ASC Lecture 3 → segments and addressing

Notes	lecture notes 3

buffer → prepares the execution of the next instruction

segment part of the address specification can be CS, DS, SS, ES (limited to 16 bits)

💡 Handling the segment part is the task of the OS, handling the **offset** part is the job of the programmer

The address registers are EIP + the segment registers

Segments

A segment represents the *logical section* of a program's memory, featured by its *basic address*, by its *limit (size)* and by its *type*.

In 80×86-based processors, a segment can mean:

1. A block of memory of *discrete size* called a **physical segment** that has:
    - 64K bytes for 16-bits processors
    - 4G bytes for 32-bits processors
2. A variable-sized block of memory, called a **logical segment** occupied by a program's code or data

CS (code segment), DS (data segment), SS (stack segment), ES (extra segment), FG, GS (last 2 are extra segments that don't have a special role)

Necessity of the segments:

→ the **code segment** and the **stack segment** (running of the program will be made) are mandatory for running the program

→ for the programmer to be written down, only the code segment is mandatory

→ we can have more than one of each segment at a time (multi-module programming)

one program is always composed by one or more segments (CS, DS, SS, ES) of one or more of the specified types

💡 at any given moment only **ONE** can be **ACTIVE**

→ WHO tells me what segment is active? **The segment registers: CS, DS, SS, ES tell us which segment is active**

→ "CS contains the current active code segment" is **false**, it cannot contain a whole **segment**:

In 16 bits programming, the segment registers CS, DS, SS, ES contain **the starting addresses** of the currently active segments

Under 32 bits programming, these segment registers are containing **the segment selectors** of the currently active segments

EIP → extension of IP (instruction pointer) **contains the offset of the current instruction of the current code segment**

💡 The main task of an assembler is to **generate** the corresponding bytes

e.g. CS:EIP → The FAR address of the currently executed instruction

```
mov CS, [var]
mov eip, eax
```

EIP is automatically incremented by the current execution and it **cannot** be changed, since it is managed by the BIU

CS - contains the segment selector (a number, an entry, an index in a table) of the currently active code segment and it can be changed if the execution will switch to another segment

An **offset** is always **relative** to a certain segment (you cannot work with independent offsets) → it ALWAYS has an *associated segment*

Addressing

1KB = 2^10 bytes
1MB = 2^20 bytes - we need 20 bits addresses
1 GB = 2^30 bytes

Addresses of a memory location are **a number of consecutive bytes** from the beginning of the RAM memory and the beginning of the memory location

!! REGISTERS AND FLAGS ARE NOT IN MEMORY, they are entities at the level of the microprocessor

Every memory locations has an address associated to it, but some also have *names* (e.g. variables) **only for human purposes**. For the processor, that name memory location **has no importance**.

Every variable for the processor can have 2 meanings:

1. it's *address*

   → this is implicitly

2. it's *content*

   → you cannot access it without **going to the address first**

offset ⇒ the address of a location relative to the beginning of a segment (deplasament)

You will never work with full addresses, but rather *address specifications* ⇒ **a pair of a segment selector and an offset**

A segment selector is defined and provided **by the OS.**

$$\boxed{S_3S_2S_1S_0 : o_7o_6o_5o_4o_3o_2o_1o_0}$$

An address specification is the only form a programmer can use inside his program to express the *addresses with which he wants to work with.*

`[ ] - DEREFERENCING operator`

!! **[base] + [index * scale] + [constant]**

base: EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP

index: EAX, EBX, ECX, EDX, EBP, ESI, EDI but **not ESP**

scale: 1, 2, 4, 8

- Displacement only addressing mode → **Direct Addressing**

  ! the displacement is an offset

  ```
  v dd 87102h
  -------------
  mov eax, [v]
  ```

- **Indirect addressing** → the offset formula / anything other than just the displacement
  - **based addressing** if in the computing one of the base registers in present
  - **scale-indexed addressing** if tin the computing one of the index registers is present
- **relative addressing → jumps**

  ```
  jmp Below ;  OllyDbg sees it as jmp [0084] (jmp 84 below)
  .........
  .........
  Below:
      mov eax, ebx
  ```

FAR Adress Specification:

- $s_3s_2s_1s_0 : offset$ where the first part is constant

  Segment : offset (displacent)
  16 bits   16 bits

- segment register : offset_specification where segment registers are CS, DS, SS, ES, FS, GS
- FAR [variable] where variable is of type QWORD and contains 6 bytes (what you call in other languages *pointers)

For an instruction there are 3 ways to express the required operand:

1. **register mode**

   ```
   mov eax, 17
   ```

2. **immediate mode**

   ```
   mov eax, 17 ; 17 is speccified in immediate mode
   ```

3. **memory addressing mode**

   ```
   offset_address = [base] + [index * scale] + [constant]
   ```

Notion	Representation	Description
Address specification, logical address, FAR address	$Selector_{16}:offset_{32}$	Defines completely both the segment and the offset inside it.
Selector	16 bits	Identifies one of the available segments. As a numeric value it codifies the position of the selected segment descriptor within a descriptor table.
Offset, NEAR address	$Offset_{32}$	Defines only the offset component (considering that the segment is known or that the flat memory model is used).
Linear address (segmentation address)	32 bits	Segment beginning + offset, represents the result of the segmentation computing.
Physical effective address	At least 32 bits	Final result of segmentation plus paging eventually. The final address obtained by BIU points to physical memory (hardware).

ERRORS:

→ **memory violation error**

→ **if the processor doesn't find that specific index**

→ if you go outside the limits of that segment

→ **syntax error**

→ **runtime error**

Q: Why were the segments not extended?

→ it wasn't necessary, because they were holding the starting address of that segment in 16 bit prog., but in 32 bit prog. you **are not allowed to hold that address**, therefore now they hold the ***segment descriptor*** which is an index in a table managed entirely by the OS

Q: What does CS store?

→ CS contains **the segment selector** corresponding to the active code segment

Q: Is this code correct?

```
mov eax, [8:1000h]
```

→ it is an instruction exposed to runtime error, even though there is not syntax error

Q: Why can't we define variables in code segment, like in C?

```
segment code
    start:
        v db 17
        v1 dw 54321
        mov ax, [v]
```

→ even if it is *a valid line*, at runtime the first 2 lines will be considered as **instructions**

→ therefore the first 3 bytes will be generated in a chaotic way

Q: How can you *correct* this code, without using a data segment / can you **define** variables in the code segment?

```
segment code
    start:
```

```
jmp REAL_START
    v db 17
    v1 dw 54321

REAL_START:
    mov ax, [v]
```