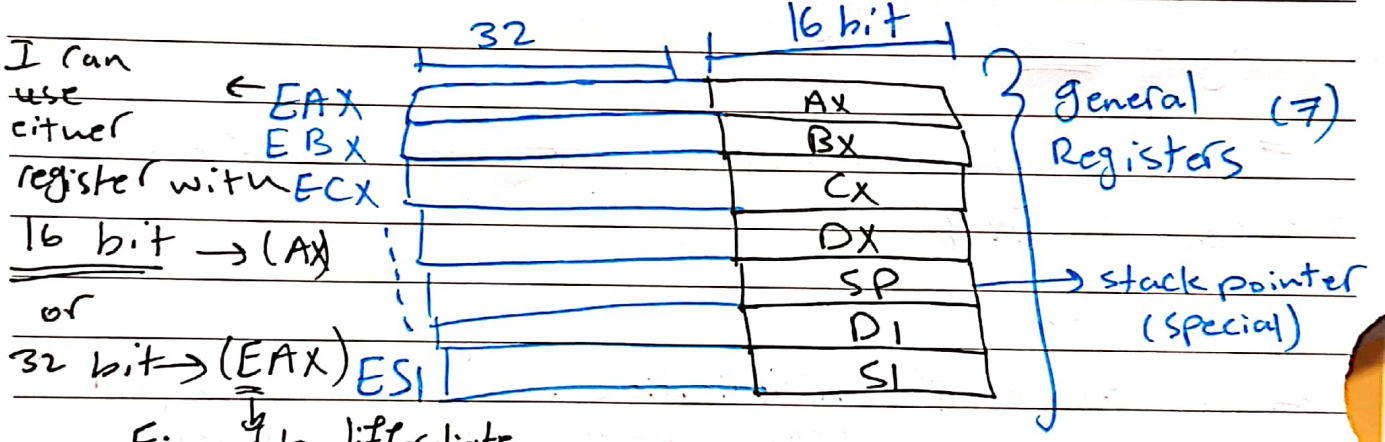
**chapter(2)

8086 upto ↗ above invisible appeared registers

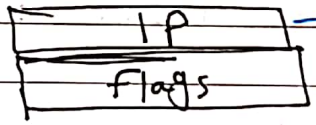
** كل معالج جديد بيجي بعده من نفس ال family يحتوي على نفس ال component آسابقة ولكن يضيف حاجة

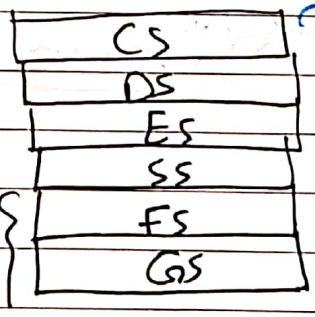
Each new processor contain same component of the previos but adds other new ones.

Example: I can't delete old register or chang its name, I can only add new registers without making any change on the old ones.



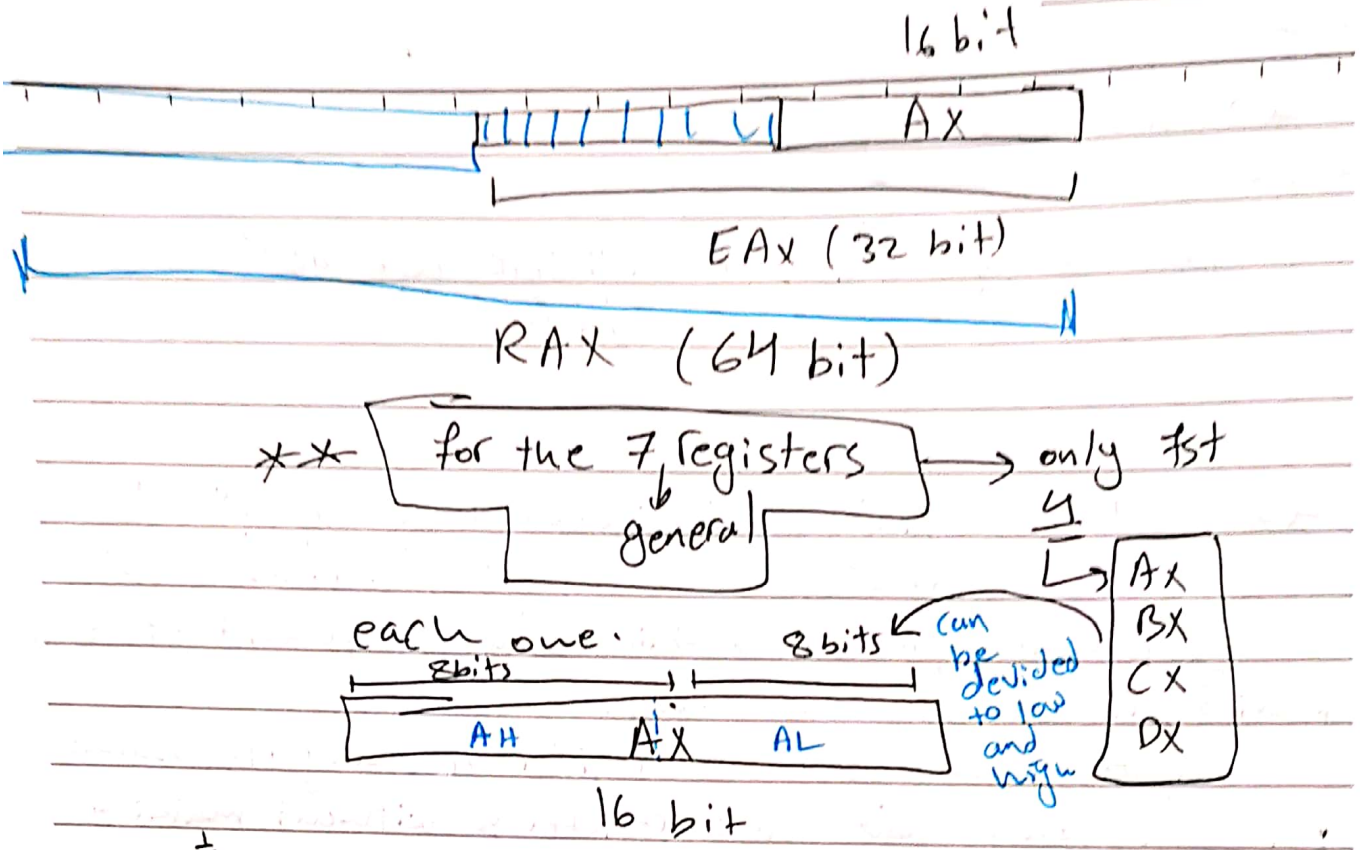
E: used to differentiate between a 16 bit or 32 bit register.





segment Registers are only 16 bit, there's no other choices

instead of having larger Registers they added two new ones (FS, GS)



* Changing the value of AX means also changing AL, AH, ** changing the value of AL, or AH means changing AX

*** but changing data in AL, ~~or AH~~ doesn't change the data in AH

**** the result of multiplication or devision is put in the accumulator (EAX) ; if its not enough in size the rest of the result will be put on DX (data register)

* 2^{50} = peta → 52 bit address bus.

** 4 Peta byte of memory → is the maximum

BX → holds offset address

CX → count → hold special instruction
for example: (loop)
↳ counter inside CX.

- BP → No 8 bit size choice.
- DI source distination.
- SI

MOV SI, DI ✓
MOV DI, SI ✓

**Source register can be used as distination and distination can be used as source. (in MOV instruction)
* in some instructions (DI) must be distination and (SI) must be source.

- R8 - R15 → all general
→ it has 8 bit size but it doesn't have two choices between high and low like the previus registers
→ only low (first 8 bits)

[0 7]

Special Registers:- RIP → the value changes automatically.
تلقائيا

Five Apple

**flags that are affected with logical & arethmitic pocesses :- ~~AC~~ A, Z, S, T, O, C, P

⟹ after making for example (add) operation between two binary numbers, I'll get the result (new number) and the value in the flags will be changed due to theirs function.

- P (Parity) → over flow not counted with the bits.
- others are defined in slides.

** Segment Registers

Purpose: Connecting each one of them on the begging of a certain segment, because any program will be downloaded randomly on the RAM.

** If the program has only one segment it will be (code) CS ; No Program without code.

** Real mode memory from 00000 → FFFFF

Segment address offset is 64 k byte from the begginning address

→ 10000 (beginning)
+ F000 (offset) → it could be ≤ F000
1F000 (actualadress) → F000 is the max

* when aprogram is randomly downloaded on RAM beginning & ending adress may be changed but the displacement or offset between them will be the segment

Five Apple

** effective address will change but the offset won't, that's why I deal with offset.

→ if the beginnig address was 10000
it will be in the segment 10000

* all beginning adresses start with zero at the LSB, so in segment we ignore the fst zero.

EA = segment * 10H + offset
↓ ↓ address
effective

put zero on right.

** Hexadecimal addition & Subtraction :-

→ add every two digits as decimal and you'll get the result X:

X:
- decimal 0-9 ≡ in hexadecimal still 0-9
- decimal 10-15 ≡ A-F
- X > 15, X ≥ 16
  ↳ then X = X - 16 = the result and carry 1

→ in subtraction :-
subtract each two digits as decimal

X =
- → 0-9 ≡ 0-9
- → 10-15 ≡ A-F

* don't forget in exam to write

A = 10

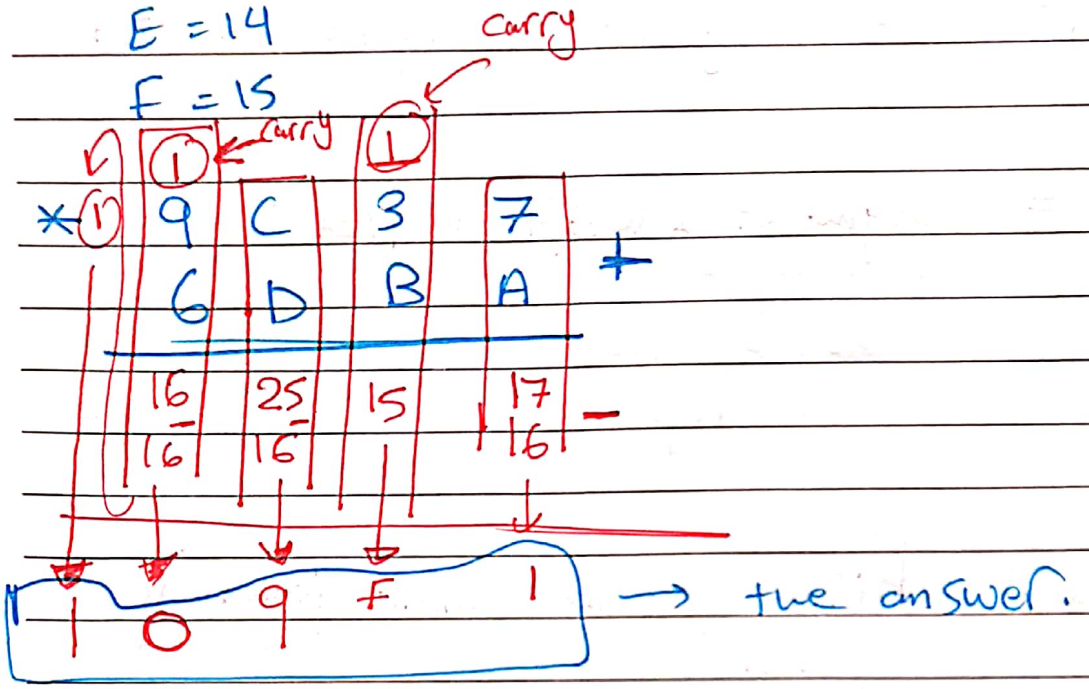
B = 11

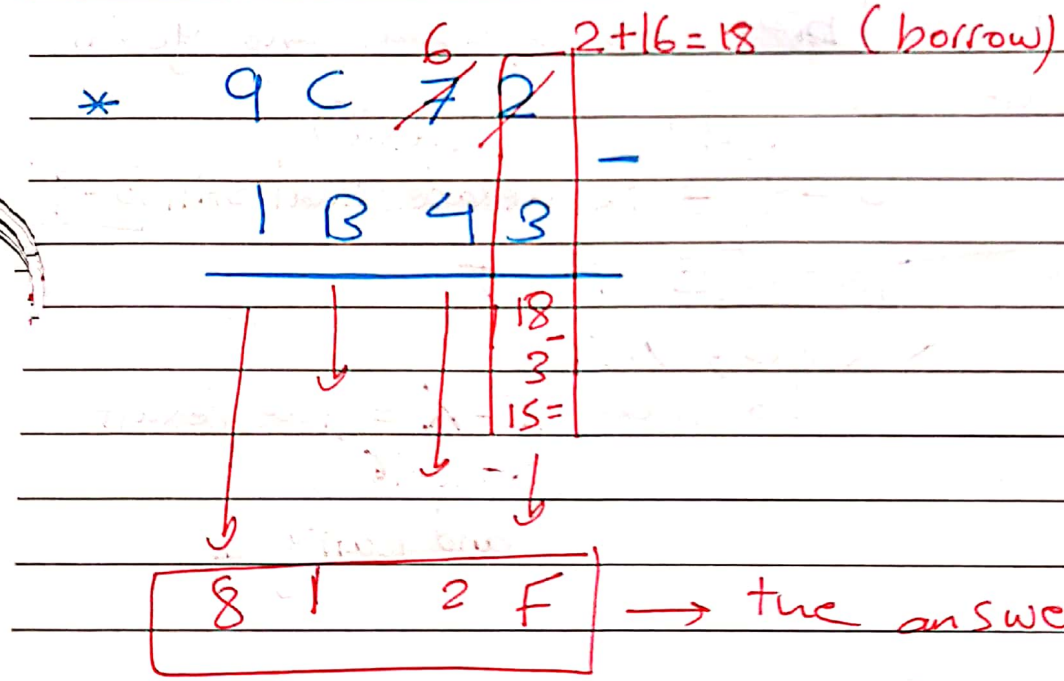
C = 12

D = 13

E = 14

F = 15

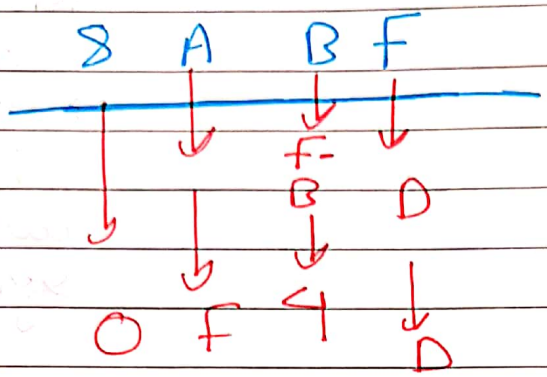




9+16 = 25
25 - A = 15 → F

0+16 = 16
16 → 15 => F

12+16 = 28
15
13 = D



0 F 4D → the answer.

* Segment * 10H + offset = effective address

→ Convert 08F1 : 0100 to a linear address :-

08F10
+ 00100
19010 → EA ≡ linear

** the address of the segment is stored on the segment Register

*** → if the code segment starts at 10000 then (1000) will be stored on CS
- what about the offset?
it will be stored on a special Register called (IP) / instruction pointer

Five Apple

* then the effective address = 10H * segment ⊕ IP

IP → offset for the next instruction of the code segment.

- SS → stack pointer (special Register)
- BP → base pointer (general)

* the default of the basepointer is (stack segment)

segment	offset	
CS	IP	16 bit.
SS	SP or BP	
DS	BX, DI, SI (distination, source.)	
ES	DI or string instructions	

the default for DI is DS
but if I had string instructions
=> with ES
extra segment.

*for the 32 bit table

each Register that store an offset is changed to its 32 bit

example: IP → EIP
SP → ESP
⋮

FS, GS → No defaults

the main change that

EAX, EBX, ECX

→ as a 32 bit Registers can store an offset and they can be default for Data Segment

but as a 16 bit

AX, CX, DX, they're not able to store an offset.

* Segment Size = Ending address − starting address

Figure 2-5

code segment → Ending 0A0EF
09 0F0
00 0F

F 14+16 = 30
9,16↑

Ending 0A0EF
starting 09 0F0
01 00

Effective mode :-

- selectors & Descriptors
  - * using segments indirectly

selector leads me to the Descriptor which lead me to the segment.

Descriptor → 8 bytes that have all the information about the segments.

* In real mode the Register selects segment directly

* each descriptor 8 byte.
  - local table → 64k byte. max
  - global table → 64k byte. max

both local & global : $\frac{64k\ byte}{8\ byte} = 8k$

8k = 8*1024 → descriptor in the table.

Five Apple

** limit ≡ size for a segment

figure 2-6

[1] descriptor (1) :-

limit → $L_0 - L_{15}$ (16 bit) = 2 byte

[2] starting address for the segment.

$B_0 - B_{23}$ (24 bit) → 3 bytes.

From point 1 ⇒ max size of the segment = 2^{16} = 64k byte.

From point 2.

memory size = $2^{24} = 2^4 \cdot 2^{20}$

= 16 M

by number of the base bits I can detairmine witch descriptor I must use given anadress bus size, because both lead me to know the memory size.

descriptor (2) :-

1 limit ($L_0 - L_{19}$) = 20 bit → max segment size = 2^{20} = M

2 base ($B_0 - L_{31}$) = 32 bit → memory size = 4 G byte.

address bus size.

3 A : Available. → 0 → not available

→ 1 → available

• what does available mean? some descriptors are not connected to

Segment, so limit & Base may contain rubbish data in this case $A_V = 0$

ow $A_V = 1$

4 D ← 16 bit → D = 0 (AX) → by default the segment register = 16

32 bit → D = 1 (EAX)

↳ by default (32)

5 G ← 0 → limit size between 0 - 1M

00000 FFFFF

0000 0FFF FFFFFFFF

1 → 4k → 4G

↓ as if the segment size may be as the memory size.

adds 3 Fs FFF to the begning of the limit. and the ending.

- there is 1 unused bit.

* descriptor (3) :- any descriptor with base lagrer than 32 bit is called 64-bit

L with D together → tell us we're working on 16 or 32 or 64 bit.

if L=0 ← D=1 → 32

D=0 → 16

if L=1 → directly 64

* What is access right?

① P — P=0 → undefined → my descriptor is not connected to a certain segment.

P=1 → connected to a segment and gives me its info.

② DPL, compared to RPL

00 → highest
01
10
11 → lowest priority

③ S — 0 → describes system
1 describes code or data segment.

④ E | ED/C | R/W

3 bit, we read them togather

*E=0 → describe data segment = doesn't describ code segment.

ED=0 data is filled ↑ upward

ED=1 to be filled downward (stack)

W → 0 → may not be written
1 → data may be written.

Five Apple

if E=1 → describe code segment.

C=0 → ignore DPL

C=1 → enable DPL

R → may be read
→ may not be read
} reading ≡ run

(5) A

* contents of segment Register :- (16 bit)

① RPL → fst 2 bits.

RPL must be ≥ DPL

01 ≥ 11

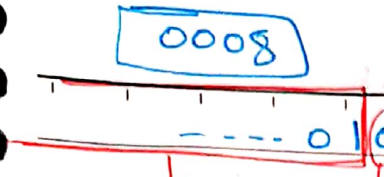
10 < 00 X

bit 2 → ② TI → 0 → global
→ 1 → local

③ seletor → 13 bit

2 → 15

→ to select one from the 8192 descriptors.



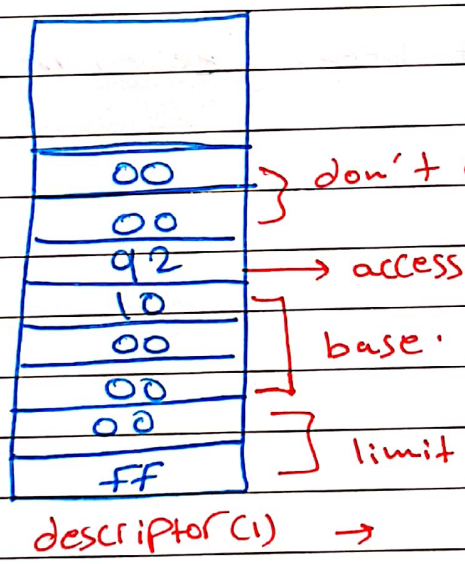
RPL → 00 → the higuest in priority
00 ≥ DPL for swle.

Q: is access is granted? Yes, RPL 00

fig 2-9

① I'll choose global table

then go to des criptor (1)



Q: what is the starting address for the target segment?

100000 H

Q: what is the segment size? 00FF

in decimal? 255 decimal

Five Apple

Q: What is the last address?

Starting + limit = 1000FF H

10010010
P DPL E ED
S

Q: is this a System described? Yes / No

S=1

Q: is this a code segment? yes / No

E=0

(data)

Q: this is (data/ stack) segment?

From ED

Q: Can you modify this segment? yes.

DS.

not stack → not SS

not code → not CS

Q) has this segment been accessed before? No

A=0

Five Apple