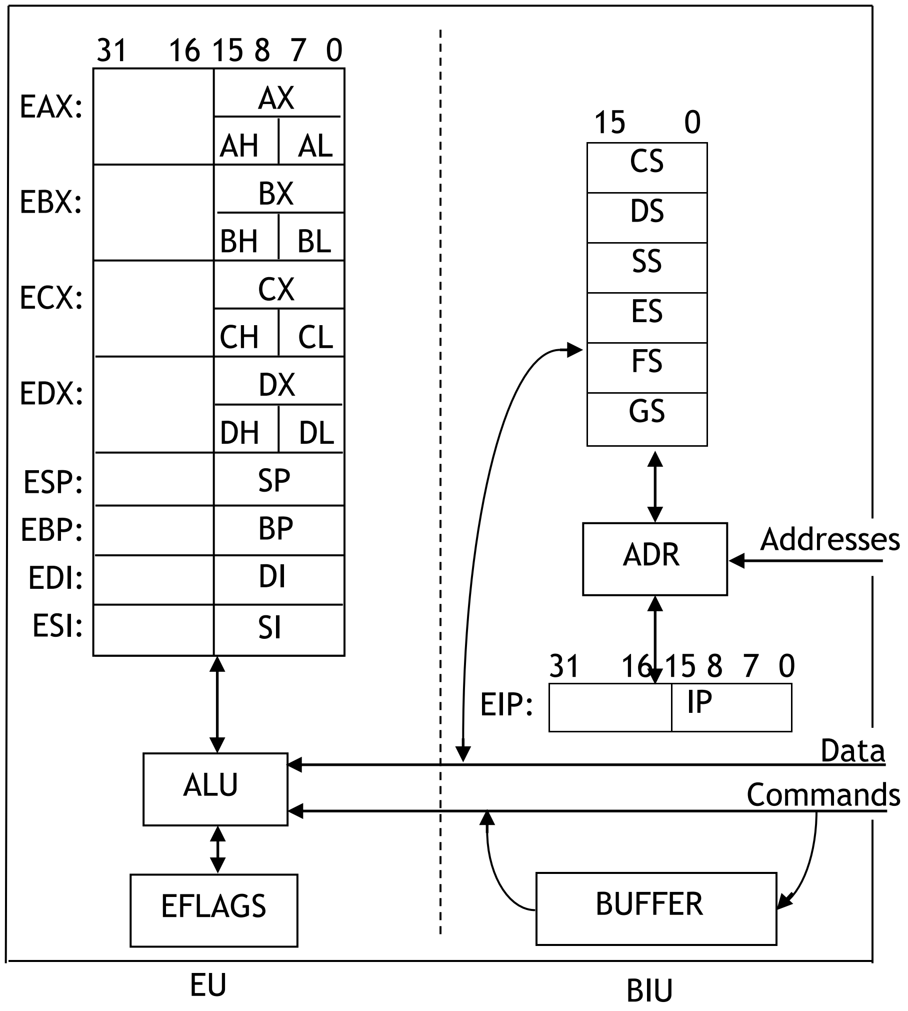
<u>2.6. THE x86 MICROPROCESSOR ARCHITECTURE (IA-32)</u>

<u>2.6.1. x86 Microprocessor's structure</u>

The x86 microprocessor has two main components:

- **EU** (*Executive Unit*) – run the machine instr. by means of **ALU** (*Arithmetic and Logic Unit*) component.
- **BIU** (*Bus Interface Unit*) - prepares the execution of every machine instruction. Reads an instruction from memory, decodes it and computes the memory address of an operand, if any. The output configuration is stored in a 15 bytes buffer, from where EU will take it.



EU and **BIU** work in parallel – while **EU** runs the current instruction, **BIU** prepares the next one. These two actions are synchronized – the one that ends first waits after the other.

2.6.2. The EU general registers

EAX - *accumulator*. Used by the most of instructions as one of their operands.

EBX – *base register*.

ECX - *counter register* – mostly used as numerical upper limit for instructions that need repetitive runs.

EDX – *data register* **-** frequently used with EAX when the result exceed a doubleword (32 bits).

"Word size" refers to the number of bits processed by a computer's CPU in one go (these days, typically 32 bits or 64 bits). Data bus size, instruction size, address size are usually multiples of the word size. So, for a CPU the "word size" is a basic attribute/feature influencing the above mentioned elements.

Just to confuse matters, for backwards compatibility, Microsoft Windows API defines a WORD as being 16 bits, a DWORD as 32 bits and a QWORD as 64 bits, regardless of the processor. So, WORD and DWORD DATA TYPES are ALWAYS on 16 and 32 bits respectively FOR THE ASSEMBLY LANGUAGE , regardless of the CPU's "word size" (16, 32 or 64 bits CPU).

ESP and **EBP** are *stack* registers. The stack is a LIFO memory area.

Register **ESP** (*Stack Pointer*) points to the last element put on the stack (the element from the top of the stack).

Register **EBP** (*Base pointer*) points to the first element put on the stack (points to the stack's basis).

EDI and **ESI** are *index registers* usually used for accessing elements from bytes and words strings. Their functioning in this context (*Destination Index* and *Source Index*) will be clarified in chapter 4.

EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI are doubleword registers (32 bits). Every one of them may also be seen as the concatenation of two 16 bits subregisters. The upper register, which contains the most significant 16 bits of the 32 bits register, doesn't have a name and it isn't available separately. But the lower register could be used as single so we have the 16 bits registers **AX, BX, CX, DX, SP, BP, DI, SI.** Among these registers, AX, BX, CX and DX are also a concatenation of two 8 bits subregisters. So we have **AH, BH, CH, DH** registers which contain the most significant 8 bits of the word (the upper part of AX, BX, CX and DX registers) and **AL, BL, CL, DL** registers which contain the least significant 8 bits of the word (the lower part).

2.6.3. Flags

A *flag* is an indicator represented on 1 bit. A configuration of the FLAGS register shows a synthetic overview of the execution of the each instruction. For x86 the EFLAGS register (the status register) has 32 bits but only 9 are actually used.

31	30	…	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	…	x	OF	DF	IF	TF	SF	ZF	x	AF	x	PF	x	CF

We have 2 flags categories:
a). reporting the status of the LPO (having a so called previous effect) – CF, PF, AF, ZF, SF, OF
- ADC ; Conditional JUMPS (23 instructions – ja = jnbe; jg = jnle ; jz; …)

b). flags to be set by the programmer having a future effect on instructions that follows – CF, TF, IF, DF
- HOW ?... by using SPECIAL intructions – 7 instructions

CF (*Carry Flag*) is the transport flag. It will be set to 1 if in the LPO there was a transport digit outside the representation domain of the obtained result and set to 0 otherwise. For example, in the addition

1001 0011 +	147 +		93h +	-109 +
0111 0011	115	there is transport and CF is set therefore to 1	73h	115
1 0000 0110	262		106h	06

CF flags the UNSIGNED overflow !

PF (*Parity Flag*) – Its value is set so that together with the bits 1 from the least significant byte of the representation of the LPO's result an odd number of 1 digits to be obtained.

AF (*Auxiliary Flag*) shows the transport value from bit 3 to bit 4 of the LPO's result. For the above example the transport is 0.

ZF (*Zero Flag*) is set to 1 if the result of the LPO was zero and set to 0 otherwise.

SF (*Sign Flag*) is set to 1 if the result of the LPO is a strictly negative number and is set to 0 otherwise.

TF (*Trap Flag*) is a debugging flag. If it is set to 1, then the machine stops after every instruction.

IF (*Interrupt Flag*) is an interrupt flag. If set to 1 interrupts are allowed, if set to 0 interrupts will not be handled.
More details about IF can be found in chapter 5 (Interrupts).

DF (*Direction Flag*) – for operating string instructions. If set to 0, then string parsing will be performed in an ascending order (from the beginning to its end) and in a descending order if set to 1.

OF (*Overflow Flag*) flags the **signed overflow**. If the result of the LPO (considered in the signed interpretation) didn't fit the reserved space, then OF will be set to 1 and will be set to 0 otherwise.

Flags categories

The flags can be split into 2 categories:

a). with a previous effect generated by the Last Performed Operation (LPO): CF, PF, AF, ZF, SF and OF
b). having a future effect after their setting by the programer, to influence the way the next instructions are run: CF, TF, DF and IF.

Specific instructions to set the flags values

Considering the b) category, it is normal that the assembly language to provide specific instructions to set the values of the flags that will have a future effect. So, we have 7 such instructions:

CLC – the effect is CF=0
STC – sets CF=1
CMC – complements the value of the CF ; 3 instructions for CF

CLD – sets DF=0
STD – sets DF=1 ; 2 instructions for DF

CLI – sets IF=0
STI – sets IF=1 ; 2 instructions for IF – they can be used by the programmer only on 16 bits programming ; on 32 bits, the OS restricts the access to these instructions !

Given the major risk of accidentally setting the value from TF and also its absolutely special role to develop debuggers, there are NO instructions to directly access the value of TF !!

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, a segment represents a logical section of a program's memory, 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:

$$\mathbf{a_7a_6a_5a_4a_3a_2a_1a_0 := 00000000 + o_7o_6o_5o_4o_3o_2o_1o_0}$$

$$\Rightarrow$$

$$\mathbf{a_7a_6a_5a_4a_3a_2a_1a_0 := o_7o_6o_5o_4o_3o_2o_1o_0}$$

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 <u>protected mode on 32 bits</u>.

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 active segment of any type. Registers **CS** (*Code Segment*), **DS** (*Data Segment*), **SS** (*Stack Segment*) and **ES** (*Extra Segment*) from **BIU** contain the values of the selectors of the active segments, 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 the offset of the current instruction 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	$Selector_{16}$:$offset_{32}$	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).

2.6.5. Machine instructions representation

A x86 machine instruction represents a sequence of 1 to 15 bytes, these values specifying an operation to be run, the operands to which it will be applied and also possible supplementary modifiers.

A x86 machine instruction has maximum 2 operands. For most of the instructions, they are called *source* and *destination* respectively. From these two operands, only one may be stored in the RAM memory. The other one must be either one **EU** register, either an integer constant. Therefore, an instruction has the general form:

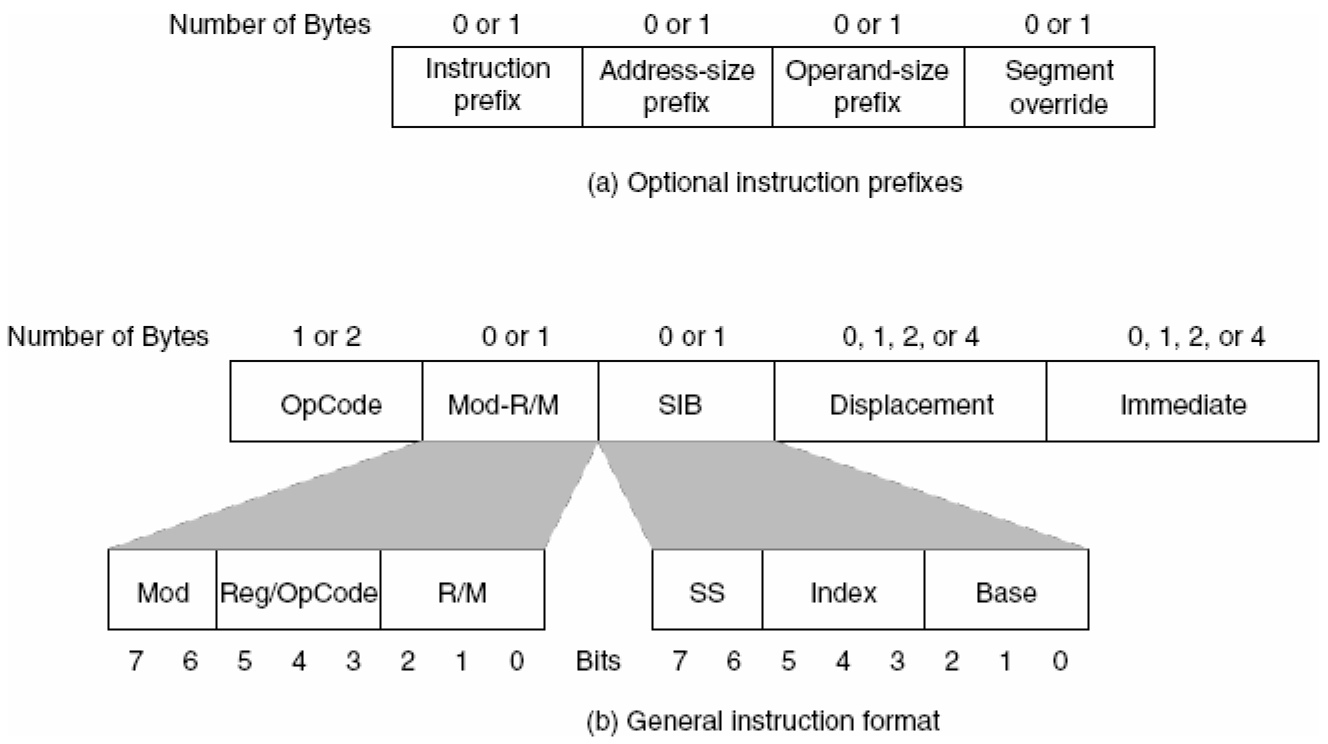
instruction_name *destination***,** *source*

The internal format of an instruction varies between 1 and 15 bytes, and has the following general form (*Instructions byte-codes from OllyDbg)*:

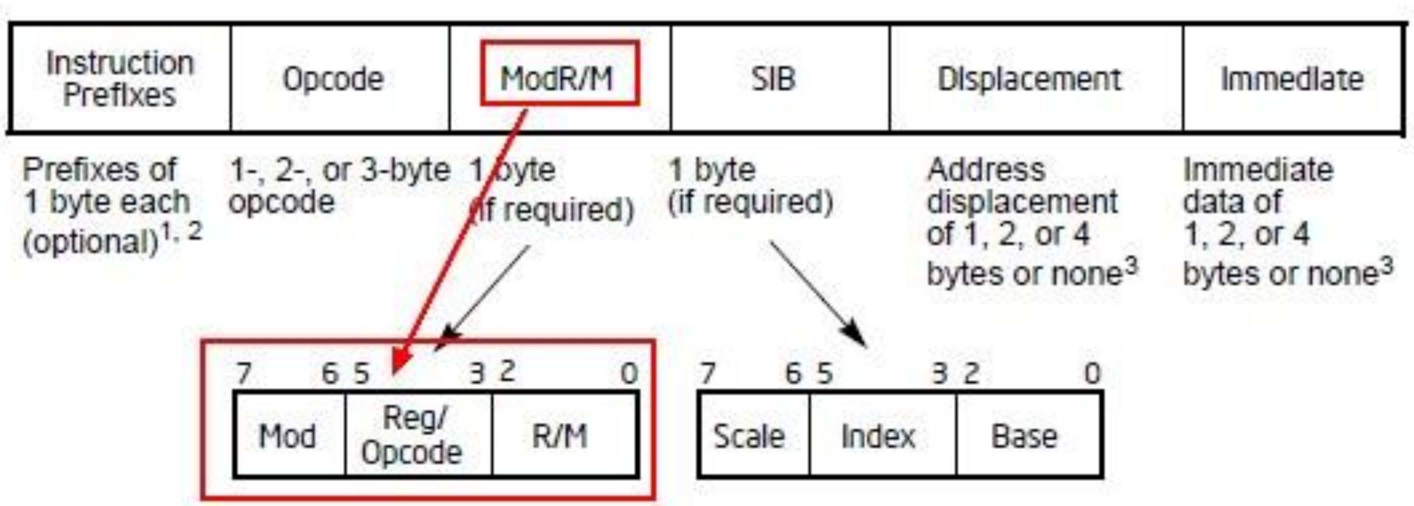
[prefixes] + code + [ModeR/M] + [SIB] + [displacement] + [immediate]

The *prefixes* control how an instruction is executed. These are optional (0 to maxim 4) and occupy one byte each. For example, they may request repetitive execution of the current instruction or may block the address bus during execution to not allow concurrent access to operands and results.

The operation to be run is identified by 1 to 2 bytes of *code* (opcode), which are the only mandatory bytes, no matter of the instruction. The byte *ModeR/M* (register/memory mode) specifies for some instructions the nature and the exact storage of operands (register or memory). This allows the specification of a register or of a memory location described by an offset.



Although the diagram seems to imply that instructions can be up to 16 bytes long, in actuality the x86 will not allow instructions greater than 15 bytes in length.



For more complex addressing cases than the one implemented directly by ModeR/M, combining this with SIB byte allows the following formula for an offset (*http://datacadamia.com/intel/modrm*):

[base] + [index × scale] + [constant]
(SIB) (displacement+immediate)

where for base and index the value of two registers will be used and the scale is 1, 2, 4 or 8. The allowed registers as base or/ and as indexes are: EAX, EBX, ECX, EDX, EBP, ESI, EDI. The ESP register is available as base but cannot be used as index (http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0100_sib_byte_layout).

Most of the instructions use for their implementation either only the opcode or the opcode followed by ModeR/M.

The *displacement* is present in some particular addressing forms and it comes immediately after ModeR/M or SIB, if SIB is present. This field can be encoded either on a byte or on a doubleword (32 bits).

The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or *direct*) addressing mode. **The displacement-only addressing mode consists of a 32-bit constant that specifies the address of the target location**. The displacement-only addressing mode is perfect for accessing simple scalar variables. Intel named this the displacement-only addressing mode because a 32-bit constant (displacement) follows the MOV opcode in memory. **On the 80x86 processors, this displacement is an offset** from the beginning of memory (that is, address zero).

Displacement mode, **the operand's offset** is contained as part of the instruction as an 8-, 16-, or 32-bit displacement. The displacement addressing mode is found on few machines because, as mentioned earlier, it leads to long instructions. In the case of the x86, the displacement value can be as long as 32 bits, making for a 6-byte instruction. Displacement addressing can be useful for referencing global variables.

As a consequence of the impossibility of appearing more than one ModeR/M, SIB and displacement fields in one instruction, the x86 architecture doesn't allow encoding of two memory addresses in the same instruction.

With the *immediate value* we can define an operand as a numeric constant on 1, 2 or 4 bytes. When it is present, this field appears always at the end of instruction.

2.6.6. FAR addresses and NEAR addresses.

To address a RAM memory location two values are needed: one to indicate the segment and another one to indicate the offset inside that segment. For simplifying the memory reference, the microprocessor implicitly chooses, in the absence of other specification, the segment's address from one of the segment registers CS, DS, SS or ES. The implicit choice of a segment register is made after some particular rules specific to the used instruction.

An address for which only the offset is specified, the segment address being implicitly taken from a segment register is called a *NEAR address*. A NEAR address is always inside one of the 4 active segments.

An address for which the programmer explicitly specifies a segment selector is called a *FAR address.* So, a FAR address is a COMPLETE ADDRESS SPECIFICATION and it may be specified in one of the following 3 ways:

- $s_3s_2s_1s_0$ *: offset_specification* where $s_3s_2s_1s_0$ is a constant;
- segment register: offset_specification, where segment registers are CS, DS, SS, ES, FS or GS;
- FAR [variable], where variable is of type QWORD and contains the 6 bytes representing the FAR address.

The internal format of an FAR address is: at the smallest address is the offset, and at the higher (by 4 bytes) address (the word following the current doubleword) is the word which stores the segment selector.

The address representation follows the little-endian representation presented in Chapter 1, paragraph 1.3.2.3: the less significant part has the smallest address, and the most significant one has the higher address.

2.6.7. <u>Computing the offset of an operand. Addressing modes.</u>

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

- *register mode*, if the required operand is a register;   mov eax, 17
- *immediate mode*, when we use directly the operand's value (not its address and neither a register holding it); mov eax, 17
- *memory addressing mode*, if the operand is located somewhere in memory. In this case, its offset is computed using the following formula:

$$\textit{offset\_address} = [\text{ base}] + [\text{ index} \times \text{scale }] + [\text{ constant }]$$

So *offset_address* is obtained from the following (maximum) four elements:

- the content of one of the registers EAX, EBX, ECX, EDX, EBP, ESI, EDI or ESP as base;
- the content of one of the registers EAX, EBX, ECX, EDX, EBP, ESI or EDI as index;
- scale to multiply the value of the index register with 1, 2, 4 or 8;
- the value of a numeric constant, on a byte or on a doubleword.

From here results the following modes to address the memory:

- *direct addressing*, when only the *constant* is present;
- *based addressing*, if in the computing one of the base registers is present;
- *scale-indexed addressing*, if in the computing one of the index registers is present.

These three mode of addressing could be combined. For example, it can be present direct based addressing, based addressing and scaled-indexed etc.

A non direct addressing mode is called ***indirect addressing*** (based and/or indexed). So, an indirect addressing is a one for which we have at least one register specified between squared brackets.

In the case of the jump instructions another type of addressing is present called *relative* addressing.

Relative addressing indicates the position of the next instruction to be run relative to the current position. This ”distance“ is expressed as the number of bytes to jump over. The x86 architecture allows relative SHORT addresses, described on a byte and having values between -128 and 127, but also relative NEAR addresses, represented on a doubleword with values between -2147483648 and 2147483647.

```
Jmp Below2  ; this instruction will be translated into (see OllyDbg) usually in something as Jmp [0084]↓
………….
………….
Below2:
   Mov eax, ebx
```