Assembly.

(spring 2016)

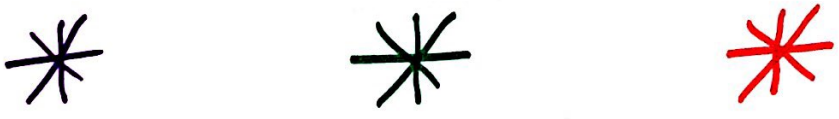
Dr.: Saadeh Sweadan.

Slides: Khalid Darabkeh.

Produced by: Mohammad Abuhashia.

Contents:

- Summary For CH_1.
- Notes from the second half of CH_2 ⟶ to the last of CH_6



Assembly

CHAPTER "1":

* Peripheral devices: it is not part of the essensial compute (mem. or Up) ; it is internal device or external.
* 1946: first computer.
* edge-triggered clocking methodology: update(change) ⟹ on a clock edge.
* level- " " " : update(change) ⟹ on the high level (level 1)
* cycle time: cycle time = sec. per cycles.
* clock rate (freq.): clock rate (freq.) = cycles per sec.

$\Rightarrow$ $\text{freq.} = \frac{1}{\text{cycle time}}$ $\quad T = \frac{1}{f}$

* The three parts of the computer:
① CPU. ② Memory. ③ I/O. ⟹ They all comected by buses.
* bus: a wire carry the information from place to place.

Ex. (Vax) ⟹ 32-bit address bus.
& 32-bit data bus.

* Types of buses: Address / Data / control.
  - Address: to assigne an address to a device.
  - Data: to get/give info. from/for a device.
  - control: provide read or write signals to the device to indicate if the CPU want to get or send info.

* It is impossible for two devices to have the same address.

The per formance of the CPU is Better when... ⟺ it have more (large number) of Data buses.

* Why the DATA-buses are bi-directional ?

since the CPU use them to receive or to send data.

8-bit bus ⟹ send 1 byte. ...

16-bit bus ⟹ send 2 bytes.

* Types of Data-bus:
  - ① Internal (back side) → between CPU and cache.
  - ② External (front side) → between cache & memory.

* Why the Address-bus is unidirectional ?

since the CPU use it only to send out addresses.

$$\text{Number of memory locations addressable} = 2^{\text{Number of address bits.}}$$

* All memories share 2 features:

① Each info. unit is the same size.

② info unit has a numbered address so → it can be uniquely referenced.

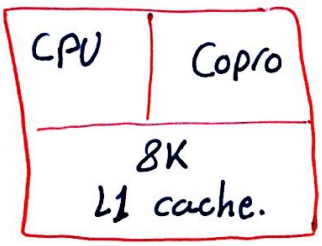
* Memory Cell characterized by:
  - ① address.
  - ② content.

* CPU contains:
  - ① Registers
    - A special purpose register
    - B General purpose register.
      - it can store Data/Addresses.
  - ② ALU
  - ③ CU
    - store the meaning of each instruction and what the CPU should do when receiving instruction.

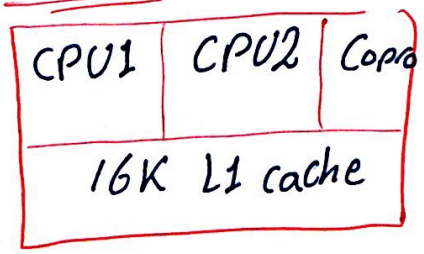
* **Assembler**: translator is used to translate the assembly code into machine code.

* Example for fourth-generation languages: SQL.

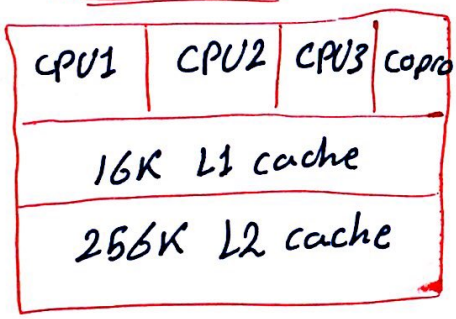
80486DX



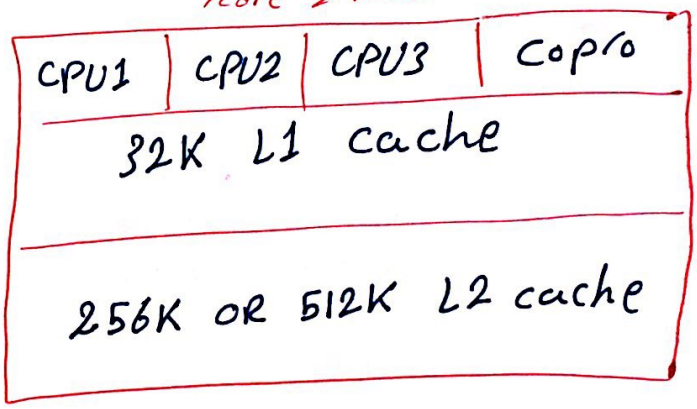
Pentium



Pentium Pro

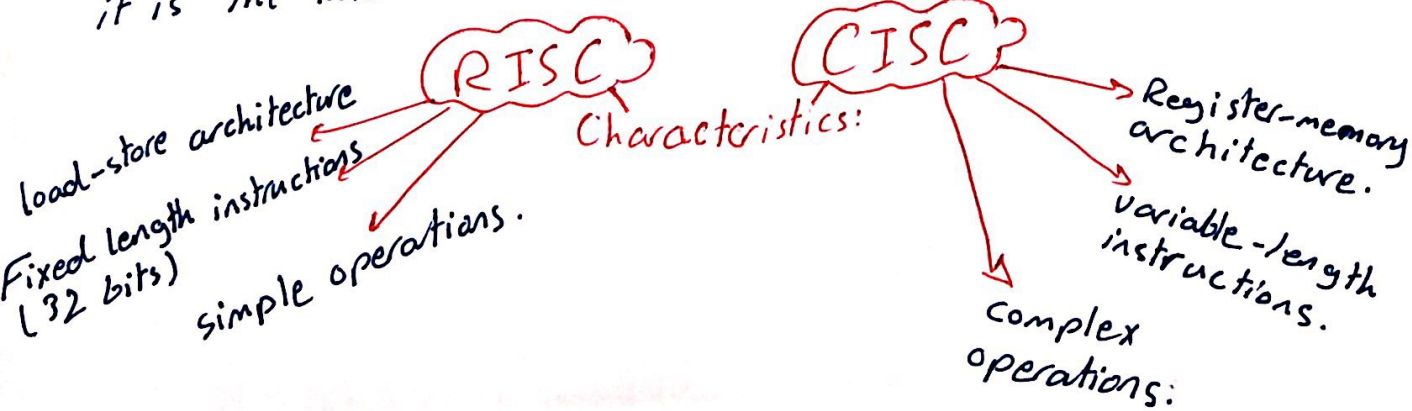


Pentium II / Pentium III / Pentium 4 / core 2 module.



* Multithreaded applications:
if the program is written to take advantage of multiple cores (to increase speed of execution.

* ISA: (instruction set architecture)
it is the interface between hardware & software.



* Pipelining: a ways of speeding up execution of instructions.
(Key idea): overlap execution of multiple instructions.

* VLIW: very Long instruction word.

* I/O characteristics: ① Behavior. ② Partner. ③ Data Rate.

* Three main parts of memory system:
① TPA (640K bytes) ② system areer (384K bytes) ③ XMS.

* The type of Mp determine whether XMS exists or Not.

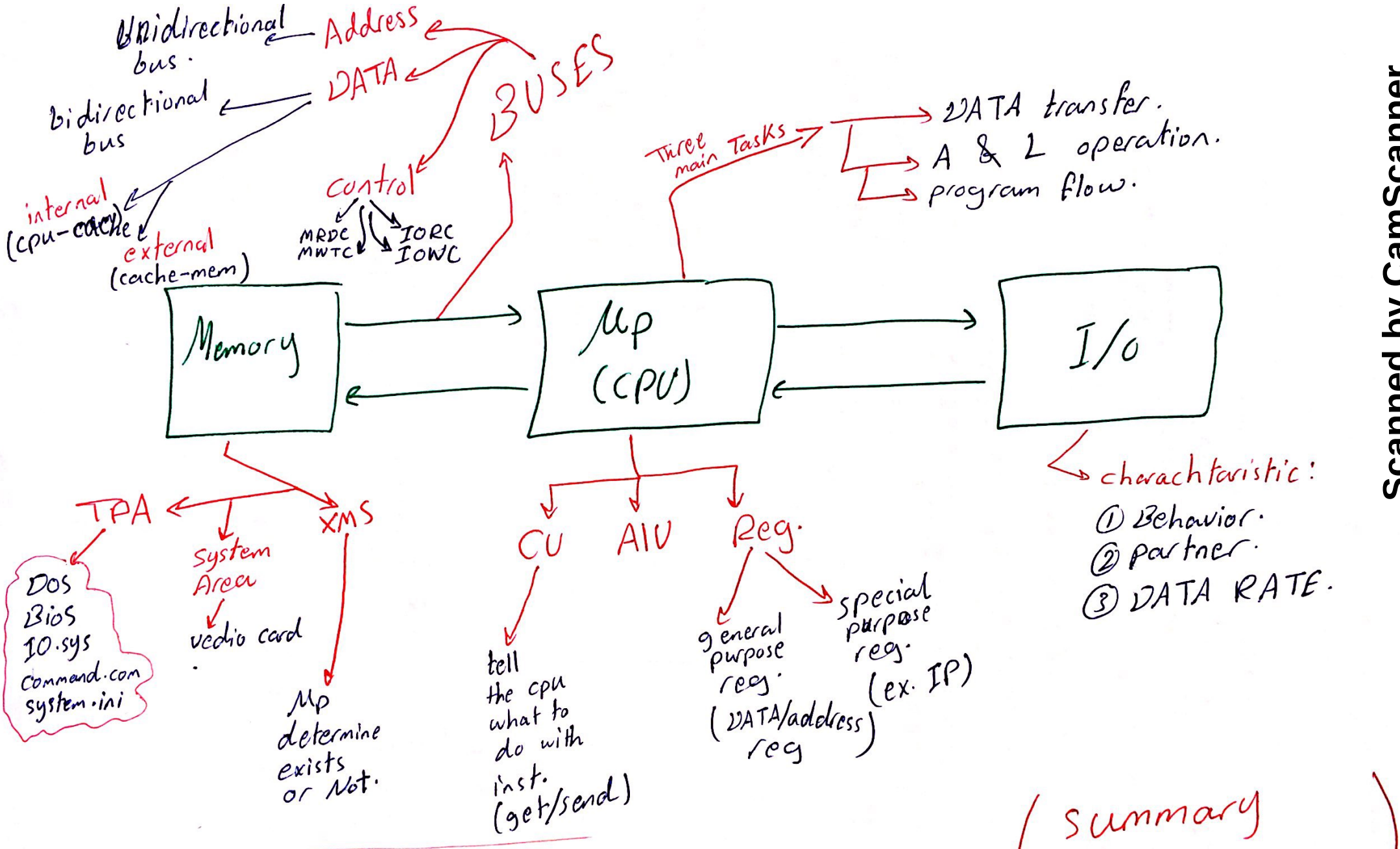
* How O/s stop program x? it issues an interrupt.

TPA ⇒ DOS / BIOS / IO.sys. / COMMAND.COM / system.INI

- contains programs allow DOS to use I/O devices.
- controls the operation of the computer from the Keyboard when operated in the DOS mode.
- windows use it to load drivers used by windows.

* DOS device drivers: have extension of .SYS → ex. Mouse.sys.
In DOS version 3.2 and higher: extension .EXE → ex. EMM386.EXE

* Three main Tasks for Mp:
① Data transfer. ② simple arithmetic and logic operations. ③ program flow via simple decision.



command.com : related to the Keyboard without it we can't use the Keyboard. (used by DOS).

(summary for #CH1 #)

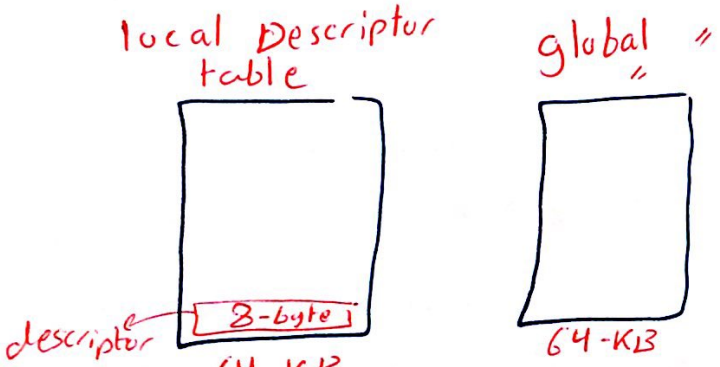
Assembly "Notes"

#CH2#

* If EA = 9D3C5 and the offset = F4C8 find the segment ?

```
  9 D 3 C 5
    F 4 C 8   -
-------------
  8 D E F D
```

⇒ segment = 8DEFD



⇒ To know how many Descriptors we have:

$$\frac{64KB}{8B} = \frac{2^6 * 2^{10}}{2^3} = \frac{2^{16}}{2^3}$$

$$= 2^3 * 2^{10} = (8)(1024) = 8192 \text{ descriptors}$$

Limit:

$L_{15} \cdots L_0$

16-bit ⇒ 2^{16} = 64K (size)

Limit $L_{19} \cdots L_{16}$ $L_{15} \cdots L_0$

⇒ 20-bit ⇒ (size) = 1M

AV: (Available)
- 1 (Not used)
- 0 (used).

D
- 0 ⇒ instruction set (IS) 16-bit
- 1 ⇒ (IS) 32-bit.

G
- 0 ⇒ Limit 00000H ⟶ FFFFFH
- 1 ⇒ with constant: ex. $L_{19} \cdots L_0$ FFF (const.) ⇒ 32-bit.
  - Least: 00000FFF (4K)
  - Most: FFFFFFFF (4G)

L	D	
0	0	⟶ IS = 16-bit (AX)
0	1	⟶ IS = 32-bit (EAX)
1	X	⟶ IS = 64-bit (RAX)

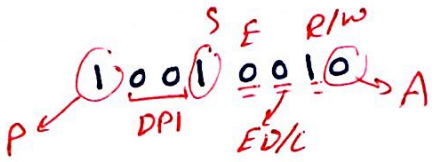
* Access rights:

* إذا أتى مثال عن segment ليس من الضروري أنه يعطي نوع هل هو code أو DATA يمكنه معرفة ذلك من خلال قيمة R أو W أو من الأصل من خلال قيمة E فإذا كانت:

$E=0 \Rightarrow$ DATA seg.

$E=1 \Rightarrow$ Code seg.

* EX. 92H →



we have for (92):

$P=1 \Rightarrow$ valid base & limit.

DPI = 00 (high level)

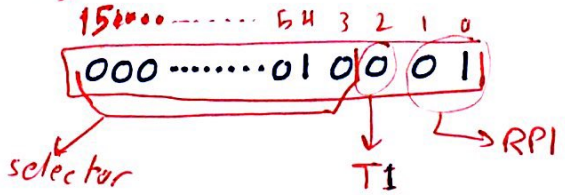
$S=1 \Rightarrow$ DATA seg. descriptor.

$E=0 \Rightarrow$ DATA seg.
- $ED=0 \Rightarrow$ upward (DATA seg.)
- $W=1 \Rightarrow$ DATA can be written.

$A=0 \Rightarrow$ (Not accessed).

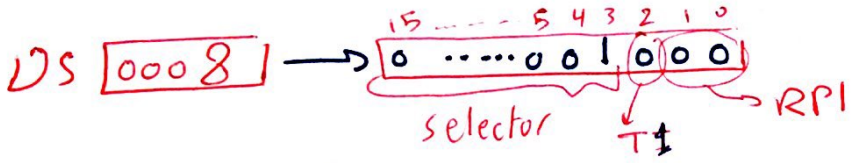
Ex. for figure (2-8):

DS = 17[redacted] →



* selector = (2)
(it will select descriptor Number 2)
* TI = 0 (global des. table).
* RPI = 01 if (01 ≥ 11) and that's right so the access is granted

* for figure (2-9):



* selector = 1 (descriptor 1)
* TI = 0 (global).
* RPI = 00 ≥ 11 ⇒ access granted.

---

* size of segment = limit = 00FFH (2-bit) = 256 B
* starting address = Base = 100000H
* Ending address = size(limit) + starting address = 1000FFH

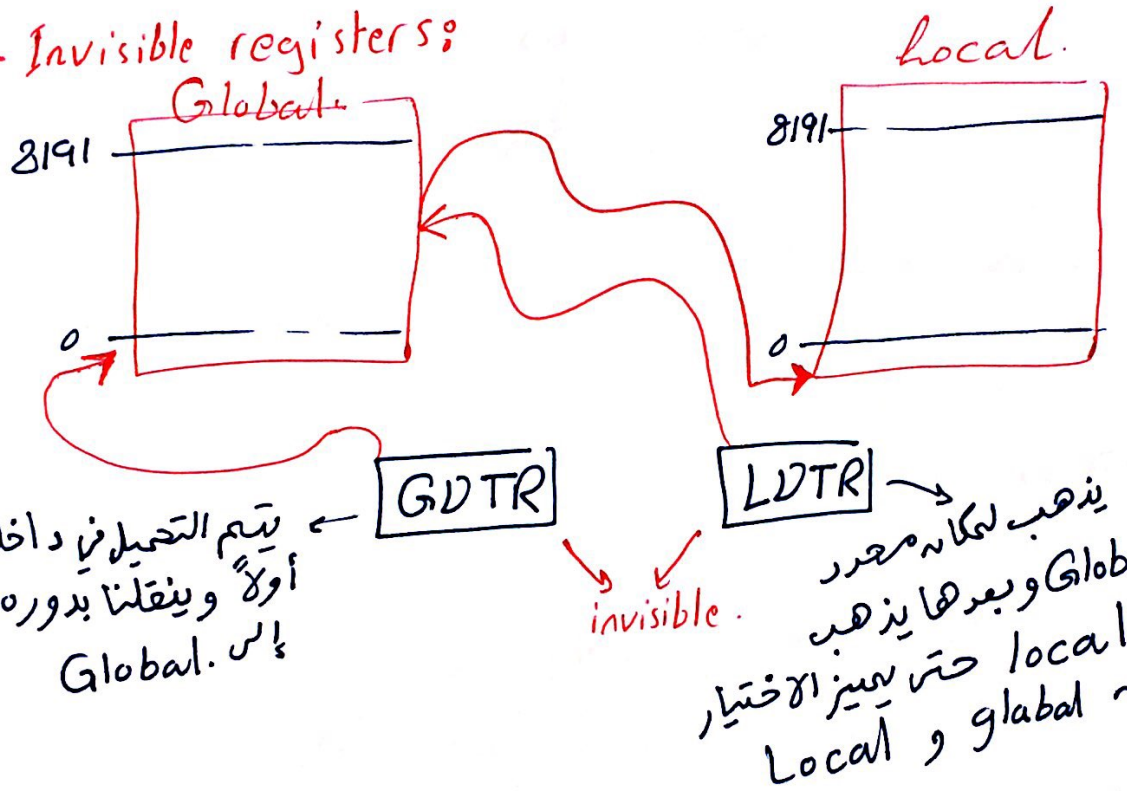
* Flat mode memory:

- Range: 0000000000 ⟶ FFFFFFFFFF

⇒ 10 * 4 = 40-bit

⇒ size: 2^{40} = 1 TB

* Invisible registers:



يتم التحميل في داخله أولاً وينقلنا بدوره إلى Global.

يذهب لمكان محدد في Global وبعدها يذهب لـ local حتى يميز الاختيار بينه Local و glabal

---

CH3

Assembly language:

- consists: LABEL / OPCODE / OPERAND / COMMENT.

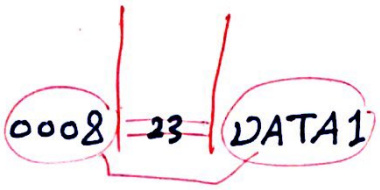
* جميعها قد تكون موجودة أو غير موجودة ماعدا operation code = (OPCODE) فهو لا يمكن الاستغناء عنه.

- comment:

* فقط للملاحظات يستفيد منه المستخدم فقط ولا يؤثر في التنفيذ.

Ex:

offset	LABEL	OPCODE	OPERAND
0008	DATA1	DB	23H



DATA1/0008 بنفس العنوان ولكنه التعامل مع DATA1 أسهل من استخدام الرموز 0008 *

---

start: mov AL, BL

≡

Jump start

* إذا كنا نعلم بهذا المثال أن start تقابل 000A يمكنه حذف start من المثال ونضع مكانها 000A ⇐

⇒ Jump 000A

ولكنه التعامل مع start أسهل.

---

* LABLE:

* يجب أن يبدأ بـ @ , $, _ , ? أو بحرف ولا يجوز أن يبدأ برقم

DATA1 ✓
@DATA1 ✓
2DATA1 X

Ex (3-2):

0100 ---
0103 --- } → the first instruction take 3-byte

0109
010B } → take 2-byte.

* Mov.

Destination, source
(Dest) (src)

same size.

8-bit	8-bit
16-bit	16-bit
32-bit	32-bit

* Ex. MOV AL, CL

5H 3H

⇒ 3H 3H

MOV CX, AL
MOV AL, CX

⇒ syntax error [mixed-size]

---

* Figure (3-2): BX = 0300H

MOV AX, BX

AX = 0300H

* يأخذ قيمة BX وينسخها على AX.

* دائماً يجب كتابة الجواب (بالهكسا) حتى لو أعطيت الأرقام Decimal

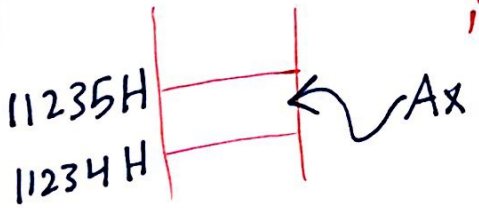
MOV CH, 3AH

* يسمى [immediate]

CH = 3AH

MOV [1234H], AX

* It will take AX and put it in the offset (1234H)



EA = 10000H + 1234H = 11234H / 11235H

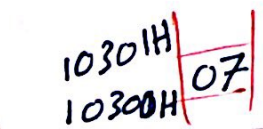
MOV [BX], CL

16-bit 8-bit

BX = 0300H

EA = 10000H + 0300H = 10300H

* هنا مع أنها الأحجام مختلفة ولكنه أي شيء داخل أقواس يصبح لذلك memory [BX] ⇒ offset وليست (reg on Destination.)



⟸ (7) [illegible]

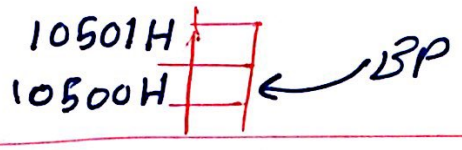
* figure 3-2:

MOV [BX + SI] , BP

Base index Base pointer

EA = DS * 10H + Base + Index

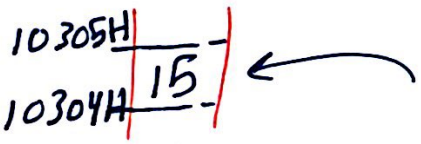
= 10000H + 0300H + 200H = { 10500H / 10501H



MOV CL , [BX + 4H]

EA = DS * 10H + Base + relative

= 10000H + 0300H + 4H = { 10304H / 10305H



* لو افترضنا قيمة CL هي 15H ⇐

CL = 15H

MOV ES, DS ✗ (Not allowed seg. to seg.)

MOV CS, AX ✗ (Not allowed since the destination can't be a segment (CS)).

MOV AX, CS ✓ (allowed).

* Immediate addressing: it isn't for the destination we can use it just for source.

MOV CH, 3AH ⇒ CH = 3AH

MOV 3AH, CH (invalid) [since the destination can't be immediate]

* إذا كان حجم src أقل من حجم dest ينسخ عليه ويملئ المتبقي أصفار.

MOV EAX, 1356H

EAX = 0001 3456 H

MOV AX, 100 → إذا لم يحدد هل هي Binary or Hexa ⇒ By Default (Decimal)

$(100)_{10} = (64)_{16}$

⇒ AX = 0064 H

MOV AX, 100H

AX = 0100H

MOV AL, 10110111B

⇒ The result of excution: AL = 10110111B

OR AL = B7H

* الأفضل أنه نكتبها بـ (Hexa)

MOV AL, 'A' ← ascii

AL = 41H

MOV AL, 'a'

AL = 61H

MOV AL, 5

AL = 05H

MOV AL, '5'

AL = 35H

* كيفية التمييز أنه هل AH reg. أم رقم بـ (Hexa)

MOV AL, AH ← Reg.

MOV AL, 0AH ← Number.

AL = 0AH

* (0) هو عبارة عن "Identifier" لا يدخل في الحجم.

MOV AL, 0B2H

AL = B2H

* مع أنه حجم الـ src كأنه أكبر من dest ولكن (0) identifier لا يدخل بالحجم لذلك يمكن النسخ على AL.

MOV AL, 023H

* حتى يكون identifier (0) يجب أنه يتبع بحرف لذلك في هذا المثال يدخل (0) في الحجم لأنه لم يتبع بحرف

⇒ invalid [mixed size]

since: AL (8-Bit) & 023 (16-Bit)

MOV BL, 44

$(44)_{10} = (00101100)_2 = (2C)_{16}$

BL = { 00101100B, OR, 2CH }

MOV AX, 44H

AX = 0044H

AX = 44 ⇒ X

AX = 44H ⇒ X

MOV AX, 'AB'

in Ascii: 'B' 'A'

low high

⇒ AX = 9291H

B A

MOV AL, 'AB' ⟶ X (mixed size)

MOV AX, '4B' AX = 9234H

MOV ESI, 12 ESI = 0000000CH

MOV [1234H], AX assume AX = 0ABCDH

EA = DS * 10H + offset

= 10000H + 1234H

= { 11234H, 11235H }

11235	AB
11234	CD

0008H Number DB 15H

MOV AL, [0008H]

* (0008H) هي رقم offset لذلك ينظر للقيمة التي بداخلها وينسخها على (AL).

AL = 15H

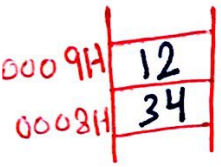
0004 DATAH DW 0AAAAH

identifier

⇒ 00AAAH ✗ (mixed size)

Let BX = 0008 , CX = 1234H

MOV [BX], CX ⟺ same MOV [0008H], CX



* After excuation BX will not change ⇒ BX = 0008H

MOV [DI], 10H (ambiguous)

⇐ حجم DI غير معروف

⇒
MOV Byte PTR [DI], 10H
MOV Word PTR [DI], 10H

* These two instructions are correct since we know the size of DI.

MOV [AX], BX (invalid) AX, EX, DX can't be offset.

↳ it can be for BP, DI, SI, BX

MOV AX, [BX] ✓

MOV [EAX], BX ✓ (offset can be any extended reg. except ESP)
EDX

---

* Example (3-7):

MOV AX, ES:[046CH]

EA = (ES)*(10H) + offset = (0)*(10H) + 046CH = 046CH

* Base-Plus-Index addressing:

Ex. MOV DX, [BX + DI]

BX → Base, DI → index.

$\Rightarrow EA = seg * 10H + Base + index.$

- we use this type for two-dimensions array

{ Base → beginning location for memory array.
Index → for an element in the array.

---

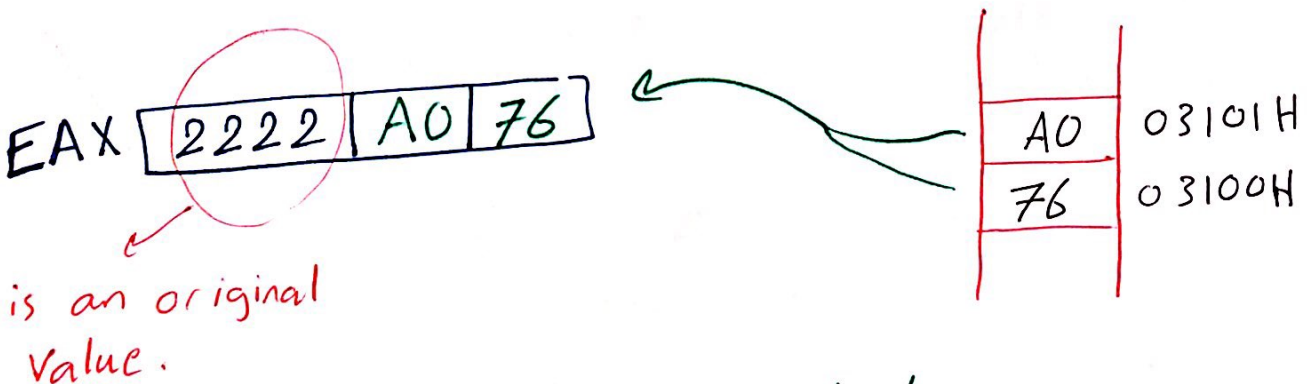
* Register Relative Addressing:

Ex. MOV AX, [BX + 1000H]

$\Rightarrow EA = seg * 10H + Base + relative.$

$\Rightarrow$ if BX = 0100H , DS = 0200H

$\Rightarrow EA = 02000H + 0100 + 1000 = \begin{cases} 03100H \\ 03101H \end{cases}$



* we have to know that changing in the value of AX ⇒ will change the value of (EAX)

EAX = 2222A076H

* Base relative-plus-index:

Ex. MOV AX, [BX + SI + 100H]

$\Rightarrow$ EA = seg * 10H + Base + Index + Relative.

MOV AL, [BX + 1000H]

This example can be written in another form if we know that: 0009 File DB 1000H

$\Rightarrow$ MOV AX, File [BX]

---

* Scale index addressing:

MOV Reg, [Base + f * Index]

(Base: 32-bit reg.; f: 1, 2, 4, 8)

EA = seg * 10H + Base + (f * index)

MOV CL, [EBX + 4 EDX] EA = DS * 10H + EBX + 4 * EDX

MOV CL, [BX + 2*EDI]

(wrong instruction)

since BX is a 16-bit register.

MOV AX, [EBX, 5*EDI]

(wrong instruction)

since the scale factor = 5

MOV AX, [4 * EDI]

(valid instruction)

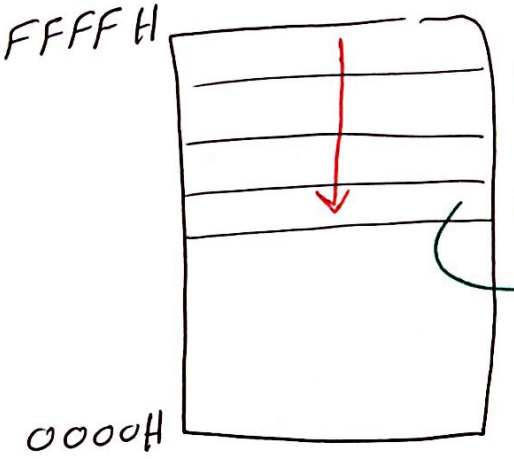
it will consider that the Base has (0) value.

MOV AL, [EBX + 2 * EDI + 2]

we don't need to know it's a hexa or decimal value since (0 → 9) is the same for both systems.

---

* stack Mode:



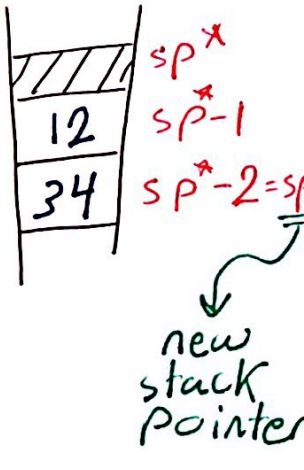
* مع كل إضافة لـ DATA قيمة (sp) تنقص بمقدار (2 OR 4 Byte)

---

* PUSH instruction:

if BX = 1234 H (high 12, low 34)

PUSH BX



* يتم تطبيقه من stack Mode في (Little indian) كما هو لا يختلف عن DATA seg.

---

* POP instruction:

⇒ Can't be for immediate.

PUSH Word Ptr [BX] → ✓ 2 Byte.

PUSH DWord Ptr [BX] → ✓ 4 Byte.

PUSH Byte Ptr [BX] → X → since PUSH & POP can used just for word or Dword

PUSH [BX] → Ambigious.

Example (3-15):

high low

AX = 1000 H

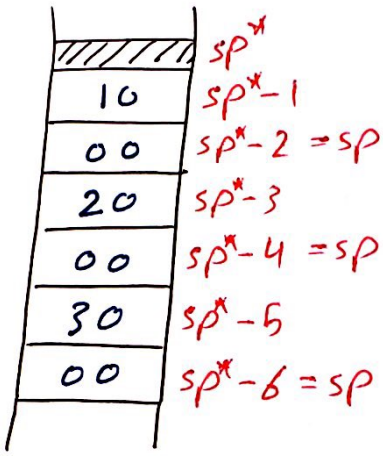
BX = 2000 H

CX = 3000 H

PUSH AX

PUSH BX

PUSH CX



(last in - first out)

⇒

POP AX

AX = 3000H

sp = sp* - 4 (Top stack)

POP CX

CX = 2000H

sp = sp* - 2 (Top stack)

POP BX

BX = 1000H

sp = sp* (Top stack)

Example (slide 67):

(a) EA = seg * 10H + offset (sp)

= 35000 + FFFE

= 44FFE

(c) upper range:

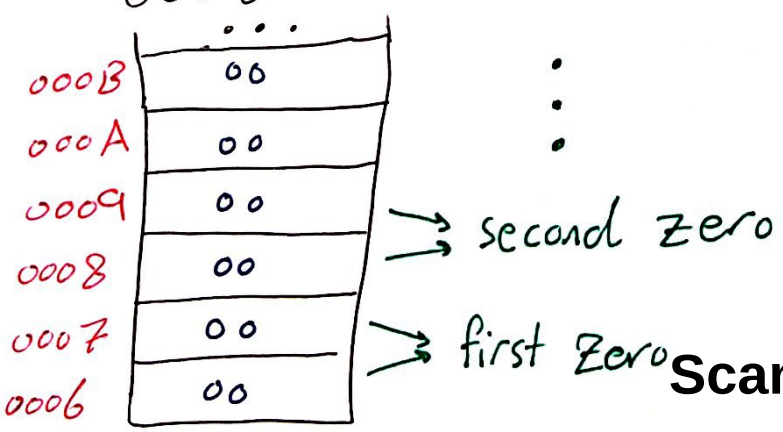
we can write it:

3500:FFFF

OR: 44FFF

0006 MARKs DW 0,0,0,0,0,0,0,0

(8-elements)



for (200 elements):

```
MARKs   DW   200 DUP(0)

Table1  DW   10 DUP(?)  => 10 words uninitialized.
Name1   DB   30 DUP('?') => 30 bytes initialized (Ascii)

matrix  DW   10 DUP(5 DUP(0))
```

same as:

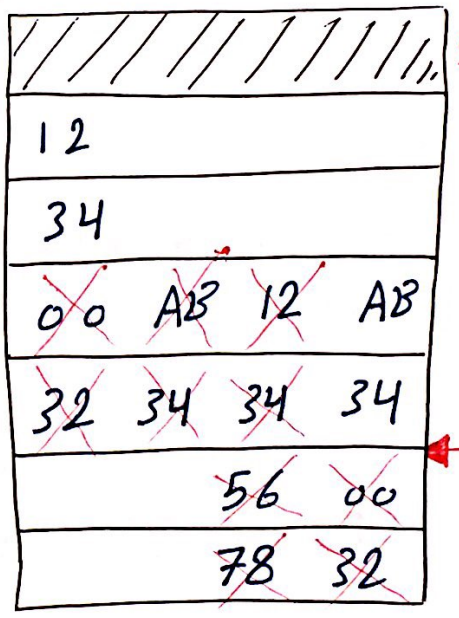
```
matrix  DW   50 DUP(0)
```

---

* Example:

```
DATA1 DD   12345678H
MOV SP, 3F22H
MOV AX, 1234H
MOV DX, 0ABCDH
MOV DI, 232AH
MOV SI, 50
PUSH AX
PUSH 32H
POP CX
MOV DL, AL
PUSH DX
POP BX
PUSH DATA1
POP DX
PUSH SI
POP DATA1
PUSH BX
```

(Hexa) تتم تحويله لـ = 0032H



* The last value of SP = 3F1EH

* Series of changing in this example:

CX = 0032H

DL=34H ⇒ DX = AB34H

BX = AB34H

DX = 5678H

DATA1 = 1234H

#CH4#

MOD

R/M (memory) ⇐
- 00 → No displacement.
- 01, 10 → There is a displacement.

R/M (Register) ⇐
- 11

Ex. for (00): MOV AX, [BX] (indirect reg.)
MOV AX, [BX+DI] (Base-index reg.)

* The difference between (01) & (10) MOD:

(01) ⇒ Byte. ex. → MOV AL, [BX + 10H] (relative) ← Byte
MOV AL, [BX+SI+10H]

(10) ⇒ Word. → word.
ex. MOV AL, [BX + 1000H]
MOV AL, [BX+SI+1000H]

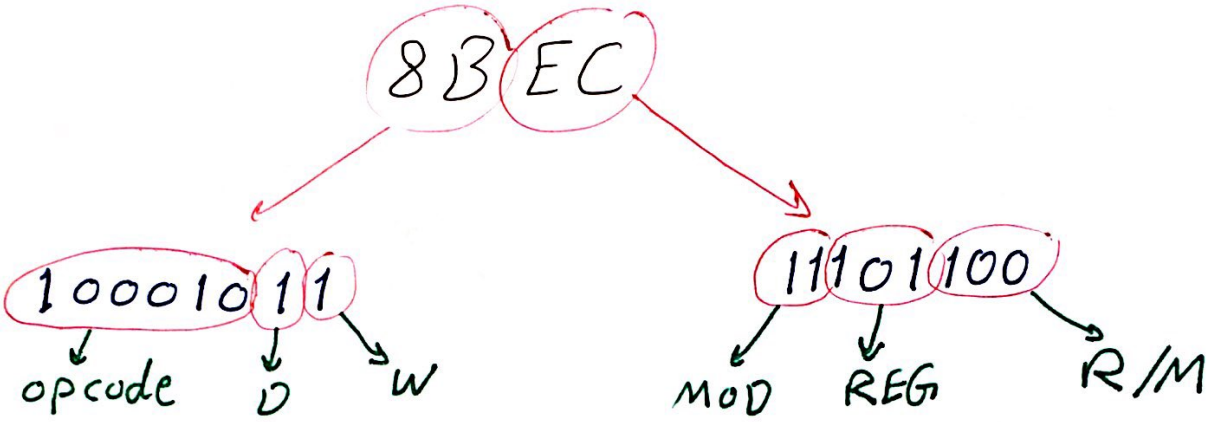
* (immediate / scale addressing):

There is No specified MOD for them

⇒ They depend on instruction.

* figure (4-4):

(MOV BP, SP) ≡ (8BECH)



opcode = 100010 ⇒ MOV

D = 1 ⇒ Transfer to Reg.

W = 1 ⇒ Word.

* from the table in slide (7):

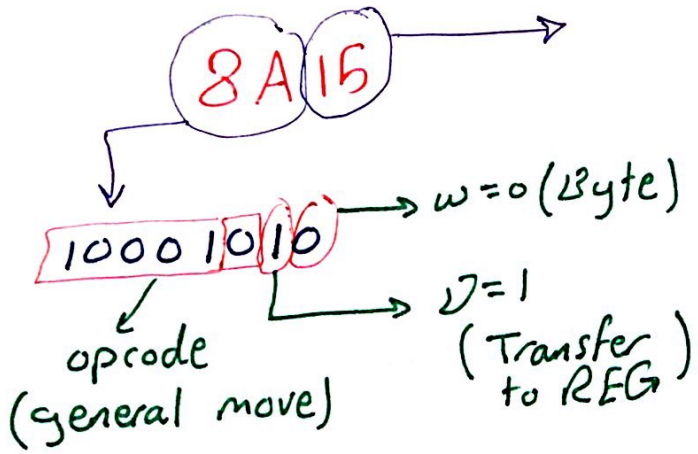
from REG = 101 we know that it is (BP).

and from R/M (100) ⇒ (SP).

⇒ 8BEC ≡ MOV BP, SP

* figure (4-5):

- Transfer from (8A15H) To (MOV DL,[DI]).



since it is a general move
so we have:
MOD-REG-R/M

15 ⇒ 00010101

MOD = 00
REG = 010
R/M = 101

* since the MOD is (00) so there is No displacement and we know also that the R/M is memory.

⇒ The form of instruction will be:

MOV REG, Memory

* from the tables: 010 ⇒ DL , 101 ⇒ [DI]

MOV DL,[DI]

- Transfer from (MOV DL,[DI]) To (8A15H).

-No immediate, No disp., No relative ⇒ general move

opcode ⇒ 100010

Now we need to know about D & w:
since the destination to Reg ⇒ D=1 , and DL is a Byte ⇒ w=1

⇒Now we have (10001010) = 8A

Next page

*since the source is a memory and There is No disp. we can know that MOD = 00

from the general move we have: MOD - REG - R/M

REG (DL) = 010 (from table)

R/M [DI] = 101

$\Rightarrow$ 00 010 101

(15)

$\Rightarrow$ 8A15 H

---

*figure: (4-6) MOV [1000H], DL

[1000H] → R/M, DL → Reg.

opcode: 100010

*now we want D & W: D=0, W=0

D=0 → since we transfer from reg.

W=0 → since DL is a Byte

(10001000)

88

$\Rightarrow$ MOD - REG - R/M

here we have the special addressing mode $\Rightarrow$ MOD = 00 & R/M = 110

since Reg DL so Reg = 010

00010110

16

The form will be:

(opcode-D-L)(MOD-Reg-R/M)(low disp)(high disp)

$\Rightarrow$ 88160010 H

* figure (4-7):

88 56 00 H

88 → opcode-D-W

56 → MOD-Reg-R/M

00 → disp.

88

⇓

10001000

General move

D = 0 (from Reg)

W = 0 (Byte)

Now we know the form:

MOV R/M, Reg.

56

⇓

01010110

MOD

R/M (memory) with disp. one Byte.

R/M [BP]

Reg. (DL)

⇒ MOV [BP+00H], DL

---

* figure (4-9): MOV Word Ptr [BX+1000H], 1234H

⇒ Type: move immediate with memory.

⇒ 1100011 opcode

Now for D & W:

D = 1 (always D = 1 for this type since the destination can't be immediate)

W = 1 since it is word Ptr.

⇒ MOD-Reg-R/M

10 - 000 - 111

10 → R/M (memory) with disp. 2Byte.

000 → since we don't care for the reg, we have immediate.

111 → BX

⇒ C7 87 00 10 34 12 H

* لو حذفنا الـ disp. في المثال السابق الاختلاف سيكون بإلغاء الـ disp من الحل و MOD = 00

C7 07 34 12 H

* immediate with reg.:

MOV AX, 0AB CDH

لو كانا الرقم Decimal نحوله Hexa.

opcode:

1011 w rrr

⇒ 1011 (1) 000 → reg (AX)

w=1 (word)

⇒ BB CD AB H

BB: low imm. | CD AB: High imm.

⇒ the Backward for (BB CD AB H)

BB

1011 (1) 000 → reg (Ax)

w=1 (word)

immediate with reg.

⇒ MOV REG, imm.

⇒ MOV AX, 0ABCDH

MOV EAX, 0AB CD H

since EAX (32-bit) ⇒ immediate = 0000AB CD H

10111000

⇒ B8

* we have to notice that EAX (32-bit) so we have 66H prefix.

⇒ 66 BB CD AB 00 00 H

* Segment MOV inst. :

MOV BX, CS

opcode :

100011d0

⇒ 100011(0)0 → d=0 (from reg)

8C

⇒ MOD - SEG - R/M

11 - 001 - 011

R/M (reg) | CS | BX

⇒ CB

⇒ 8CCB H

⇒ Backward: 8CCB H

8C ⇒ 10001100

Type: move seg.

d=0 (from reg) ⇒ seg.

⇒ MOD-SEG-R/M

CB: 11 001 011

R/M (reg) | CS | BX

⇒ MOV BX, CS

---

* 32-bit addressing:

MOV EAX, [EBX + 4*ECX]

→ since there is reg (EAX) 32-bit ⇒ 66H prefix

→ since there is reg (EBX) in offset ⇒ 67H prefix

Type: general move.

100010dw

d=1 → (src → reg)

w=1 → (32-bit)

⇒ 8B

MOD - Reg - R/M

00 - 000 - 100

R/M (memory) | EAX | scaled (from SIB)

No disp

04

⇒ for the scaled:

*4 ⇒ (10)
Base ⇒ EBX
Index ⇒ ECX

SS - index - Base
10 - 001 - 011
⇒ 8B

⇒ 67 66 8B 04 8B H

---

* Backward: 6766 8B 04 8B H

- 6766: 32-bit reg in offset. / 32-bit reg
- 8B: قد تكون disp. أو immediate أو scaled نعرف ذلك أثناء التحويل.

8B:
10001011
(general mov) ⇒ MOV reg-R/M

04 ⇒ 00 000 100
- 00: No disp.
- 000: EAX
- 100: R/M (mem.) (scaled)

for the scaled (8B):
SS - index - Base
10 00 10 11
- 10: x4
- 001: ECX
- 011: EBX

⇒ MOV EAX, [EBX + 4 * ECX]

---

* لو أخذنا على المثال السابق داخل (1000H) offset ⇐ الاختلاف سيكون:

67 66 8B 04 8B 00 10 H

MOV [1000H], AL

Type:
direct using accumulater.

$\Rightarrow$ 1010000 dw
w=0 (Byte)
d=0 (from reg)

Here: No need for MOD-reg-R/M

$\Rightarrow$ A00010 H

* لو كانت AX بدلاً من AL الجواب A10010 H
* لو كانت EAX بدلاً من AL الجواب 66 A10010 H

* Load Effective Address:

0008 DATA1 DW 1234H

MOV BX, DATA1
BX = 1234H

MOV BX, offset DATA1
BX = 0008H

$\Rightarrow$ LEA REG, MEM $\Rightarrow$ Reg = offset

* The form of LEA:

L (seg) — DS/SS/ES/FS/GS

Dest., SRC — Dest: Reg (16-32 bit); SRC: mem.

LCS BX, [DI] (invalid) — cs can't be changed.

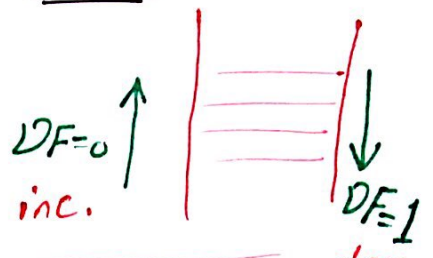
* String inst.:

MOV string (MOVS)

هي الـ inst. الوحيدة المسموح فيها أن يكون memory ← Dest & SRC

* Direction Flag:

DF → 0 (auto increment) ⇒ CLD
DF → 1 (auto decrement) ⇒ STD



MOV [DI], 15H

⇒ EA = DS * 10H + offset

↓ By Defualt.

MOVS [DI], 15H

⇒ EA = ES * 10H + offset

---

* Load string (LODS):

DF=0 → SI = ±1 → DF=1

(Byte) 1-	LODSB	AL = DS:[SI]	, SI = ±1
(word) 2-	LODSW	AX = DS:[SI]	, SI = ±2
(Dword) 3-	LODSD	EAX = DS:[SI]	, SI = ±4

consider:

(Byte) 4-	LODS List	AL = DS:[SI]	, SI = ±1
(word) 5-	LODS DATA1	AX = DS:[SI]	, SI = ±2
(Dword) 6-	LODS FROG	EAX = DS:[SI]	, SI = ±4

(From Array To Accumulator)

---

* Store String (STOS):

Byte 1-	STOSB	ES:[DI] = AL	, DI = ±1
word 2-	STOSW	ES:[DI] = AX	, DI = ±2
Dword 3-	STOSD	ES:[DI] = EAX	, DI = ±4

consider:

Byte 4-	STOS List	ES:[DI] = AL	, DI = ±1
word 5-	STOS DATA3	ES:[DI] = AX	, DI = ±2
Dword 6-	STOS DATA4	ES:[DI] = EAX	, DI = ±4

(From Accumulator To Array)

* لا يتم استخدام REP مع LODS :

- لأن التكرار عليها لا يستفاد منه لأنه عند نسخ قيمة على Accumulator سيستمر بتغيير القيم المخزنة على Accu. لذلك سيصبح بدون فائدة التخزين عليه.

---

*(MOVS) move string :

(Byte) 1 - MOVS B ES:[DI] = DS:[SI] , SI, DI = ±1

(word) 2 - MOVSW ES:[DI] = DS:[SI] , SI, DI = ±2

([word) 3 - MOVSD ES:[DI] = DS[SI] , SI, DI = ±4

consider:

Byte 4 - MOVS Byte1, Byte2 ES:[DI] = DS[SI] , SI, DI = ±1

word 5 - MOVS word1, word2 ES:[DI] = DS[SI] , SI, DI = ±2

Dword 6 - MOVS Dword1, Dword2 ES:[DI] = DS[SI] , SI, DI = ±4

ملاحظة مهمة جداً :

Byte1, Byte2 ⇐ لا تعني أنه الحجم هو Byte ولكنه هي كاسم فقط

ونعرف الحجم من التعريف الخاص بهما

Byte1 DB...

Byte2 DB---

---

* In string (INS):

(Byte)	1 - INSB	ES:[DI] = [DX]	, DI = ± 1
(word)	2 - INSW	ES:[DI] = [DX]	, DI = ± 2
(Dword)	3 - INSD	ES:[DI] = [DX]	, DI = ± 4
DB	4 - INS list	ES:[DI] = [DX]	, DI = ± 1
DW	5 - INS DATA4	ES:[DI] = [DX]	, DI = ± 2
DD	6 - INS DATA5	ES:[DI] = [DX]	, DI = ± 4

* Out string (OUTS):

Byte	1 - OUTSB	[DX] = DS:[SI]	, SI = ± 1
word	2 - OUTSW	//	, SI = ± 2
Dword	3 - OUTSD	//	, SI = ± 4
DB	4 - OUTS DATA7	//	, SI = ± 1
DW	5 - OUTS DATA8	//	, SI = ± 2
DD	6 - OUTS DATA9	//	, SI = ± 4

* Miscellaneous DATA:

- Exchange: (XCHG)

XCHG Dest, Src
R/M R/M

src = old dest.
Dest = old dest

ex. DL = 3H, CL = 7H

XCHG DL, CL

DL = 07H & CL = 03H

* Conditions:
- If dest & src reg. must be same size.
- can't be from mem. to mem.
- Can't be Immediate for both dest & src.

✗ CHG AX, DS (invalid)

since seg. takes only MOV/PUSH/POP.

✗ CHG AL, 15H (invalid)

since Not allow to use immediate.

✗ CHG [DI], DATA1 (invalid)

since can't be from mem → mem.

MOV BL, CL
MOV CL, AL
MOV AL, BL

⇒ ✗ CHG AL, CL

↙ This inst. can represent by using 3 mov.

- Translate: (XLAT)

* There is Two inst. must be done before using XLAT

① MOV AL, index

② MOV BX, offset Array

⇒ where:
AL = index Number
0 → (n-1)
BX = offset Array

⇒ EA = seg (DS) * 10H + BX + AL

XLAT لا يجوز جمع reg مختلفات بالحجم إلا في
8-bit + 16-bit
يمكن أن نجمع مختلف بالحجم

Example (4-9) slide 33 :

we have 10 elements ⇒ MOV AL, 4

index Num (0 → 9)

The fourth element is 66H

⇒ AL = 66H

* we can represent XLAT using:

```
SUB   AH, AH    ---- (AH = 0)
MOV   SI, AX    ---- (SI = 00X)
MOV   AL, [BX+SI]
```

---

* IN & OUT:

Form: IN Dest, src → input device → immediate (8-bit) / DX (16-bit)

Dest → Accumulator { AL, AX, EA }

Form: OUT Dest, src → Acc. (AL / AX / EAX)

Dest → output device → imm. (8-bit) / DX (16-bit)

MOV AL, 15H → AL = 15H

IN AL, 15H → Bring Byte from an input device has the Number 15H.

⇒ DX = 1234H

MOV AX, DX → AX = 1234H

IN AX, DX → Bring Word from input device has DX.

IN AL, 1234H (Invalid) can't be a 16-bit immediate.

* figure (4-18):

OUT 19H, AX
- DATA bus (D0 - D15)
- Address bus (A0 - A15)
- Port control ($\overline{IOWC}$)

(D0 - D15) و (A0 - A15) حسب حجم Accum.

$\overline{IOWC}$: out ⇐ inst. وقت write استخدمنا لو كانت
Read نستخدم ⇐ IN

IN AL, P8 where P8 = imm. (8-bit)

* لو لم يعرف P8 ستكون العبارة خاطئة
لأنه لا يوجد P8 من Hexa

IN AL, DL (Invalid) → since we put on src DX or ½ byte immediate.

OUT AL, DX (Invalid)
↳ since the Accum. must be on src and DX on dest.

---

* MOVSX & MOVZX :

* Form:

MOVSX ⟹ dest, src
MOVZX ⟹ reg, R/M

- signed ← MOVSX
- unsigned ← MOVZX

16 ← 8
32 ← 16
64 ← 32

MOVZX
Dest | src
...0000 | src

MOVSX
Dest | src
MSB
...0000 | 0 | src
...1111 | 1 | src

MOVZX DL, AL (invalid) → since src & dest are same size.

MOVSX EDX, AL (invalid)
↳ since the size here 32 ← 8

MOVSX EDX, AL

⇒ we can make it write:

- MOVSX BX, AL
- MOVSX EDX, BX

- MOVZX DATA, AX (invalid)
- we can't use mem. in this inst.

Ex. AL = A3 H

① MOVZX BX, AL

BX = 00AL H

② MOVSX BX, AL

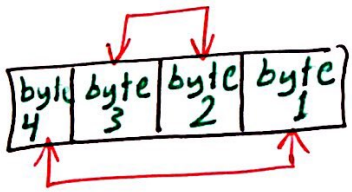
BX = FFA3 H

---

* BSWAP:

⇒ Form:

BSWAP (32-bit Reg)



ex. EAX = 12 34 56 87 H

⇒ BSWAP EAX ⇒ EAX = 87 56 34 12 H

---

* Conditional Move: (CMOV)

⇒ Form:

CMOV dest, src

(Reg), (R/M)

16-bit ← 16-bit

32-bit ← 32-bit

* Condition:
  - True ⇒ Dest = src.
  - False ⇒ No move.

---

* لأن CMOV ظهرت فيما بعد يجب تعريفها داخل البرنامج حتى يميزها (assembler) من خلال وضع

.586 OR .686

* Types of conditional move:

1] Flag Conditional Move:

* Zero flag (ZF):

CMOVE or CMOVZ → ZF=1 move. / ZF=0 No move.

CMOVNE or CMOVNZ → ZF=1 No move. / ZF=0 move.

* carry flag (CF):

CMOVC → CF=1 move. / CF=0 No move

CMOVNC → CF=0 Move. / CF=1 No move.

* sign flag (SF):

CMOVS SF=1 (move)

CMOVNS SF=0 (move)

* parity flag (PF):

CMOVP (CMOVPE) → PF=1 (even) move.

CMOVNP (CMOVPO) PF=0 (odd) move

* overflow flag (OF):

CMOVO OF=1 move

CMOVNO OF=0 move.

2] Unsigned CMOV:

(Above):

CMOVA → CF=0 and ZF=0

CMOVAE → CF=0

(Below):

CMOVB → CF=1

CMOVBE → CF=1 OR ZF=1

(Equal):

CMOVE → ZF=1

CMOVNE → ZF=0

3] Signed CMOV:

CMOVG CMOVL CMOVE

CMOVGE CMOVLE CMOVNE

CMOVA AL,CL
CMOVB DATA1,BX
CMOV AX,CX
} (invalid) → since 8-bit. → can't be mem. → No such move.

Ex. (slide 46):

3800
AA00 -
1100

CF=0

CMOVA we check for CF=0 & ZF=0

⇒ DX=0AA00H

CMOVAE DX,BX ⇒ DX=0AA00H

* EQU:

```
TEN EQU 10

MOV AL,10   <=>   MOV AL, TEN
```

↳ (TEN) is an immediate value Not a memory.

```
0000  DATA1  DB  15H
0001  DATA2  DW  1234H
```

↳ By default data2 take this value

* By using ORG we change it:

```
0000  DATA1  DB  15H
ORG 100H
0100H DATA2
```

* PROC & ENDP:

```
<Name> PROC <Near,far>   uses AX BX CX
   ____
   ____
   ____
   ____
   RET
<Name> ENDP
```

Near → as local. far → as global

uses: تستخدم للمحافظة على القيم.

global means: can use by any prog.

local means: can use by current prog.

#CH5#

* Addition:

* form:

1 - ADD
2 - ADC
3 - XADD
4 - INC

Dest , src
R/M , R/M/imm.
(same size)

8	8
16	16
32	32
64	64

* effects all flags (C, Z, S, P, O, A)

1 - Dest = dest + src

2 - dest = dest + src + cf

3 - dest = dest + src & src = old dest.

4 - dest = dest + 1

* form:

operand
R/M (directive)
8-bit
16
32
64

* effects all flags except Cf.

ADD DS, AX (Invalid) → Not allowed for seg.

MOV DS, AX (Valid)

① ADD:

Ex. DL = 33H

ADD DL, 12H

$$\begin{array}{r} 12H \\ 33H + \\ \hline 45H \end{array}$$

⇒ cf = 0, zf = 0, of = 0, Af = 0, Sf = 0, Pf = 0

45 → 01000101 (odd)

DL = 45H

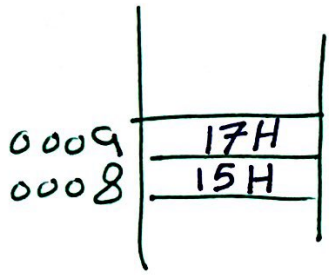
Ex in slide (9):

Let: 0008 NUMB DB 15H, 17H

```
MOV DI, offset NUMB
MOV AL, 0
ADD AL, [DI]
ADD AL, [DI+1]
```

→ DI = 0008

→ AL = 0



$$\Rightarrow \begin{array}{r} 15H \\ 0 + \\ \hline 15H \\ 17H + \\ \hline 2C \end{array}$$

⇒ AL = 2C H

in this example we can replace { MOV AL, 0 / ADD AL, [DI] } by one instruction

⇒ MOV AL, [DI] ⇒ Nothing will change.

Ex (5-5) slide (11):

AX ⇒ scaled (*2)

for EAX ⇒ scaled (*4)

for RAX ⇒ scaled (*8)

Ex. given: $AL = (27)_{10}$, $BX = 0008H$

and | 27H | 0008H | $\Rightarrow$ $(27)_{10} = (1B)_H$

ADD [BX] , AL

$$\begin{array}{r} 1B \\ 27 + \\ \hline 42H \end{array}$$

$\Rightarrow$ | 42H | 0008H

if he asked about BX $\Rightarrow$ $BX = 0008H$ (No change on its value)

$\Rightarrow$ cf=0 / Af=1 / sf=0 / of=0 / zf=0 / pf=1

Ex. $AL = FFH$ ADD AL,5

$$\begin{array}{r} FF \\ 5 + \\ \hline 104H \end{array}$$

$\Rightarrow$ $\boxed{AL = 04H}$ cf=0 of=1

---

(2) ADC :

figure (5-1): for example take:

ADD AX, CX
ADC BX, DX

$N_1 =$ 1234 (BX) 5678 (AX)

$N_2 =$ 567A (DX) BCDE (CX)

[1]

6 8 A F (BX) 1 3 5 6 (AX)

$\Rightarrow$ $BX = 68AFH$

$AX = 1356H$

cf=0

---

(3) XADD :

ex. AL=03H, CL=07H

XADD AL, CL $\Rightarrow$

$\boxed{AL = 0AH}$

$\boxed{CL = 03H}$

④ INC :

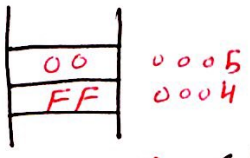
ex. INC AL

⇒ AL = AL + 1

ADD AL, 1 ⇒ effect cf.

INC AL ⇒ don't effect cf.

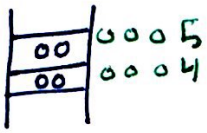
Ex. BX = 0004H ,



INC Byte Ptr [BX]

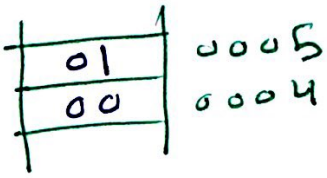
FF
1 +
100

⇒



INC Word Ptr [BX]

⇒



---

* # Examples from past papers #

① IF AX = 8345H and CX = 9FDAH , BX = 037AH and DX = 00D7H find the value of AX, BX , CX, DX , CF After execution the following code :

ADD AX, CX
ADC BX, DX

```
    1
  037A   8345
  00D7   9FDA   +
 ------------
  0452   231F
   ↓      ↓
   BX     AX
```

⇒

AX = 231F H
BX = 0452 H
CX = 9FDA H
DX = 00D7 H
CF = 0

② If BL = 2DH and DL = 4AH what is the value of BL & DL after execution : XADD BL, DL

⇒

BL = 77 H
DL = 2D H

* SUBTRACTION:

1 - SUB dest = dest - src
2 - SBB dest = dest - src - cf
3 - CMP dest - src
4 - DEC dest = dest - 1

* form:
Dest , src
R/M , R/M/imm.
(same size)
→ No mem. to mem.
→ No segment register.
* effects all flags.

* form:
operand
R/M
8, 16, 32, 64
* effects all flags except cf.

① SUB:

Ex. BX = A2C3H , CX = 8D37H

```
 A2C3
 8D37 -
------
 158C
```

SUB BX, CX ⇒ BX = 158C H

zf=0 / cf=0 / Af=1 / sf=0 / pf=1 / of=0

MOV CH, 22H
SUB CH, 44H

```
 22
 44 -
----
 DE
```

⇒ CH = DE H

cf = 1
sf = 1
zf = 0
pf = 1

Af = 1
of = 0
↳ لاختلاف الإشارة بين الرقمين

22 + (-44)

(2) SBB :

Ex. given : cf=0 , AH=15H , AL=12H

SBB AH, AL

$$\begin{array}{r} 15 \\ 12 - \\ 0 - \\ \hline 3 \end{array}$$

⇒ AH = 03H

⇒ the last value of Cf = 0

---

(3) CMP :

Ex. AL = 72H

CMP AL, 10H

AL = 72H

$$\begin{array}{r} 72 \\ 10 - \\ \hline 62\ H \end{array}$$

↳ Pf = 0 (odd)
Zf = 0
Cf = 0

JAE Next.

↳ (Jump if above or equal)

↳ case(1) : if AL was 72H

CMP AL, 10H ⇒ cf = 0 True condition

⇒ so it will go to the label (Next) somewhere in the code.

case(2) : if AL was 05H

CMP AL, 10H ⇒ Cf = 1 wrong condition.

---

(4) DEC :

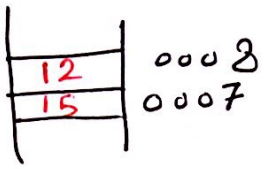
Ex. BH = A0H

DEC BH ⇒

$$\begin{array}{r} A0 \\ 1 - \\ \hline 9FH \end{array}$$

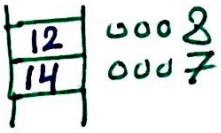
⇒ BH = 9FH

Ex. BX = 0007H , given



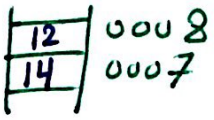
DEC Byte Ptr [BX]

15
1-
14

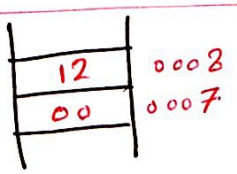


DEC Word Ptr [BX]

1215
1-
1214



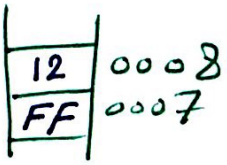
---

if the memory was 

in the last example:

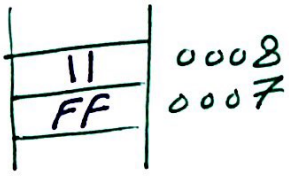
DEC Byte Ptr [BX]

00
1-
FF

⇒ 

DEC Word Ptr [BX]

1200
1-
11FF

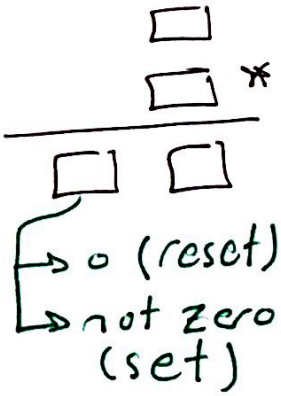
⇒ 

CMP XCHG ⇒ X

---

* Multiplication:

unsigned: MUL
signed: IMUL
operand (multiplier) R/M



result = multiplicand * multiplier
↓ Acc.
↓ R/M

16-bit = 8-bit * 8-bit
AX Acc. R/M
AL/AH

32-bit = 16 * 16
DX,AX AX R/M

64-bit = 32 * 32
↓ ↓ ↓
EDX,EAX = EAX * R/M

Ex. AL = 03H , CH = 04H

MUL CH

$3 * 4 = (12)_{10}$

$= (C)_{16}$

⇒ AX = AL * CH

⇒ **AX = 000CH** , cf = 0 , of = 0

Ex. MUL RCX

⇒ RDX - RAX = RAX * RCX

Example (5-13) ⇒ slide (33):

```
MOV BL, 5
MOV CL, 10
MOV AL, CL
MUL BL
MOV DX, AX
```

⇒ BL = 5H
CL = 0AH
AL = 0AH

AX = AL * BL

```
  5
 10 *
 50
```

↓ $(32)_{16}$

⇒ **AX = 0032 H**

⇒ **DX = 0032 H**

Examples from Past Papers

① given: AX = 0345H, SI = 01DAH, BX = 037A H, DI = 05D7H

what is the value of AX, BX, SI, DI, Cf after execution of:

```
SUB AX, DI
SUBB BX, SI
```

```
 0345
-05D7
------
 0D6E
```

AX = FD6EH , **cf = 1**

```
 037A
 01DA -
    1 -
------
 019F
```

⇒ **BX = 019F H**

No change on DI & SI.

⇒ **cf = 0**

---

② What is the value stored in AX, CL, DX, BL after execution:

```
MOV BL, 5FH
MOV CL, 20H
MOV AL, CL
MUL BL
MOV DX, AX
```

AX = AL * BL

```
  5F
 20 *
------
  00
 0BE0 +
------
 0BE0
```

BL = 5F H

CL = 20 H

AX = 0BE0 H

DX = 0BE0 H

* Types of IMUL :

① IMUL one operand.

* form:

IMUL operand

⇒ double width = Acc. * op.

double width	Acc.	op.
16	8	8
32	16	16
⋮	⋮	⋮

② IMUL two operands.

*form:

IMUL Dest, src
Reg, R/M
16 - 16
32 - 32
→ must be same size.

⇒ Dest = Dest * src

ex. IMUL CX, BX

⇒ CX = CX * BX

⇒ All the following instruction are (invalid):

IMUL [BX], CX ⇒ since the dest can't be memory.

IMUL AL, CL ⇒ since IMUL take the sizes 16-bit or 32-bit.

MUL CX, BX ⇒ since MUL take one operand.

③ IMUL two operands with immediate.

* form:

IMUL Dest, src
Reg, imm.
16 — 8
16
or
32 — 8
32

⇒ Dest = Dest * src

Ex.

IMUL BX, 1234H (valid)
IMUL BX, 12H (valid)
IMUL BL, 12H (invalid)
↳ dest. takes only 16-bit or 32-bit.

** Always the immediate in the instruction IMUL must be one byte (8-bit) OR same size with the reg.

④ IMUL three operands.

* form:

IMUL Dest, src1, src2
Reg, R/M, imm.
16 — 16 — [8, 16]
32 — 32 — [8, 32]

⇒ Dest = src1 * src2

imm — [one byte, same size.]

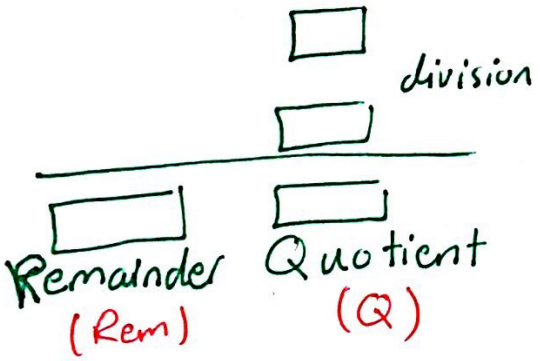
Ex. Determine if the following inst. valid or invalid:

IMUL EBX, ECX, 13H (valid)
IMUL DX, [SI], 1234H (valid)
IMUL [BX], DX, 15H (invalid) ⇒ since dest can't be mem.
IMUL BX, CX, DX (invalid) ⇒ since src2 must be imm.

---

* Division:

result = Dividend / Divisor



* form:

unsigned ← DIV
← IDIV signed
operand
Divisor
R/M
8/16/32

* No affect on flags.
* No use for immediate in Division.

* for the signed division:

dividend	divisor	Q	Rem
+	+	+	+
+	−	−	+
−	+	−	−
−	−	+	−

⇒ Rem has the same sign of the dividend.

AH	AL	=	Acc.	/	OP.
Rem	Q		AX		R/M
(8-bit)			16-bit		8-bit

DX	AX	=	DX-AX	/	R/M
Rem	Q		32-bit		16-bit
(16-bit)					

EDX	EAX	=	EDX-EAX	/	R/M
Rem	Q		64-bit		32-bit
(32-bit)					

* 8-bit division : ⇒ (means divisor 8-bit)

⇒ How to divide CL/BL:

*Unsigned:

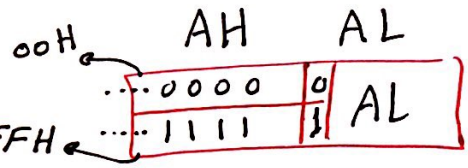
① MOV AL, CL
MOV AH, 0 ⇒ or (SUB AH, AH)

② MOVZX AX, CL

* signed:

① MOVSX AX, CL

② CBW (convert Byte to Word)



AX ⇒ DIV BL or IDIV BL

Ex. Given AX = 11H, CL = 8H

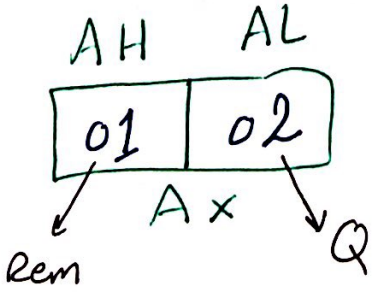
DIV CL

$(11)_{16} = (17)_{10}$

$(8)_{16} = (8)_{10}$

$17/8 = \overset{Q}{2}\ \overset{Rem}{1}$

⇒



Ex. Given AX = 21H , BL = FBH

IDIV BL

$(21)_{16} = (33)_{10}$

$(FB)_{16} \Rightarrow$ 11111011 (sign number)

⇓

00000101 ⇒ +5

$\Rightarrow (FB)_{16} = (-5)_{10}$

33/-5 = Q: -6, Rem: +3

$(-6) \Rightarrow$ +6: 00000110

⇓

11111010 $\Rightarrow (FA)_{16}$

$(+3) \Rightarrow (03)_{16}$

$\Rightarrow$ AH: 03, AL: FA

AX → Q: FA, Rem: 03

---

* DIV Byte Ptr [BP]

* لو لم يقم بتعريف حجم [BP] ⇐ invalid

⇒ (ambiguous)

Example from Past papers

What is the value stored in AL, AH, BL, BH after execution the following code:

MOV AL, 0F5H
MOV AH, 00H
MOV BX, 5A08H
DIV BL

} → AX = 00F5H

BX = 5A08H

⇒ AX (Rem | Q) = AX / BL

$(F5)_{16} = 11110101 = (245)_{10}$

$(08)_{16} = (8)_{10}$

Q Rem

245/8 = 30 5

$(30)_{10} = (1E)_{16}$, $(5)_{10} = (05)_{16}$ ⇒ AX: AH 05, AL 1E

AL = 1EH | AH = 05H | BL = 08H | BH = 5AH

* 16-bit division:

=> How to divide CX/BX:

DX AX

* Unsigned:

```
MOV AX, CX
MOV DX, 0  => or (SUB DX, DX)
```

* signed:

CWD (convert word Dword)

DX		
0000	0	AX
FFFF	1	

=> DIV BX

or IDIV BX

Example 5-15 slide (46):

	Q	Rem
-100/9 =	-11	-1

-11 ⇒ +11 ... 0000 1011

-11 ... 1111 0101

FFF5 H

-1 ⇒ +1 ... 0001

... 1111 1111

FFFF H

=> Ax = FFF5 H

Dx = FFFF H

* 32-bit division:

=> How to divide ECX/EBX:

EDX EAX

* Unsigned:

```
MOV EAX, ECX
MOV EDX, 0 => or (SUB EDX, EDX)
```

* Signed:

CDQ (convert Dword to Quadword)

EDX		
00000000	0	EAX
FFFFFFFF	1	

=> DIV EBX

or IDIV EBX

* Example (see slide 49):

13/2 ⇒ Q = 6, Rem = 1 ⇒ multiply remainder by 2

BL = 2

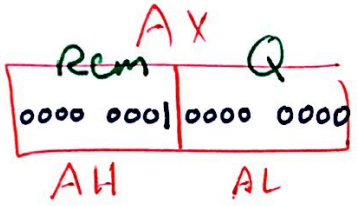
1 ⇒ AH

ADD AH, AH ; AH = 2

CMP AH, BL ; AH - BL

2 - 2 = 0 ⇒ Cf = 0

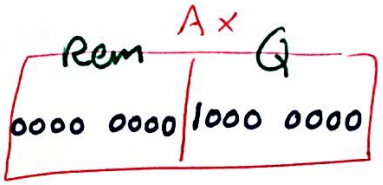
MOV AL, 0



This number in binary in Decimal will be : (256)

⇒ 256/2 = 128

⇒



⇒ 0110 • 1000 0000 ⇒ 6.80

---

* BCD Arithmetic:

decimal 12 ⇒ Binary 1100 (4-bit) ⇒ BCD packed 0001 0010 (8-bit) ⇒ BCD unpacked 0000 0001 0000 0010 (16-bit)

* DAA (Decimal Adjust after Addition).
* DAS (" " " Subtraction). → doesn't take operand.

⇒ Acc. (AL)

* BCD addition Correction:

1- ADD numbers as Hexa.
   Dest ⇒ AL (result) Hexa.

2 - DAA (convert result hexa ⇒ BCD)

a) If AL and OF > 9H or AF = 1
   ⇒ Add 06H ⇒ AF = 1

b) If AL and F0 > 9H or CF = 1
   ⇒ Add 60H ⇒ CF = 1

c) If Both
   ⇒ Add 66H

* Note that 0F means the least significant four bits.
  and F0 means " most " " ".

Ex. (slide 56)

```
 27H
 34H +
 ----
 5BH
```

⇒ The wanted value : 27 + 34 = 61

DAA:
```
 5B
 06 +
 ----
 61H
```

* we add 06 since the Least significant four bits > 9H.

```
 (1)
  29H
  69H
 ----
  92H
DAA 06H +
 ----
  98H
```

```
  52H +
  61H
 ----
  B3
  60 +
 ----
cf 1 ← 13
```

* we add 60 since the most significant four bits > 9H.

⇒ AL = 113 ✗

⇒ AL = 13H
  & CF = 1

①
39 H
87 H +

C 0
66 +
---
cf 26
1

AL = 26 H
CF = 1

* Here both least & most significant four bits > 9 H.

30
24 +

54

⇒ DAA it is a correct answer so no addition but we can add 00

54 + 00 = 54 H

Ex (5-18) slide (54): wanted value : 1234 + 3099 = 4333

ADL
1234 H
3099 H +
---
① ①
42 CD
00 66
---
43 33

(DAA) → we use it twice once for (CD) and once for (42).

---

* BCD Subtraction Correction:

1- SUB number as Hexa.
Dest ⇒ AL (result) Hexa.

2- DAS (convert result Hexa ⇒ BCD)

a) If AL and OF > 9H or AF = 1
⇒ SUB 06H AF=1

b) If AL and F0 > 9H or Cf=1
⇒ SUB 60H Cf=1

c) If Both
⇒ SUB 66H Cf=AF=1

*ex. slide (58)

```
 71H
 43H -
 ----
  2E
```

wanted answer

71 - 43 = 28

⇒ DAS :

```
  2E
  06 -
 ----
  28H
```

AL = 28H

---

① →

```
  32
  41 -
 ----
  F1
 60 -
 ----
  91
```

(DAS)

wanted answer: → Not in BCD.

32 - 41 = -9

⇒ 91 - 100 = -9

since there was a borrow.

Cf = 1

```
 46H
 46H +
 ----
  8B
  06 +
 ----
  91H
```

⇒ Cf = 0

---

* Ascii Arithmetic :

```
 5 ⇒ 35
 4 ⇒ 34
 ---
 9   69
```

→ actual but wanted (39)

* Ascii Correction after addition :

1- ADD Numbers as Hexa.

AL = result in Hexa , AH = 0

2 - AAA (convert Hexa ⇒ BCD unpacked)

a) AL = AL + 6 , AH = AH + 1

↳ if AL and oF > 9H or AF = 1

b) AND AL 0F → to remove tags.

3 - ADD tag (BCD → ascii)

OR AL, 30H

ADD AL, 30H

(OR AX, 3030H)

* Ex. slide (62):

A = 35 H ⇒ 35 + 32 = 67 + 00 = 67 (AAA)

AND 0F → 07

OR 30 → 37 H

wanted answer '7' ⇒ 37H

AX = 0031

0031 H + 0039 H = 006A (AAA)

1 + 06 = 01 70 (AND) 0F = 01 00

⇒ ADD: 01 00 + 30 30 = 31 30

AL = AL + 6

AH = AH + 1

* Ascii Correction after subtraction:

1- SUB Numbers as Hexa

⇒ AL = result in Hexa, AH = 0

2- AAS

a) if AL AND 0F > 9H or AF = 1

⇒ AL = AL - 6, AH = AH - 1

b) AND AL 0F → remove tag 3

Ex. '5' - '2'

'5' → 35

'2' → 32

35 - 32 = 03 - 0 (AAS)

03 AND 0F

03 OR 30

'3' ← 33 Hexa

Ex. slide (63):

'8' - '9'

38 - 39 = FF (AH) - 06 (AAS)

FF AND 0F = 09

09 OR 30 = 39 Hexa

means that it's negative answer

answer + 10 ⇒ 09 - 10 = -1

* Ascii Correction after Multiplication:

1- Remove tag from multiplication operands

⇒ AND AL, 0F H
AND 0P, 0F H ⇒ (Ascii → BCD unpacked)

2- Mul as Hexa (BCD unpacked ⇒ Hexa)

Ax = result (hexa)

3- AAM ⇒ AL = (AX) mod (0A $(10)_{10}$)
AH = (AX)/ (0A $(10)_{10}$)

⇒ Rem = AL
Q = AH (BCD unpacked)

4- Add tag (BCD ⇒ Ascii)

⇒ OR Ax, 3030 H

* Note: for BCD Numbers apply steps ② & ③

---

* 2A H /0AH ⇒ 42/10 ⇒

AH	AL
04	02

30 30
34 32 → '2'
'4' ←

Ex. '4' * '5'

4 * 5 = 20 ⇒ 00010100 = 0014H = Ax

AAM ⇒ 0200H
OR 3030
Ax = 3230 H

---

34 AND 0F → 04
35 AND 0F → 05

04 * 05 = 20 = 0014 H

*Ascii Correction after Division:

1- Remove tag (Ascii ⇒ BCD)
⇒ AND AX, 0F0FH
AND OP, 0FH

2- AAD (Ax → BCD ⇒ Hexa)
Ax = AL + (AH * 0AH)

3- DIV (result ⇒ BCD) ⇒ AL = Q
AH = rem

4- ADD tag (BCD ⇒ Ascii)
OR AX, 3030H

*Note: for BCD Numbers apply steps ② & ③ just.

*Ex. slide (67):

59/8 ⇒ 3539
0F0F] AND
0509 H (AAD)

AL + (AH * 0AH)
09 + (05 * 0A) = 0032H

(DIV): 59/8
⇒ rem AH 03, Q AL 07 ⇒ (OR) 3030H ⇒ AX: AH 33 (Rem.), AL 37 (Q)

⇒ in the last example: if he asked to find AL & AH we can directly do:

59 ⇒ 3539
Rem Q
59/8 ⇒ 03 07

03 ⇒ 33H
07 ⇒ 37H

⇒ AH = rem = 33H
AL = Q = 37H

* Basic Logic Instructions

dest = dest (and) src ← 1 - AND
dest = dest (or) src ← 2 - OR
dest = dest ⊕ (XOR) src ← 3 - XOR
Dest • (AND) src ← 4 - TEST
↳ No answer

form

Dest , src
R/M , R/M/imm.
[same size]

CF = OF = 0
(others result)

5 - NOT
6 - NEG

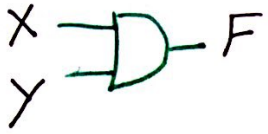
form

operand
R/M
8/16/32/64

No affected flags.

(others result)
CF = 0 or 1
↙
when src = 0

① AND:



$X \cdot 0 = 0$ (mask/clear)

$X \cdot 1 = X$ (filter)

X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

$F = X \cdot Y$

XXXX XXXX Unknown Number

1111 0000 (AND)

XXXX 0000

(pass) (Mask)

Ex. if AL = 2AH

AND AL, 40H

⇒ 0010 1010

0100 0000 AND

0000 0000

AL = 00H
ZF = 1
PF = 1 (even)
SF = 0

Ex. What is the clears bit after execution:

AND DI, 4FFFH

4FFF ⇒ 0 1 0 0 1111 1111 1111

(15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0)

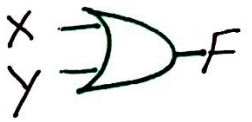
⇒ clears bit = 12, 13, 15

Ex. AND RAX, 128

$2^7 = 128$ ⇒ bit(7) will be 1 and the rest zeroes.

---

② OR



$F = X + Y$

$X + 0 = X$

$X + 1 = 1$

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

XXXX XXXX

1111 0000 (OR)

1111 XXXX

Ex. AL = 0D4H

OR AL, 15H ⇒

```
1101  0100
0001  0101   (OR)
1101  0101
```

AL = D5H
ZF = 0
PF = 0 (odd)
SF = 1 (neg.)

Ex. OR RBP, 1000H

1000H ↓

0001 0000 0000 0000 (12)

⇒ The bit that will be set = 12.

Ex. OR DH, 0A3H

⇒ set = 0, 1, 5, 7

(7) (5) (1) (0)
1 0 1 0 0 0 1 1

Ex. OR EBP, 10

(A)H

.... 0000 0 (3) 1 0 (1) 1 0

set = 1, 3

OR EBP, 10H

.... 0 0 0 (4) 1 0000

set = 4

---

(3) XOR:

F = X ⊕ Y

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	0

```
XXXX  XXXX
1111  0000   (XOR)
X̄X̄X̄X̄  XXXX
```

X ⊕ 0 = X

X ⊕ 1 = $\bar{X}$

Ex. AL = 5CH

XOR AL, 23H ⇒

```
0101  1100
0010  0011   XOR
0111  1111
```

AL = 7FH
PF = 0 (odd)
ZF = 0
SF = 0

Ex: XOR ESI, 100 → 01100100 → (64)H

....00 011 0 01 00

inverts (Complement) = 2, 5, 6

Ex: XOR ESI, 100H

....0001 0000 0000

inverts = 8

Ex. slide (82):

OR CX, 0600H

0000 0110 0000 0000

set = 9, 10

AND CX, FFFCH

1111 1111 1111 1100

clear = 0, 1

XOR CX, 1000H

0001 0000 0000 0000

inverts = 12

---

(4) Test:

→ with JZ / JNZ

JZ → jump if zero

JNZ → jump if not zero

* Here No value will store in the dest.

---

(5) & (6) ⇒ NOT & NEG:

NOT:

XXXX XXXX (NOT)

$\bar{X}\bar{X}\bar{X}\bar{X}$ $\bar{X}\bar{X}\bar{X}\bar{X}$

NEG:

XXXX XXXX (NEG)

$\bar{X}\bar{X}\bar{X}\bar{X}$ $\bar{X}\bar{X}\bar{X}\bar{X}$ + 1

Ex: CH = 2FH

NOT CH

0010 1111 (NOT)

1101 0000

⇒ CH = D0H

NEG CH

0010 1111 (NEG)

1101 0001

⇒ CH = D1H

---

* Shift & Rotate:

* shift:

logical left ← 1 - SHL

Arithmetic left ← 2 - SAL

Logical right ← 3 - SHR

Arithmetic right ← 4 - SAR

* Rotate:

No carry ← 1 - ROR

carry ← 2 - RCR

No carry ← 3 - ROL

carry ← 4 - RCL

form ⇒ Dest, SRC

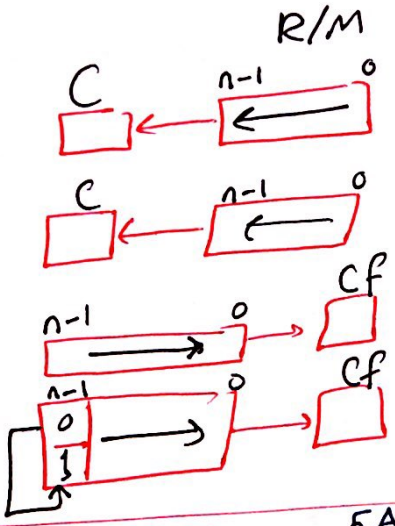
R/M, imm/reg

CL

* Shift or Rotate dest. by the number of bits in SRC.

* Shift:

(unsigned) SHL
(signed) SAL
(unsigned) SHR
(signed) SAR



These two instructions do the same work.

Ex. AL = 5A H, CL = 3

SHL AL, CL

Cf 0 1 0 1 1 0 1 0 ⇒ 0 1 0 1 1 0 1 0 0 0

⇒ AL = D0 H

Ex. state if invalid or Not:

SHL DL, AH
SHR DX, CX
} → (Invalid) ⇒ since src just imm. or CL

SHR AX, 20 (valid) ⇒ AX = 0000H

SHL [BX], CL (Invalid) ⇒ ambiguous (since CL doesn't determined the size.)

SHL Byte Ptr [BX], CL (valid)

---

*Note: Segment shift Not allowed (segment just in MOV, PUSH, POP)

Ex. DX = 8D0CH ⇒ 1000 1101 0000 1100

[1] SAR DX, 10

(15) 1000 1101 0000 1100 (0)

↳ Cf

0

* آخر قيمة خرجت تكون هي قيمة Cf

⇒ 1111 1111 1000 11

DX = FFE3H

[2] SHR DX, 10

⇒ 0000 0000 0010 0011

⇒ DX = 0023 H

Ex. (slide 93):

$10 \rightarrow 8 \rightarrow 2^3 \Rightarrow (3)$
$10 \rightarrow 2 \rightarrow 2^1 \Rightarrow (1)$

```
SHL  AX*, 1     ; AX = AX* . 2
MOV  BX, AX     ; BX = 2 AX*
SHL  AX, 2      ; AX = 8 AX*
ADD  AX, BX     ; AX = AX + BX
```

$= 8AX^* + 2AX^* = 10\,AX^*$

$18 \rightarrow 16 \rightarrow 2^4 \Rightarrow (4)$
$18 \rightarrow 2 \rightarrow 2^1 \Rightarrow (1)$

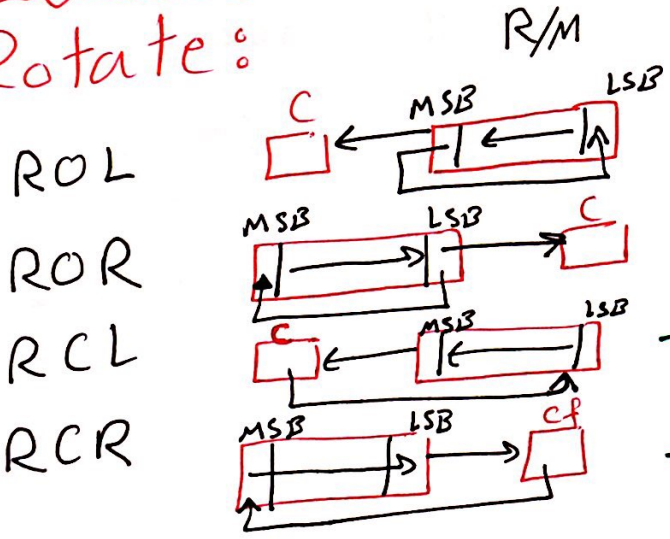
```
SHL  AX*, 1
MOV  BX, AX
SHL  AX, 3
ADD  AX, BX
```

$5 \rightarrow 4 \rightarrow 2^2 \Rightarrow (2)$
$5 \rightarrow 1 \rightarrow 2^0 \Rightarrow (0)$

```
MOV  BX, AX*    ; AX = AX*
SHL  AX, 2      ; AX = 4 AX*
ADD  AX, BX     ; AX = AX + BX
```

$= 4AX^* + AX^*$
$= 5\,AX^*$

* Rotate:

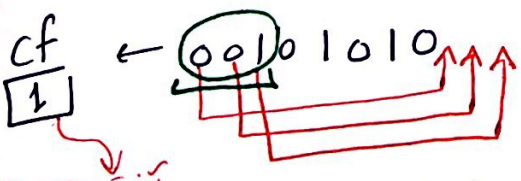


No Rotation for the carry flag (we must know the value of cf before execute the inst.

Rotation for the Carry flag.

Ex. AL = 2AH, CL = 3, Cf = 0

1 ROL AL, CL

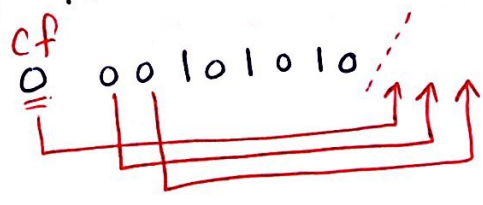


آخر قيمة دوّر Rotation لها

$\Rightarrow$ 01010001

$\Rightarrow$ AL = 51H

2 RCL AL, CL



$\Rightarrow$ (1)01010000

New cf

CF = 1

AL = 50H

Ex. AX = 203DH , CF = 1

1 ROR AX, 63

* لأن 63 عدد كبير نأخذ 63 mod 16 = 15

صار آخر bit انعملها Rot.

AX ⇒ 0010 0000 0011 1101 CF [0]

⇒ 0100 0000 0111 1010 ⇒ **AX = 407AH**

---

2 RCR AX, 63

⇒ 63 mod 17 = 12 (17: one bit for CF)

0010 0000 0011 1101 1 → CF

12-bit rotation

⇒ 0000 0111 1011 0010 [0] (New value of CF) ⇒ **CF = 0**

⇒ **AX = 07B2H**

---

* إذا كان حجم الـ imm. أكبر من حجم Reg *

- No carry → (imm) mod (size of reg)
- with carry → (imm) mod (size of reg + 1)

*Bit scan inst. :

BSR
BSF

form ⇒

Dest	src
Reg (32)	Reg (32)

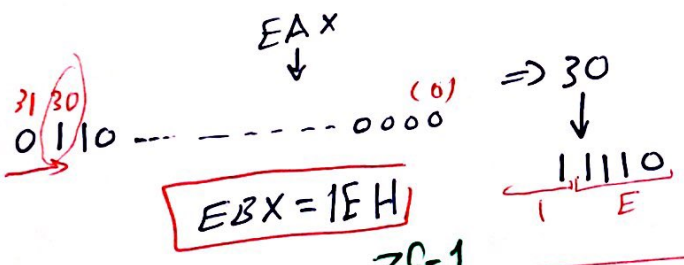
⇒ Dest = index of first bit = 1 in src
- Left → right (BSF)
- right → Left (BSR)

ZF = 1 → if (1) found.
ZF = 0 → if no found src = 0

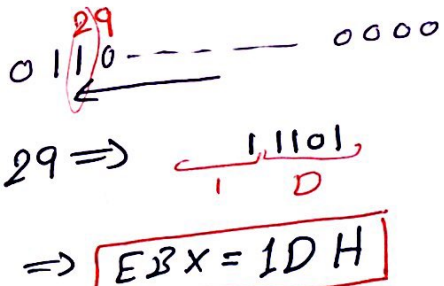
Ex. slide (99) :

EAX = 60000000 H

① BSF EBX, EAX



② BSR EBX, EAX



---

* String Comparisons :

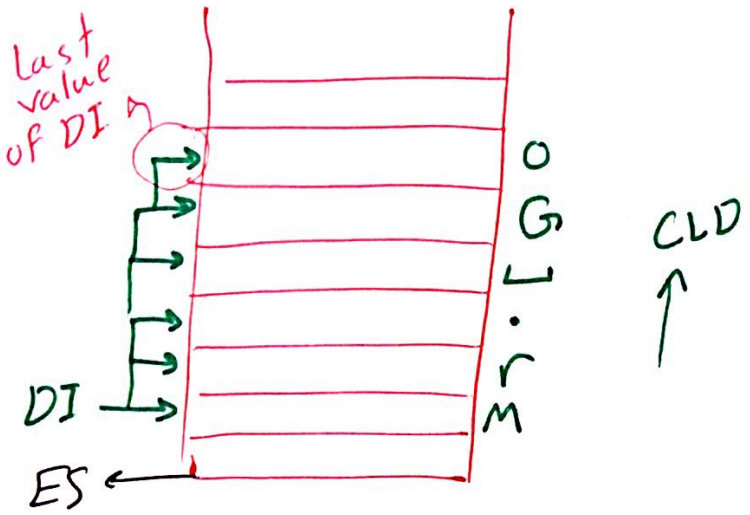
* scan string (SCAS) :

Byte 1- SCASB
word 2- SCASW
Dword 3- SCASD

ES:[DI] ? AL ; DI = ±1
ES:[DI] ? AX ; DI = ±2
ES:[DI] ? EAX ; DI = ±4

DF = 0 → DI = +1
DF = 1 → DI = -1

*Ex. slide (102):



* It will do 4 comparisons until find (G).

REPNE ⟹ Zf = 1 (means that we found the wanted value).

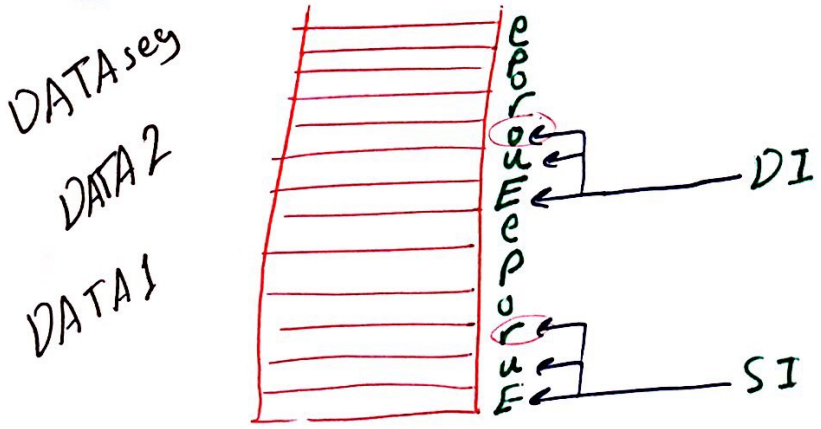
* Zf will equal to zero if we didn't find the wanted value through the loop.

---

* compare string (CMPS):

Byte	1 - CMPSB	ES:[DI] ? DS:[SI]	; DI, SI = ± 1
word	2 - CMPSW	ES:[DI] ? DS:[SI]	; DI, SI = ± 2
Dword	3 - CMPSD	ES:[DI] ? DS:[SI]	; DI, SI = ± 4

Ex. slide (105):



REPE ⟹ Zf = 0
⟹ get out from the loop.

* لو كانت الكلمتين بالمثال السابق Europe ⟸ سيكون سبب الخروج
مثال loop ⟸ انتهاء Counter Cx ⟸ Zf = 1

---

* DOS interrupt 21H: (slide 107):

DATA1 DB 6, ?, 6 DUP (0FFH)
max actual

10	11	12	13	14	15	16	17
6	-	FF	FF	FF	FF	FF	FF
	3	55 U	63 S	41 A	0D CR		

#CH6#

1 ——— .
2 . ——— * if the program executed line by line we call this program linear program, if we use call/return/jump instructions ⟹ Non linear program.
3 . ——
4 . ———
⋮
n . ———

* figure (6-1) slide (8) :

⟹ Jump Group : short / Near / Far ← * حفظ الأشكال الخاصة بكل نوع (لا تعطى بالامتحان).

* Short Jump :

form : JMP short disp ← one byte.

⟹ New IP = IP + disp

Ex. slide (10) :

10000 JMP short 04 ⟶ it will take one byte for JMP and one byte for disp so Next inst. ⟹ on offset 10002

10002 Next inst.

CS = 1000H ⟹ IP = 10002 − 10000 = 0002

⟹ New IP = 0002 + 04 = 0006H

* we could use label with JMP (easier than hexa addresses)

```
10000   JMP short 04
10002   Next inst.
10006
```

⇒

```
JMP Label
Label:
```

Ex. slide(12):

IP = 0009

⟹ New IP = 0020

To find the disp

```
0020
0009 -
-------
0017 H
```

disp = 17H

JMP short next

JMP start ⟹ IP = 0024, New IP = 0002

disp ⟹

```
0002
0024 -
-------
DE H
```

disp = DEH

---

* Near Jump:

form: JMP Near disp (2 bytes disp.)

New IP = IP + disp

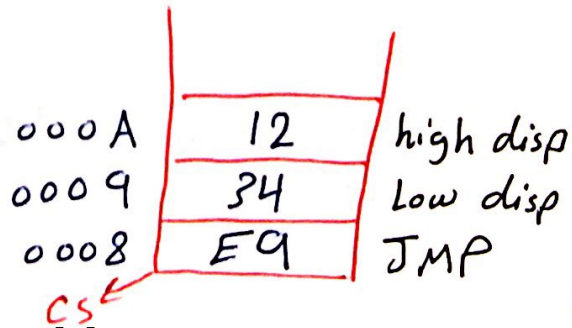
Ex. find the New IP?

0008 JMP 1234H

New IP = 123FH

⇒ 0003 Next inst

```
New IP =  0003
          1234 +
          ------
          123F H
```



* figure (6-3) slide(15):

⇒ 10000 JMP 0002H

disp = 0002 H
IP = 0003 H +
0005 H

⇒ New IP = 0005H

---

*Ex. slide (17)

JMP next ⇒ IP = 000AH
New IP = 0200H

disp

0200H
000AH −
01F6 H

high → 01 ← low.

disp = 01F6H

JMP start ⇒ New IP = 0002 H
IP = 0205 H −
disp = FDFD H

FD
FD
E9

we can write this inst. like this

E9 FD FD (low, high)
OR
E9 FDFDH

⇒ JMP next
E9 F6 01
OR
E9 01F6H

---

* Far Jump:

form: JMP Far ptr 4 Bytes

Ex:

20ABC JMP Far Ptr 10001234H
CS = 2000H
it will take 5 addresses
⇒ IP = 0ABC + 5 = 0AC1 H

New IP = 1234 H

⇒ New CS = 1000 H

EA = 11234H

10	high CS
00	Low CS
12	high IP
34	Low IP
EA	JMP

inst. ⇒ EA 34 12 00 10
JMP, low IP, high IP, Low CS, high CS

**

```
JMP Far Ptr Next
Next:
```

Another way ⟹ to write this instruction

```
JMP Next
EXTRN Next: Far
Next::
```

---

* Jumps with Register Operands:

Indirect jump [reg]
↳ takes short Near far

JMP Near 0010
JMP Next
⇒ New IP = IP + disp.

JMP REG
↳ 16-bit (Near)
↳ 32-bit (Far)

(16-bit)
New IP = Reg.

(32-bit)
New IP = Low half of reg.
New Cs = High half of reg.

Ex. BX = 0020H

0008 JMP BX
000B → Next inst
⇒ IP = 000BH

↷ New IP = 0020H

BX = 0020H
it represents New IP Not a disp.

Ex. EAX = 12345678H

JMP FAR PTR EAX

⇒ New IP = 5678H
New CS = 1234H

* Index (double-indirect jump):
  ↳ it is just for Near & Far only.

JMP [ ]
- if Near ⇒ EA = 2 bytes ⇒ represents New IP.
- if Far ⇒ EA = 4 bytes ⇒ represents New IP & New CS.

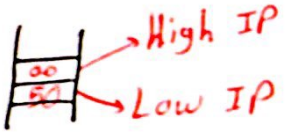
---

*Ex. given That BX = 0008H and

10	000B H
00	000A H
00	0009 H
50	0008 H

① 0006 JMP Near [BX]

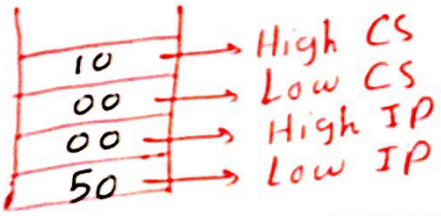
since it is Near ⇒ EA = 2 bytes.
⇒ so it will represents the New IP



⇒ New IP = 0050H

② JMP Far Ptr [BX]

since it is Far ⇒ EA = 4 bytes ⇒ representing the New IP & New CS.



⇒ New IP = 0050H
New CS = 1000H

---

* Conditional Jump: just take → short / Near (No use for Far)

⇒ form:

J(condition) Label
- If FALSE (goes to the Next inst.)
- If TRUE (goes to Label)

1 Test Flag:

JS cf=1, sf=1	JZ ≡ JE zf=1	JC Cf=1	JP ≡ JPE (even) PF=1	JO of=1
JNS sf=0	JNZ = JNE Zf=0	JNC Cf=0	JNP ≡ JPO (odd) PF=0	JNO of=0

2 Unsigned Compare: → it come with CMP dest, src

JA zf=0 & cf=0	JB cf=1	JE Zf=1
JAE cf=0	JBE cf=1 OR zf=1	JNE Zf=0

3 Signed Number Compare:

JG JGE JL JLE JE JNE

* Loop:

→ form:

```
MOV CX, Count
Label
  ____
  ____
  ____
  ____
  ____
Loop Lable.
```

Must Not contain inst. change the value of CX, since it will cause Logical Error.

Loop

- Conditional Loop.
  - Loop NE (Loop NZ)
    it will stay inside the loop when:
    CX / ECX != 0
    Zf = 0
    - LoopNED
      ECX != 0
      Zf = 0
    - LoopNEW
      CX != 0
      Zf = 0
  - Loop E (Loop Z)
    it will stay inside the loop when: CX / ECX != 0
    Zf = 1
    - LoopED
      ECX != 0
      Zf = 1
    - LoopEW
      CX != 0
      Zf = 1
- Unconditional Loop.
  - Loop CX (16-bit) / ECX (32-bit)
    - Loop D (ECX) → 16 OR 32
    - Loop W (CX) → 16 OR 32

---

in Example (slide 40):

when BL = 34H ⇒ AL = 34H ⇒ Zf = 1 ⇒ exit from the loop

LoopNE (CX != 0)

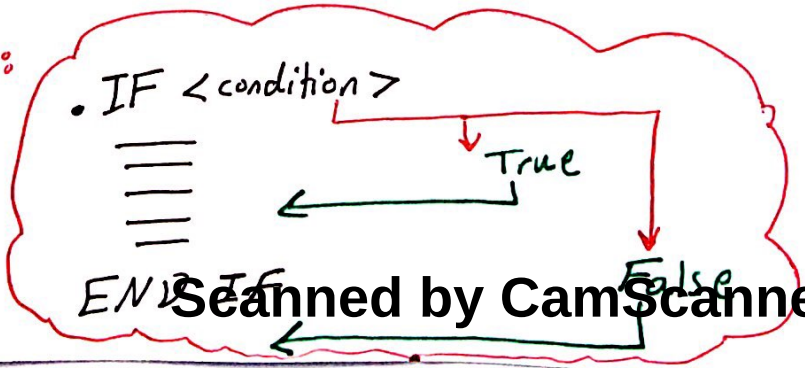
---

* Controlling the flow of the program:

IF:

1 Simple If:

⇒ form:



2 IF-ELSE:

=> form:

```
IF <condition>
    ≡≡≡  (TRUE)
EISE
    ≡≡≡  (FALSE)
END IF
```

3 Nested IF:

=> form:

```
IF   <condition>
    ≡≡≡
    EISE   IF <condition>
        ≡≡≡
        EISE
        ≡≡≡
    END IF
```

While :

①
```
. While <condition>
   ____
   ____
   ____
   ____
End W
```

Ex. ⇒

```
MOV CX,10
. While <cond.>
        ↳ CX!=0
   ____
   ____
   ____  → DEC CX
   ____
END W
```

---

②
```
. Do
   ____
   ____
   ____
   ____
. while <condition>
```

Ex. ⇒

```
MOV CX,10
. do
   ____  → DEC CX
   ____
. while <cond.>
        ↓
      CX!=0
```

---

*Notes on ① & ② :

* If the Condition is True they will do the same work.
* If the Condition is False (Do while) will execute at least one time.
* In Do while & while The Condition to continue the process of execution.

---

③
```
. Repeat
   ____
   ____
   ____
. Until <condition>
```

Ex. ⇒

```
MOV CX,10
. Repeat.
   ____  → DEC CX
   ____
   ____
. Until <cond.>
        ↳ CX=0
```

*Notes on ③ :

* In Repeat-until The condition to stop the process of execution.
* We can Reduce the size of .Until <cond.> ⇒ put: Until CXZ

* Procedures :

It take Near OR Far , No use for Short.

(Local) (Global)

* CALL :

Near Call :

⇒ form: Call 2 bytes → disp.

take 3 bytes

⇒ New IP = IP + disp.

Ex. find the New IP ?

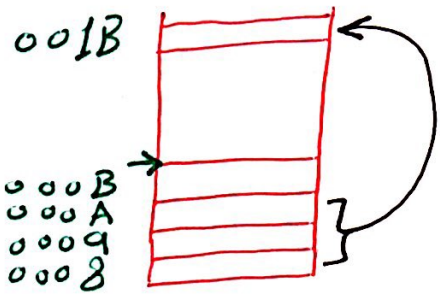
0008 Call 0010H

takes 3 bytes

000B Next inst

⇒ New IP = 000B + 0010

= 001BH



---

* figure 6-6 slide(54):

we can Know the inst ⇒ Call 0FFF

IP = 0003 , disp = 0FFF

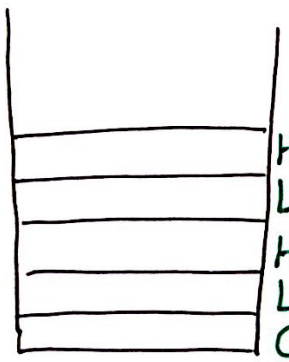
⇒ New IP = 11002H

Far Call:

form:

Call | 4 bytes

H CS | L CS | H IP | L IP



takes 5 bytes

Ex. CS = 1000H

0008 call far 20001234H

⇒ IP = 000DH

CS = 1000H

New IP = 1234H

New CS = 2000H

---

* figure 6-7 slide (57):

the inst. will be: call 11000002H

New IP = 0002H

New CS = 1100H

→ New EA = 11000 + 0002

= 11002H

procedure

Indirect Call:

BX = 2000H

Call BX ⇒ BX = 2000H (Not a disp)

⇒ New IP = 2000H

EBX = 2000ABCDH

Call EBX ⇒ New CS = 2000H

New IP = ABCDH

* Index Call :

given: BX = 0008H

20	000B
00	000A
00	0009
04	0008

Call [BX]

[1] - Near : ⇒ New IP = 0004H

[2] - Far : ⇒ New CS = 2000H
New IP = 0004H

---

* In Ex 6-16 slide (59):

Call disp → we Know the value of disp from:

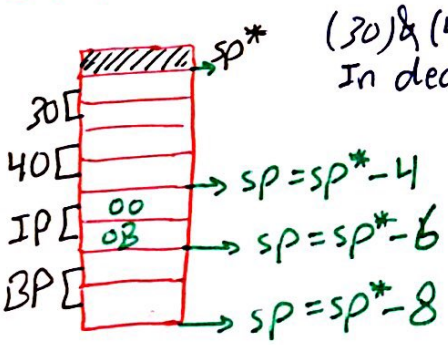
disp = New IP − IP

$$\text{disp} = \begin{array}{r} 0110\,H \\ 0107\,H \\ \hline 0009\,H \end{array} -$$

disp = 0009H

*Note that here Hexa Numbers so disp ≠ 3 (subtract as Hexa)

Ex. (6-17) slide (65):



(30) & (40) In decimal.

* AX, [BP + 4]

we didn't use BX or CX since (BP) has by defualt (SS).

New IP = 0071H

* If we want to Know the disp:

disp = 0071H − 000BH ⇒ disp = 0066H

* In this Ex. we can avoid using PUSH & POP by using:

PROC NEAR uses BP

* In this Ex. if POP BP wasn't exist it will cause Logical Error.

The End