→ Push / Pop:-

→ immediate value

Push word ptr 20h → 20h 16-bit value
(word ptr: size specifier)

{ mov ax, 20h
  push ax

pop 20h (x)

x = 9 * 6 + 2;
↓
l-value

Constant → (constant can't be l-value)
(Size can't be determined
(It is stored in code segment not in data segment)

Intrinsic Data Types:-

8-bit / 16/32/64-bit

format:
[name] directive initializer, [initializer]----

(→ optional)

1. db (Define byte)

```
char / value  db  20h      ; 1-byte
next          db  ? [80] (not intialized) ; 2-byte
list          db  1,2,3,4  ; 4-bytes
                  [0] [1] [2] [3]
list2         db  10 dup(0)
```

↳ duplicate operator → array

list2 db 10 dup (?)

db 100 ; 0 → 255

no-name

char2 db 'A' ; ASCII code (65)

ii) dw (define word)

valuex dw 100h :-(100h = 256)
listw dw 400, 500, 600, 700, 800 (16-bit each variable)
[0], [2], [4], [6], [8]

dw ?

push valuex
pop listw[2]

iii) dd (Define double word)

big dd 7A040h
barray dd 47Ah, 42h, 67h, 0AAh → [12]
AAh dd ? [4] [8]
[0]

iv) DQ (Define Quad word)

vbig dq 743298Ah

Addressing Modes of x-86 Microprocessor:-

1. Register Addressing Mode:-
   a. Both destination and source are registers of the same size
   b. CS can't be destination.
   c. Both dest. and source can't be segment registers

e.g.:- mov ax, bx ✓
mov cx, dl X. (Size mismatch)
mov al, ah ✓

mov ds, es ✗ (C point)
mov cs, ax ✗ (16 point)

Formats:-

mov reg8, reg8
mov reg16, reg16
mov segreg, reg16
mov reg16, segreg

2. Immediate Addressing:-
A constant immediately follows the op-code.

e.g.-

mov al, 'A'
mov bx, 42h
mov cl, 01101111b
mov dl, 10
mov cx, 'A B'
low → high
mov ah, 0ah

3. Register Indirect Addressing:-

Register indirectly refers to memory.

e.g.-

mov ax, [bx]

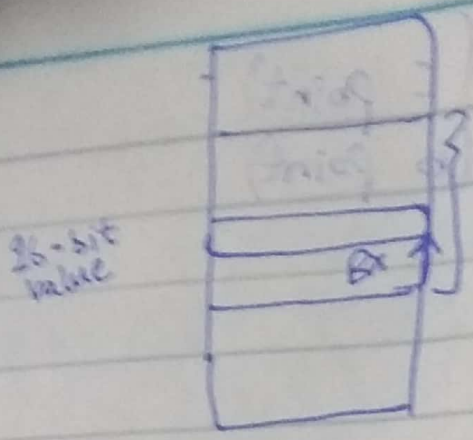
⇒ bx contains the offset of data in data segment.

Load effective address

mov bx, offset list2
mov ax, [bx]

OR

lea bx, list2



0. mov al, 80
1. mov [bx], al
2. lea bx, next

mov bx, 1 (If no name)
↓
address of memory offset
0 → value
1 → next

mov [bx], [dx] → X

→ Memory to memory transfer is not allowed

mov ax, [bx]
mov [bx], ax

Problem :- It can't determine size of data to be transfered.

mov [bx], 200

Sol 1 :- mov word ptr [bx], 200 } or

Sol 2 :- mov ax, 200
mov [bx], ax

mov cl, [sp]

⇒ 8- bit contents at offset given by SP from stack segment.

mov cl, ds: [sp]

segment override

mov es: [bx], dx

Indexed Addressing:-

list dw 22 dup(0) → displacement ↓ replace with offset on runtime

mov ax, list [si]

0, 2, 4

arrb db 200 dup(10)

mov arrb[di], cl

0, 1, 2, 3, ---

mov ax, [bx + si]

mov ax, es: [bx + si]

Base: bx, bp

index: di, si

⑤ Base-Plus-Index:-

Base : BX, BP

Index : DI, SI

Eg:- mov [BP + DI] , CX (DS)

mov SS : [BP + DI] , DX

offset in

mov es : [BX + DI], word ptr 200h

segment prefix size specifier

⑥ Based Index plus displacement:-

→ (name of variable)

→ Name is replaced by displacement on runtime.

TwoD db , 10 dup (5 dup(50))

```
mov di, 0
mov si, 2
mov bx, 30
mov dl, TwoD [bx+si]
```

disp ↙ ↘ index

base

	0	1	2		9	
0	50	50			50	10
1						10
2						10
3			X			10
4						

TwoD [3,2]

mov TwoD [bx + di] , al

```
mov   al, 30
mov   dl, 35
add   al, dl
mov   dl, al
mov   ah, 2
int   21h
```

Output: A

mov dl, 6
, ASCII code in dl
, Service # to display a char

1) Data Movement :- mov, lea, push, pop

Lahf, sahf, LxS

2) Arithmetic :-
ADD, SUB, INC, dec, mul, div, neg, cmp (comparison)

3) Logical :-
and, or, xor, not

4) Program Constraints :-
Jmp, int, call, jx, loop

div:-

```
mov dl, 8
mov bx, 18
```

1 → Q

```
10 | 18        Q  R
   | 10        1  8
   | 8  → R
```

```
mov ax, bx
mov bl, 10
div bl  ------------>  ; ax  (numerator)
                       bl → denominator
mov R, ah
mov Q, al
```

8-bit → 16 bit
al, al accumulator

Lahf:- (Load ah from flags)

order
Lahf
dest, source
8-bit "lowest" flags

sahf (store ah into flags)

sahf
source dest
lower 8-bit flags

LXS (X = D/E/S)

```
mem 32 dd ?
LDS bx, mem 32
```

Copy ← higher 16-bit (segment part)
bx ← lower - 16-bit (offset part)

mul (Multiply):-

Generally, mul reg8 ; ax = al * reg8

eg; mul dl ; ax = al * dl

Generally, mul reg16 ; dx:ax = ax * reg16

mul cx ; dx:ax = ax * cx

div (divide):-

Generally, div reg8 ; ax / reg8 → Q: al, R: ah

div reg16 ; dx:ax / reg16 → Q: ax, R: dx

jmp:-

Generally jmp target-label

Formats:- jmp disp8
jmp disp16
jmp reg16

e.g.:- mov dx, 10
mov ax, 20
jmp next ; 4 byte
mov cx, ax ; 4 byte
mul cx ; 2 byte
inc cx
next: add bx, ax

IP = IP + 10 (disp)

2. Call and return

→ Call disp16
→ Call reg16

mov dx, 10
mov ax, 20
call next
next: mov cx, ax
mul cx
inc cx
jmp over
next: add bx, ax
dec bx
ret
over:

1. Push IP
2. IP = IP + disp

1. Pop IP

3. int

format: int : nn

eg:- int 21h

Program flow control:-

Conditional

1. FLAG based

JC	(Jump if carry)	CF = 1
JNC	(Jump if no carry)	CF = 0
JS	(Jump if sign)	SF = 1
JNS	(Jump if no sign)	SF = 0
JZ	(Jump if zero)	ZF = 1
JNZ	(Jump if not zero)	ZF = 0
JO	(Jump if overflow)	OF = 1
JNO	(Jump if no overflow)	OF = 0
JP	(Jump if parity)	PF = 1
JPE	(Jump if parity even)	PF = 1
JNP	(Jump if no parity)	PF = 0
JPO	(Jump if parity add)	PF = 0

2. Relational - operator Based:-

{ JA (Jump if above) >
JNBE (Jump if not below or equal) !<=
(greater)

{ JNA (Jump if not above) !>
JBE (Jump if below or equal) <=

JB (Jump if below) <

JNAE (Jump if not above or equal) ! >

JNB (Jump if not below) ! <

JAE (Jump if above or equal) >=

JE (Jump if equal) =

JNE (Jump if not equal) !=

ah { even: Rem=0
   { odd: Rem=1 ← n1/2

ax = n { al ← n1
       { ah ← 0