ASC Lecture 5 → FAR and NEAR addresses + Pointer arithmetic

≡ Notes	lecture notes 5

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

Even though the elements of this formula are optional, at least **one** has to be present:

→ `[base]` can be a register

- if only the base is present we call it **base addressing**
- can take any register: EAX, EBX, ECX, EDX, ESP, EBP

→ `[index]` can be a register

- if only the index is present we call it **index addressing**
- can take any register **except** **ESP**

→ `[scale]` $\in 1, 2, 4, 8$

→ `[constant]` an integer

- `offset_spec = [base] + [index]` ⇒ **indirect addressing**
- `offset_spec = [constant]` ⇒ **direct addressing**

FAR addresses and NEAR addresses

When dealing with **complete address** specification it is called the *FAR address* (both parts, not only the offset) ⇒ a *NEAR address* is **an incomplete** address specification, having only the offset.

A far address can be specified as follows:

- $s_3s_2s_1s_0 : offset$ where $s_3s_2s_1s_0$ is **constant**
- $segment_{register} : offset$ where the segment register is **CS, DS, SS, ES, FS, GS**
- $FAR[variable]$, where the type is a *qword* and contains 6 bytes representing the ***far address*** (4 bytes for the *offset* + 2 bytes for *segment*)

! little endian representation *only applies in memory*

The general structure of an instruction looks like this: `instr_name [destination], [source]`

`mul bx` ← source, destination is implicitly `dx:ax`

`inc esi` ← destination

`cbw` ← has **both operands implicit**

e.g. memory: | | 32h | B2h | 9Ah | 56h | 78h | 1Ch | DEh | | let 32h be at position 17

```
mov ebx, 17
mov eax, ebx ; eax = 17
mov eax, [ebx] ; a register doesn't have an address
; this is an "into the memory" addressiing mode, not an imediate one
; essentially goes into memory at 17 and wants to take out 'x' bytes
; eax = 569AB231h
```

```
mov eax, ebx + 2 ; stupid syntax error
mov eax, [ebx + 2] ; eax = 1C78569Ah
```

```
mov eax, [ebx + 2 * esp - 7] ; synax error, esp cannot be an index
mov eax, [ebx - edx] ; syntax error, there's no such a thing as "-register"
```

```
mov eax, [ebx + esp]
mov eax, [ebx * 2 - 7]
mov eax, [ebx * 3 - 7] ; seems incorrect but it changes to [ebx + ebx * 2 - 7]
mov [ebx * 2 - 7], eax
```

```
mov eax, [a] ; direct addressing BUT "a" is NOT a constant -> incorrect
```

In any programming language, any declared variable will have a *fixed* address, containing the *segment* and the *offset*, which is determinable only at assembly or compile time

Pointer arithmetic

Pointer arithmetic represents the set of arithmetic operations allowed to be performed with pointers, this meaning **using arithmetic expressions which have addresses as operands**. In addressing system operations with pointers are performed.

At the low level framework, working with addresses is vital; this is why every time we work with addresses they **must obey** *the offset specification formula*.

The **contents** of a variable are ***variable***, but the address is **constant.**

When you allocate a variable, it's address is **fixed** and the *offset* is a constant determinable at any time.

#TODO

Q: To what is little endian applied?

→ when you do:

```
a db 1, 2, 3, 4, 6
-> memory: 01 02 03 04 06, NOT 06 04 03 02 01 little endian
```

Little endian is applied to structures that are at least 2 bytes long (words, dwords, qwords)

Q: Why did the designers choose little endian representation?

→ in a loop, for example, you want to take measures so that *n* can be as high as possible, BUT from a statistical point of view, **most loops, even if n is a dword, the value doesn't exceed 100.**

→ usually we work with small values ⇒ they wanted to offer the optimal solution and came to the conclusion that in most cases the most significant bytes **are 0** ⇒ ***the most used byte is the last one***

→ therefore they chose to place it *first* so you have fast access to it

Q: Why do we need the offset specification formula?

→ They are used to compute the pointer that redirects you somewhere else

→ you look at the data in memory and have to decide how to interpret it

⇒ every program you write must obey this specification formula

Q: Which are the arithmetic operations that are allowed with pointers in computer science?

→ Any operation that makes sense: adding a constant to a pointer, subtracting a constant from a pointer, **subtracting 2 pointers!!**