2.6.4. Address registers and address computation

Address of a memory location – nr. of consecutive **bytes** from the beginning of the RAM memory and the beginning of that memory location.

An uninterrupted sequence of memory locations, used for similar purposes during a program execution, represents a *segment*. So, <u>a segment represents a logical section of a program's memory</u>, featured by its *basic address* (beginning), by its *limit* (size) and by its *type*. Both basic address and segment's size have 32 bits value representations.

In the family of 8086-based processors, the term **segment** has two meanings:

1. A block of memory of discrete size, called a *physical segment*. The number of bytes in a physical memory segment is
   - (a) 64K for 16-bit processors
   - (b) 4 gigabytes for 32-bit processors.
2. A variable-sized block of memory, called a *logical segment* occupied by a program's code or data.

We will call *offset* the address of a location relative to the beginning of a segment, or, in other words, the number of bytes between the beginning of that segment and that particular memory location. An offset is valid only if his numerical value, on 32 bits, doesn't exceed the segment's limit which refers to.

We will call *address specification* a pair of a *segment selector* and an *offset*. A ***segment selector*** is a numeric value of 16 bits which selects uniquely the accessed segment and his features. A segment selector is defined and provided by the operating system !In hexadecimal an address **specification** can be written as:

$$S_3S_2S_1S_0 : O_7O_6O_5O_4O_3O_2O_1O_0$$

In this case, the selector $s_3s_2s_1s_0$ shows a segment access which has the base address as $b_7b_6b_5b_4b_3b_2b_1b_0$ and a limit $l_7l_6l_5l_4l_3l_2l_1l_0$. The base and the limit are obtained by the processor after performing a segmentation process.

To give access to the specific location, the following condition must be accomplished:

$$o_7o_6o_5o_4o_3o_2o_1o_0 \leq l_7l_6l_5l_4l_3l_2l_1l_0.$$

Based on such a specification the actual segmentation ==address computation== will be performed as:

$$a_7a_6a_5a_4a_3a_2a_1a_0 := b_7b_6b_5b_4b_3b_2b_1b_0 + o_7o_6o_5o_4o_3o_2o_1o_0$$

where $a_7a_6a_5a_4a_3a_2a_1a_0$ is the computed address (hexadecimal form). The above output address is named a *linear address. (or segmentation address).*
An address specification is also named FAR address. When an address is specified only by offset, we call it NEAR address.

A concrete example of an address specification is: **8:1000h**

To compute the linear address corresponding to this specification, the processor will do the following:
1. It checks if the segment with the value 8 was defined by the operating system and blocks the access such a segment wasn't defined; (memory violation error…)
2. It extracts the base address (B) and the segment's limit (L), for example, as a result we may have B – 2000h and L = 4000h;
3. It verifies if the offset exceeds the segment's limit: 1000h > 4000h ? if so, then the access would be blocked;
4. It adds the offset to B and obtains the linear address 3000h (1000h + 2000h). This computation is performed by the **ADR** component from **BIU**.

This kind of addressing is called *segmentation* and we are talking about the *segmented addressing model*.

When the segments start from address 0 and have the maximum possible size (4GiB), any offset is automatically valid and segmentation isn't practically involved in addresses computing. So, having $b_7b_6b_5b_4b_3b_2b_1b_0 = 00000000$, the address computation for the logical address $s_3s_2s_1s_0 : o_7o_6o_5o_4o_3o_2o_1o_0$ will result in the following linear address:

$$a_7a_6a_5a_4a_3a_2a_1a_0 := 00000000 + o_7o_6o_5o_4o_3o_2o_1o_0$$

$$a_7a_6a_5a_4a_3a_2a_1a_0 := o_7o_6o_5o_4o_3o_2o_1o_0$$
$$\Rightarrow$$

This particular mode of using the segmentation, used by most of the modern operating systems is called the *flat memory model*.

The x86 processors also have a memory access control mechanism called *paging*, which is independent of address segmentation. Paging implies dividing the *virtual* memory into *pages*, which are associated (translated) to the available physical memory. (1 page = 2^12 bytes = 4096 bytes).

The configuration and the control of segmentation and paging are performed by the operating system. Of these two, only segmentation interferes with address specification, paging being completely transparent relative to the user programs.

Both addresses computing and the use of segmentation and paging are influenced by the execution mode of the processor, the x86 processors supporting the following more important execution modes:

- *real mode*, on 16 bits (using memory word of 16 bits and having limited memory at 1MiB);
- **protected mode on 16 or 32 bits, characterized by using paging and segmentation;**
- *8086 virtual mode,* allows running real mode programs together with the protected ones;
- *long mode on 64 and 32 bits,* where paging is mandatory while segmentation is deactivated.

In our course we will focus on the architecture and the behavior of x86 processors in protected mode on 32 bits.

The x86 architecture allows 4 types of segments:

- *code segment*, which contains instructions ;
- *data segment*, containing data which instructions work on;
- *stack segment*;
- *extra segment;* (supplementary data segment)

Every program is composed by one or more segments of one or more of the above specified types. At any given moment during run time there is only at most one <u>active</u> segment of any type. Registers **CS** (*Code Segment*), **DS** (*Data Segment*), **SS** (*Stack Segment*) and **ES** (*Extra Segment*) from **BIU** contain <u>the values of the selectors of the active segments</u>, correspondingly to every type. So registers CS, DS, SS and ES ==determine== the starting addresses and the dimensions of the 4 active segments: code, data, stack and extra segments. Registers FS and GS can store selectors pointing to other auxiliary segments without having predetermined meaning. Because of their use, CS, DS, SS ES, FS and GS are called *segment registers* (or *selector registers*). Register **EIP** (which offers also the possibility of accessing its less significant word by referring to the **IP** subregister) contains <u>the offset of the current instruction</u> inside the current code segment, this register being managed exclusively by **BIU.**

Because addressing is fundamental for understanding the functioning of the x86 processor and assembly programming, we review its concepts to clarify them:

Notion	Representation	Description
Address specification, logical address, FAR address	$\text{Selector}_{16}:\text{offset}_{32}$	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).