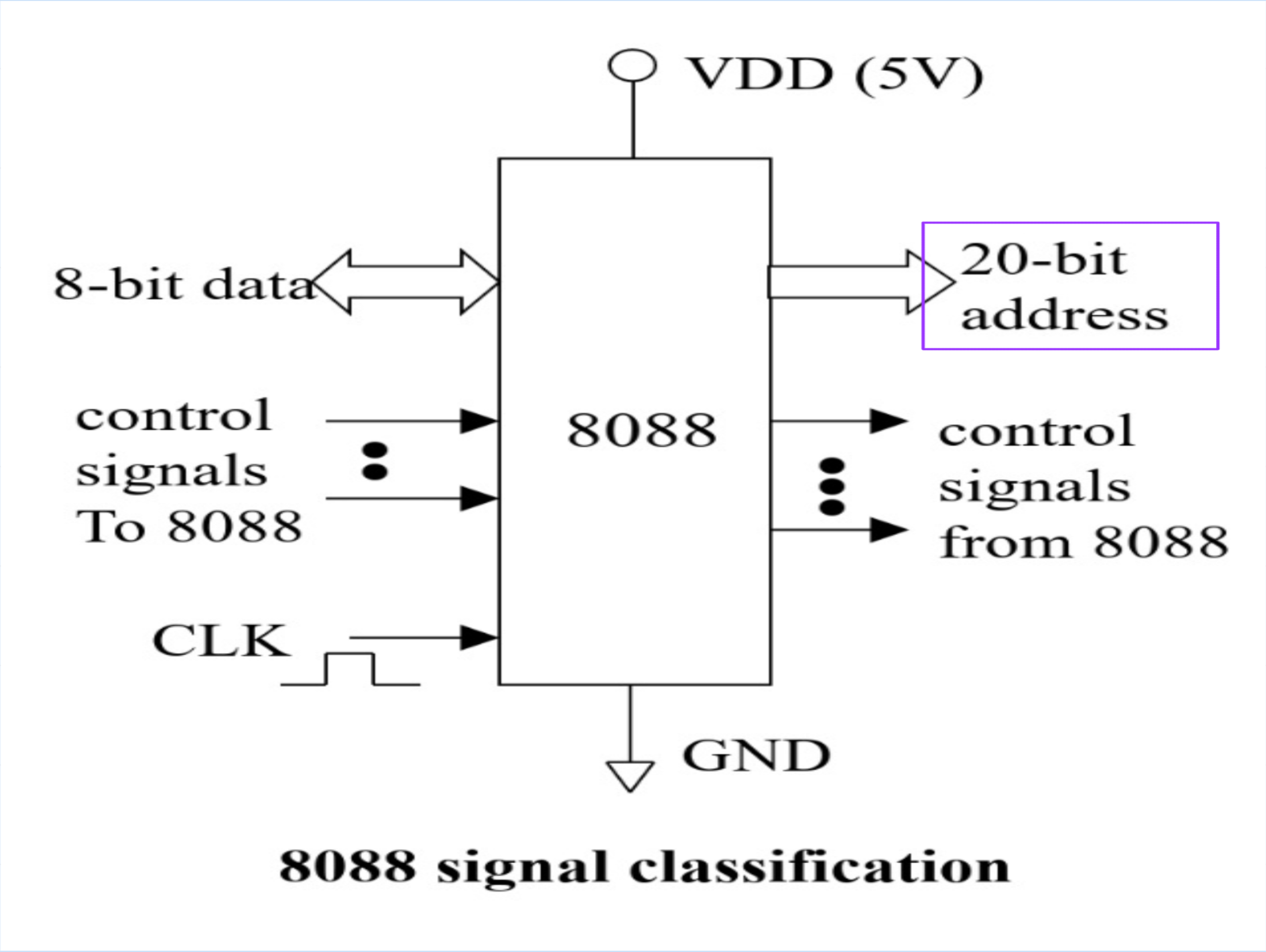
Assymply

⇐ كل نوع آلة اله اسمبلي خاص

⇐ غاية الاسمبلي احسن واسرع
- حرية اكبر

⇐ المشكلة انها اصعب من Hgih leol

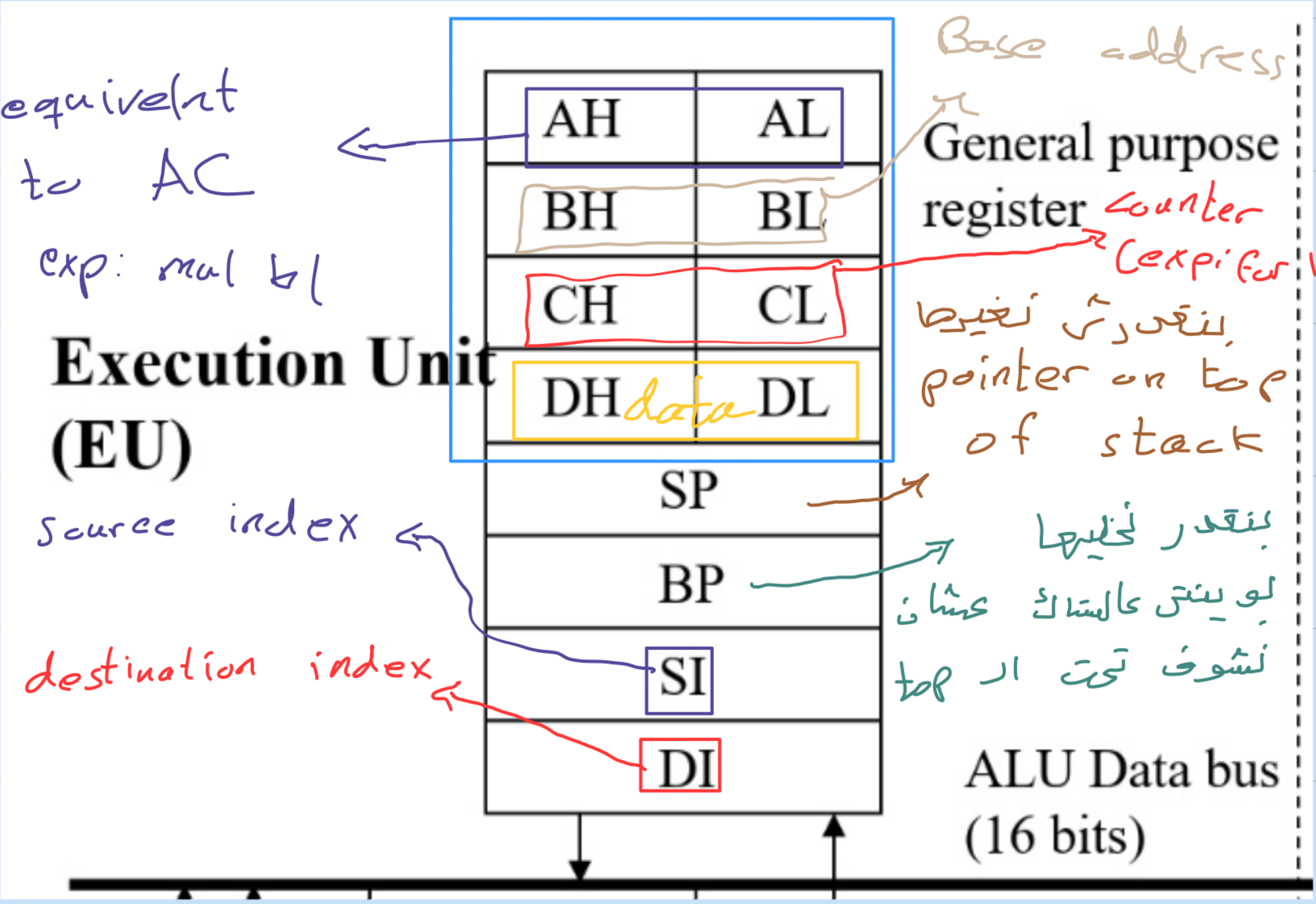


8088 signal classification

2^{20} memory cell

⇐ ال Intel يعني ال cell ⇐ 1 Byte

كل ريجيستر 16 Bit



Ax, Bx, cx, Dx

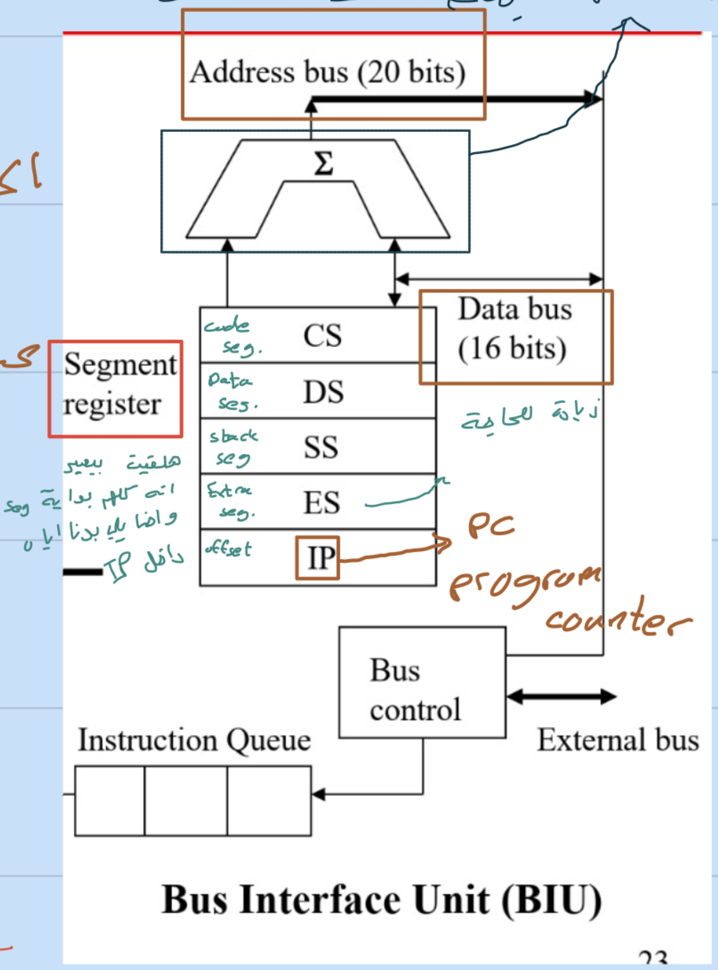
بنقدر نتعامل معهم كتقسيم 8 bit

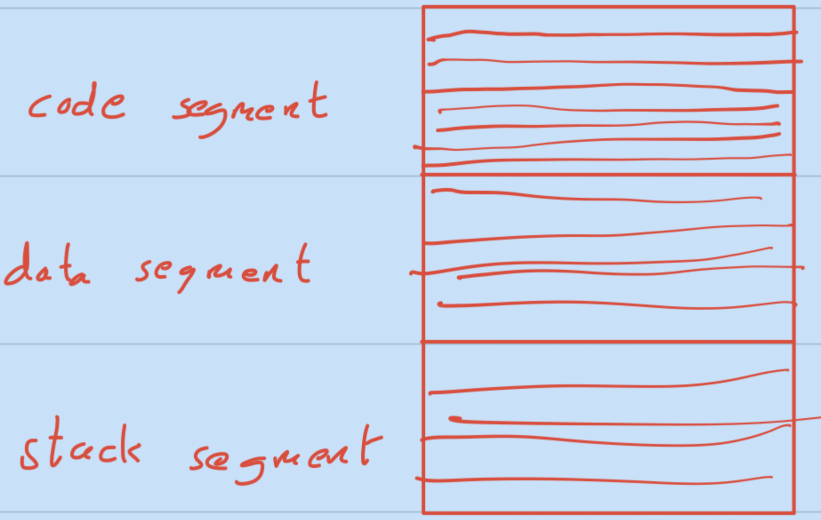
جوا CPU يلتي تكوين 20bit

المشكلة انه 16 bit وهو

هنا الادريس 20bit

$2^{20} = 1MB$





بنقسم الميموري لـ 3 اقسام وبخلي بوينتر للكل موع

انه وين بتبلش

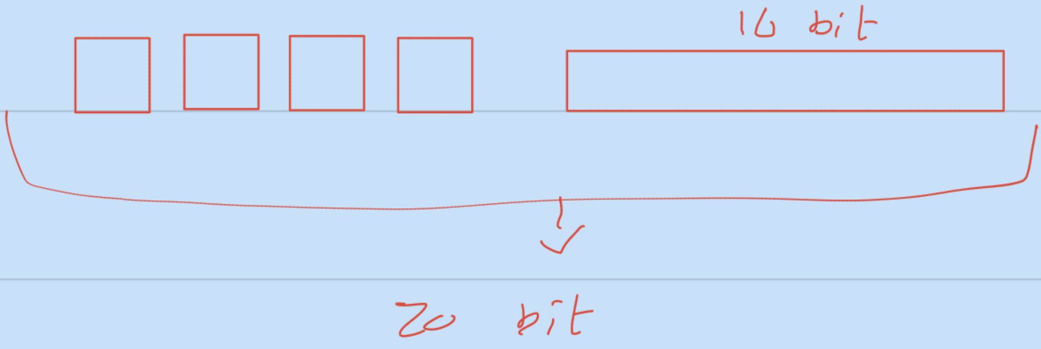
⇐ ممكن يكون بينهم تقاطع يعني مثلاً CS = DS

عادي جداً بس بتصير انه 64k الهم مع بعض

⇐ بس لما ما حلينا المشكلة لانه تاعات ال segment

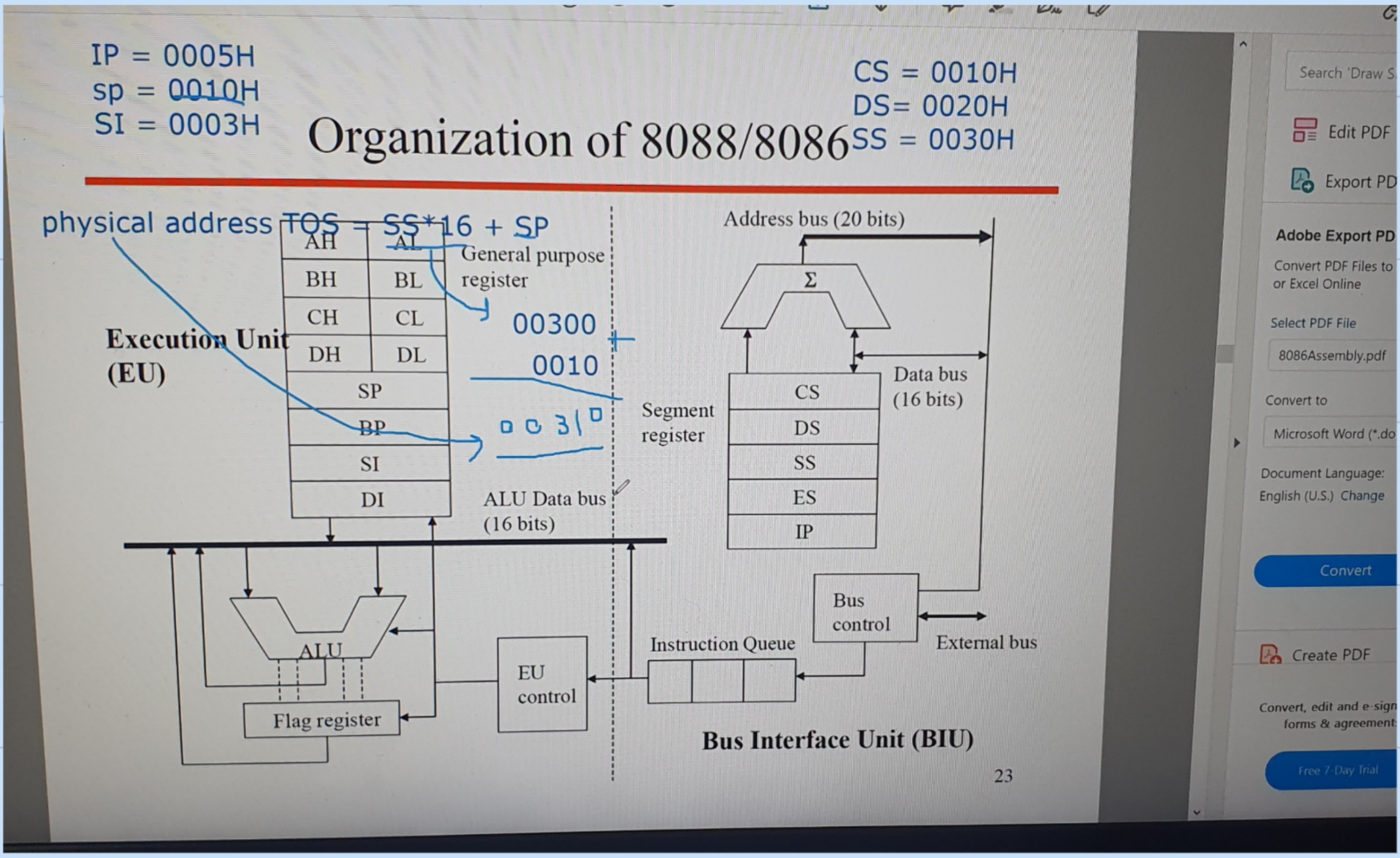
هي 16bit [illegible] 20

⇐ الحل : بنسوي shift lift 4 bit



ps: 4 bit shift ⇒ $* 2^4$ ⇒ * 16 + offset

⇐ دايما اول 4 bit رح تكونوا اصفار (segments)



IP = 0005H
sp = 0010H
SI = 0003H

CS = 0010H
DS= 0020H
SS = 0030H

Organization of 8088/8086

physical address TOS = SS*16 + SP

00300 +
0010
00310

phisical address of next instruction

⇒ CS * 16 + IP

=> 00100 + 0005 = 00105

phisical address of second opperand in

ADD Al, [SI + 4]

SI + 4 => offset address

Data segment * 16 + (4 + SI)

0020 0 + {4 + 0003} = 00207

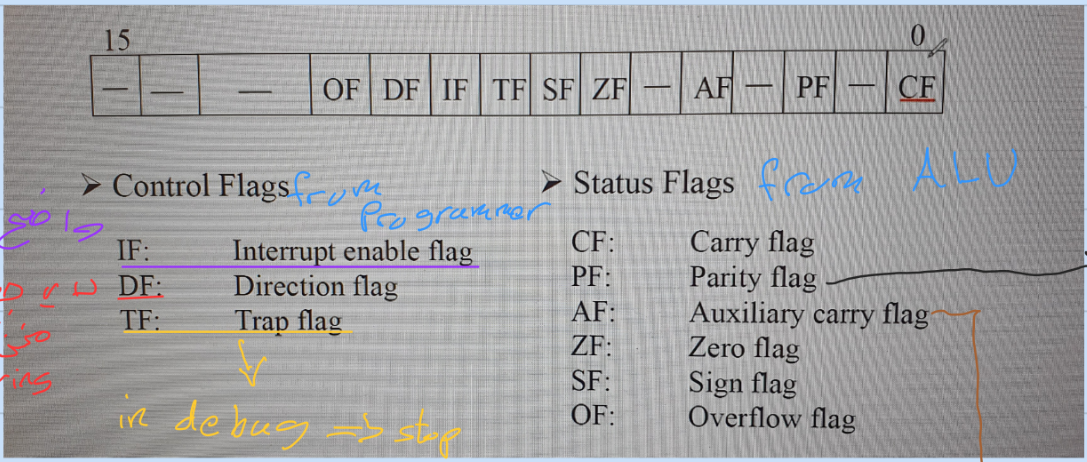
logical Address:

Seg. address : offset address

Exp: ADD Ax, CS:[Bx]

8086
8186
8286 → Ax, Bx.... كل هان ٧ ت الي يستخدم 16 bit
8386
8486 → EAx, EBx... ⇐ 32 bit بعدها صار
⋮

all flags is in spicial Regester



15															0
—	—	—		OF	DF	IF	TF	SF	ZF	—	AF	—	PF	—	CF

➢ Control Flags from Programmer

0 او 1

IF: Interrupt enable flag

[illegible] string DF: Direction flag

TF: Trap flag

in debug ⇒ stop program

➢ Status Flags from ALU

CF: Carry flag

PF: Parity flag → number of ones in first 8 bit (least sig.) if odd 0 even 1

AF: Auxiliary carry flag

ZF: Zero flag

SF: Sign flag

OF: Overflow flag

4 bit	4 bit

AF

Exp

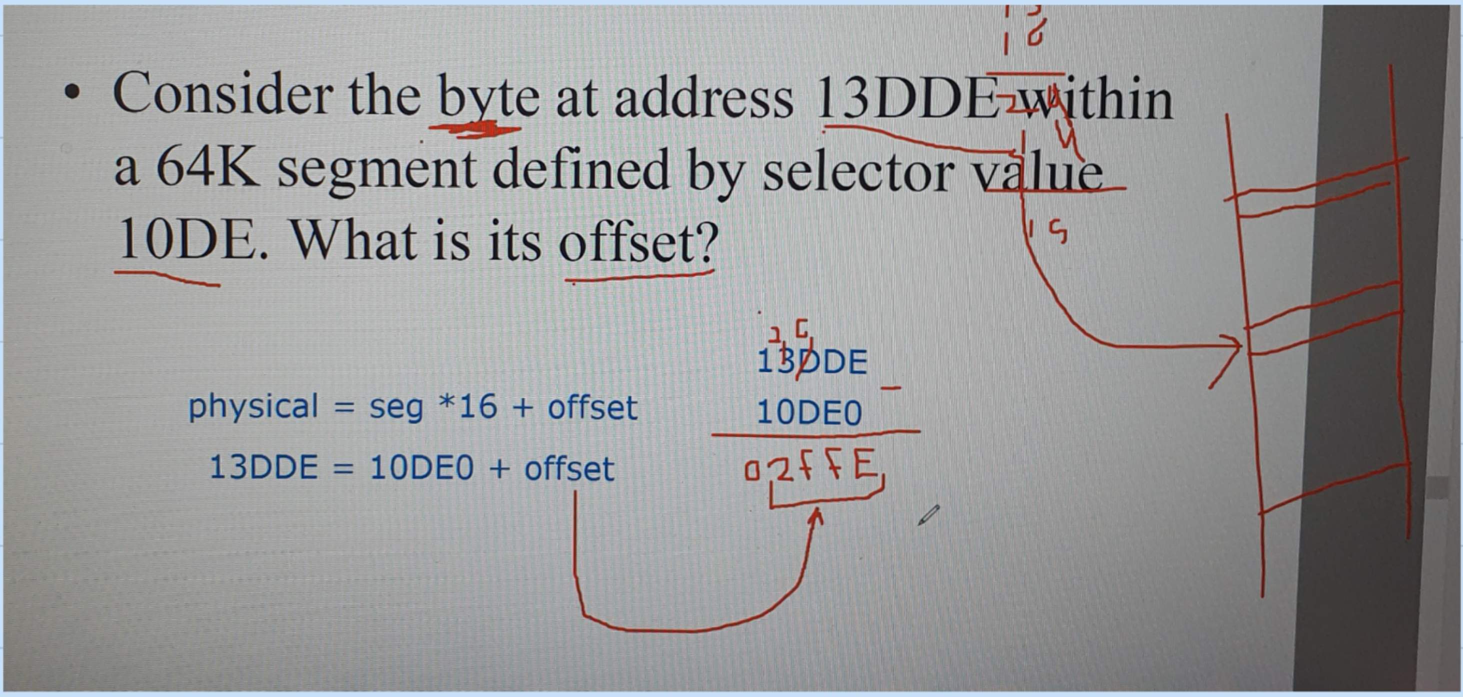
this

1 1 1 1
1 0 0 1   1 1 0 0

1 0 0 1 0 1 1 1 ⇒ used for BCD

carry & جدول ⇐

we have no 3 operand in intel

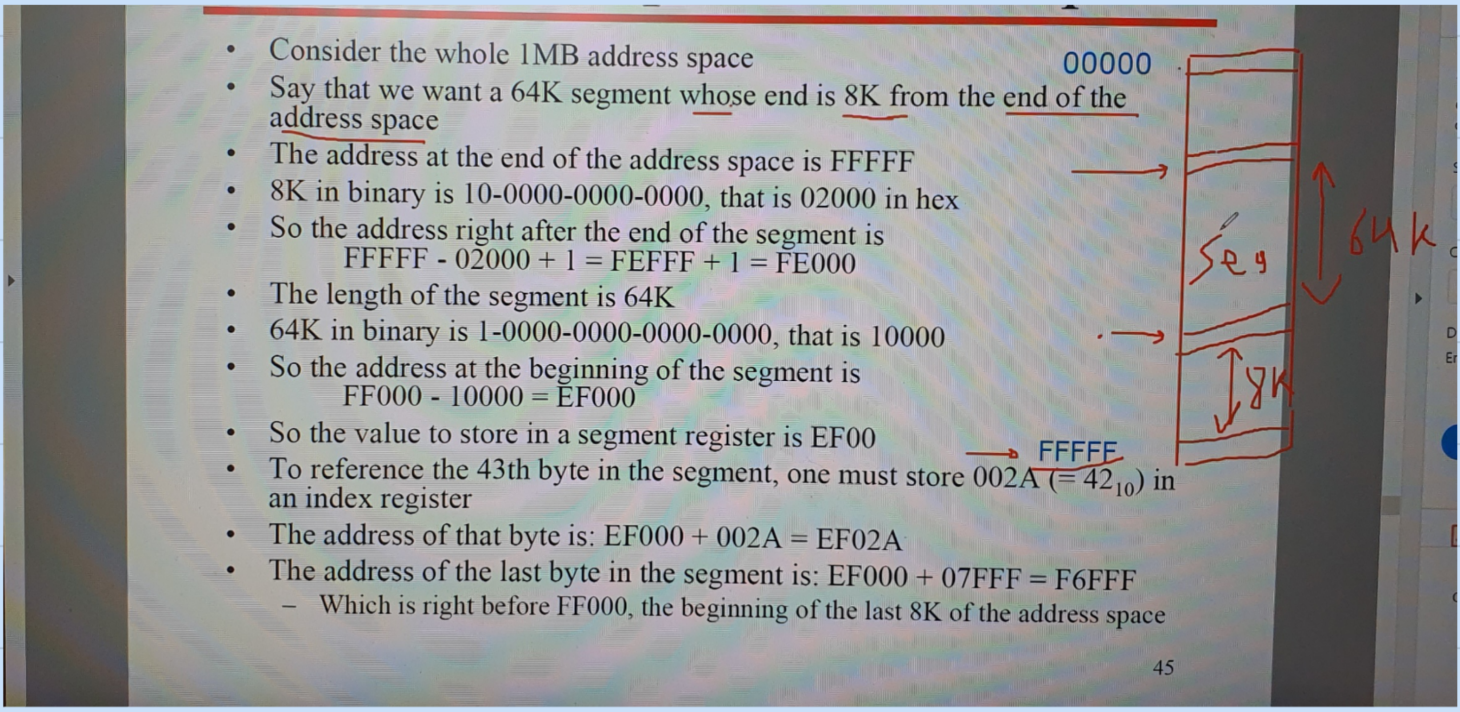


- Consider the byte at address 13DDE within a 64K segment defined by selector value 10DE. What is its offset?

physical = seg *16 + offset

13DDE = 10DE0 + offset

$$\begin{array}{r} 13DDE \\ -\ 10DE0 \\ \hline 02FFE \end{array}$$



- Consider the whole 1MB address space
- Say that we want a 64K segment whose end is 8K from the end of the address space
- The address at the end of the address space is FFFFF
- 8K in binary is 10-0000-0000-0000, that is 02000 in hex
- So the address right after the end of the segment is
  FFFFF - 02000 + 1 = FEFFF + 1 = FE000
- The length of the segment is 64K
- 64K in binary is 1-0000-0000-0000-0000, that is 10000
- So the address at the beginning of the segment is
  FF000 - 10000 = EF000
- So the value to store in a segment register is EF00
- To reference the 43th byte in the segment, one must store 002A (= 42_{10}) in an index register
- The address of that byte is: EF000 + 002A = EF02A
- The address of the last byte in the segment is: EF000 + 07FFF = F6FFF
  - Which is right before FF000, the beginning of the last 8K of the address space

Assimply

=> files is .ASM

=> it ignore capital and small letter

=> comment is semicolon ;

=> start with dot (.) is directive => not inst.

(اشي لا يعتبر شي لل CPU <= تعني import in java

.model tinny => same segment for code, data, stack

.model small => each one have segment

.model large => same small but extra for code

.stack => build stack to use it (SP)

.data => for write the data of program

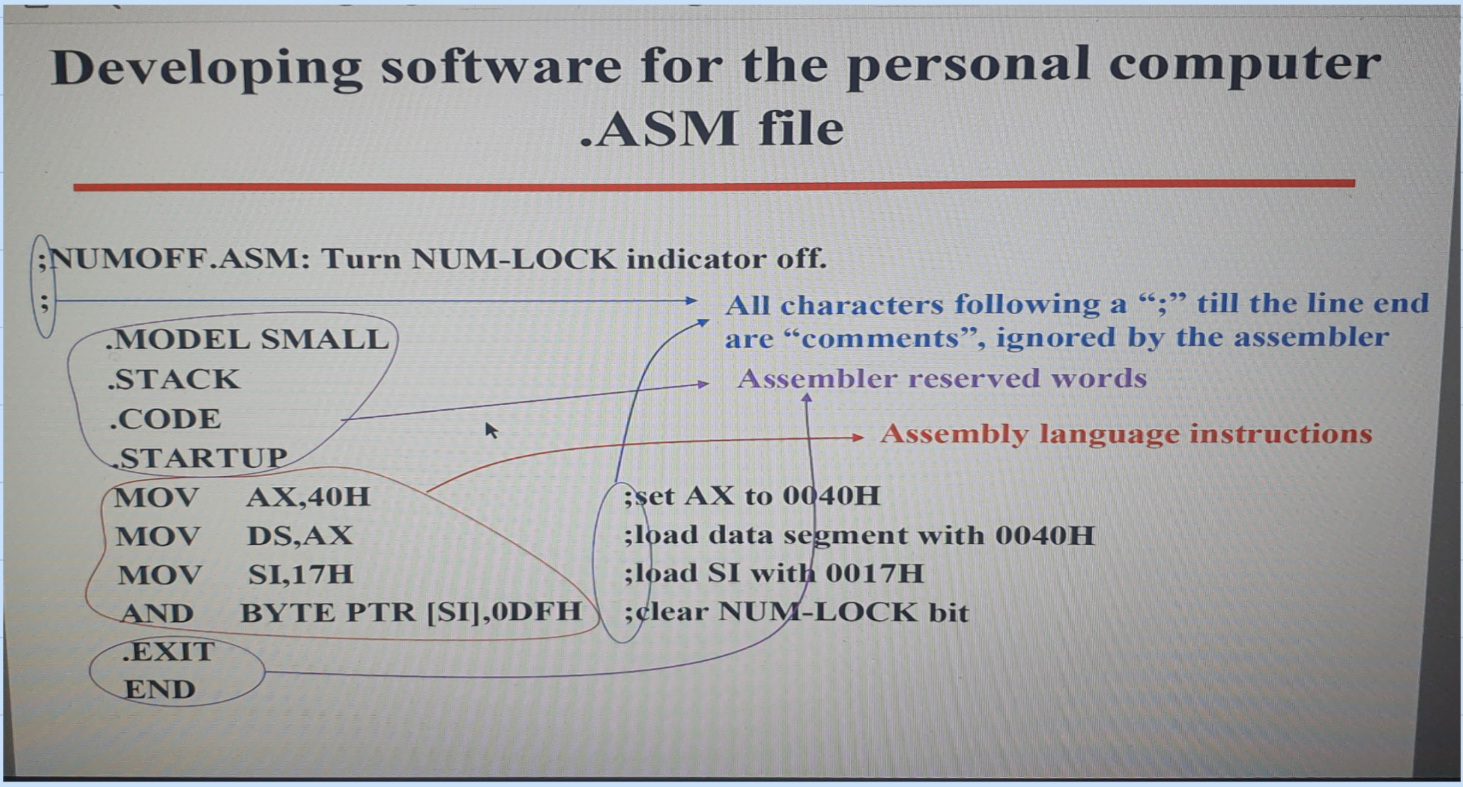
.code => for start writing the code after it

.start up => set DS pointer to .data

=> if we haven't .data we can ignore it

.Exit => return to operating system

=> in end we write "END"



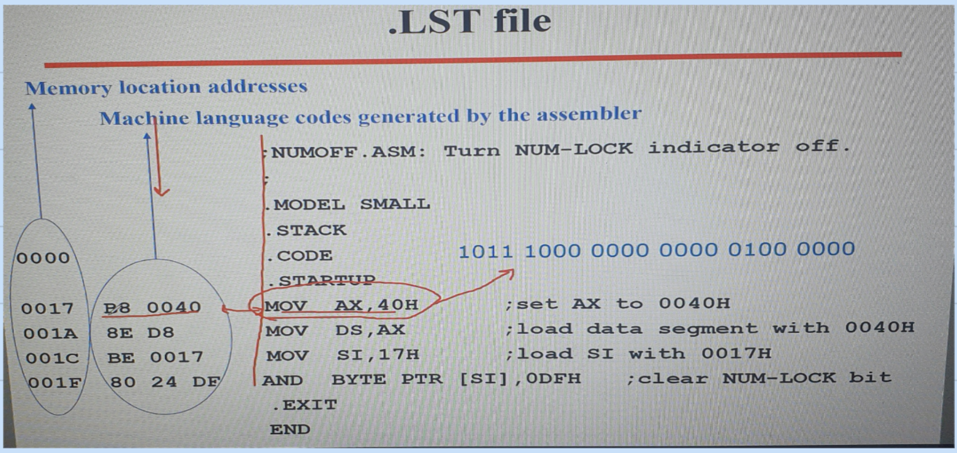
Developing software for the personal computer .ASM file

```
;NUMOFF.ASM: Turn NUM-LOCK indicator off.
;
.MODEL SMALL
.STACK
.CODE
.STARTUP
MOV     AX,40H              ;set AX to 0040H
MOV     DS,AX               ;load data segment with 0040H
MOV     SI,17H              ;load SI with 0017H
AND     BYTE PTR [SI],0DFH  ;clear NUM-LOCK bit
.EXIT
END
```

All characters following a ";" till the line end are "comments", ignored by the assembler

Assembler reserved words

Assembly language instructions



.LST file

Memory location addresses

Machine language codes generated by the assembler

```
                    ;NUMOFF.ASM: Turn NUM-LOCK indicator off.
                    ;
                    .MODEL SMALL
                    .STACK
                    .CODE            1011 1000 0000 0000 0100 0000
                    .STARTUP
0017   B8 0040      MOV   AX,40H          ;set AX to 0040H
001A   8E D8        MOV   DS,AX           ;load data segment with 0040H
001C   BE 0017      MOV   SI,17H          ;load SI with 0017H
001F   80 24 DF     AND   BYTE PTR [SI],0DFH    ;clear NUM-LOCK bit
                    .EXIT
                    END
```

.LST

write instruction in hexadecimal

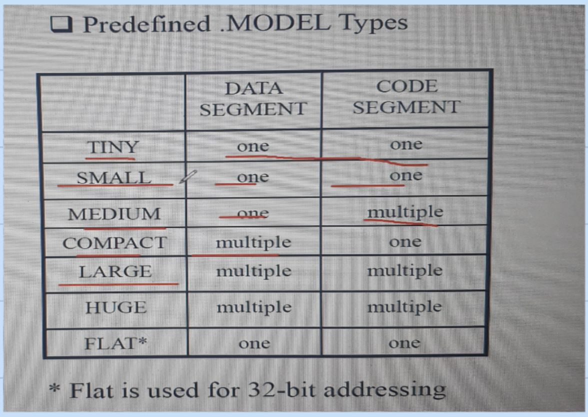
⇒ intel is lettel indian

we can write lable befor instruction

⇒ when do loop or if (Jump)

label 1: mov Ax, 20 H

↳ address of instruction in code segment

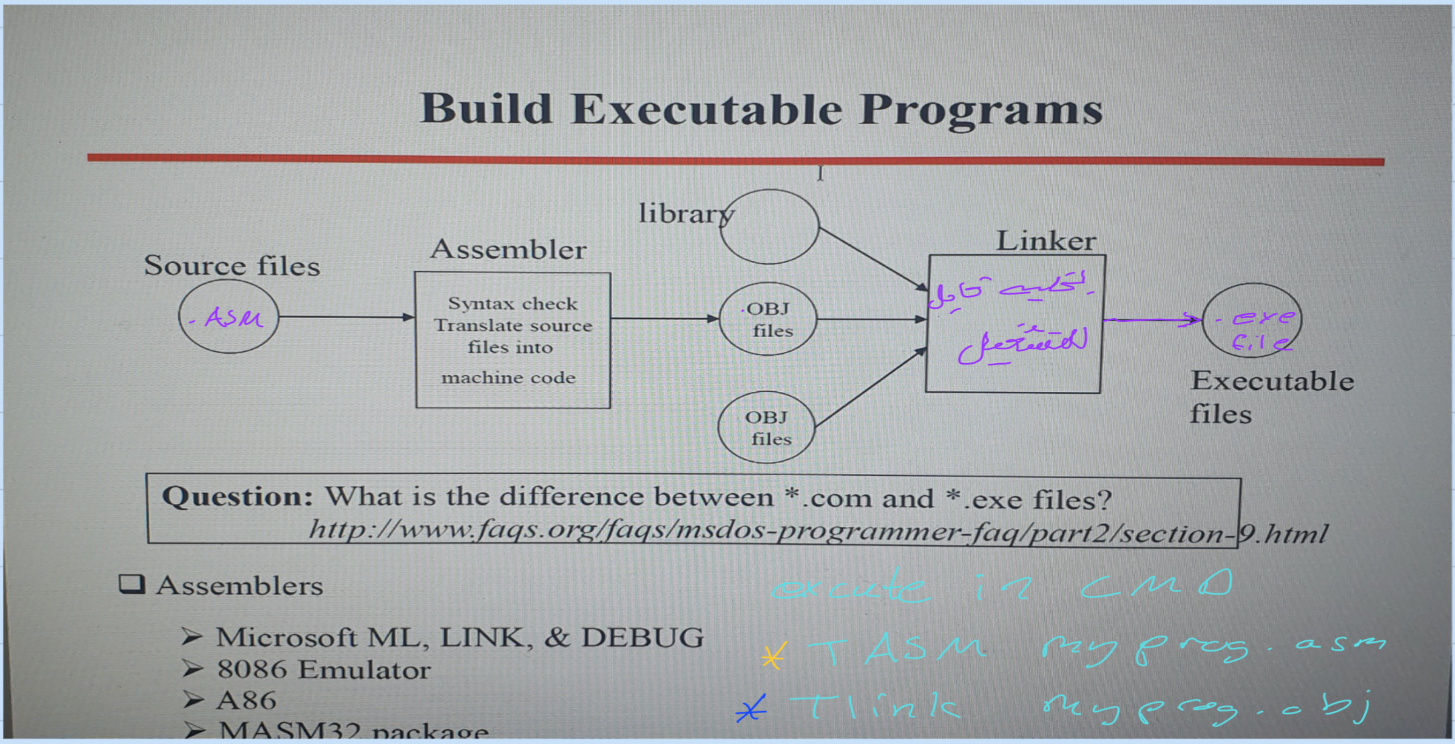


❑ Predefined .MODEL Types

	DATA SEGMENT	CODE SEGMENT
TINY	one	one
SMALL	one	one
MEDIUM	one	multiple
COMPACT	multiple	one
LARGE	multiple	multiple
HUGE	multiple	multiple
FLAT*	one	one

* Flat is used for 32-bit addressing

.model جميع انواع



Build Executable Programs

Question: What is the difference between *.com and *.exe files?
http://www.faqs.org/faqs/msdos-programmer-faq/part2/section-9.html

❑ Assemblers

- ➢ Microsoft ML, LINK, & DEBUG
- ➢ 8086 Emulator
- ➢ A86
- ➢ MASM32 package

excute in CMD

* TASM myprog.asm

* Tlink myprog.obj

In microsoft:

- ML /c /Fl myprog.asm
- Link myprog

.com old exutable file (now is .exe)

using constants:

binary: 1101 b

hexa : 1101 H => if start with letter
0A5H

decimal: 1101

negative => mov Ax, -1

using char and string

we can use single or double quotes

'A' "hello"

"A" 'hello'

=> each char is one byte

labels: ⇒ uniqe

data label ⇒ label (not have :)

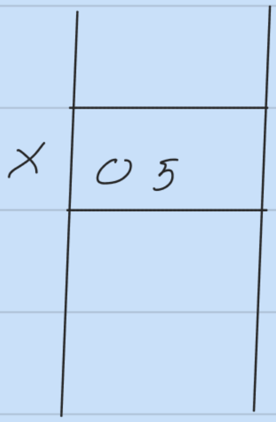
Code label ⇒ label: mov Ax, Bx

Data Allocation

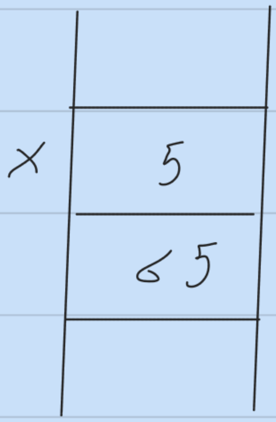
after .data we write all variables

.data

$\underbrace{X}_{\text{address of the data}}$ $\underbrace{DB}_{\text{define Byte}}$ 5



```
x   DB   5, 'A'
mov  AL, [x]      } → same
mov  AL, x        }
mov  BL, [x+1]     65
mov  BL, x[1]      65
mov  BL, x+1       5+1 = 6
```



x DB ? define without initialize value

DB Byte 1 Byte char

D B	Byte	1	Byte	char
D W	word	2	Byte	int
D D	double word	4	Byte	float, long
D Q	Quad word	8	Byte	double
D T	ten byte	10	Byte	

A DB 0
DB 1
DB 'A'

B Dw 'A', 16H, 34

address = B + 2

Bcz word is two Bytes

String :

CR enter

LF go to start line

message DB 'Bye', 0DH, 0AH, '$'

new line

end of string

if we want define 200 element ?

marlss Dw 200 dup(0) 400 Byte

or anything

we can use it nested

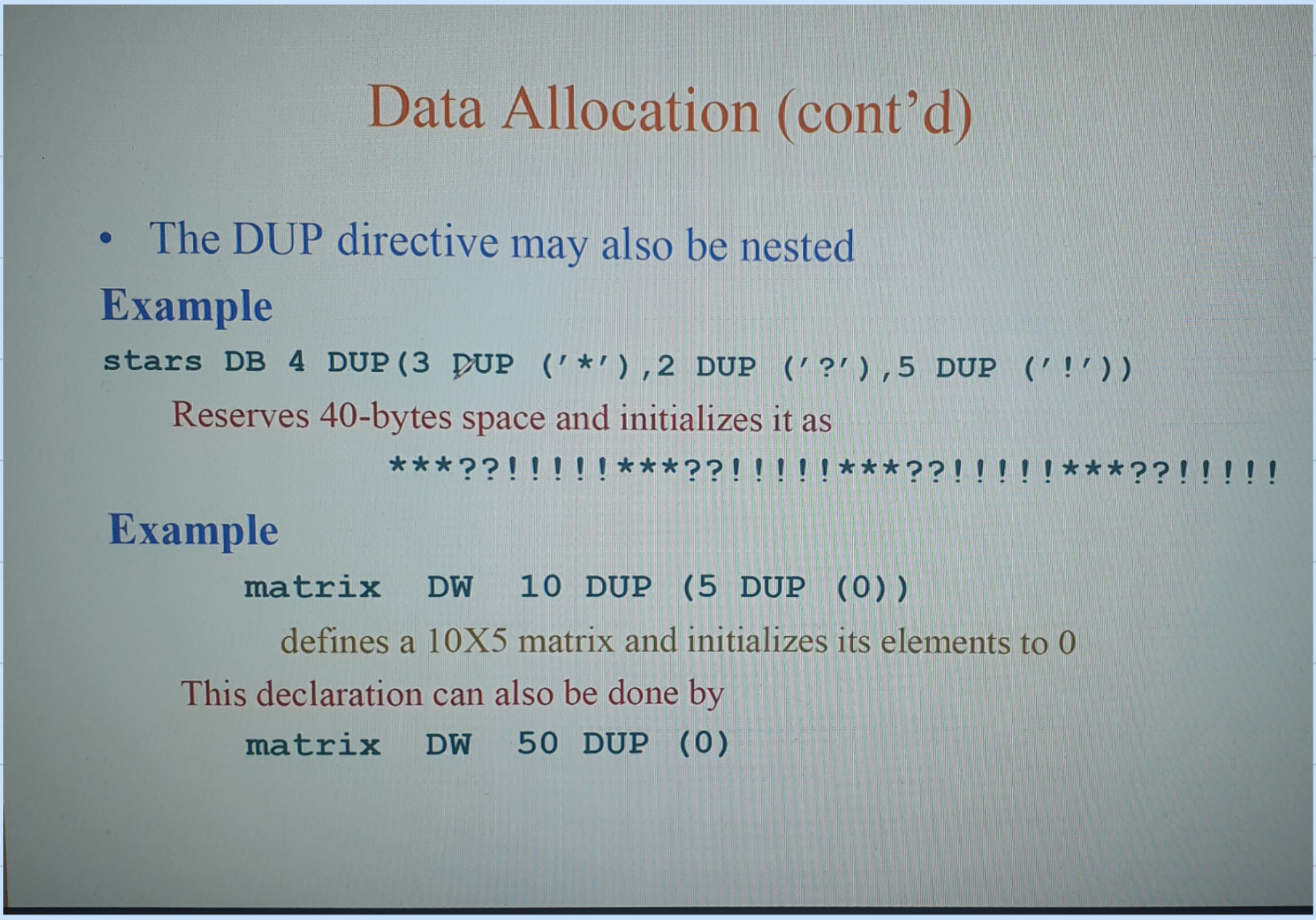
x DB 100 dup (10 dup(0)) 1000 Byte

for matrix 100 x 10

it will store as one element

But the defferent when we access it

⇒ address = (r -1) * r_size + (c -1)



Data Allocation (cont'd)

- The DUP directive may also be nested

Example

```
stars DB 4 DUP(3 DUP ('*'),2 DUP ('?'),5 DUP ('!'))
```

Reserves 40-bytes space and initializes it as

```
***??!!!!!***??!!!!!***??!!!!!***??!!!!!
```

Example

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

defines a 10X5 matrix and initializes its elements to 0

This declaration can also be done by

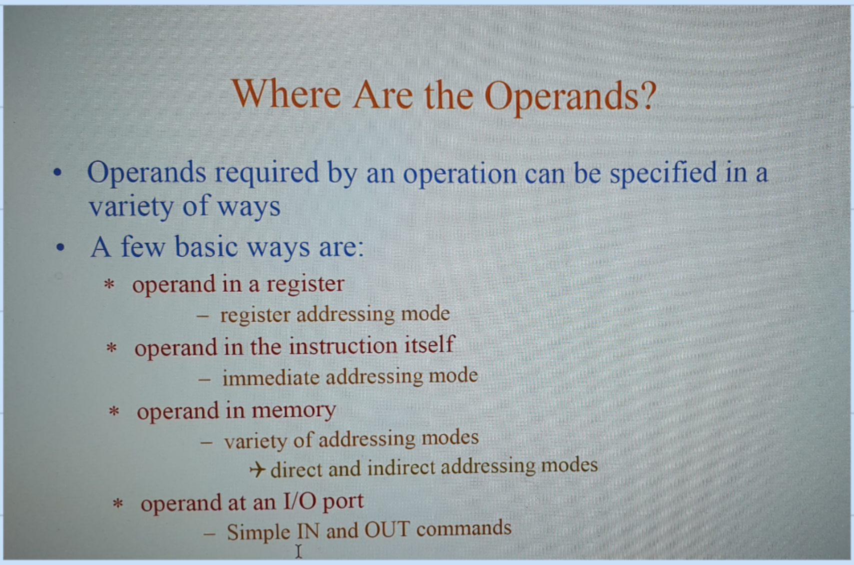
```
matrix   DW   50 DUP (0)
```

constant like PI :

PI EQU <3.1416>

بتنحطش فالميموري

⇐ الاسمبلر بحولها لقيمتها قبل ما يحول ل
machine code



Where Are the Operands?

- Operands required by an operation can be specified in a variety of ways
- A few basic ways are:
  - * operand in a register
    - – register addressing mode
  - * operand in the instruction itself
    - – immediate addressing mode
  - * operand in memory
    - – variety of addressing modes
      - ✈ direct and indirect addressing modes
  - * operand at an I/O port
    - – Simple IN and OUT commands

Regester addrissing

oberand is in regester

mov Ax, Bx

mov Al, Cl

$\underbrace{\text{movzx}}$ Ax, Al

↳ if we have extended ⇒ for unsigned

⇒ add zeros

$\underbrace{\text{movsx}}$ Ax, Al

↳ if we have extended ⇒ for signed

⇒ add most sig. bit

immediate (constant)

data is part of instruction

mov Al $\boxed{75}$

mov AL, $\boxed{+5}$

Direct addressing

data is in data segment

segment: $\underbrace{\text{offset}}$ → called effictive address

mov Al, [20]

فعلياً بنحطها متغير

```
.data
x DB 5
:
.code
.startup
mov Al, [X]
```

```
y DB 0, 1, 2, 3, 4
:
mov Ax, [y+2]
```

$\underbrace{\text{Ax}}$ 2 Byte then it have 0302 (littel indian)

03 → AH, 02 → Al

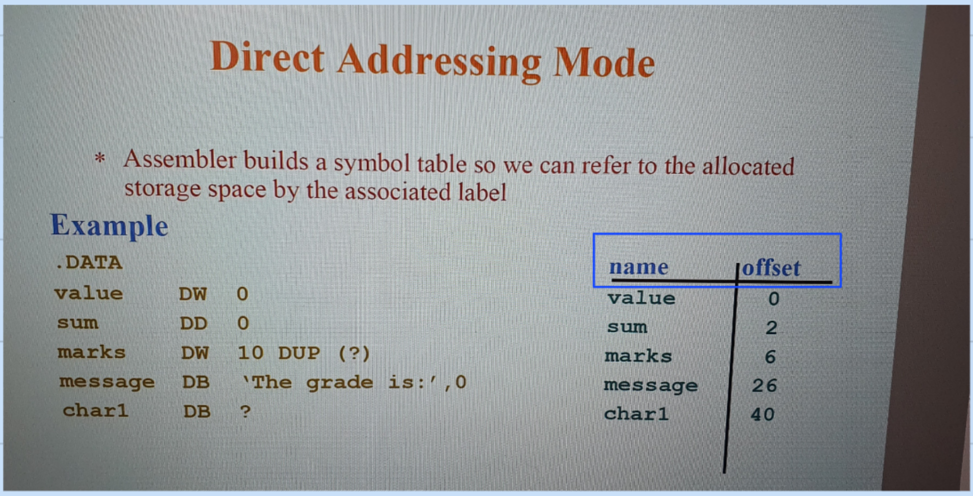
X DW 1234H, 8, 34 H

mov Al, [x+4]

x	34H
1	12H
2	08H
3	00
4	34H
	00

Al = 34H

Direct



Direct Addressing Mode

- Assembler builds a symbol table so we can refer to the allocated storage space by the associated label

Example

```
.DATA
value     DW   0
sum       DD   0
marks     DW   10 DUP (?)
message   DB   'The grade is:',0
char1     DB   ?
```

name	offset
value	0
sum	2
marks	6
message	26
char1	40

يعني ال يعطيك ذي offset حق الاسم

Register indirect

store address in regester

=> we stre in BX, BP, SI, DI

mov DL, [SI]

more flexable since we can change value

Exp: in array we can increment, decrement

$\underbrace{\text{mov BX, offset Array}}$ ↳ offset of store the array

↕ same

lea BX, Array load effictive address

Based address

use base regesters ⟹ BX or BP

mov AX, [$\underbrace{BX+4}$] ↳ BX يجب الرايح بدون تغيير

Indexed Addressing

use SI, DI

Based Indexed Addressing

Contain base and index Regester

Bcz if we use just one of them then we can't get the address that for array start mov [BP + SI] , AH

same

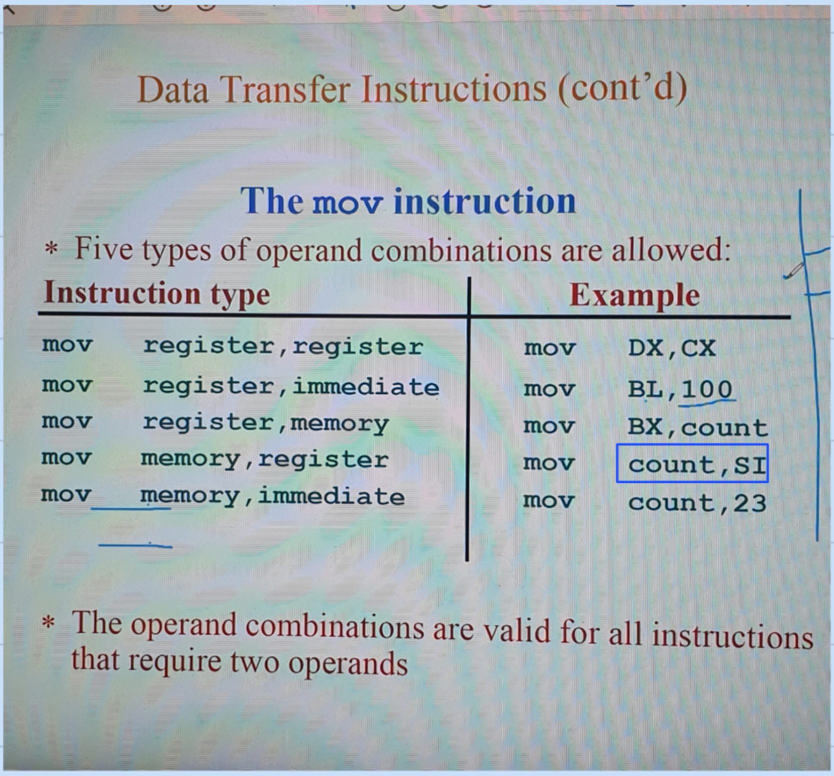
mov [BP] [SI] , AH

calculate phisical address we get ses. then add BP and SI

Based Indexed with displacement

Same as Based Indexed Addressing But add displacement

mov cl, [Bx + DI + 2080H]



Data Transfer Instructions (cont'd)

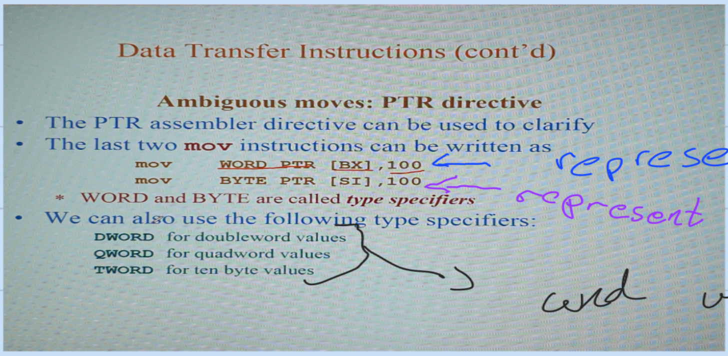
The mov instruction

* Five types of operand combinations are allowed:

Instruction type	Example
mov register,register	mov DX,CX
mov register,immediate	mov BL,100
mov register,memory	mov BX,count
mov memory,register	mov count,SI
mov memory,immediate	mov count,23

* The operand combinations are valid for all instructions that require two operands

بتخزن ف count, count+1 لانه بتوسعش



Data Transfer Instructions (cont'd)

Ambiguous moves: PTR directive

- The PTR assembler directive can be used to clarify
- The last two **mov** instructions can be written as

```
mov     WORD PTR [BX],100
mov     BYTE PTR [SI],100
```

* WORD and BYTE are called ***type specifiers***

- We can also use the following type specifiers:

```
DWORD   for doubleword values
QWORD   for quadword values
TWORD   for ten byte values
```

represent in word at [Bx]

represent in Byte

and we can use any thing to represnt as it

xchg instruction

تبديل الداتا الموجودة

used for change indian

xch Al, AH

xlat instruction

xlatb $\boxed{\text{mov Al, [Bx + Al]}}$ بس هاي مخفوة

xlatw

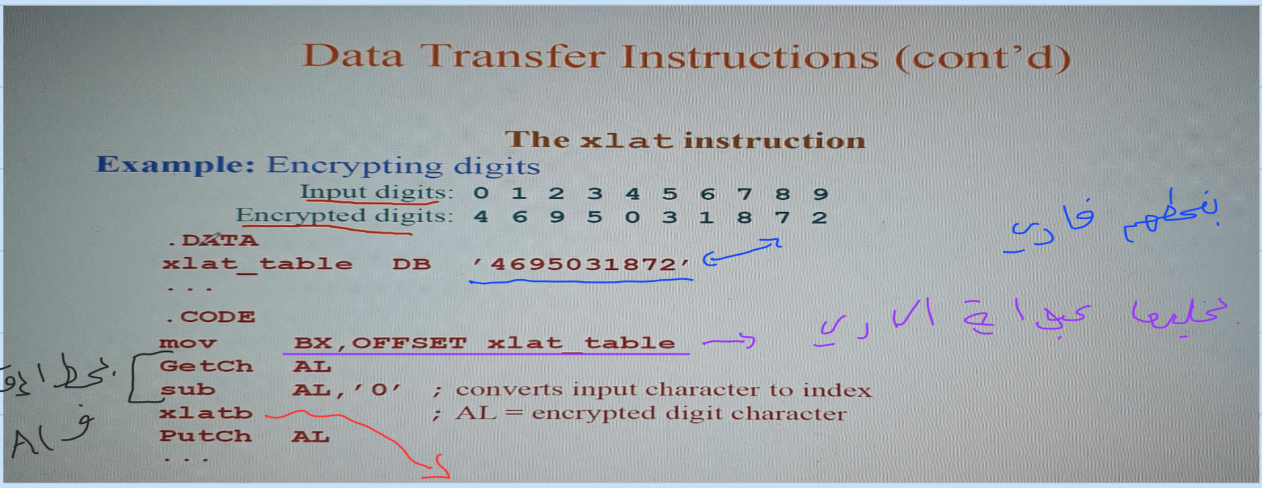
A DB -1, 4, 5, 2, 6

lea BX, A

mov Al, 3

xlatb

Al = [A+3] = 2



Data Transfer Instructions (cont'd)

The xlat instruction

Example: Encrypting digits

Input digits:	0	1	2	3	4	5	6	7	8	9
Encrypted digits:	4	6	9	5	0	3	1	8	7	2

```
.DATA
xlat_table    DB    '4695031872'
. . .
.CODE
mov     BX,OFFSET xlat_table
GetCh   AL
sub     AL,'0'    ; converts input character to index
xlatb             ; AL = encrypted digit character
PutCh   AL
. . .
```

بنطعم فاري

خليها عبارة الاري

نحط الرقم

في Al

بعدين يجب الاندكس المقابل للرقم من السترينغ

print on screen

1. we must store fuction number in AH
2. set the parameter
3. excute (INT 21H)

print char

1. function 02H print one char
2. mov char to DL
3. INT 21H

print string

1. function 09H print string
2. mov string offset to string LEA msg
   it will print from offset to $
3. INT 21H