16 bit offset specification formula : $\boxed{[BX \mid BP] + [SI \mid DI] + [const]}$

offset specification

base pointer

indexed addressing

based ~~addressing~~ addressing

direct addressing

! The address of a variable is <u>constant</u>

* A variable is not a constant, but it's offset is !

the second part of the address is determined at <u>load time</u> !

```
mov  ax, [eax + edp *4 - 7]
mov  eax, [bx + si + 6]
```

you CANNOT have a mixture of these

the destination decides the number of bytes taken from the next address

Machine Language is the set of machine instructions to which the processor reacts

Assembly language is a programming language in which the basic instructions correspond to the machine operations

* every variable name is replaced with its **offset** and later with the whole add.

The basic elements with whom the assembler works:

- labels
- instructions
- directives (# in C for ex. # define ...)
  - they are directed to the processor
  - indication given to the compiler & assembler
- location counter → int the number of bytes generated so far ⇒ the offset RIGHT NOW
  - ↳ an R-value
  - it cannot be assigned sat

$\boxed{\text{Symbols} = \text{Mnemonics} + \text{labels}}$

→ i.e. the coursor in a word doc. the address where we're right now

Every segment has its own location counter $\boxed{\$}$

[v] — dereferencing operator (specific to NASM)

$$ - start of current section

$ - $$ = the distance between the beginnig of the segment AS A SCALAR

ex)
```
        a     db   17, -2, offh, 'xyz'
              db   'a', -101, 251, -3, 7eh
              db   _ _ _ _ _ _
        lga   dw   $ - a
        lga   dw   lga - a
        b     dw   _ _ _ ______ _ _
```
→ pointer arithmetic will ALWAYS be BYTES

⇒ mov ax, b-a (pointer arithmetic ⇒ eax - the length of the array)

or do like this:

what else can we write for the same result?

The assembler/compiler has 2 main tasks:

1) Check the sintactic correctness and validity of your sourcecode

2) Generating the corresponding bytes for the instruction and directives met in the source code

The assembler is a tool for generating BYTES

mov eax, $\boxed{\$ - a}$ NO! we are in the code segment ⇒ FALSE

mov eax, lga - a CORRECT, cause lga is a variable

$\boxed{\text{EQU}}$ defines a constant ⇒ they don't have allocated memory location

⇒ is a syntactical error to dereference a constant

mov eax, [lga] → SYNTAX ERROR

* You cannot sub. from an offset a for address

A code label you define with „:"

Source line format is:

$\boxed{\text{[label [...]][prefix es][mnemonic] [operands] [; comment]}}$

```
ex:   here :      jmp  here        ; label + mnemonic + operand + comment
      repz   cmpsd                 ; prefix + mnemonic + comment  -
      start :                      ; label + com.
```

47:35

Why are „:" optional when it comes to labels

a ↗ (db) 14, -2, off — mnemonic — operands

data label ⇒ not needed

the assembler is generating the corresponding bytes

Two categories of labels:

a) Code labels → can also appear in the data segment

b) Data label → can also appear in the code segment

C and assembly are value oriented languages

The value associated with the label in assembly language is an integer number representing the address of the instruction or directive following the label.

CALL is a jump that stores the value to the stack

expressions must be evaluatable at assembly time except the offset specification formula

```
mov eax, [ebx+2]     - correct
mov eax, ebx+2       - syntax error because it is not evaluatable at assemb. time
mov eax, [v+2]
mov eax, [v]+2       NO   cause at assembly time you don't know the contents of the variable
```

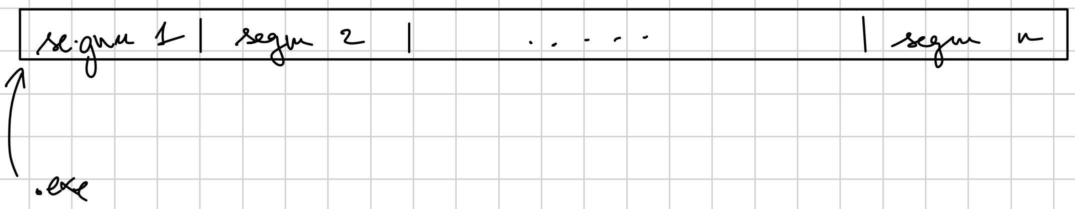
Operands types:
- immediate operands
- register operands
- memory operands

?? : offset 1
?? : offset 2

nobody knows the segment part until loading time

the linkers are responsibles for the correctness / validity of the program

Direct addressing is when you don't have the destination a register



'19:24 the most imp part

* what you did until now were NOT for addresses

* every offset is part of one of the 4 types of segments
CS DS ES SS

```
jmp FAR CS: ....
jmp FAR DS: ... or jmp far [label2]
```

* ES can only be used in explicit specifications like ES: [ver]

```
mov eax, [v]          ; mov eax, DWORD PTR DS:[405000]

mov eax, [ebx]        ; mov eax, DWORD PTR DS:[ebx]
mov eax, [ebp]        ; ————————||———————— SS: [ebx]
mov eax, [ebp*2]      ; ——||—— DWORD PTR SS: [ebp+ebp]
mov eax, [ebp*3]        ————————||———————— SS: [ebp + ebp*2]
         [ebp*4]        ————————||———————— DS : [ebp*4]
```

---

ex de încercat

```
mov eax, [ebx +esp] ✓ ; mov eax, DWORD PTR [SS: esp + ebx]
mov eax, [esp + ebx] ✓

mov eax, [ebx + esp*2]    you cannot have esp as an index

mov eax, [ebx + ebp*2] ✓
```

```
mov eax, [ebx + ebp]  } any of them can              DS : ebx + ebp
                         be the base / index => Different:
mov eax, [ebp + ebx]  }                               SS: ebp + ebx
```

$[ebx*2+ebp] \rightarrow$ SS:

$[ebx*1+ebp] \rightarrow$ SS:

$[ebp*1+ebx] \rightarrow$ DS:

$[\underbrace{ebx*1}_{index}+\underbrace{ebp*1}_{base}] \rightarrow$ SS:

$[\underbrace{ebp*1}_{index}+\underbrace{ebx*1}_{base}] \rightarrow$ DS:

$\underbrace{\text{ex with CS:}}$ $\xrightarrow{\text{deduction}}$

jmp et 1 ; jmp short 00102h → if you stay into the [-127, 127] range it is considered SHORT and can be done on a byte