Far addresses and near addresses

when we deal with a complete address specification it is called a far address (both part, not only the offset)

near address = incomplete address specification (has only the offset)

jumps are made on near addresses and work only with the offset

- A far address can be specified as follows:
  - $s_3 s_2 s_1 s_0$: offset — specification where $s_3 s_2 s_1 s_0$ is a constant
  - segment register : offset — specification, where segment registers are CS DS SS ES, FS, GS
  - FAR [variable], where the type is QWORD and contains 6 bytes representing the far address
    4 bytes for offset + 2 bytes for segment

Out of all 4 segments ONLY the code segment is mandatory

little - endian chapter 1 1.3.2.3

↳ the less significant part has the smallest address, and the most significant one has the higher add.

↳ applies only in the memory (memory representation technique)

12 34 56 78 h ⟶ memory: 78, 56, 34, 12, h

why did they choose to represent it this way? 20:00

a db 1h, 2, 3, 4, 5, 6 is NOT represented 6 5 4 3 2 1
but 1 2 3 4 5 6

you cannot talk about little - endian when it comes to bytes but to: WORDS
DWORDS
QWORDS

1 22:35

- if you define a variable in a loop from 1, n. let n be a double - word. you want to give the possibility for n to be as big as needed
  but most loops look like this for (i=1, i<=100, i++)
  usually the values are small and the higher values are almost always 0.
  the little byte is the MOST USED one

general structure of an instruction:

inst_name [destination], [source]

optionality

mul bx ← source

inc esi ← destination

cbw ← doesn't have operands because they are implicit

i) immediate mode

mov eax, 17

no need for further evaluation, done immediately

ii) register mode

iii) memory addressing mode

if you have memory addressing mode opperand it's offset MUST obey the following specification formulas

offset_spec = [base] + [index * scale] + [constant]

all are optional but at least one has to be present

register → base addressing

register → index addres.

indirect addressing

direct addressing

offset = deplasament / how far from the beggining of the address you are

as a base you can use any of the base registers

index —— // —— EXCEPT for ESP

scale is a constant, factor of multiplication, with values 1, 2, 4, 8

Why should we know about it?

little-end in order to know memory alloc.

- formula de la adresa

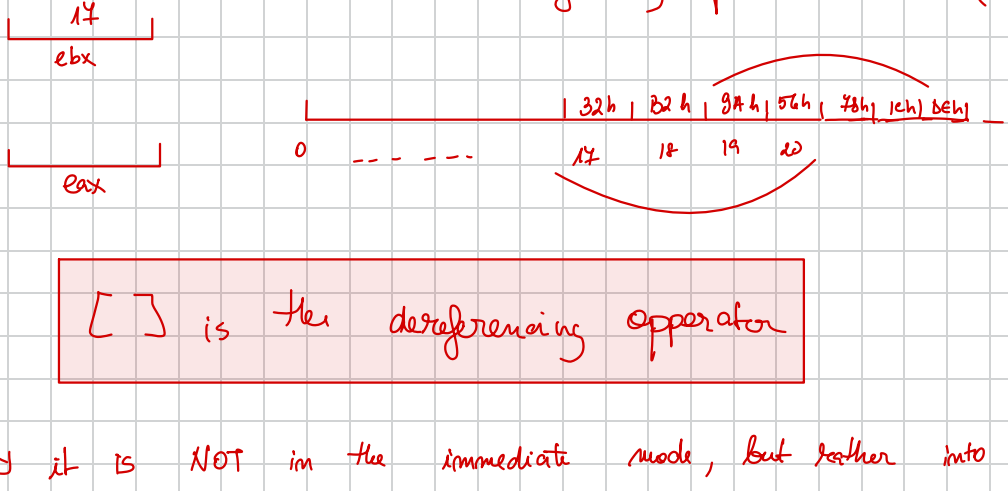
```
mov ebx, 17
mov eax, ebx      ; eax = 17
mov eax, [ebx]
```

'50:50

* a register doesn't have an ADDRESS

take out the value

* is a dereferencing operator in C (*p pointer)



[ ] is the dereferencing opperator

it is NOT in the immediate mode, but rather into the memory memory

mov eax, [ebx] means go into memory at 17 and take out "x" bytes from memory

⇒ eax = 56 9A B2 32 h

```
mov eax, ebx+2   ⟶ stupid syntax error message
mov eax, [ebx+2]
```

eax = 1C 78 56 9A h

```
mov eax, [ebx + 2*esp -7]   → syntax error
```

```
mov eax, [ebx+esp]   → correct because we don't use a scale ⇒ esp can be the base
```

```
mov eax, [ebx*2 -7]   → correct, no base register
```

'66:15 `mov eax, [ebx*3 -7]` → seems incorrect but it is interpreted as

```
mov eax, [ebx + ebx*2 -7]
```

```
mov eax, [ebx -edx]   → syntax error
```

* there's no such a thing as "— register"

```
mov [ebx*2 -7], eax   is also fine
```

mov eax, [a] → direct addressing but "a" is NOT a constant

↑ we are not concerned with the values but with addresses

1:75:00

in any programming language, any declared variable will have a fixed address, and it has 2 parts: segment, offset and its offset is always a constant determinable at assembly or compile time

the .exe file always contains the code segment and the data declared globally

* at the low level framework, working with addresses is vital
* every time we work with addresses they must obey to the offset specification formula

the contents of a variable are variable, but its address is constant

when you allocate a variable its address is fixed

the offset is a constant determinable at any time

Why * pointer arithmetic?

what are the admissable operations are possible (from an arithmetic POV) on pointers?

A: Anything that makes sense / is reasonable

Addition NO adding two pointers makes no sense!! why would you need to do with such a value

add the values of two variables and then interpret it as a pointer, that's allowed

* offset_spec = [base] + [index * scale] + [constant] → formula for computing a pointer using registers

! a register is NOT a pointer

Multiplication NO

Divide NO

Subtraction YES

the difference between 2 pointers tells us how many bytes there are between the two variables in memory

1). Subtracting two addresses → you obtain something useable
Address – address = ok (q-p = subtraction of 2 pointers = sizeof(array) in C, the number of bytes between these 2 addresses in assembly)

Why do we need pointer arithmetic?

It allows me to walk by inside memory, while keeping the value inside

2). Adding a numerical constant to a pointer
Address + numerical constant (identification of an element by indexing – a[7]) , q+9

3). Subtracting a numerical constant from a pointer
Adress - numerical constant - a[-4] , p-7;
*(a-4) - useful for reffering array elements

Dereferencing a pointer means accessing the value stored at the memory address pointed to by that pointer

"cum te privesc te evaluează ca o numeric"

adding a constant to a pointer is still a pointer

subtracting two pointers obtains a SCALAR