Instruction set of 8086 :-

An instruction is a basic command given to a µp to Perform a specific operation on the given data.

An instruction has 2-group of fields one is "opcode" which defines the operation to be performed by instruction. the other field called operand, specifies data on which the Computer has to perform the specified operation.

The Collection of instruction that a micro Processor is designed to execute is known as instruction set.

The 8086 µp instructions can be categorised in to the following types.

1.) Data transfer instructions
2.) Arithematic and logical instruction
3.) Branch instruction.
4.) String "
5.) Loop "
6.) Flag manipulation instructions
7.) Shift and Rotate "
8.) Machine Control "

1) Data Copy/Transfer instructions :- These types of instructions are used to transfer data from source operand to destination operand. All the Store, MOVE, LOAD, EXCHANGE, INPUT, and OUTPUT belongs to this Category.

2) Arithematic and logical instructions :- All the instructions Performing arithematic, logical increment, decrement, Compare and scan instructions belong to this.

3) Branch instructions:- These instructions transfer Control of Execution to the specified address. All the Call, Jump, interrupt and return instructions belong to this class.

4) Loop instructions:- If these instructions have REP Prefix with CX used as Count register, they can be used to implement unConditional loops. The Loop, LOOPNZ and LOOPZ instructions belong to this Category.

5) Machine Control instructions:- These instructions Control machine status. NOP, HLT, WAIT, LOCK instructions belong to this

6). Flag Manipulation instructions:- All instructions which directly affect the flag register, come under this group.
CLD, STD, CLI, STI.

7) Shift and Rotate instructions:- These instructions involve the bitwise shifting (or) Rotation in either direction with (or) with out Count in CX.

8) String instructions:- These instructions involve string manipulation operations like load, move, Scan, Compare, store etc;

(i) DATA transfer instruction:-

SYNTAX	ACTION PERFORMED:-
1) MOV Destination, Source	→ Data is Copied from source to Destination.
2) PUSH Source	→ Two bytes is Copied from source to stack
3) POP Destination	→ Two bytes is Copied from stack to destination.
4) PUSHF	→ Copies Flag Register on-to stack
5) POPF	→ Copies 2-bytes from top of stack to flagRegister
6) XCHG OPRecord 1, Operand 2	→ Exchange the data stored in Operand 1 & OPR2
7) XLAT	→ Translate a byte from one code to another.
8) LEA Register, Source	→ Loads the Effective Address from source to Register.
9) LDS Register, Memory	→ Loads two Consecutive words from memory to Register & DS.
10) LES Register, Memory	→ Loads two Consecutive words from memory to Register & ES.
11) IN AX, DX	→ one byte is Copied from Port specified in DX, to AL
12) OUT Port, AL	→ one byte from AL is Copied into specified Port.
13) OUT DX, AX	→ one byte from Ax is Copied into Port specified DX.
14) LAHF	→ Copy lower byte from flag register to AH.
15) SAHF	→ Copy the byte from AH to lower byte of flag register.

1) MOV instruction:-

(copied ←)

Syntax:- MOV Destination, Source

Action:- This instruction copies data from source to destination.

Source:- Immediate data / register / memory location.

Destination:- Register / memory location

Note:- (i) Size of destination & source should be same.
(ii) Destination cannot be IP register (or) DS register
(iii) Both the operands cannot be segment registers
(iv) " " " " memory locations

Flags:- NO flags are affected.

Eg:-
```
MOV DS, 1234H    ; invalid
MOV AX, 1234H    ; valid
MOV DS, AX       ; valid
MOV AL, BX       ; invalid
```

```
MOV DS, ES       ; invalid
MOV IP, AX       ; invalid
MOV [BX] [SI]    ; invalid
MOV 1234H, BX    ; invalid
MOV [BX], AL     ; valid.
```

About Stack Pointer:- (SP register)

→ A stack is a Last In First out read/write memory. The items that go in last will come out first. This is because all read (POP) and write (PUSH) operations will takes place from one end called top of the stack (TOS).

→ The SP always contains the memory address of the last byte of the currently PUshed item on TOS ie it always points to the TOS. Stack is normally used by subroutines (or) interrupts for saving certain registers such as the Program Counter and status register.

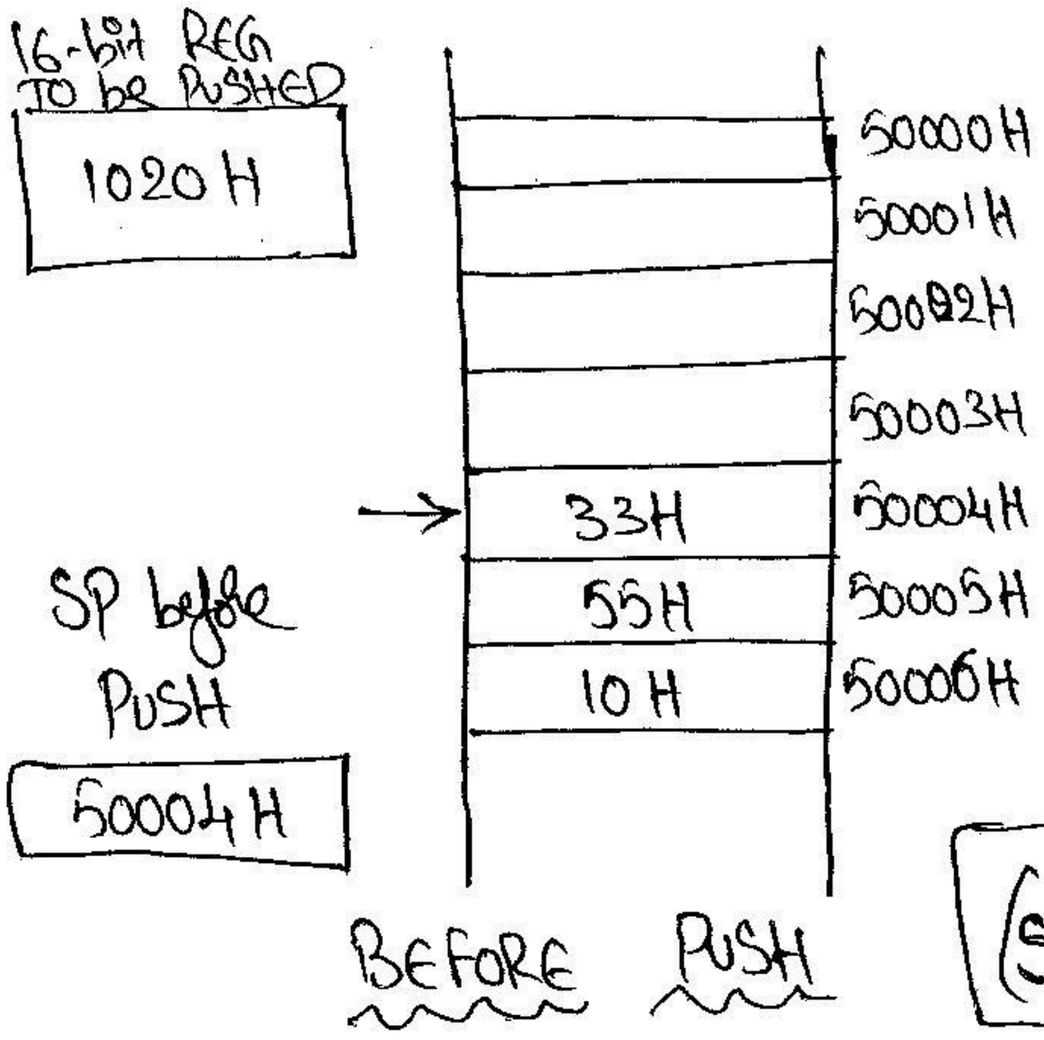
2) PUSH and POP operations:-

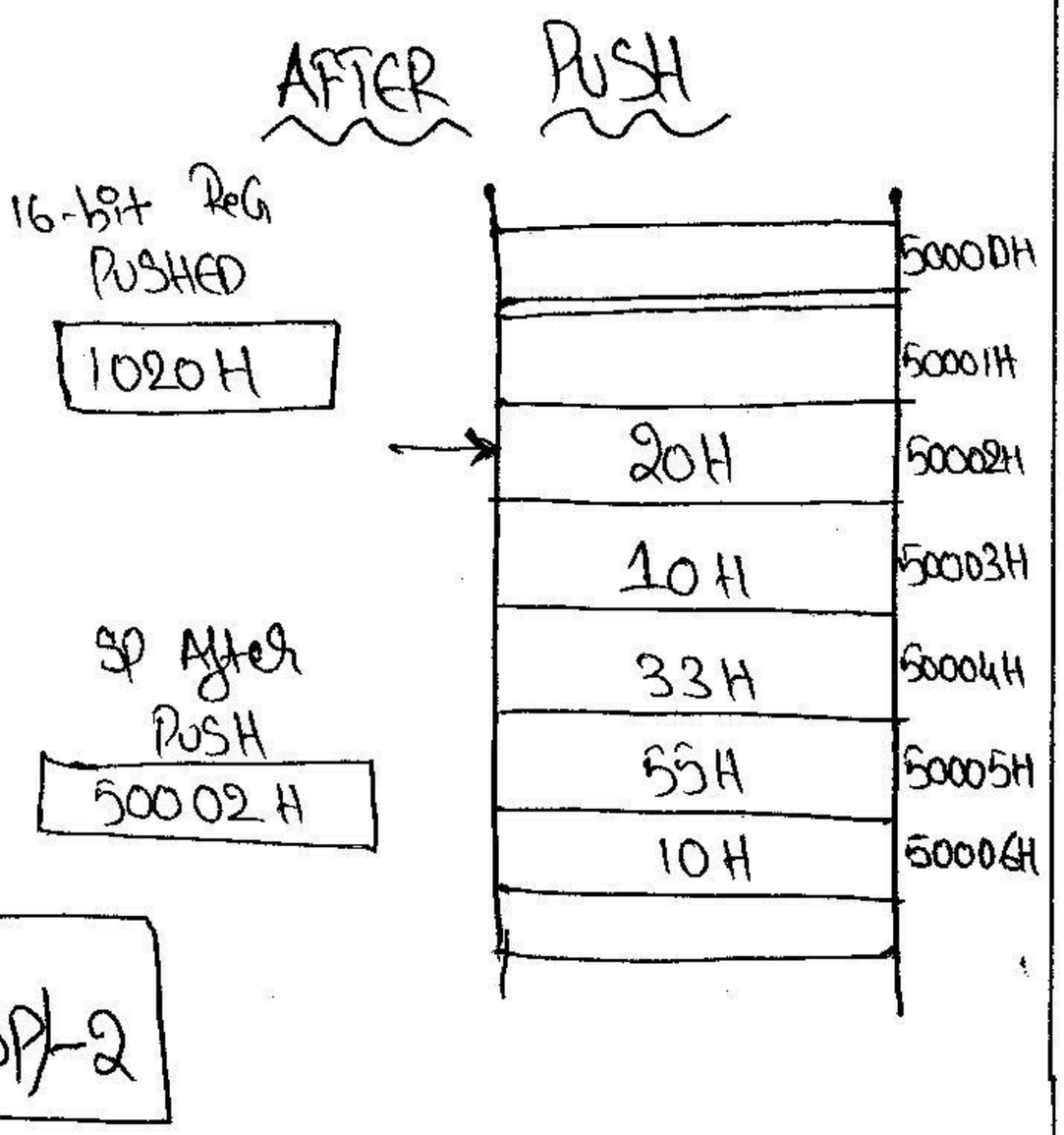
* → If the stack is accessed from the top, the stack pointer is decremented after a PUSH operation and incremented before POP.

*) → On the other hand, if the stack is accessed from the bottom, SP is incremented after a PUSH and decremented after POP.

*) → Typical microprocessors access stack from top.

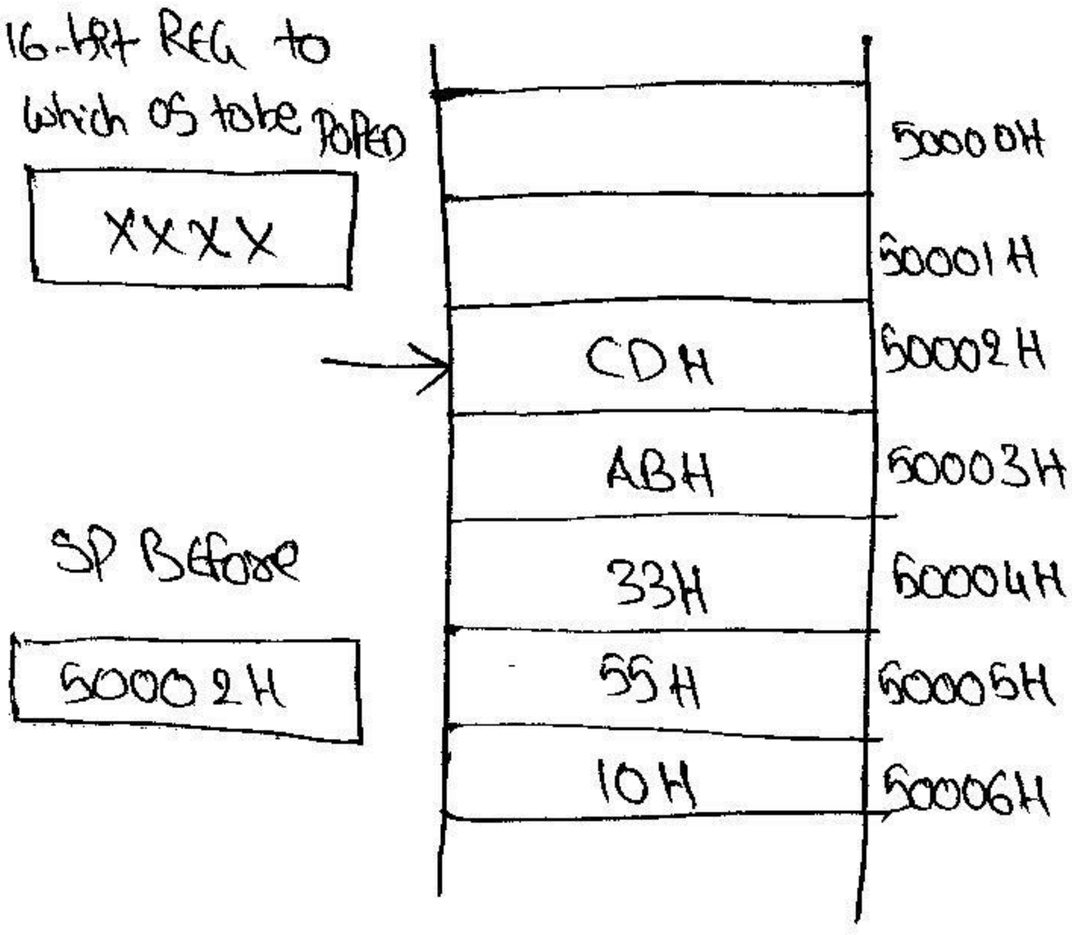
Eg:- In 8086 MP PUSH and POP operations can be done only on 16-bit data. Hence SP is incremented (or) decremented by value of 2-always.



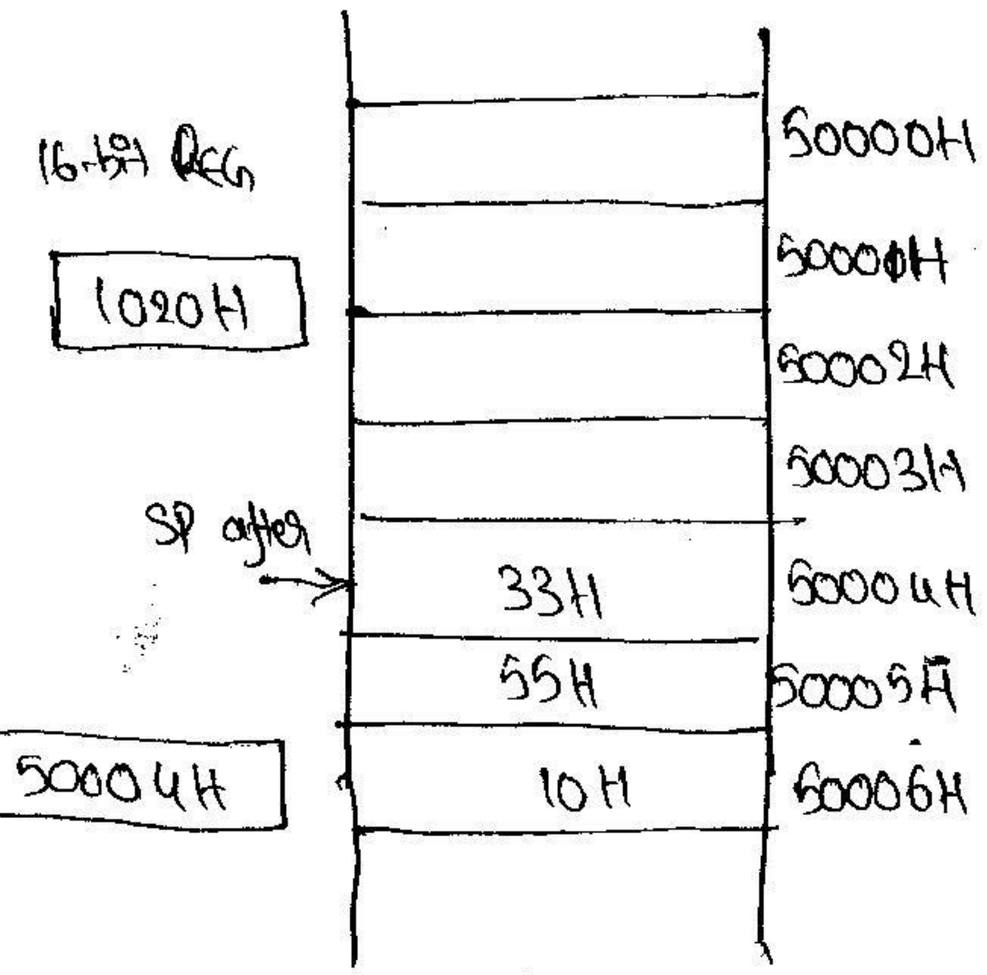


(SP) = (SP) - 2

POP:-



(SP) = (SP) + 2



Syntax:- *) PUSH REG (Push the Content on to Stack top)

Eg:- PUSH BX

*) PUSH SREG

Eg:- PUSH DS

*) PUSH Memory

Eg:- PUSH [5000H]

The instruction Pushes the contents of specified register/memory location on the stack. The Stack Pointer is decremented by '2' after each execution of instruction.

No Flags are Affected here.

POP (POP from Stack):- This instruction when executed loads the specified reg/memory location with the Content from the stack pointed by Stack Pointer. The Stack Pointer is automatically incremented by 2 after this instruction execution.

Flags:- No flags affected here.

Syntax:- *) POP REG (POP-off the Contents from top of Stack)

Eg:- POP AX

*) POP SREG

Eg:- POP DS

*) POP Memory

Eg:- POP [5000H].

---

3) XCHG (Exchange):-

Here the Contents of operand 1, & Operand 2 are Exchanged, the Contents of specified Source and destination operands which may be register (or) one may be memory location.

However, Exchange of data Contents of two memory locations is not Permited.

Eg:- (*) XCHG BX, 1234H [BX+SI] ; It is also written as XCHG 1234H [BX+SI], BX. The Contents of DS, whose offset address is given by EA = 1234H + BX + SI to be exchanged with "BX" Reg.

(*) XCHG BX

(*) XCHG [5600H], AX

Flags :- No flags affected.

---

4) I/O (Input/output) operations :-

The I/O operations are performed through I/o ports. The I/O devices are connected through I/o ports so as to have synchronization with fast MP & slow I/o devices.

The I/o Port has 8-I/p (or) 8 o/p Pins through which data can be transmitted (or) received. At any one instance of time only one byte of data can be transmitted (or) received.

*) IN Accumulator :- The IN instruction will copy data from a Port to the AL (or) Ax Register

Syntax :- IN AL, Port no:- (or) IN Ax, Port no:-

Eg:- IN AL, 0300H ; Here 0300H is address of 8-bit Port
IN AX, 0034H ; Here 34H is address of 16-bit "
IN AX ; Here 16-bit Port address is in Dx Reg.

*) OUT (output to the Port) :- This instruction is used for writing to an o/p Port. It copies a byte from AL (or) a word from Ax to the specified Port.
The address of o/p Port specified in the instruction directly (or) implicitly in Dx.

Eg:- Syntax :- OUT Port no:, AL (or) OUT Port no:-, Ax

Eg:- OUT 03H, AL ; This sends AL data to Port address is 03H
OUT DX, Ax ; (or) OUT DX, AL.

5) PUSHF :- (Push flags to Stack) :-

The Push flag instruction pushes the flag register on to the stack ; first the upper byte and then the lower byte is Pushed on to it. The SP is decremented by 2, for each Push operation. The general operation of this instruction is similar to PUSH operation.

6) POPF : Pop flags from stack :-

The POP flags instruction loads the flag register Completly (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by '2' for each POP operation.

7) LAHF (Load 'AH' from lower byte of flag) :-

This instruction loads the 'AH' Reg by the lower byte of the flag register which Contains the status flags.

i.e Carry, Parity, auxiliary, Carry, zero, & Sign flags.

By this instruction we can see the contents of the status flags.

8) SAHF (Store 'AH' to lower byte of flag Register) :-

Stores 'AH' Reg in the lower byte of the flag register which Contains the status flags.

i.e Carry, Parity, auxiliary, Carry, zero & Sign flags.

This instructions sets and Resets the Conditions Code flags (except overflow)

9) LEA (Load Effective Address) :-

Effective address of the source operand is Computed & this effective address is loaded into a 16-bit register.

Flag :- No flag is affected.

Note :- EA represents the displacement (or) offset of the desired operand from the segment base

Syntax :- LEA register, Memory

Eg:- [illegible]

(*) LEA DS, 56H[SI]; invalid ∵ improper use of segment reg, Source can not be segment.

(*) LEA BX, 1234H ; valid.

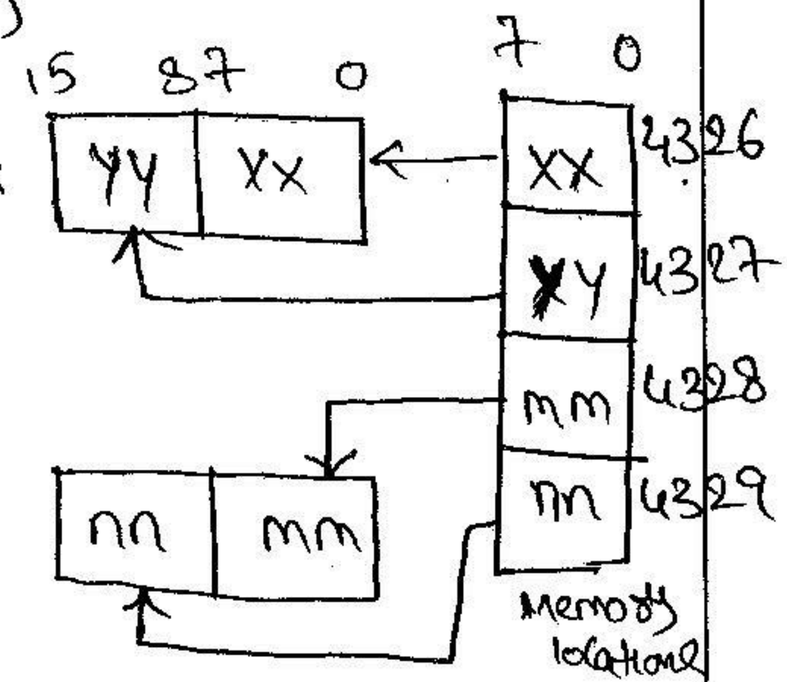
(*) LEA CX, [BX+DI]

---

10) LDS :- This instruction copies a word from two memory locations into the register specified in the instruction. and then copies a word next two memory locations in to the DS register.

Syntax:- LDS REG, Memory

Eg:- LDS BX, 5000H
LDS BX, [4326]



11) LES :- Same as LDS explanation except that replaces DS by ES. This instruction is used for re-initializing the ES segment

Eg:- LES BX, [789A]
LES DI, [BX]

---

12) XLAT :- This instruction is useful to convert from one code to another (translation) using look up table.

To execute this instruction, BX is initialized to contain the start address of the loop up table.

The instruction get a byte from a location pointed by [BX+AL] and is transferred to AL register i.e

(AL) ← DS : [(BX) + (AL)] //.

❖ ARITHMETIC INSTRUCTIONS

❑ ADD Add

Operands:- REG,MEMORY => ADD AX,0100H
Memory,REG
REG,REG => ADD AX,BX
memory , immediate => ADD AX,[1000H]
immediate, Memory

The content of the both the operands are added & the result is stored in destination.

Eg: ADD AX,Y ; AX=AX+Y

❑ ADC Add With Carry

Operands REG,MEMORY => ADC AX,0100H
memory,REG =>
REG,REG => ADC CL,BL
memory , immediate => ADC AX,[0300H]
immediate,Memory

The instructions adds the source operands to destination operation along with carry flags& the result is stored in destination operand.

Eg: MOV AX,1234H
ADC BX,FFDBH

❑ SUB SUBTRACT

Operands REG,MEMORY => SUB AX,0100H
memory,REG => SUB 0100,AX
REG,REG => SUB AX,BX
memory , immediate => SUB AX,[1000H]
immediate,Memory => SUB [5000H] , 0100

The instructions SUBTRACTS the source operands to destination operation and result is stored in destination operand.

Eg: MOV AL,05H
SUB AL,02H ; AL=AL-02H

❑ SBB SUBTRACT With BARROW

Operands REG,MEMORY => SBB AX,0100H
memory,REG => SBB 0100,AX
REG,REG => SBB AX,BX
memory , immediate => SBB AX,[1000H]
immediate,Memory => SBB [5000H],0100

The instructions adds the source operands to destination operation along with the values of carry flags& the result is stored in destination operand.

Eg: MOV AX,5678H
SBB AX,4321H

Dest operand = [(Dest operand) – (src operand) – (carry flag)]

Unsigned:- (++ (8) ~~xxx~~)

Result >FF (8) $(255)_{dec.}$ then CF=1

Signed :- (+ (8) -)

i.e -128 & +127
small large

Starting bit if 1-> -ve, 0-> +ve } -> in terms of binary 1 bit is 0 (8) 1

MSB LSB

Hexa to Binary format { 0 - Positive
1- '-ve' -> 2 Complent

X + (-Y)

X (+ve) 0285 H =>

(-Y)(-ve) 9563 H =>

Unsigned operation:- If 2-operands are given then two terms are 'tve' so we can Perform Add, Sub, Multiply, Division.

Signed operation:- If 1 operand is +ve & another is -ve i.e from given operands Convert into binary if MSB of 1st bit is 1 -> i.e -ve so do 2's Complement unless changed to 'tve'.

0 -> i.e +ve

* If the result of Signed operation is '-ve' i.e 1st bit is '1' then again Perform 2's complement for the result etc.

- **INC INCREMENT**

Operands REG => INC AX

memory => INC 0100

INCREMENTS THE OPERAND BY '1'

This instruction increments the content of the specified register (or) memory location by '1'. All the condition code flags are affected except the CF (carry flag).

- **DEC DECREMENT**

Operands REG => DEC BX

memory => DEC 0100

DECRMENTS THE OPERAND BY '1'

This instruction decrements instructions subtracts 1 from the content of the specified register (or) memory location . All the condition code flags are affected except the CF (carry flag).

- **MUL MULTIPLICATION (UNSIGNED MUL)**

Operands REG => MUL BH ; (AX) ← AL * BH

memory => MUL 0100H ; (AX)← AX * 0100

The instruction is used for the multiplication of 2 unsigned numbers . The content of specified register (or) memory location can be multiplied by the contents of AL (8-bit) or (16-bit) reg .

The result is placed in AX (for 8-bit manuplation) or DX:AX (if 16-bit)

In case of 16-bit multiplication ,lower significant word is placed AX and higher significant word is placed in DX register .It will affect higher CF and OF

Eg:- 1. MUL BH ; (AX)← AL* BH

2. MUL CX,(DX)(AX)← (AX)*(CX)

3. MOV AL, 0FDH

MOV CL, 05H

MUL CL

- **IMUL (signed multiplication)**

The instruction is used for the multiplication of 2 signed numbers. The instruction multiplies a signed byte in source operand by a signed word in AX. The source can be REG (or) memory operand not immediate data.

The CF and AF are affected

Eg: 1. IMUL CL

2. IMUL CX

3. IMUL [SI]

4. MOV AL. -03H

MOV CL, -05H

IMUL CL

Binary Multiplication:- 2-8bit unsigned binary no:-

```
                    128 64 32 16 8 4 2 1
130   =             1  0  0  0  0 0 1 0
X 240 =             1  1  1  1  0 0 0 0
-------------------------------------------
                    0  0  0  0  0 0 0 0
                 0  0  0  0  0 0 0 0  x
              0  0  0  0  0 0 0 0  x  x
           0  0  0  0  0 0 0 0  x  x  x
        1  0  0  0  0 0 1 0  x  x  x  x
     1  0  0  0  0 0 1 0  x
  1  0  0  0  0 0 1 0  x
-------------------------------------------
  1  1  1  1  0 0 1 1  0  0  0  0  0
  |  |  |  |  |  |  |  |  |  |  |  |
  7     9     E     0
```

⇒ 79E0 H ⇒ 31,200

```
32
64
128
256
2048
4096
8192
16384
-------
31200
-------
```

Binary Division:-

$\frac{32780}{140} \Rightarrow$ 10001100) 0000000011101010 = EA H = 234 (Quotient)

```
1000000000001100
 10001100 1111
-----------------
 01110100 11111
  10001100 1111
-----------------
  01011000 1111
   10001100 111
-----------------
   00100101101
    10001100 1
-----------------
    00010100 1  = 14 H = 20 (Rem)
```

2) $\frac{10}{5} \Rightarrow$ 0101) 1010 = 2

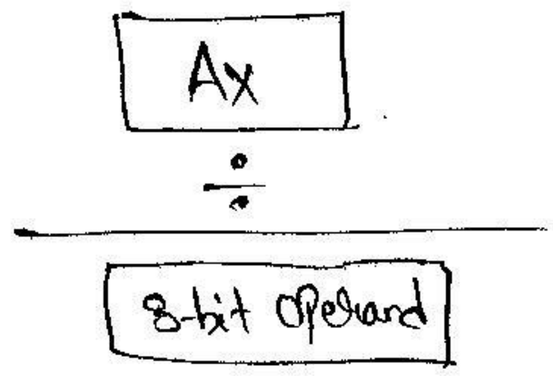
```
5 ↑   8 4 2 1 → 2
      0 0 1 0
    ---------
0101) 1 0 1 0
      1 0 1 0
    ---------
      0 0 0 0 → 0
```

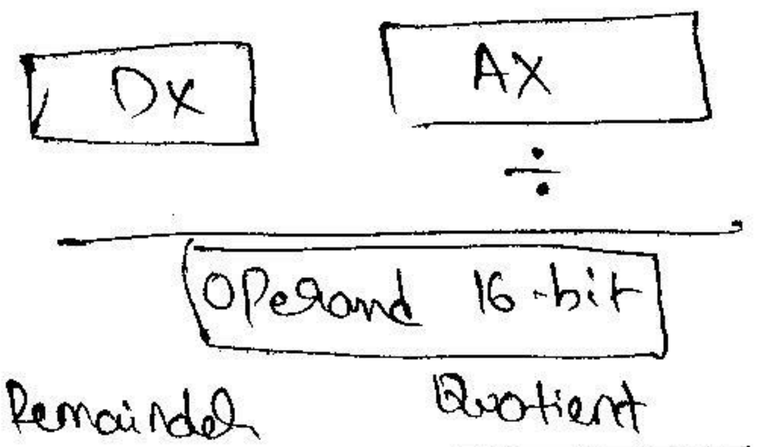
Division instructions :- (DIV)

DIV (unsigned division) :- The instruction Performs unsigned division.
It divides an unsigned word (or) double word by a 16-bit (or) 8-bit operand.
The divided must be in Ax for 16-bit operation and divisor must be specified using any one of addressing modes except immediate.

If the divisor is 16-bit wide, then the dividend is the DX:AX registed pair after the division the quotient will be stored in to AX and the remainder into DX.

If the divisor is 8-bits the dividend is Ax. and this case the quotient will be stored in AL & the remainder in AH.





All the flags are undefined.

Eg:-
```
MOV  AX , 00C8H
MOV  CL , 06 H
DIV  CL
```

	Quotient	Remainder
1) 2C58/56 =	84H	00H
2) 37D7/97 =	8E	65 H

IDIV (Signed DIVISON) :-

The instruction performs the same operation as the DIV instruction but with signed operands.
It is used to divided a 16-bit signed no. by an 8-bit signed no:- (or) to divided a 32-bit signed no:- by 16-bit signed no:-
no flags affected:

Eg:-
```
IDIV  CL
IDIV  BP
```

	Q	R
85112/0085 =	01000	0012

NEG (Negate) :-

This instruction Produces the two Complement of the Specified operand & Stored the result in the same operand. MP Performs the negate operation by subtracting the operand from '0'. This is done to represent a negative number. All the flag bits are modified.

Eg:- MOV AL, 15H
NEG AL

CMP (Compare) :-

This instruction Compares a byte (or) word from the specified Source with a byte (or) word from specified destination.

For Compalision it subtracts the source from destination operands but does not store the result any where, but the flags are changed.

If both operands are Equal, Zero flags is set,
If source is greater than destination CF = 1 (or) else CF = 0
Set Reset

Eg:- CMP AL, 01H
CMP BH, CL
CMP 0100
CMP [5000H], 0100H
Cmp BX, [SI].

CBW :- (Convert Signed byte to Signed word) :-
this instruction Copies the sign of a byte in AL to all the bits in AH. AH is said to be sign extention of AL.

The CBW operation must be done before a signed byte in AL can be divided by another signed byte with the IDIV instruction.

Eg:- AX = 00000000 10011011 = -155 decimal
CBW ; AX = 11111111 10011011 = -155 decimal.

CWD (Convert Signed word to Signed Double word):-

CWD Copies the sign bit of a word in AX to all the bits of the DX register. i.e it extends the sign of AX in to all of DX. The CWD operation must be done before Signed word in AX can be divided by another Signed word with the IDIV instruction.

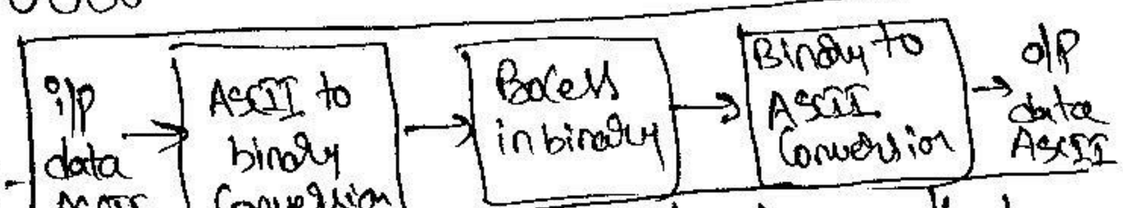
DX = 00 000000 00 000000

AX = 1111 0000 11 000111 = -3897 decimal

CWD ; DX : AX

1111 1111 11 11 1111 1111 0000 11 000111 = -3897 decimal.

AAA (ASCII Adjust after Addition):-



This instruction is executed after an Add instruction that adds two ASCII codes operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to ~~it~~ a unpacked decimal digits.

The numerical data entered in to the computer from a keyboard is usually in ASCII Code. The digits 0-9 are represented as 30H to 39H the 8086 MP allows to add the ASCII codes of two decimal digits without masking off the "3" Bcoz in the upper nibble of each. The AAA instruction is used to make sure the result is the correct unpacked BCD.

Eg:- 6 + 9 ⇒ ASCII it is 36 = 0011 0110

39 = 0011 1001

0110 1111 ⇒ 6F

∴ F = 15.

The AAA instruction is used to adjust the result in the AL register after the addition of two ASCII no:- the adjustments is done as follows.

(i) If the 4 LSB's of the AL register are between 0 and 9 and AF is zero, then 4 MSB's of AL register are cleared and 4 LSB's remain unaltered.

```
MOV AL, 6
MOV BL, 9
ADD BL, AL
AAA
```

(ii) If the 4 LSB's of AL register are b/w A and F (or) AF flag is one then 06 is added to AL register & is added to AH register, AF flag is set to '1' and 4 MSB's of AL are cleared.

b/w (0-9) means

Eg:- ① AL = 67 ⇒ AL = 07 (is cleared with zero)
Before AAA After AAA

Before AAA ⇒ 67 = 0110 0111 (MSB, LSB) ; AF = 0
After AAA ⇒ 07 = 0000 0111 ; AF = 0
MSB cleared ↓ remains same

Eg: ② AH = 00 & AL = 5A (A → b/w A & F) before AAA

here LSB's of AL are greater than 9. hence '06' is added to 'AL' reg.

5A
06

60

AF = '1'

& 4 MSB's are AL cleared to zero ∴ AH = 00

AH is incremented by '1'

AH = 01 ; AL = 00 ← After AAA.

Eg ③:- 6 and 8 addition.

36
38

6E

Step:-2:-
6E
06

74

Step:-3 AL = 04

Step 4:- AH is incremented by '1'.

∴ Result :- 0104 → which is unpacked BCD
14 → Packed BCD.

AAS (ASCII adjust AL after subtraction):-

This instruction is used to adjust the result in AL register after performing subtraction, when the operands are ASCII

Op:- The adjustments is done as follows.

(i) If the least significant (LS) Hex digit of AL register is less than (or) Equal to 9 and AF is '0', then the MS Hex digit of AL register is cleared (i.e made 0) and LS Hex digit is unaltered.

(ii) If the LS Hex digit is of AL register is greater than '9' (or) AF is '1', the following adjustments are made.

(a) 6 is subtracted from LS hex digit of AL register
(b) The 4 MSB's of AL reg are cleared.

(c) Contents of AH are decremented by '1'.

(d) Carry and Auxilary Carry flag are set to '1'.

ASCII 9 − ASCII 5

39
35 ∴ AL = 04
04

(2) ASCII 5 − ASCII 9

35 = 0011 0101
39 = 0011 1001
1111 1100 = FC //
sub 0110
0000 0110 => FF06 //
(cleared) (AH is decrement by '1')

AAM (ASCII Adjust after Multiplication):-

Before Multiply two ASCII digits first mask the upper 4-bits of each. This leaves unpacked BCD in each byte.

The AAM instruction is used to adjust the result in AL register after multiplication of two unpacked BCD nos. to give result in BCD the following steps are taken.

The AL register is divided by 10(0A hex). Quotient is stored in AH and the remainder is stored in AL.

Eg:- MOV AL, 3 ; → Multiplier in unpacked BCD form
MOV BL, 9 ; → " " " " "
MUL BL ; → Result 001B H is in AX here AH=00 ; AL=1B
AAM ; → AX = 0207 H
Q R
(or)
AX = 3030H ; AX = 3237 H.

Eg:2 MOV AL, '3' ; multiplier in ASCII
MOV BL, '9' ; Multiplicand in "
AND AL, 0FH ; multiplier in unpacked BCD form.
AND BL, 0FH ; " " " "
MUL BL ; Result 001BH is in AX
AAM ; AX = 0207 H
(or) AL, 30 H ; AL = 37 H.

0011 X 1001
0011
0000X
0000XX
0011X
001 1 0 1 1 = 1B

001B
16
8
2
1
27

10) 27 (2 → Q
20
7 → R

Eg:-3)
```
AL = 05
BH = 09
MUL BH
AAM.
```

AAD (ASCII Adjust before DIVISION):-

The AAD instruction Converts 2-unpacked BCD digits in AH & AL to the equivalent binary no:- in AL. This adjustment must be made before dividing the two unpacked BCD digits. in AX by an unpacked BCD byte.

The AAD instruction adjusts the numerator in AX before dividing two unpacked decimal no:-

The denominator is a single unpacked byte

The adjustment step are as follows

(i) AH is multiplied by 10 (0A H) and added to AL. and sets AH to 0.

Eg:- If AX is 0207 before then AX changed to 001B H after. AAD.

AAD instruction reverse the changes done by AAM.

Eg:- Divide 27 by 5

```
MOV AX, 0207 ; dividend in unpacked BCD form
MOV BL, 05 H ; divisor in unpacked form
AAD ; AX = 001BH
DIV BL ; AX = 0205 H ;//.
```

Step:-2
; Multiply AH with 10 => 02 X 10 = 20 Add to AL

Step:1
Eg: 0207 / 05 ; AH = 02, AL = 07

```
  20
  07
 ---
  27
```

Res

Step:3 27 5 => Q ; 5 → Q → AL ; 2 → R → AH

27 / 5 =

5) 27 (5 → Q
25
---
2 → R

❖ **Bit Manipulation Instructions**

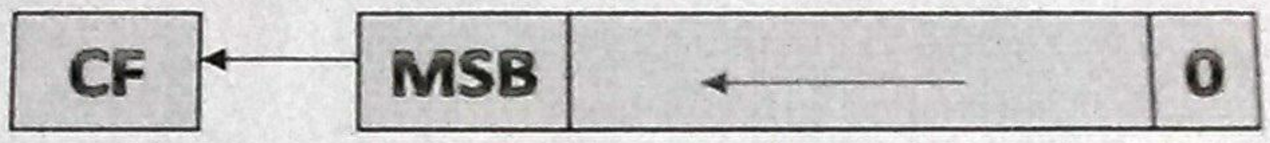
Shift instructions

❑ **SHL/SAL (shift logical /arithmetic instruction)**

- This instruction shifts each instruction in each destination

SHL <reg. / Mem>

CF ← R(MSB) ; R(n+1) ← R(n) ; R(LSD) ← 0



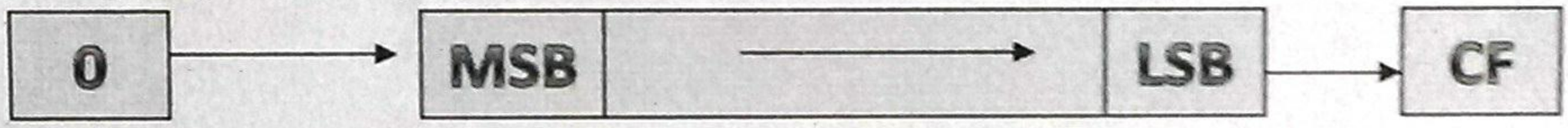
- These instructions shift the operand (word or byte) bit by bit to the left and insert zeros in the newly introduced least significant bits.
- The number of bits to be shifted if 1 will be specified in the instruction itself if the count is more than 1 then the count will be in CL register.
- The operand to be shifted can be either register or memory location contents but cannot be immediate data.

All the flags are affected depending upon the result.The shift operation will considering using carry flag.

❑ **SHR : Shift Logical Right**

SHR <reg. / Mem>

0 →L(MSB) ; R(n+1) → R(n) ; R(LSB) →CF

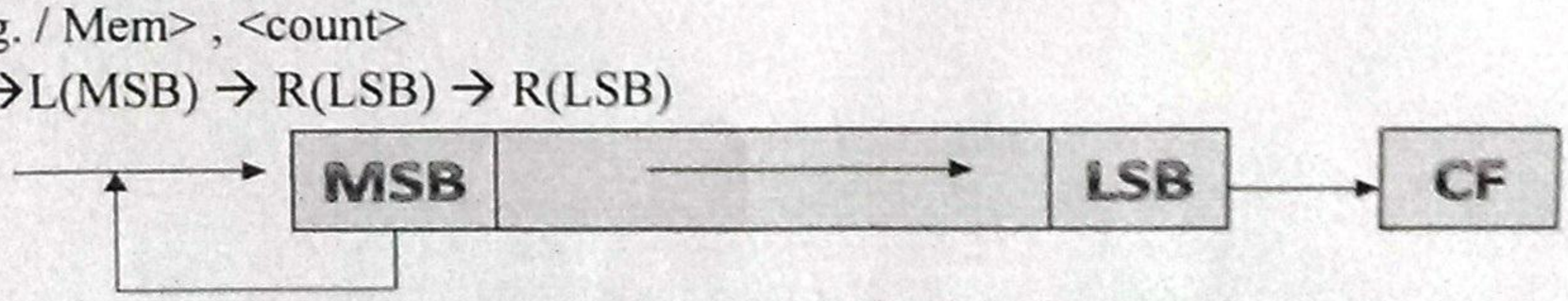


- These instructions shift the operand word or byte bit by bit to the right and insert zeros in the newly introduced Most significant bits.
- The result of the shift operation will be stored in the register itself.
- The number of bits to be shifted if 1 will be specified in the instruction itself if the count is more than 1 then the count will be in CL register.
- The operand to be shifted can be either register or memory location contents but cannot be immediate data.
- All the flags are affected depending upon the result.The shift operation will considering using carry flag.

❑ **SAR : Shift Logical Right**

SAR <reg. / Mem> , <count>

L(MSB) →L(MSB) → R(LSB) → R(LSB)

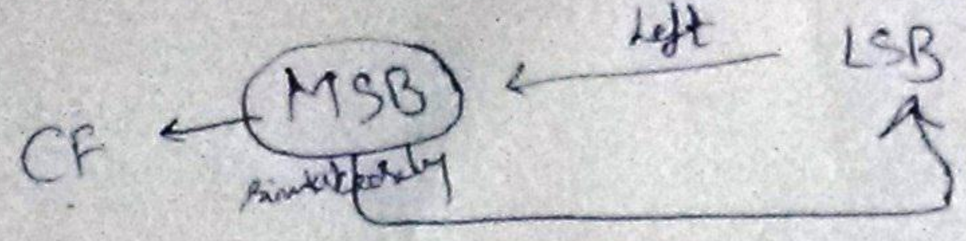


- These instructions shift the operand word or byte bit by bit to the right.

SAR instruction inserts the most significant bit of the operand in the newly inserted bit positions.

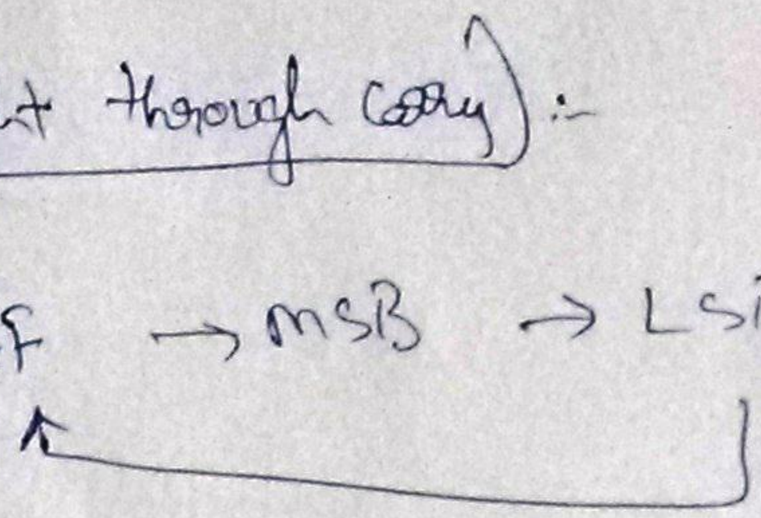
- The result will be stored in the register or memory itself.

ROL (Rotate left without carry):-



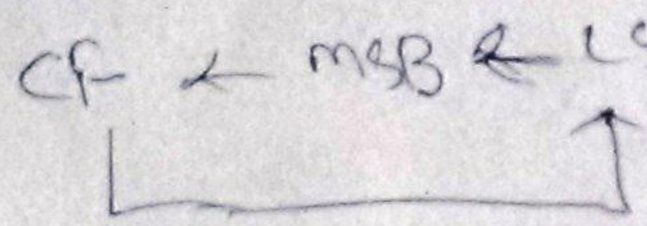
Eg:- ROL AX,1 ; If BH = 10101110
ROL BL,CL ROL BH,1
CF = 1 ; BH = 01011101
LSB
OF = 1.

RCR (Rotate Right through carry):-



RCR BX,1
RCR AX,CL

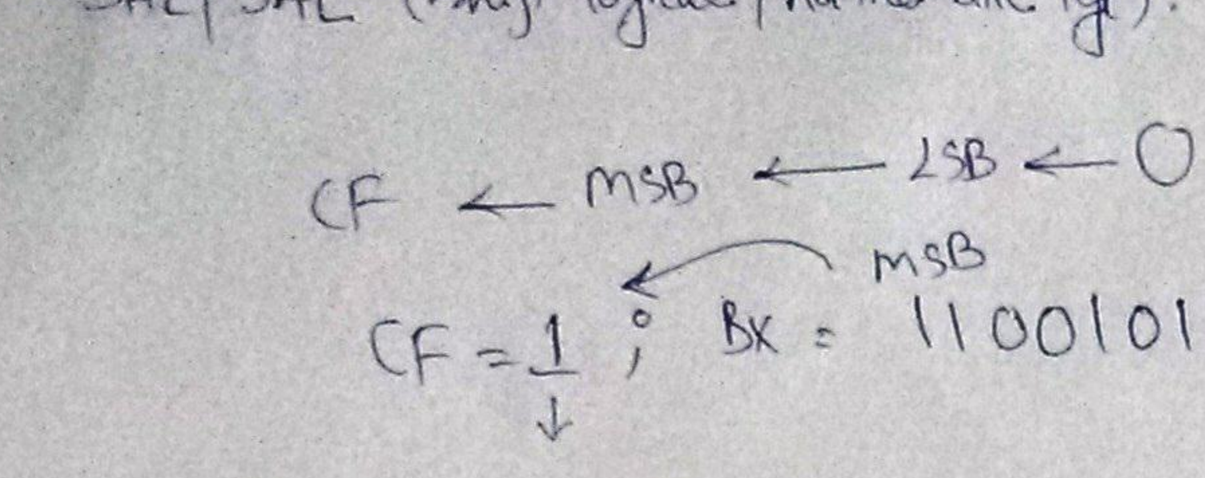
RCL (Rotate left through carry):-



RCL DX,1
RCL AX,CL

Shift instructions.

SHL/SAL (Shift logical/Arithmatic left):-



Eg:- BX = 11100101 11010011
SAL BX, 01

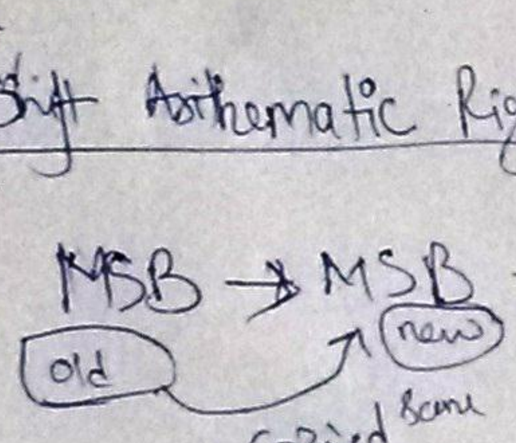
SHR (Shift logical Right):-

0 → MSB → LSB → CF

Before Eg:- SI = 10010011 10101101 ; CF = 0

After:- SHR SI, 1 ⇒ 01001001 11010110 CF = 1
MSB LSB

SAR (Shift Arithmatic Right):-



Before Eg:- AL = 00011101 = +29 decimal ; CF = 0
MSB LSB

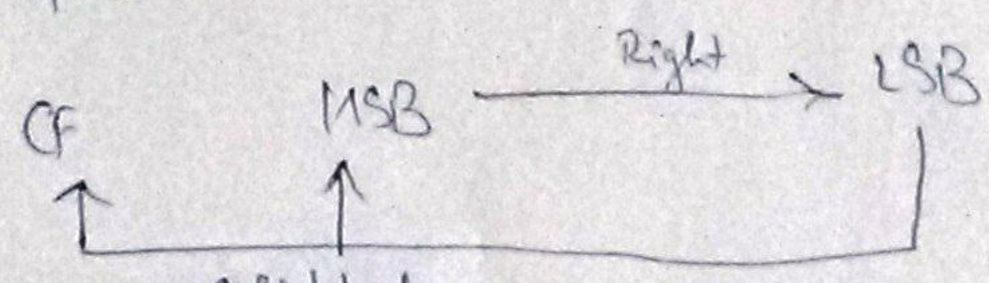
After:-
SAR AL, 1

AL = 00001110 = +14 decimal ; CF = 1
MSB
old new

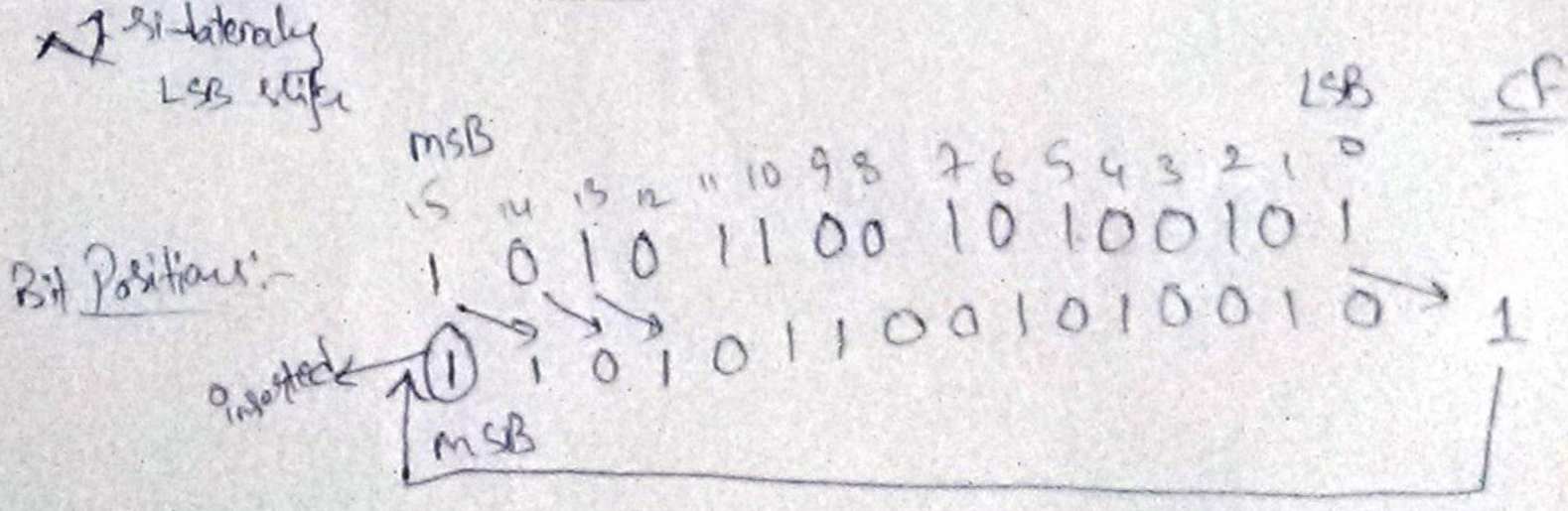
Rotate instructions:-

ROR (Rotate right with out carry):-

The LSB is pushed on to CF and similarly it is transferred into MSB position at each operation. The remaining bits are shifted Right.



Eg:- ROR BL, 01
ROR AL, CL



- The number of bits to be shifted if 1 will be specified in the instruction itself if the count is more than 1 then the count will be in CL register.
- The operand to be shifted can be either register or memory location contents but cannot be immediate data.
- All the flags are affected depending upon the result.The shift operation will considering using carry flag.

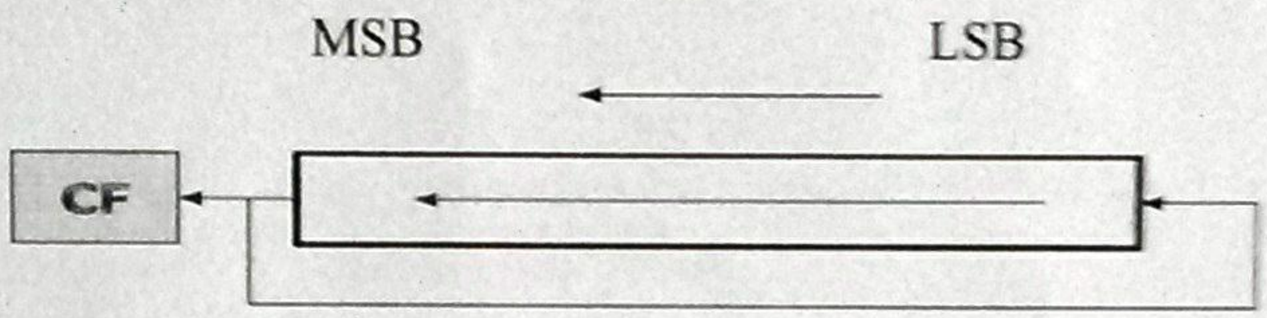
❖ **Rotate instructions**

ROL, RCL, ROR, RCR

❑ **ROL : Rotate left without carry**

ROL <Reg. / Mem> , <Count>

R(n+1) ← R(n) ; CF ← R(MSB) ; R(LSB) ← R(MSB)

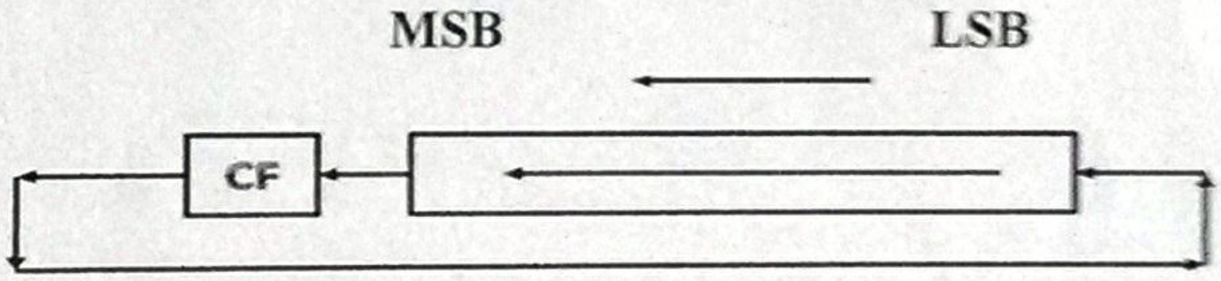


- This instruction rotates all the bits in a specified word or byte to the left by the specified count (bit-wise) excluding carry.
- The MSB is pushed into the carry flag as well as into LSB at each operation. The remaining bits are shifted left subsequently by the specified count positions.
- The PF, SF, and ZF flags are left unchanged in this rotate operation . The operand can be a register or a memory location.
- The count will be in instruction if it is 1, and in CL register if greater than 1.

❑ **RCL : Rotate left through i.e., with carry**

RCL <Reg. / Mem> , <Count>

R(n+1) ← R(n) ; CF ← R(MSB) ; R(LSB) ← CF

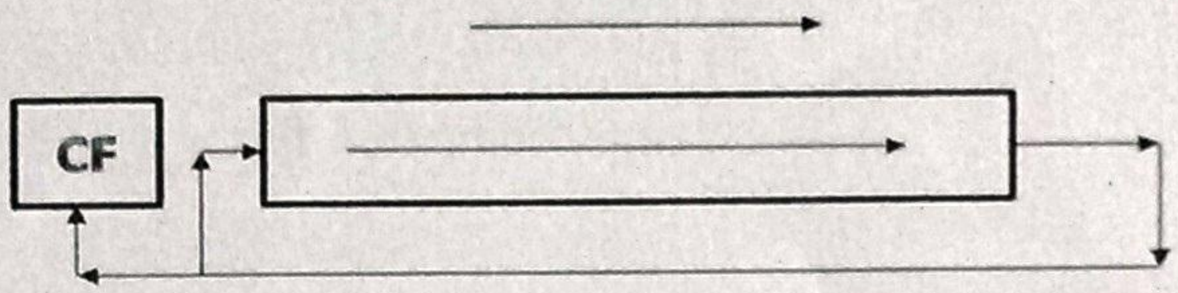


- This instruction rotates all the bits in a specified word or byte to the left by the specified count (bit-wise) including carry.
- The MSB is pushed into the CF and CF into LSB at each operation. The remaining bits are shifted left subsequently by the specified count positions.
- The PF, SF, and ZF flags are left unchanged in this rotate operation . The operand can be a register or a memory location.
- The count will be in instruction if it is 1, and in CL register if greater than 1.

- **ROR : Rotate right without carry**

ROR <Reg. / Mem> , <Count>

R(n) ← R(n + 1) ; R(MSB) ← R(LSB) ; CF ← R(LSB)



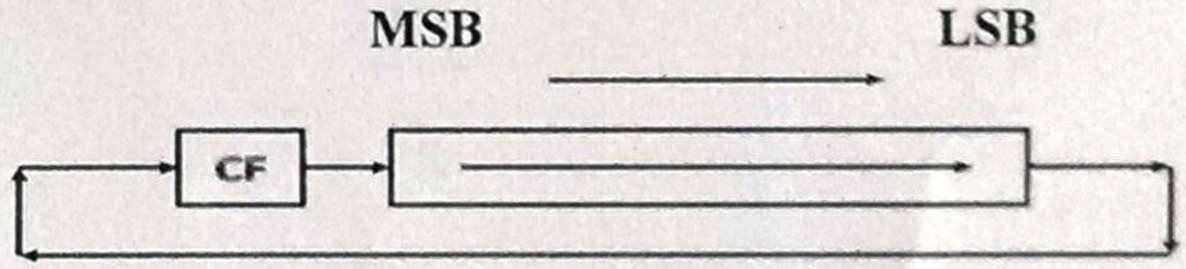
This instruction rotates all the bits in a specified word or byte to the right by the specified count (bit-wise) excluding carry.

- The LSB is pushed into the carry flag as well as the MSB at each operation. The remaining bits are shifted right subsequently by the specified count positions.
- The PF, SF, and ZF flags are left unchanged in this rotate operation . The operand can be a register or a memory location.
- The count will be in instruction if it is 1, and in CL register if greater than 1.

- **RCR : Rotate right through i.e., with carry**

RCR <Reg. / Mem> , <Count>

R(n) ← R(n + 1) ; R(MSB) ← CF ; CF ← R(LSB)



- This instruction rotates all the bits in a specified word or byte to the right by the specified count (bit-wise) excluding carry.
- The LSB is pushed into the carry flag as well as the MSB at each operation. The remaining bits are shifted right subsequently by the specified count positions.
- The PF, SF, and ZF flags are left unchanged in this rotate operation . The operand can be a register or a memory location.
- The count will be in instruction if it is 1, and in CL register if greater than 1.

❖ Processor Control Instructions

The Processor control instructions include flag manipulation and processor control instructions. These instructions control the functioning of the available hardware (programmer accessible hardware) inside the processor chip.

These are categorized into two types:

a) Flag manipulation instructions
b) Machine control instructions

The flag manipulation instructions directly modify some of the flags of the 8086 flag register.

The machine control instructions controls the bus usage and execution.

The processor control group includes instructions to set or clear carry flag, direction flag, and interrupt flag.It also includes the HLT, NOP, LOCK and ESC instructions which controls the processor operation

The Various Flag manipulation instructions are

CLC, CMC, STC, CLD, STD, CLI, STI

The Various machine control instructions are

WAIT, HLT, NOP, ESC, LOCK

- **Flag manipulation instructions**

➢ **CLC : Clear Carry**

The carry flag is reset to zero i.e., CF = 0 CF ← 0

➢ **CMC : Complement the carry**

The carry Flag is Complemented i.e., if CF = 0 before CMC then after CMC CF =1 and vice versa. CF ← ~ CF

➢ **STC : Set Carry**

The carry flag is set to one i.e., CF = 1 CF ← 1

➢ **CLD : Clear direction**

The direction flag is cleared to zero i.e., DF = 0 DF ← 0

➢ **STD : Set direction**

The direction flag is set to 1 i.e., DF = 1 DF ← 1

➢ **CLI : Clear Interrupt**

The Interrupt flag is cleared to zero i.e., IF = 0 IF ← 0

➢ **STI : Set Interrupt**

The Interrupt flag is set to 1 i.e., IF = 1. IF ← 1

- **Machine control instructions**

➢ **WAIT : Wait for Test input pin to go low or an interrupt signal**

This instruction causes the processor to enter into an idle state or wait state and continue to remain in that state until a signal is asserted on the TEST input pin or until a valid interrupt signal is received on the INTR or NMI interrupt input pin. If a valid interrupt signal occurs while the 8086 is in idle state, the 8086 will return to the idle state after the interrupt service procedure executes. It returns to the idle state because the address of the WAIT instruction is the address pushed on to the stack when the 8086 responds to the interrupt request.

WAIT affects no flags

The WAIT instruction is used to synchronize the 8086 processor with the external hardware such as the 8087 math processor.

- **HLT : Halt Processing**

The HLT instruction will cause the 8086 to stop the fetching and execution of the instructions. The 8086 will enter a halt state i.e., used to terminate a program.The only ways to get processor out of Halt state are with an interrupt signal on INTR pin, an interrupt signal on NMI pin, or a valid reset signal on RESET input.

- **NOP : No Operation**

No operation is performed for three clock periods
This instruction simply uses up three clock cycles and increments the instruction pointer to point to the next instruction.
The NOP instruction does not affect any flag.
The NOP instruction can be used to increase the delay of a delay loop.
When hand coding, a NOP can also be used to hold a place in a program for instruction that will be added later.

- **ESC:**

Frees the bus for an external master like a coprocessor or peripheral devices.

- **LOCK:**

Prefix which may appear with another instruction.

When executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely.

Used in case of multiprocessor systems.

Assembler Directives :-

Assembler is a program which translates assembly language program in to machine language program.

*) The Assembler decides the address of each label and substitutes the values for each of constants and variables. It then forms the machine code for the mnemonics and data in the assembly language program.

*) The assembler will find out the syntax errors, logical errors, programming errors.

*) for executing a program an Assembler needs some hints from the programmer. these types of ~~dgtp~~ hints are given to the assembler using some predefined alphabetical strings called Assembler Directives.

*) An Assembly language program contains 2-types of statements

a) Instructions
b) Directives.

a) The instructions are translated into machine code where as directives are not translated in to machine codes.

b) The Directives are statements to give direction to assembles to perform rest of assembly process.

*) Popular assembly's are.

Intel 8086 macro assembler (ASM86)
Borland Turbo assembler (TASM)
Microsoft macro assembler (MASM).

a) Data defination and storage allocation directives.

DB, DW, DD, DQ, DT.

b) Program organization directives.

SEGMENT, ENDS, ASSUME, END

c) Alignment directives.

EVEN, ORG

d) Value Returning attribute directives

OFFSET, SIZE, LENGTH

e) Procedure defination directives

PROC, ENDP

f) Data Control directives

PUBLIC, EXTRN, PTR

g) Branch displacement directives

SHORT, LABEL, NEAR, FAR.

1) SEGMENT & ENDS :-

SEGMENT is used to inform the assembler that a segment is starting

```
Name SEGMENT
```

ENDS assembler directive is used to inform the assembler about the end of segment.

```
Name ENDS
```

Eg:- *) CODE SEGMENT ; memory segment having name CODE start here

=

Program

=

CODE ENDS ;

*) DATA SEGMENT

=

Data

=

DATA ENDS

*) EXTRA SEGMENT

=

Data

=

EXTRA ENDS

*) STACK SEGMENT

=

stack of Data

=

STACK ENDS %

2) ASSUME :- This assembler directive is used to inform the assembler which memory segment is used for CODE, DATA, STACK, & data.

Eg:- ASSUME CS:CODE, DS:DATA, ES:EXTRA

3) PAGE :- PAGE length, width.

It is used to indicate the format of one Page. length indicates no: of lines to be Printed in one page. width indicates no: of Characters to be Printed on line.

Eg:- PAGE 35, 85

one Page = 35 lines

line = 85 characters.

4) TITLE :- TITLE text (maximum 60 charact).

Eg:- TITLE 8086 MICROPROCESSOR.

5) ORG (originate) :- ORG EA

ORG Ad is used to inform the assembler about starting Effective address from where assembler should start memory location allotment.

Eg:- ASSUME CS:CODE, DS:DATA, ES:EXTRA

CODE SEGMENT

ORG 7000H (This instruction tells Program '1' are alloted memory location starting EA = 7000H).

=

Progr (1)

=

ORG 9500H

=

Progr (2)

CODE ENDS

DATA SEGMENT

=

DATA

DATA ENDS ||.

6) END:- informs assembler that assembling operation should be stopped.

7) EQU :- (Equate) :- This directive is used to assign a value to the data name

Eg:- Rajesh EQU 5H.

8) ENDS : [END SEGMENT] :- This directive is used to indicate ending of a named segment.

Eg:- CODE ENDS

DATA ENDS

STACK ENDS

:

9) DB :- [Define Byte] :-

[List of Data bytes (or) no:- of memory locations]

This directive is used to store a byte (or) byte of memory locations or available memory for storing variables constants (or) strings.

Eg:- RANKS DB 01H, 02H, 03H, 04H

Eg:- DB 95H, 76H, 73H, 0A1H.

All four bytes are alloted memory locations.

Eg:- MESSAGE DB 'Good morning';

Eg:- DB 100 DUP (00H).

100 memory locations are alloted to 100 bytes and in all memory locations (00H) is stored by the loader.

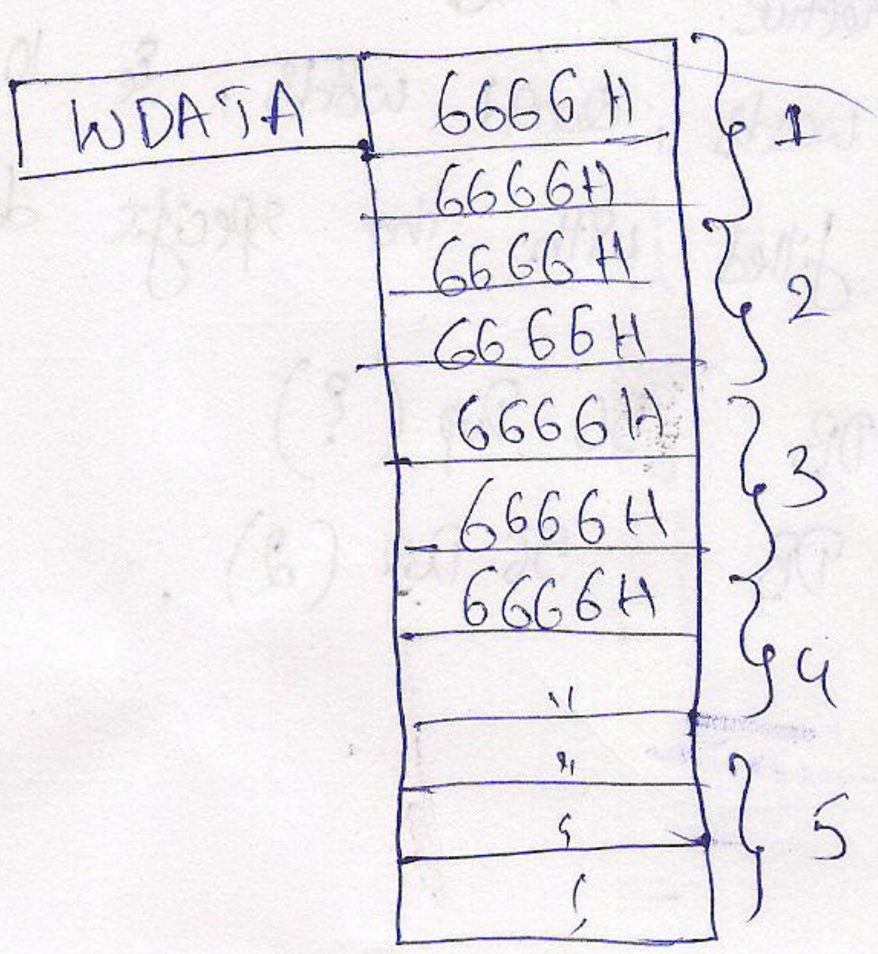
10) DW [Define word] :-

This directive is used to store a word (or) words of memory location in the available memory for storing variables constants (or) strings.

Eg:- WORDS DW 4567H, 78ABH, 045CH.

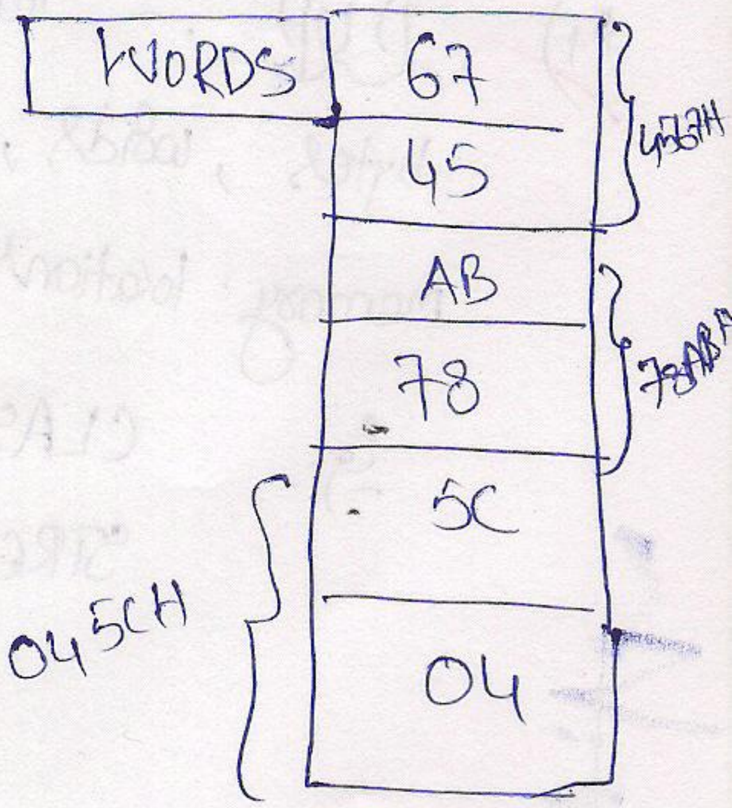
PRODUCTS DW 2 DUP(0) → It stores 2 words of consecutive locations & initialize all the 4-bytes with zero's.

WDATA DW 5 DUP(6666H) → stores 10 bytes of memory locations & initializes all words locations with 6666H.



Word => 2-bytes 8bit & 8bit

(or)



11) DD (Define Double word):- This directive is used to store double words of 2-words in available memory for storing variable constraints (or) strings.

Eg:- Rajesh DD 12345678H

NUMBERS DD 5 Dup(0) → Reserves 20 bytes of consecutive memory locations & initialize them with zeros.

12) DQ (Define Quad word):-

This directive is used to store '1' Quad word in the available memory for storing variables, constrains of strings.

Eg:- Rajesh DQ 123456789ABCDEFGH

NUMBER DQ 0

BIG_NUMBER DQ 2345987401928928

NUMBERS DQ 10 Dup(0).

13) DT (Define Ten Bytes):-

This directive is used to store 10 bytes in available memory for storing variables, Constants & strings.

Eg:- Rajesh DT 32 35 36 91 32 36 91 52 78 H

RESULT DT 20 Dup(0)

NUM1 DT 0

14) DUP :- This directive is used to reserve a series of bytes, words, double words, Quad words, & 10 bytes. Here reserved memory locations are filled with the specific data (or) left uninitialised.

Eg:- CLASS DB 56 Dup(?)

STRENGTH DB 56 Dup(2).

15) GROUP :- This directive is used to form a logical group of SEGMENTS for similar purpose.

Eg:- CRIT GROUP : CESM, CRMT, SSRR

16) PUBLIC :- A large program may contain a no: of program modules. Each module is assembled, tested and debugged individually. The PUBLIC directive is used to inform the assembler that the specified name (or) label can be accessed by other program modules

17) EXTRN :- It is used to tell the assembler that the names (or) labels of procedures following the directives are in some other assembly modules.

i.e if a variable (or) label is used in an instruction of another program, it must be specified external using EXTRN directive.

```
MODULE 1    SEGMENT
            PUBLIC    FACTORIAL FAR
MODULE1     ENDS
MODULE 2    SEGMENT
            EXTRN     FACTORIAL FAR
MODULE 2    ENDS
```

18) EVEN (Allign an Even memory address) :- It updates the location Counter to the next even address if the current location Counter Contents are not Even, and assigns the following Routine (or) variable (or) Constants to address.

Eg:-
```
SALES   DB 9 Dup (?)
EVEN
RECORDS DW 100 Dup(0)
```

Eg:-
```
EVEN
ROOT  PROC
  :     :
  :     :
ROOT  ENDP
```

19) PTR (Pointer) :- It is used to declare the type of label, variable. (or) memory operand. The operator PTR is prefixed by either byte (or) word.

Eg:- MOV AL, BYTE PTR [SI]
MOV BX, WORD PTR [2000H].

20) SHORT :- It indicates assembler only 1-byte displacement is needed to code a Jump instruction.

If the Jump destination is after the Jump instruction in program the assembler will automatically reserve 2-bytes for displacement. Using SHORT operator saves 1-byte by telling assembler that it needs to store only 1 byte for this particular Jump.

21) TYPE :- It directs the assembler to decide the data type of specified label and replaces the "TYPE LABEL" by the decided data type.

for word type variable the data type is 2.
for double word type it is '4'.
for byte type it is '1'.

Eg:- MOV AX, TYPE STRING ; if string is a word 0002H is in AX.

DOS Function CALLS:-

AH	00H	: Terminate a Program
AH	01H	: Read the keyboard
AH	02H	: write standard o/p device
AH	08H	: Read standard i/p without Echo
AH	09H	: Display a character string
AH	0AH	: Buffer keyboard input
INT	21H	: CALL DOS Function.

Procedures and Macros

- **Procedures:**

When a group of instructions are to be used several times to perform a same function in a program, then we can write them as separate subprogram called procedure or subroutine.

Procedures can be called in a program using CALL instructions.

Procedures are written and assembled as separate program modules and stored in the memory. The assembling of procedures will be done without considering the main program i.e., independent of calling program.

When procedure is called in main-program control is transferred to procedure and after execution the control is transferred back to main program, procedures are called using CALL instruction and returned back using RET instruction.

- **Label PROC Far / Near**

RET Label ENDP.

- **Types of Procedures:**
  - i. Depending on Position of procedure in memory
    - o Intra-segment Procedure
    - o Inter-segment Procedure
  - ii. Depending on occurrence in program
    - o Single Procedure
    - o Nested Procedure
    - o Re-Cursive Procedure
    - o Re-Entrant Procedure
- **Intra-Segment Procedure:**

CALL → (SP) ← (SP) – 2
((SP) + 1 : (SP)) ← (IP)
(IP) ← (EA)

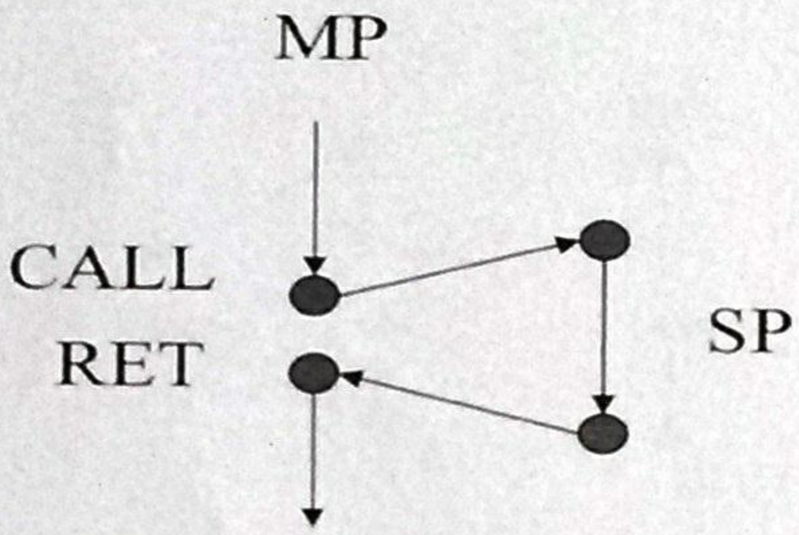
RET → (IP) ← ((SP) + 1 : (SP))
(SP) ← (SP) +2

- **Inter-Segment Procedure:**

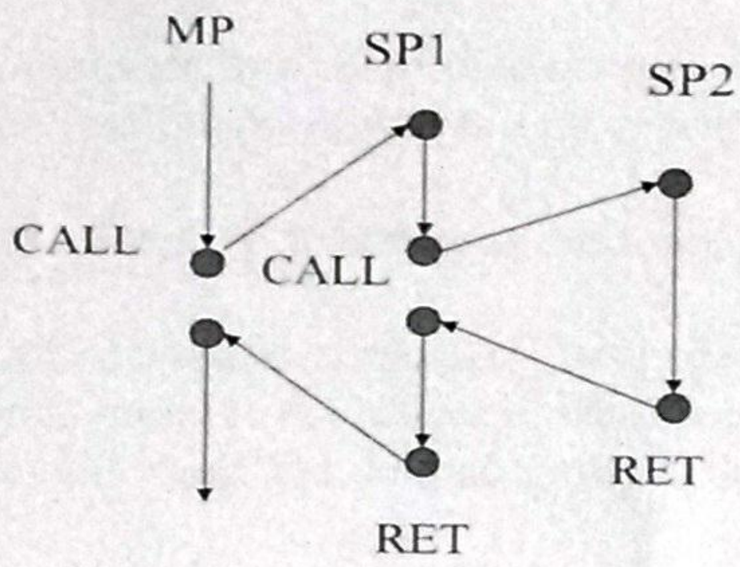
CALL → (SP) ← (SP) – 2
(((SP) + 1) : (SP)) ← (CS)
(SP) ← (SP) –2
(((SP) + 1) : (SP)) ← (IP)
(IP) ← (EA)
(CS) ← Seg. Address

- RET → (IP) ← ((SP) + 1 : (SP))
(SP) ← (SP) – 2
(CS) ← ((SP) + 1 : (SP))
(SP) ← (SP) – 2

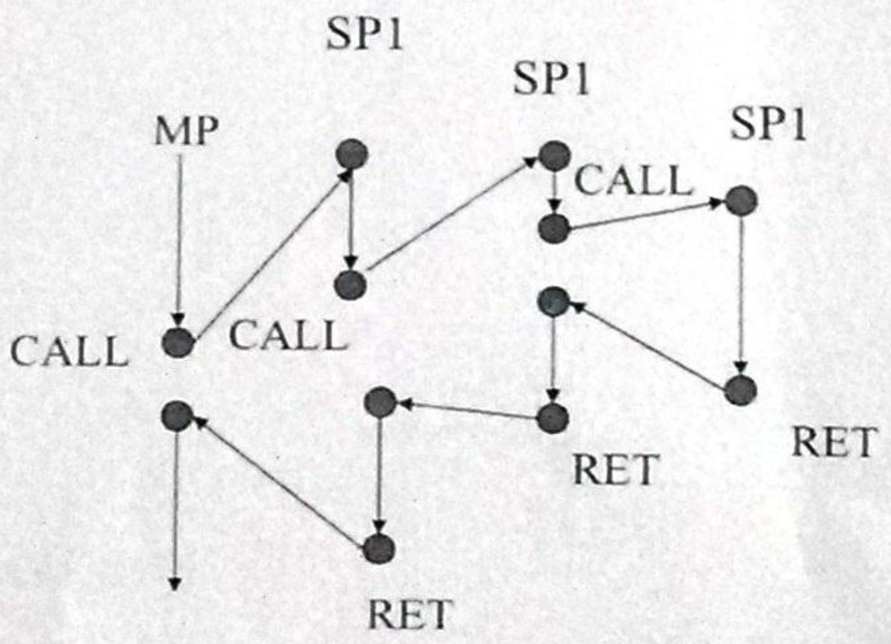
- Single Procedure:
  MP → Main Program
  SP → Sup-Program / Procedure



- **Nested Procedure**

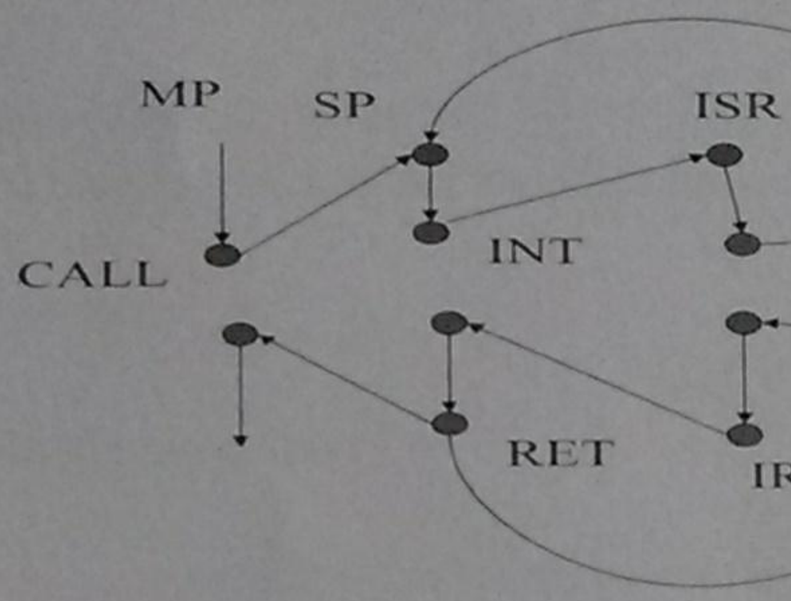


- **Re-Cursive Procedure:**



- The recursive procedures are implemented using procedure CALL itself, but care must be taken to assure that each successive call does not destroy the parameters and results generated by the previous CALL and make sure that procedure does not modify itself, i.e., each call must store its set of parameters, Registers and all temporary results in a different place in memory.

- Push all the flags and all registers used in the procedure.
- Should use only register or stack to pass parameters.
- **Re-Entrant Procedures:**



- **MACROS:**

- When a group of instructions are to be used several times to perform a same function in a program and they are too small to be written as a procedure, then they can be defined as a MACRO.
- A MACRO is a small group of instructions enclosed by the assembler directives MACRO and ENDM.
- The MACROS are identified by their names and usually defined at the start of a program.
- The Macro is called by its name in the program, whenever the MACRO is called in the program, the assembler will insert the defined group of instructions in place of CALL instruction.
- X1 MACRO
  Mov SI, OFFSET Arr
  Mov DI, OFFSET Arr1

  Mov AL, (SI)

  Add AL, (DI)

  Inc SI

  Inc DI

  ENDM
- Passing parameters in MACROS:
  X1 MACRO Arr, Arr1
  :
  END M
  :
  X1 Arr2,Arr3
  :
  X1 Arr4, Arr5

- Advantages and Disadvantages Of Procedures and Macros:

Procedures: Advantages:

1. Machine codes for the group of instructions in the procedure has to be put in memory only once.
2. Each set of instructions can be individually assembled and debugged.

- Disadvantages:

1. Need stack operations performed.
2. Over-head time required to call the procedure and return to the calling program.

- Macros Advantages:

1. Simple
2. No over-head time required.

- Disadvantages:

Program may take up more memory due to insertion of machine codes in the program at the place of Macros.

- Comparison of Procedures and Macros:

S.No.	Procedures	Macros
1.	Machine code for instructions are stored in memory only once.	Machine codes are generated for instructions in the Macro each time it is called in the main program.
2.	Accessed by CALL and RET mechanisms during program execution.	Accessed during assembly with name given to Macro when defining it.
3.	Size is unlimited.	Size is limited to few lines.
4.	Parameters are passed in registers, Memory locations or stack.	Parameters are passed as part of statement which calls Macro.
5.	Can be present in Same (or) different segment as that of the main program.	Usually defined at the beginning of program in the same segment.