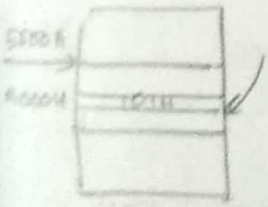


(11) Intersegment Direct mode: In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here the CS and IP of the destination address are specified directly in the instruction



eg:- JMP 5000H : 2000H ; Jump to the effective address 2000H in the segment 5000H

↓ offset addr ↓ base addr

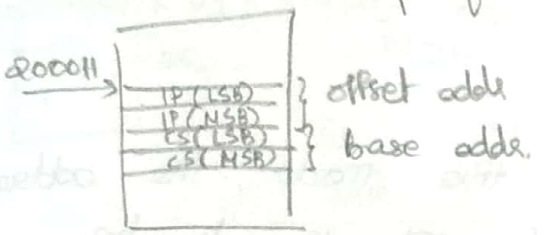


(12) Intersegment Indirect: In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e, content of a memory block containing four bytes, i.e, IP (LSB), IP (MSB), CS (LSB)

and CS (MSB) sequentially.

The starting address of the memory block may be refered using any of the addressing modes, except immediate mode.

eg:- JMP [2000H] ; Jump to other segment specified at the address 2000H in DS.



* 8086 Instruction Set

(1.) Data Copy / Transfer Instructions

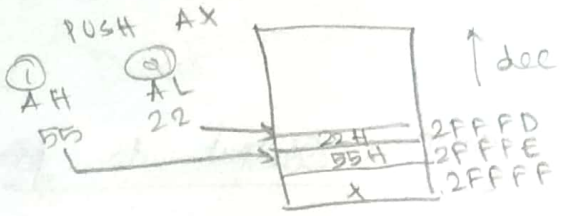
(i) MOV ; move : Move data from registers / memory location to another registers / memory location.

eg:- MOV AX, 5000H ; Immediate
MOV AX, BX ; Register
MOV AX, [SI] ; Register Indirect Indexed
MOV AX, [2000H] ; Direct
MOV AX, 50H [BX] ; Register Relative, 50H displacement

(ii) PUSH : PUSH to Stack : This instruction pushes the contents of the specified registers / memory locations onto the stack. SP is decremented by 2 after each execution

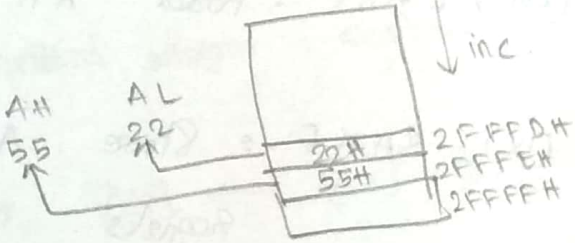
eg:- PUSH AX ; the content of register AX is pushed onto stack.

while pushing, the pointer is first decremented, and then the higher 8-bits is stored followed by lower 8-bits



(iii) POP : POP from stack : This instruction loads the while POPing, specified register/ memory location with the lower contents of the memory location of which 8-bit is first popped the address is formed using current 8 stored in AL stack segment and stack pointer. followed by the After every pop, stack pointer is incremented next 8-higher by 2. bits in AH.

eg: POP AX



(iv) XCHG : Exchange : Exchange content of destination and source. ⟷

eg:- XCHG [5000H], AX ; exchange data b/w AX and content of memory location 5000H.

XCHG BX, AX ; exchanges data b/w AX and BX.

(v) IN : Input the port : copies/reads data from the adds. of the input ports to AL or AX input device is stored into the required register

eg:- IN AL, 03H → addr of i/p device

IN AX, DX ; DX is the default Register

DX is the only register whose the addr of the i/p device is stored

(vi) OUT : Output to port : writing to output port.

eg:- OUT 03H, AL

OUT DX, AX

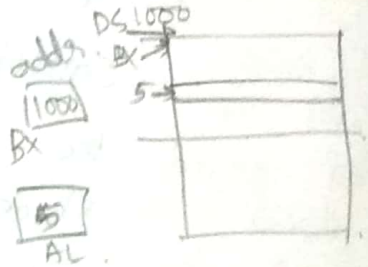
the addr of the I/O device stored in the register is loaded.

(vii) XLAT : Translate ;

AL ← DS : [BX + AL]

base addr. offset addr.

is stored to AL of DS segment



(viii) LAHF : Load AH from lower byte of flag
Store the lower 8-bit of flag reg to AH

(ix) SAHF : Store AH to lower byte of flag Registers. Store the value stored in AH to the lower 8-bit of flag register.

(x) PUSH F : PUSH flags to stack
Push the contents in F reg. to the stack.

(xi) POP F : POP flags from stack.
Pop the contents of stack to the F reg.

(2.) Arithmetic Instructions :

(i) ADD : add two values :

eg:- ADD AX, 0100H ; Immediate AX ← AX + 0100

ADD AX, BX ; Register

ADD AX, [SI] ; Register indirect

1000 +
-
-

BX = 1000
AL = 5
X <A :

ADD AX, [5000H] ; Direct

ADD [5000H], 0100H ; Immediate

ADD 0100H ; destination AX is implicit

(ii) ADC (Add with carry) : Also adds carry flag bit to the result.

eg:- ADC AX, BX

[AX] ← [AX] + [BX] + [CF] → carry flag

(iii) SUB : Subtract source operand from destination operand

SUB AX, BX ; [AX] ← [AX] - [BX]

(iv) SBB : (Subtract with borrow)

SUB AX, BX ; [AX] ← AX - [BX] - [CF]

(v) INC : Increment content of the specified reg. or memory location by 1.

eg:- INC AX ; Register

INC [BX] ; Register Indirect

INC [5000H] ; Direct

(vi) DEC : (decrement) : decrement content of register or memory location by 1.

eg:- DEC AX

(vii) CMP (Compare) : Compare content of register or memory location

eg:- CMP BX, CX

The operation used is BX − CX and the flags according to the comparison is made. CF & ZF,

If BX − CX = 0 ; CF = 0, ZF = 1

BX − CX = +ve ; CF = 0, ZF = 0

BX − CX = −ve ; CF = 1, ZF = 0

(viii) DAA (Decimal Adjust Accumulator) : Used to convert the result of the addition of two packed BCD number to a valid BCD no.

eg:- 19 + 23 = 42.

```
   0100  1001
   0010  0011
   ---------
   0110  1100   ⇒ 6CH.
         0110
   +  ---------
   0111  0010   ⇒ 72
   ---------
```

Adding correction factor (0110) to the nibble greater than 9.

(ix) Neg : (Negate) : It forms 2's complement of specified register / memory location.

(x) MUL : Multiplies an unsigned byte or word by the content of AL (if it is 8-bit)

eg: MUL BH ; [AX] ⟵ [AL] × [BH]
16 bit 8 bit 16 bit

AX → Accumulator is used to store one value and the result is stored into AX

MUL CX ; [DX] [AX] ⟵ [AX] × [CX]
16 bit 16 bit 16 bit 16 bit

default.

(xi) DIV : divides word or double word by a 16 bit or 8 bit operand.

eg:- DIV BL ⟺ $\frac{[AX]}{[BL]}$ ⇒ quotient stored in AL
remainder stored in AH.

(3.) Logical Instructions

(i) NOT : complement content of specified memory location or register. 0 → 1, 1 → 0

(ii) AND d, S : ANDs destination and source and stores the result in destination.
The AND operation is performed bitwise

S 1010 1100
d 0011 1011
1000 → d

destination & source &

(iii) OR d, S : ORs destination & source & stores the result in destination.
(bit wise)

(iv) XOR d, S : XORs destination & source & stores the result in destination.

(v) TEST d, S : ANDs destination & source but only flags are updated.
(eg:- if the result is zero, zero flag is updated)

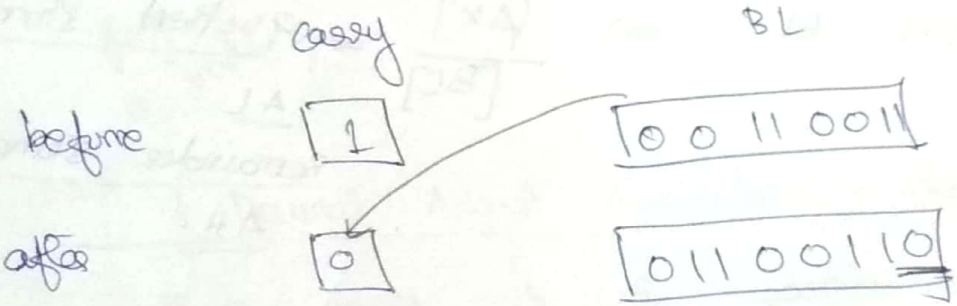
(4) Shift instructions

(i) SAL / SHL destination, count : shift logically left

[count = 1 is directly specified in the instr]

[count > 1 is given in CL register]

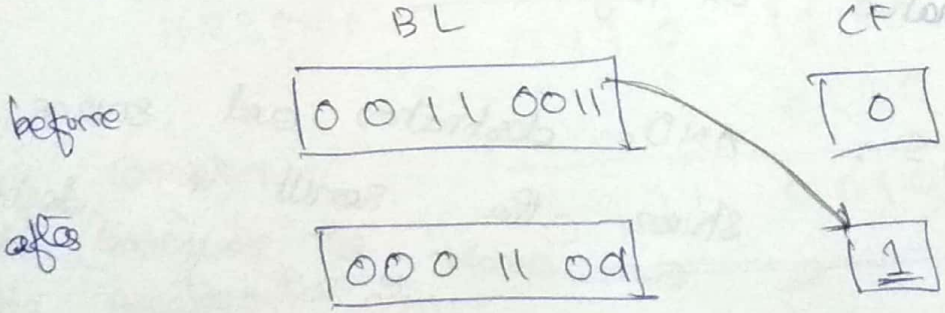
eg:- SHL BL, 1



(ii) SHR destination, count : shift logically right

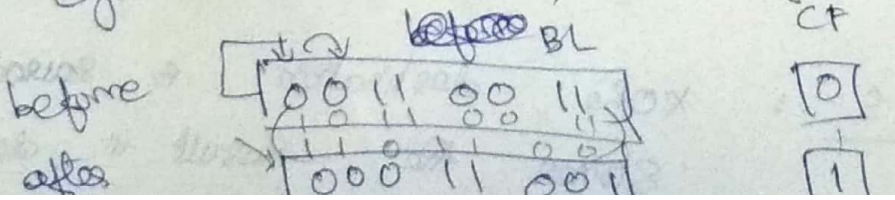
if count = 1, directly specified
count > 1, specified in CL

eg:- SHR BL, 1



(iii) SAR destination, count ; shift arithmetic right
the MSB value is repeated so that the sign doesn't change.

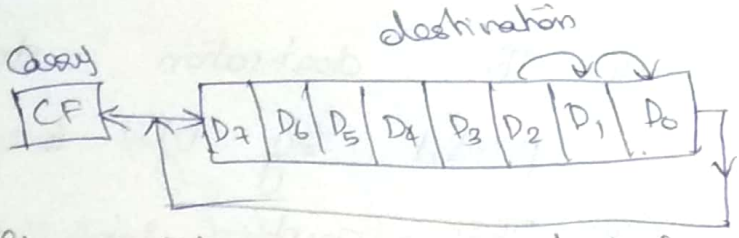
eg:- SAR BL, 1.



(5.) Rotate instructions

(i) ROR destination, count : right shift the bits of destination.

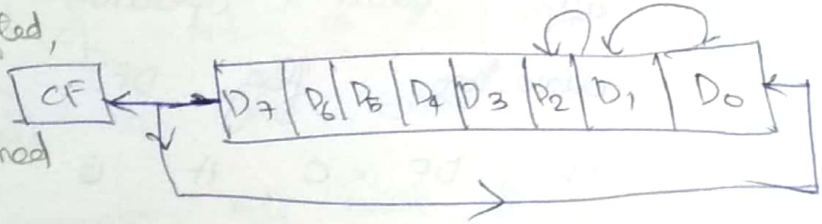
eg :- ROR BL, 1



the bits are right shifted, D_0 is stored in D_7 and a copy of it is stored in CF as well

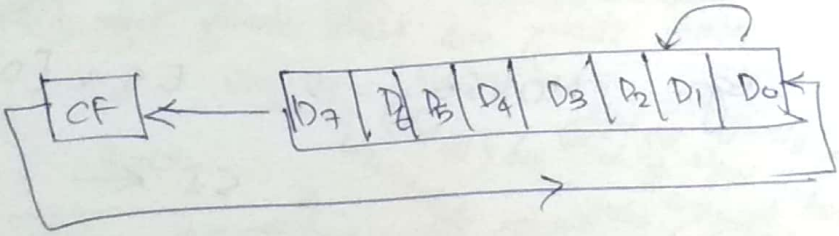
(ii) ROL destination, count : left shift the bit of destination.

the bits are left shifted, D_7 is stored in D_0 & a copy of it is stored in CF



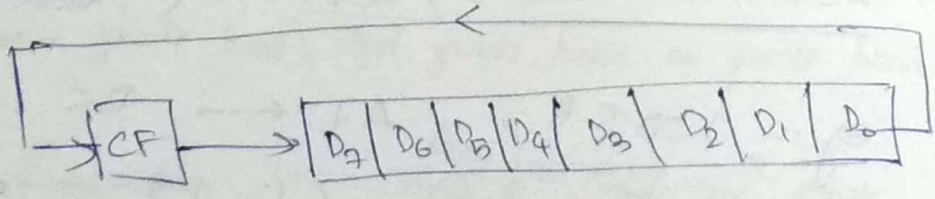
(iii) RCL destination, count : left shift through carry

the bits are left shifted, $D_7 \rightarrow CF \rightarrow D_0$



(iv) RCR destination, count : right shift through carry

the bits are right shifted, $D_0 \rightarrow CF \rightarrow D_7$



(6.) Storage Operations

Storage is a series of bytes stored sequentially in the memory. Store instruction operates

in the following manner:

(i) Source string is at a location pointed by SI in the data segment. [SI] of DS

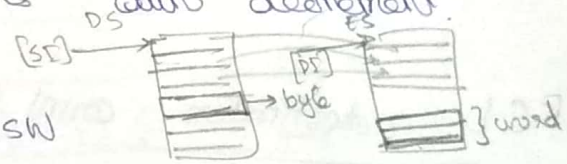
(ii) The destination string is at a location pointed by DI in the extra segment. [DI] of ES

(iii) The count for string operation is always given by CX.

(iv) SI and DI are incremented / decremented after each operation depending upon the direction flag DF.

if DF = 0, it is auto increment.

if DF = 1, it is auto decrement.



(i) MOVS : MOVSB / MOVSW

Move string ⇒ Move string byte / Move string word

eg: MOVSB ; ES : [DI] ← DS : [SI]

SB → string moved byte by byte (8 bits)
SW → string moved word by word (16 bits)
DF (Direction flag) determines direction of flow
SI & DI dec; if DF = 1
inc; if DF = 0

SI ← SI ± 1 — depending upon DF

DI ← DI ± 1 — depending upon DF

(ii) LODS : LODSB / LODSW : Load string

Load string ⇒ Load string byte / Load string word

eg:- LODSB ; AL ← DS : [SI]

the string in a given location is loaded to accumulator.
if LODSB, AL is used (8 bits)
if LODSW, AX is used (16 bits)

SI ± 1 — depending upon DF

(iii) STOS : STOSB / STOSW ; store string ⇒ store string byte / store string word.

the string in accumulator is stored to a given location
if STOSB, AL is used
if STOSW, AX is used

ES : [DI] ← AL

DI ← DI ± 1 — depending upon DF

(iv) CMPS : CMPSB / CMPSW : Compare string

Compare string ⇒ compare string byte / compare string word.

Compare a byte or word in data segment with a byte or word in the extra segment.

Comparison is done by [SI] − [DI]

Flag bits are affected, but result is not stored anywhere.

(v) SCAS : SCASB / SCASW : Scan string

Scan string ⇒ scan string byte / scan string word

Compare the content of AL or AX with a string byte (or word) in the extra segment.

SCASB ; Compare AL with ES : [DI]

DI ← DI ± 1.

(vi) REP (repeat prefix used for string instruction):

It repeats the string instructions CX number of times.

eg:- MOV CX, 0005H ⇒ the count is stored in CX

REP MOVSB : MOVSB is repeated 5 times.

(vii) REPZ / REPE (repeat on zero / equal):

Repeat string instruction till ZF = 1.

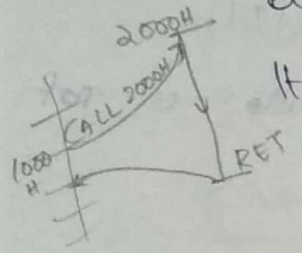
(viii) REPNZ / REPNE (Repeat on non-zero / not equal)

Repeat string instruction till ZF = 0.

(7.) Control Transfer or Branching Instruction

(i) CALL : Unconditional call (8086 offers only unconditional CALLS)

It is used to call a subroutine from a main program.



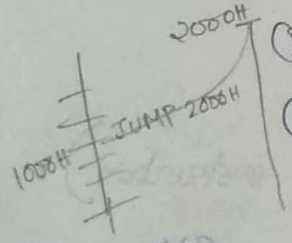
It can be of two types:

(I) Intra segment transfer – CS & IP changes

(II) Intra segment transfer – only IP changes.

(ii) RET : Return from subroutine or procedure.

(iii) JMP : JUMP (unconditional jump)



(I) Near Jump → only IP changes

(II) Far Jump → both CS and IP changes.

Unlike CALL, a RET instn cannot be given to return to the original instn.

(iv) Conditional Transfer:

(I) JZ / JE → Jump if ZF = 1

(II) JNZ / JNE → Jump if ZF = 0

(III) JS → SF = 1.

(IV) JNS → SF = 0

(V) JO → OF = 1

(VI) JNO → OF = 0.

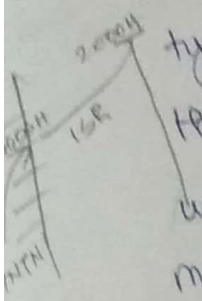
(v) INTN : Interrupt Type N (software interrupt – specified by programmer in the program)

N can be 00H to FFH (256 interrupts).

INTN is used to Jump into interrupt

source routine.

When an INTN instruction is executed, the type byte N is multiplied by 4 and the contents of IP and CS of the ISR will be taken from the hexadecimal multiplication (N × 4) as offset address and 0000 as segment address.



(vi) IRET : Return from ISR.

IRET is used to return from ISR

The instruction is used to return from the interrupt source routine.

(vii) LOOP instruction

(I) LOOP label : Jump to specified 'label' if CX not equal to 0 and decrement CX

The count is stored in CX.

eg:- MOV CX, 40H

BACK : MOV AL, BL

...

LOOP BACK;

DO CX ← CX − 1

Go to back if CX ≠ 0

0100 1101
1010 1001
0000 1001

(II) LOOPE / LOOPZ label ; (Loop on equal / loop on zero).

Loop when the comparison is equal or ZF = 1

eg:- LOOPZ BACK ; DO CX ← CX − 1

Go to 'BACK' if CX not equal to zero and ZF − 1.

(iii) LOOPNE / LOOPNZ label ; (loop on not equal / loop on non-zero).

LOOP when the comparison is not equal.

eg: LOOPZ BACK ; DO CX ← CX − 1
Go to BACK if CX not equal to 0 and ZF = 0.

(8.) Machine Control Instruction

(i) WAIT → wait for test input pin to go low.

(ii) HLT → Halt the processor.

(iii) NOP → No operation.

(iv) ESC → Escape to external device (prefix)

(v) LOCK → Bus Lock instruction prefix (prefix)

* Assembler Directives

Assembler is a program that ~~converts~~ converts assembly language program to equivalent machine codes. For doing this assembler needs some info from the programmer for the storage of particular constants or variables, logical names etc. Those type of links are given to the assembler using some predefined alphabetical strings called assembler directives.

(1.) DB : define byte : used to reserve byte or bytes of memory locations in the available memory.

used to store a list into the memory

RANKS
01H
02H
03H

[the name of the list is RANKS and stores its values in the subsequent locations.]

eg:- ① RANKS DB 01H, 02H, 03H

② List DB FFH DUP (?)

FFH → leaves 256 locations to store values
DUP (?) → stores some random values in the reserved locations.

If DUP (10) ⇒ stores the value (10) in all locations.

(2) DW : define word : used to reserve number of memory words (16-bit) instead of bytes.

used to store a list of words

eg:- WORDS DW 1234H, 4567H

WORDS DW 05H DUP (?)

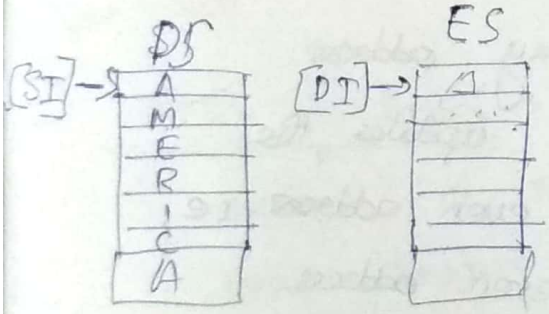
05H → size / no. of locations
DUP (?) → values

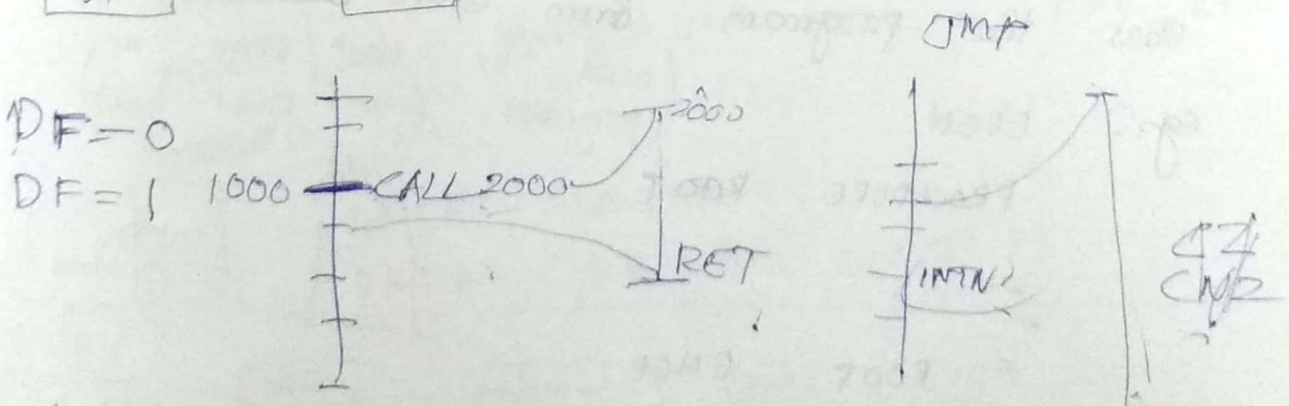
(3) DQ : Define Quad word : used to reserve 4 words of memory for the specified variable.

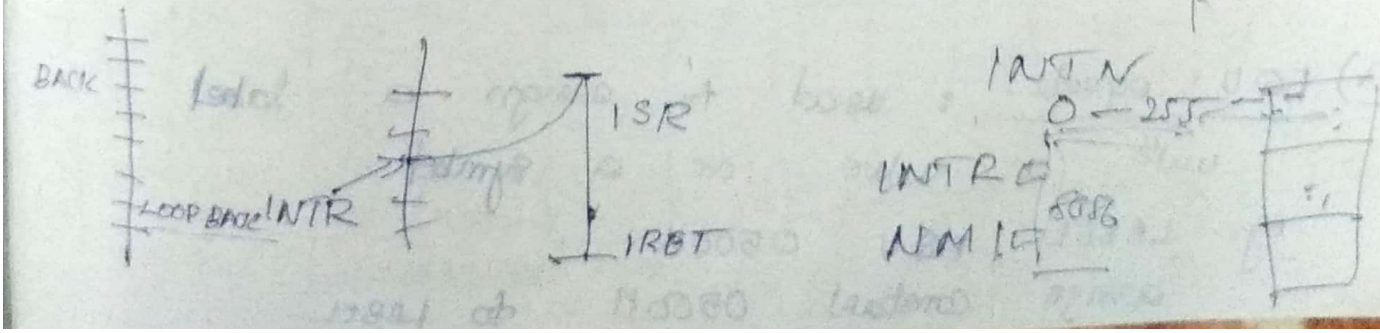
used to store 4 words in the memory

TABLE DQ 12345678, 1234,5678

name → TABLE, DQ → keyword, 4 words







(4.) DT : Define Ten Bytes — to store 10 bytes.

DD : define double word — store 2 words.

(5) ASSUME : The ASSUME directive is used to inform the assembler, the names of the logical segment to be assumed for different segments.

eg:- ASSUME CS : CODE (to assume a segment with a name)

(6) ENDS : End of Segment :

eg:- CODE ENDS (defines the end of segment)

ENDP : End of procedure (defines the end of a procedure)

(7) END : END of program : marks the end of assembly language program.

(defines the end of a complete program)

(8.) EVEN : align on even memory address.

The EVEN assemble directive updates, the location counter to the next even address, i.e, start the program from even address.

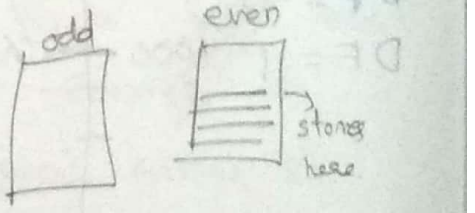
eg:- EVEN (used to start / store a program from even bank)

PROCEDURE ROOT

:

:

ROOT ENDP



(9.) EQU : equate : used to assign a label with a value or a symbol

eg:- ① LABEL EQU 0500H — 0500H is stored to LABEL

② assign constant 0500H to LABEL

ADDITION : EQU ADD

assign label

ADD area instruction is defined as ADDITION with mnemonic ADD

(10) EXTRN : external and public : EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules.

eg:-

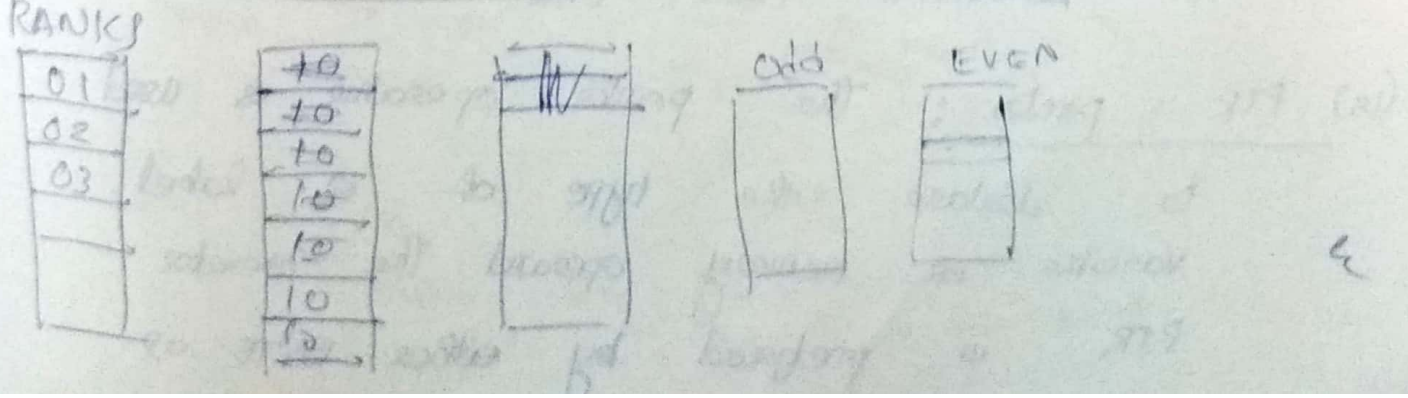
MODULE 1	SEGMENT
PUBLIC	FACTORIAL (1st declaration)
MODULE 1	ENDS
MODULE 2	SEGMENT
EXTRN	FACTORIAL
MODULE 2	ENDS.

used to inform the assembler that the variable had already been declared before.

(11) LABEL : label : The label directive is used to assign a name to the current content of the location counter. The type of the label must be specified, i.e, NEAR or FAR, BYTE or WORD, etc.

eg:

DATA	SEGMENT
DATAS	DB 50H DUP (?)



DATA_LAST LABEL BYTE FAR

DATA ENDS

(12) OFFSET : offset of a label.

used to get the offset of a variable

The directive is used to compute the offset address of the label.

eg:- MOV SI, OFFSET LIST. → LIST variable/array ka offset value is loaded to SI

offset address of label "LIST" is moved to SI.

(13) ORG : origin : It directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement.

eg:- ORG 02001H

(14) PROC : procedure : The PROC directive marks the start of a program named procedure in the statement. Also types NEAR or FAR specify the type of the procedure

PROC = Subroutine

eg:- RESULT PROC NEAR → = in the same segment
(RESULT → name of subroutine)

ROUTINE PROC FAR → = in a diff. segment
(ROUTINE → name of subroutine)

(15) PTR : pointer : The pointer operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or

WORD.

eg:- MOV AL, BYTE PTR [SI] → a byte value stored in the address pointed by SI is moved to AL

moves content of memory location addressed by SI (8 bits) to AL.

→ a word (2 locations) value stored in the address pointed by 2000H is loaded to BX

MOV BX, WORD PTR [2000H]

Move 16 bit contents of memory locations 2000H to BX, i.e, [2000] to BL, [2001H] to BH.

(16) SEG : Segment of a label : The SEG operator is used to decide the segment address of the label, variable or procedure and substitutes the segment base address in place of "SEG" label.

Q. Finding the largest data in an array.

SI points beg of list
DI points to the location where the largest data is to be stored.

CX/CL always holds the count.

```
MOV  SI, 2000H
MOV  DI, 3000H
MOV  CL, [SI]      → content of SI → 05 is moved to CL
INC  SI            → SI is incremented to point to the 1st val
MOV  AL, [SI]
DEC  CL            (CL--)
```

Step 1: INC SI (SI++)

```
MOV  BL, [SI]
CMP  AL, BL        → AL - BL ⇒ C=0
```

Jump if No Carry → JNC Step 2

MOV AL, BL (the largest value is moved to AL)

Step 2 : DEC CL (CL^{--})

JNEZ step 1 if (CL ≠ 0, move to step 1)

Jump if Non zero (CL?)

when CL = 0 ⇒ MOV [DI], AL ← final largest value is stored to DI

HLT

HLT

SI

2000	05
2001	75
2002	66
2003	98
2004	99
2005	22

DI

3000	99

Q. Verify if "Block 1" is a palindrome. If yes make "Pal = 1" in Data Segment.

storing instore always b/w DS & ES (ES holds the copy of DS)

ASSUME CS : Code , DS : Data , ES : extra

Data SEGMENT (storing the data in DS)

Block 1 DB " ABCDEEDCBA " (define byte)

Pal DB 00H Pal = 0

Data ENDS

extra SEGMENT (declaring necessary values in extra seg)

Block 2 DB 10 DUP (?)

= size of Block 2.

extra ENDS

Code SEGMENT (where the code is written)

Start : (code begins)

MOV AX, Data } the base addr of Block 1 in Data seg is moved to DS

MOV DS, AX }

MOV AX, Extra } the base addr of Block 2 in Extra seg is moved to ES

MOV ES, AX }

DS

A	
B	
C	
D	
E	
E	
D	
C	
B	
A	

ES

A	
B	
C	
D	
E	
E	
A	

DS = 0

If

LEA → Load Effective Addr

```
LEA   SI , Block 1.   → Loading the starting addr of Block 1 to SI.
LEA   DI , Block 2 + 9.   → Loading the last addr of Block 2 to DI
MOV   CX , 000AH   ← CX = 10 (count)
Back :  CLD  (clear Direction Flag) ⇒ D = 0 (INC)
        LODSB (Load storing byte where SI pointing to the accumulator)
        STD  (Set Direction flag) ⇒ D = 1 (DEC)
        STOSB (store the value stored in accl to the location pointed by DI)
        LOOP Back (still CX=0)
        LEA   SI , Block 1  (starting addr of Block 1)
        LEA   DI , Block 2  (starting addr of Block 2)
        MOV   CX , 000AH   ← CX = 10 (count)
        CLD  ( D = 0 ⇒ INC)
        REPZ  CMPSB  (Compare the value in [SI] - [DI] SI & DI)   ← Repeat if zero
        JNZ   skip   (If zero, repeat.)   ← Jump if Non zero
        MOV   Pal , 01H → Pal = 1; ⇒ Palindrome
skip :  INT 3  (Interrupt 3 = goes to the blank terminal)
```

D → Direction flag

Copies the Block 1 in DS to Block 2 in ES in the reversed order

Code ENDS → code segment ends

END → end of program.

Q. Addition of 2 16 bit datas.

```
ASSUME   CS : CODE , DS : DATA
DATA     SEGMENT
OPR 1    DW   1234H
OPR 2    DW   0002H
RESULT   DW   01H   DUP (?)
                    1 location with a random value
DATA     ENDS
```

```
CODE        SEGMENT
START       MOV   AX, DATA   } base addr of DATA
            MOV   DS, AX     } to DS
            MOV   AX, OPR1   } AX = OPR1
            MOV   BX, OPR2   } BX = OPR2
            CLC
            ADD   AX, BX  ---> AX + BX ---> AX
            MOV   DI, OFFSET RESULT
            MOV   [DI], AX
            MOV   AH, 4H
            INT   21H
            INT3  ---> back to terminal
CODE        ENDS
            END   START.
```

offset value of RESULT is loaded to DI

Q. Program to find out the no. of even & odd no.s from a given series of 16 bit hexadecimal no.s

```
ASSUME  CS : CODE , DS : DATA
DATA    SEGMENT
LIST    DW   235FH, 0A579H, 0C322H, 0C91EH,
        (creating a list of numbers)  0C000H, 0957H.
COUNT   EQU  006H  ---> COUNT = 006H => (6 numbers)
DATA    ENDS
CODE    SEGMENT
START   : XOR  BX, BX  ---> To clear BX
                            BX XOR BX ---> BX = 0
                            (same values XOR gives 0)
```

```
        XOR   DX, DX          → to clear DX
CL=6    MOV   AX, DATA  } base addr of DATA seg
        MOV   DS, AX    } is stored to DS
        MOV   CL, COUNT       → CL = COUNT = 6
        MOV   SI, OFFSET LIST → starting addr of LIST is moved to SI
AGAIN : MOV   AX, [SI]   (1st no.)
        ROR   AX, 01     → Rotate once
        JC    ODD
        INC   BX (BX++)   → else
        JMP   NEXT
ODD   : INC   DX (DX++)
NEXT  : ADD   SI, 02 (SI+2 → to next loc since it is a word)
        DEC   CL (CL--)
        JNZ   AGAIN  (till CL=0)
        MOV   AH, 4CH
        INT   21H
        INT3          → back to terminal
CODE    ENDS
        END   START.
```

DS / SI: 2357H, 0A579H, 0C322H, 0C91EH, 0C000H, 0957H

value rotated right is stored in the CF

JC — Jump if Carry (if CF = 1)

JNZ — Jump if Non Zero

even 1000 →ROR 0100 (CF 0)
odd 1001 →ROR 1100 (CF 1)

Q. Write a program to SORT a series of 10 numbers from 20000H in ascending order.

20000H = EA
∴ DS × 10H + OFA = 20000H
DS = 2000H (base addr)
OFA = 0H

```
        ASSUME CS : CODE
CODE    SEGMENT
START : MOV   AX, 2000H
        MOV   DS, AX
        MOV   CH, 09H     → CH ← pass = 9
Back 2: MOV   SI, 0000H   → SI points to the 0th location
        MOV   CL, 09H     → CL ← comparison = 9
Back 1: MOV   AX, [SI]    ; 16 bit
        CMP   AH, AL
```

In the DATA SEG, the values are defined as words (16 bits)

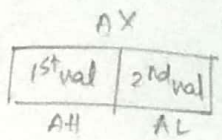
5	4	3	2	1
4	3	2	1	2
3	2	1	3	3
2	1	4	4	4
1	5	5	5	5

CL=4
Comp – CL=0 CL=0 CL=0 CL=0
Pass – CH=3 CH=2 CH=1 CH=0

2000×10 + 0000

AH | AL: 2nd val | 1st val (AX)

```
Jump if No Carry ← JNC   SKIP  ——→ else
i.e, (AH > AL)
2nd val > 1st val
                 XCHG  AL, AH   (
                 MOV   [SI], AX
                 INC   SI   (SI++)
                 DEC   CL   (CL--, decrease Comparison)
Jump if Non Zero, i.e ← JNZ   Back 1.
CL ≠ 0
Skip :           DEC   CH   (no exchange, CH--)
                            (pass--)
Jump if Non Zero ← JNZ   Back 2
i.e, CH ≠ 0
                 INT3 ——→ back to terminal.
                 CODE ENDS
                 END.
```



* Passing Parameters to Procedure

Procedure or subroutines may require input data or constants for the execution. These data or constants may be passed to the subroutine by the main program or some subroutine may access readily available data of constants available in memory.

The technique used in passing input data / parameter to procedure in assembly language programs are:

(i) using global declared variables
(ii) using registers of CPU architecture.
(iii) using memory locations.
(iv) using stack.

(v) using PUBLIC and EXTRN

(i) Using global declared variable :

eg:-

```
ASSUME   CS : CODE1, DS : DATA
DATA     SEGMENT
NUMBER   EQU  44H   GLOBAL
DATA     ENDS
CODE1    SEGMENT
START    :  MOV  AX, DATA
            MOV  DS, AX
            :
            MOV  AX, NUMBER
            :
CODE1    ENDS

ASSUME   CS : CODE2
CODE2    SEGMENT
         MOV  AX, DATA
         MOV  DS, AX
         MOV  BX, NUMBER
CODE2    ENDS
         END  START.
```

(ii) Using general purpose registers :

CPU registers can be used to store the parameters to be passed to the procedure in the available CPU registers and procedure may use the same register content for execution.

```
ASSUME      CS :  CODE
CODE        SEGMENT
START :     MOV   AX , 5555 H
            MOV   BX , f2f2H
              .
              .
            CALL  PROCEDURE 1
              .
              .
PROCEDURE 1   NEAR
              .
              .
            ADD   AX, BX
            RET
PROCEDURE 1   END P
CODE        ENDS
            END   START.
```

(iii) Memory location may also be used to pass parameters to the procedure in the same way as registers.

The main program may store the parameter to be passed to a procedure at a known memory address location and the procedure may use the same location for accessing the parameters.

eg.

```
ASSUME    CS : CODE ,  DS : DATA
DATA      SEGMENT
          NUM    DB    (55H)
          COUNT  EQU   10H
DATA      ENDS
CODE      SEGMENT                          .1000
START :                                    1001
                                           0100
                                           0100
                                           1100
          CALL  ROUTINE
          .
          .
          .
PROCEDURE ROUTINE  NEAR.
          MOV   BX , NUM
          MOV   CX , COUNT
          .
          .
ROUTINE   ENDP
CODE      ENDS
          END  START.
```

(iv) Stack memory can also be used to pass parameters to a procedure. A main program may store the parameters to be passed to a procedure in its CPU registers. The registers will further be pushed onto the stack. The procedure during its execution pops back the appropriate parameters as and when required.

eg:-

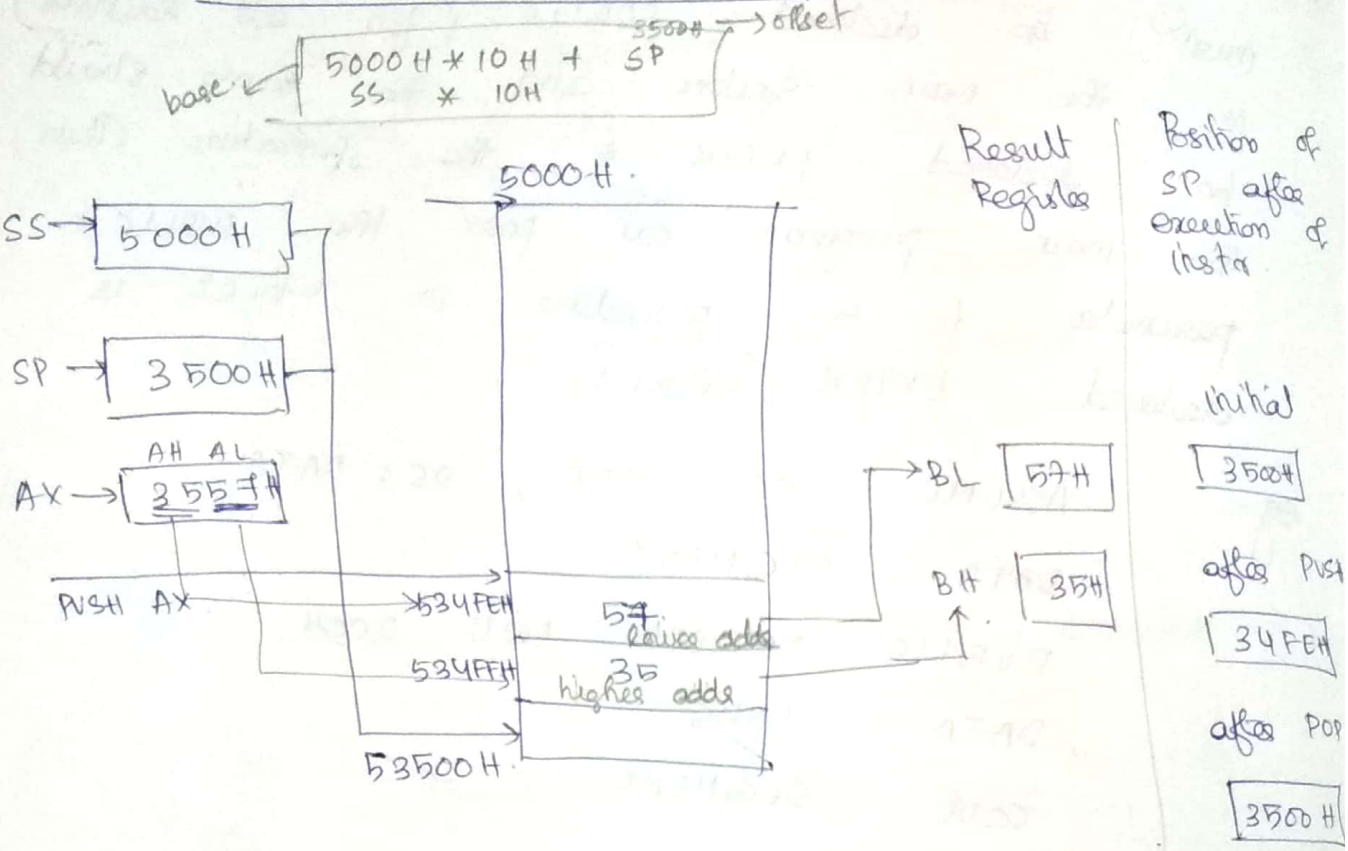
```
ASSUME    CS : CODE , SS : STACK
CODE      SEGMENT
START :   MOV   AX , STACK
          MOV   SS , AX
          MOV   AX , 557FH
          MOV   BX , 2929H
          :
          PUSH  AX
          PUSH  BX
          CALL  ROUTINE ; decrements SP by 2
          :
PROCEDURE ROUTINE NEAR
          :
          MOV   DX , SP
          ADD   SP , 02
          POP   BX  :  the data is passed
          POP   AX  :  into BX , AX
          MOV   SP , DX
          :
STACK     SEGMENT
STACK     DATA  DB  200H  DUP (?)
STACK     ENDS.
```

(v) Using PUBLIC and EXTRN directives. The variable must be declared PUBLIC (for all routines) in the main routine and the same should be declared EXTRN in the procedure. Thus, the main program can pass the PUBLIC parameter to a procedure in which is declared EXTRN (external).

eg:-

```
ASSUME    CS : CODE , DS : DATA
DATA      SEGMENT
PUBLIC    NUMBER  EQU  2004H
DATA      ENDS
CODE      SEGMENT
START :   MOV   AX , DATA
          MOV   DS , AX
          :
          CALL  ROUTINE
          :
          PROCEDURE  ROUTINE  NEAR.
          EXTRN  NUMBER
          MOV  AX , NUMBER
          :
          ROUTINE  ENDP
```

* Stack Structure of 8086 / 8088



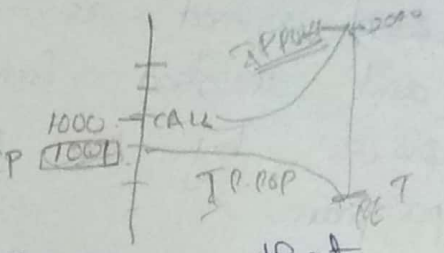
The stack contains a set of sequentially arranged data bytes, with the last item appearing on the top of the stack. This item will be popped off the stack first for use by the CPU.

→ The stack pointer is a 16 bit register that contains the offset of the address that lies in the stack segment.

→ The stack segment like any segment may have a memory block of a maximum of 64 KB locations and thus may overlap with any other segment

→ The stack segment Registers (SS) and the stack pointer (SP) register contains the base/

segment address and the offset address respectively.



* Programming with Subroutines :

Q. Write a procedure named SQUARE that squares the contents of BL & places the result in BX.

Subroutine : SQUARE

Description : (BX) = Square of (BL)

```
SQUARE  PROC  NEAR      ; (name of subroutine, intra or inter segment)
        PUSH  AX        ; Save the regsters to be used.
        MOV   AX, BX    ; place the number in AL
        IMUL  BL        ; multiply BL with (Accumulator for mul & div one value is always in reg.)
        MOV   BX, AX    ; Save the result.
        POP   AX        ; restore the regsters used.
        RET
SQUARE  ENDP
```

(IMUL: Signed multiplication)

(the reg AX might have been used in the main prgm, so its value have to be pushed to a stack to save its value. ← PUSH AX)

(the value pushed is restored. ← POP AX)

* Difference b/w Procedure (Subroutine) & MACRO

Procedure (Subroutine)	Macro
(1) A procedure (subroutine/ function) is a set of instructions needed repeatedly	A macro is similar to a procedure but is not invoked by the main program. Instead, the macro

by the programme. It is stored as a subroutine and invoked from several places by the main program.

(2) A subroutine is invoked by a CALL instruction. and control returns by a RET instr.

(3) Reduces the size of the program

(4) Executes slowly as time is wasted to push and pop the return address in the stack.

(5) Depends on stack.

code is pasted into the main program, whereas the macro name is written in the main program

A macro is simply accessed by using its name. The entire macro code is pasted at the location by the assembler

Increases the size of the program.

Executes faster as return addr is not needed to be stored into the stack, hence push and pop is not needed.

Do not depend on the stack.