2's complement. Example.

1001 0011 (= 93h = 147), so in the UNSIGNED interpretation 1001 0011= 147

Being a binary number starting with 1, in the SIGNED interpretation, this number is a negative one. Which is its value ?

Answer: Its value is: – (2's complement of the initial binary configuration)

So, we have to determine the 2's complement of the configuration 1001 0011
How can we obtain the 2's complement of a number (represented in memory so we are talking about base 2) ?

Variant 1 (Official): Subtracting the binary contents of the location from 100 ...00 (where the number of zero's are exactly the same as the number of bits of the location to be complemented).

```
1 0000 0000 –
  1001 0011
---------------
  01101101      =   6Dh = 96+13 = 109 (so the 2'complement on 8 bits of 147 is 109)
```

So, the value of 1001 0011 in the SIGNED interpretation is -109

Variant 2 (derived from the 2's complement definition – faster from a practical point of view): reversing the values of all bits of the initial binary number (value 0 becomes 1 and value 1 becomes, after which we add 1 to the obtained value.

According to this rule, we start from 1001 0011 and reverse the values of all bits, obtaining 0110 1100 after which we add 1 to the obtained value: 0110 1100 + 1 = 0110 1101

So, the value of 1001 0011 in the SIGNED interpretation is -109

Variant 3 (MUCH MORE faster practically for obtaining the binary configuration of the 2's complement): We left unchanged the bits starting from the right until to the first bit 1 inclusive and we reverse the values of all the other bits (all the bits from the left of this bit with value 1).

Applying this rule, we start from 1001 0011 and left unchanged all the bits starting from the right until to the first bit 1 inclusive (in our case this means only the first bit 1 from the right – which is the only one that is left unchanged) and all other bits will be reversed, so we obtain…0110 1101 = 6DH = 109

So, the value of 1001 0011 in the SIGNED interpretation is -109

Variant 4 (the MOST faster practical alternative, if we are interested ONLY in the absolute value in base 10 of the 2's complement):

Rule derived from the definition of the 2's complement: The sum of the absolute values of the two complementary values is the cardinal of the set of values representable on that size.

On 8 bits we can represent 2^8 values = 256 values ([0..255] or [-128..+127])

- On 16 bits we can represent 2^16 values = 65536 values ([0..65535] or [-32768,+32767])
- On 32 bits we can represent 2^32 valori = 4.294.967.296 values (…)

So, on 8 bits, the 2's complement of 1001 0011 (= 93h = 147) is 256 – 147 = 109, so the corresponding value in SIGNED interpretation for 1001 0011 is -109.

[0..255] – admissible representation interval for “UNSIGNED integer represented on 1 byte”
[-128..+127] – admissible representation interval for “SIGNED integer represented on 1 byte”

[0..65535] – admissible representation interval for “UNSIGNED integer represented on 2 bytes = 1 word”
[-32768..+32767] – admissible representation interval for “SIGNED integer represented on 2 bytes = 1 word”

Why do we need to study the 2’s complement ? is it useful for us as programmers ? In which way ?...

1001 0011 (= 93h = 147), so in the UNSIGNED interpretation 1001 0011= 147

Which is the signed interpretation of 1001 0011 ? a). 01101101 b).-109 c). 6Dh
Which is the signed interpretation of 93h ? a). 01101101 b).-109 c). 6Dh
Which is the signed interpretation of 147 ? a). 01101101 b).-109 c). 6Dh d).+147 (THE QUESTION IS TOTALLY INCORRECT ¡!!!!!! – because we cannot have DIFFERENT interpretations in base 10 of numbers ALREADY expressed in base 10)

1 0000 0000 –
1001 0011

01101101 = 6Dh = 96+13 = 109 (so the 2’complement on 8 bits of 147 is 109)

So, the value of 1001 0011 in the SIGNED interpretation is … -109

147 and -109 are two complementary values, in the sense that 1001 0011 = either 147, or -109 depending on the interpretation.

So the complement of 147 is -109. Is it also true the other way around ? Is -147 the complement of the 109 ?...

Let's check…109 = 01101101 , the 2's complement of 01101101 is 10010011 = 147, so... Which is the conclusion then ?...

Which is the binary representation for -147 ?

147 = 10010011, so … how can we obtain -147 ?

-147 DOES NOT belong to the interval [-128..+127] – admissible representation interval for "SIGNED integer represented on 1 byte", so it follows that -147 CAN NOT be represented on 1 byte !!

-147 belongs to [-32768..+32767] – admissible representation interval for "SIGNED integer represented on 2 bytes = 1 word", so IN ASSEMBLY LANGUAGE -147 can be represented ONLY on Word !

147 on WORD is 00000000 10010011, its 2's complement being 11111111 01101101, so

11111111 01101101 = FF6Dh = -147 in the SIGNED interpretation

Check: 11111111 01101101 = FF6Dh = 65389 in the UNSIGNED interpretation, the sum of the absolute values of the two complementary values being 65389 + 147 = 65536 = the cardinal of the set of numbers representable on 1 WORD, so these 2 interpretations of the binary configuration 11111111 01101101 are correct and consistent !!

Two complementary values WILL NEVER BE part of the same admissible representation interval !!!!

-128, 128; 147, -109; -1, 255;

1000 0000 = 80h = 128 (unsigned) = - 128 (in base 2 we may say that 80h has as its 2's complement exactly itself = 80h).

Which is the MINIMUM number of BITS on which we can represent -147 ?

- On n bits we may represent 2^n values: - either the UNSIGNED values [0..2^n – 1]
  - or the SIGNED values [-2^(n-1), 2^(n-1)-1]

On 8 bits we can though represent 2^8 values (=256 values), either [0..2^8-1] = [0..255] in the UNSIGNED interpretation, either [-2^(8-1), 2^(8-1)-1] = [-2^7, 2^7-1] = [-128..+127] in the SIGNED interpretation

On 9 bits… [0..511] or [-256..+255] and because -147 € [-256..+255] it follows that the MINIMUM number of bits on which we may represent -147 is 9 and -147 representation is:

(…On 9 bits we may represent 512 numbers, 512-147 = 365 = 1 6Dh = 1 0110 1101 …)

So 1 0110 1101 = 16Dh = 256 + 6*16 + 13 = 256 + 96 + 13 = 365 in the UNSIGNED interpretation !

1 0110 1101 = -(2's complement of 1 0110 1101) = -(0 1001 0011) = -(093h)= -147

As a DATA TYPE in ASM, obviously that we have to enroll any value as being a byte, word or dword, so -147 € [-32768..+32767] and accordingly to the above discussion wew have -147 = 11111111 01101101 = FF6Dh as a value represented on 1 word = 2 bytes.

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	(=6?)		106h	06

CF flags the UNSIGNED overflow !

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.

Definition. An *overflow* is mathematical situation/condition which expresses the fact that the result of an operation didn't fit the reserved space for it.

At the level of the assembly language an *overflow* is situation/condition which expresses the fact that the result of the LPO didn't fit the reserved space for it OR does not belong to the admissible representation interval OR that operation is a mathematical nonsense in that particular interpretation (signed or unsigned).

CF vs. OF. The overflow concept.

1001 0011 +	147 +		93h +	-109 +
1011 0011	179	a carry of the most significant digit occurs	B3h	-77
1 0100 0110	326	so the value 1 is placed in CF	146h	-186 !!!!
	(unsigned)		**(hexa)**	**(signed)**
	CF= 1			**OF= 1**

- 326 and -186 are the correct results in base 10 for the corresponding interpretations

BUT, in ASSEMBLY language we have ADD b+b → b, so what we obtain as interpretations on 1 byte are :

1001 0011 +	147 +		93h +	-109 +
1011 0011	179	a carry of the most significant digit occurs	B3h	-77
1 0100 0110	70	so the value 1 is placed in CF	146h	+70 !!!!
	(unsigned)		**(hexa)**	**(signed)**
	CF= 1			**OF=1**

By setting both CF and OF to 1, the « message » from the assembly language is that both interpretations in base 10 of the base 2 addition are incorrect mathematical operations !

0101 0011 +	83 +		53h +	83 +
0111 0011	115	a carry DOES NOT occur so CF=0	73h	115
1100 0110	198		C6h	198 !!!!
	(unsigned)		**(hexa)**	**(signed)**
	CF=0			**OF=1**

- 198 is the correct result in base 10 for both the corresponding interpretations

BUT, in ASSEMBLY language we have ADD b+b → b, so what we obtain as interpretations on 1 byte are :

0101 0011 +	83 +		53h +	83 +
0111 0011	115	a carry DOES NOT occur so CF=0	73h	115
1100 0110	198		C6h	- 58 !!!!
	(unsigned)		**(hexa)**	**(signed)**
	CF=0			**OF=1**

Setting CF to 0 expresses the fact that the unsigned interpretation in base 10 of the base 2 addition is a correct one and the operation functions properly. Setting OF to 1 means that the signed interpretation is NOT a correct mathematical operation !

OF will be set to 1 (*signed overflow*) if for the addition operation we are in one of the following situations (*overflow rules for addition in signed interpretation*). These are the only 2 situations that can issue overflow status for the addition operation:

0……+	or	1…..+	(Semantically, the two situations denote the impossibility of mathematical acceptance
0……		1…..	of the 2 operations: we cannot add two positive numbers and obtain a negative result
--------		-------	and we cannot add two negative numbers and obtain a positive result).
1……		0…..	

In the case of subtraction, we also have two overflow rules in the signed interpretation, a consequence of the two overflow rules for addition:

1……-	sau	0…..-	(Semantically, the two situations denote the impossibility of mathematical acceptance
0……		1…..	of the two operations: we cannot subtract a positive number from a negative one and
------		-------	obtaining a positive one, neither can we subtract a negative number from a positive one
0……		1…..	and obtaining a negative one).

1 0110 0010 -	98 -		62h -	98 -
1100 1000	200	We need significant digit borrowing for performing	C8h	-56
1001 1010	154	the subtraction, therefore CF is set to 1	9Ah	-102 !!!!
(unsigned)			**(hexa)**	**(signed)**
CF=1				**OF=1**

None of the interpretations above is mathematically correct, because in base 10, the subtraction 98-200 should provide -102 as the correct result, but instead 154 is provided in the unsigned interpretation for the subtraction - which is an incorrect result! In the SIGNED interpretation we have: 98-(-56) = -102 (Again an incorrect value!!) instead of the correct one 98+56=154 (the value of 154 result interpretation is valid only in unsigned representation).

In conclusion, in order to be mathematically correct, the results of the two subtractions from above should be switched between the two interpretations; so the two interpretations associated to the binary subtraction are both mathematically incorrect. For this reason the 80x86 microprocessor will set CF=1 and OF =1.

Technically speaking, the microprocessor will set OF=1 only in one of the 4 situations presented above (2 for addition and 2 for subtraction).

The multiplication operation does not produce overflow at the level of 80x86 architecture, the reserved space for the result being enough for both interpretations. Anyway, even in the case of multiplication, the decision was taken to set both CF=OF=0, in the case that the size of the result is the same as the size of the operators (b*b = b, w*w = w or d*d = d) (« no multiplication overflow », CF = OF = 0). In the case that b*b = w, w*w = d, d*d = qword, then CF = OF = 1 (« multiplication overflow »).

The worst effect in case of overflow is in the case for the division operation: in this situation, if the quotient does not fit in the reserved space (the space reserved by the assembler being byte for the division word/byte, word for the division doubleword/word and respectively doubleword for division quadword/doubleword) then the « division overflow » will signal a ‘Run-time error’ and the operating system will stop the running of the program and will issue one of the 3 semantic equivalent messages: ‘Divide overflow’, ‘Division by zero’ or ‘Zero divide’.

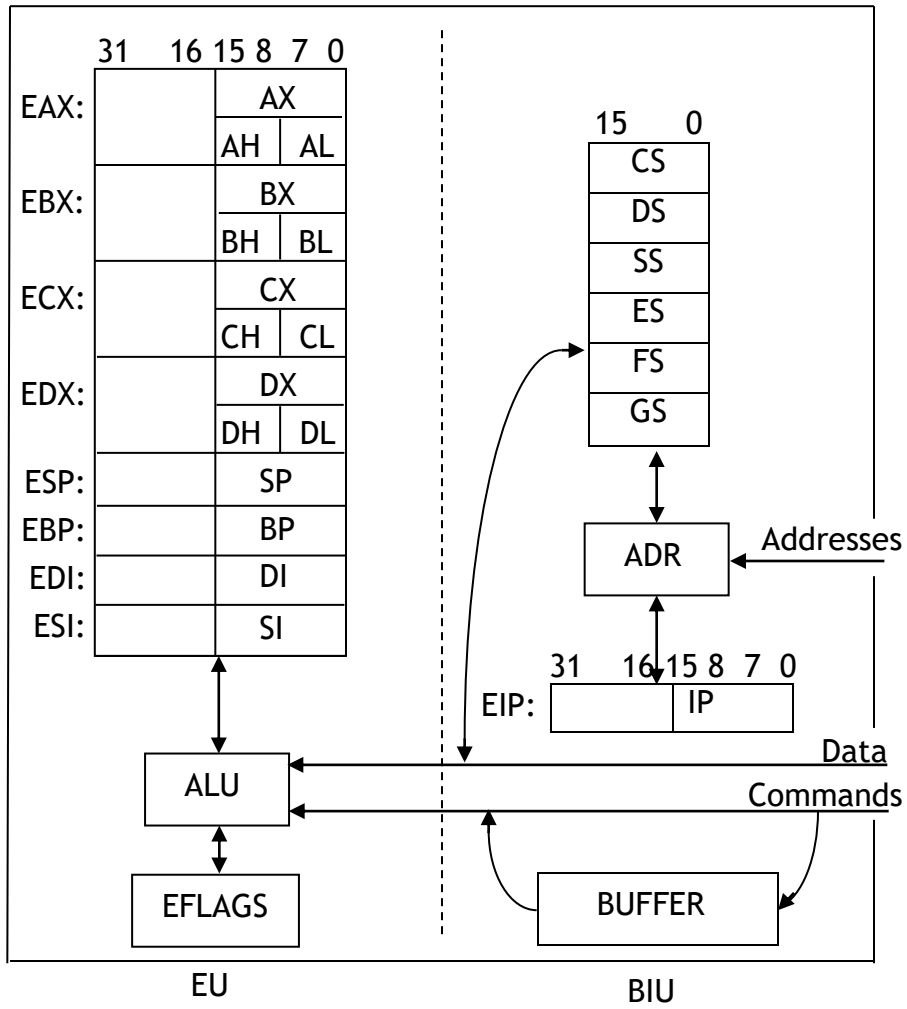
In the case of a correct division CF and OF are undefined. If we have a division overflow, the program crashes, the execution stops and of course it doesn’t matter which are the values from CF and OF…

2.6. THE x86 MICROPROCESSOR ARCHITECTURE (IA-32)

2.6.1. x86 Microprocessor's structure

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:

$$\mathbf{S_3S_2S_1S_0 : O_7O_6O_5O_4O_3O_2O_1O_0}$$

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:

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

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

⇨

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 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 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:

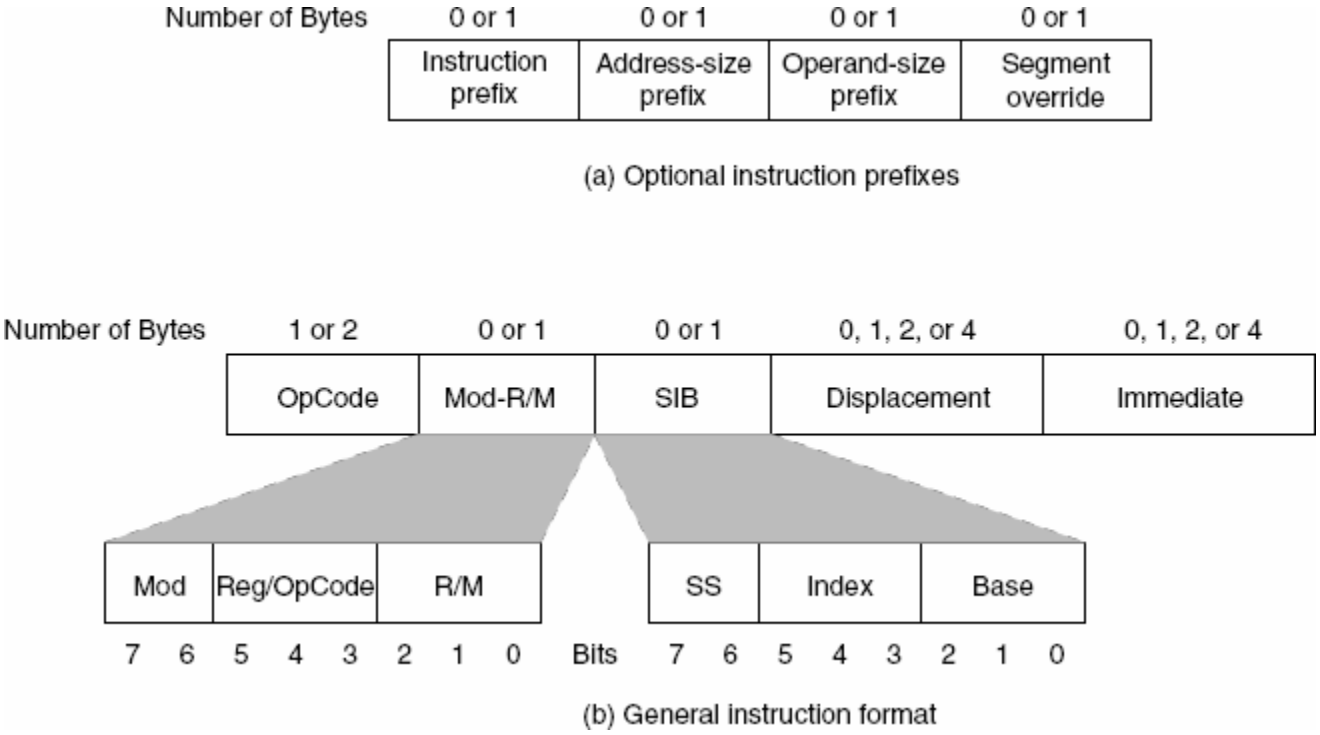
instruction_name *destination**,** source*

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

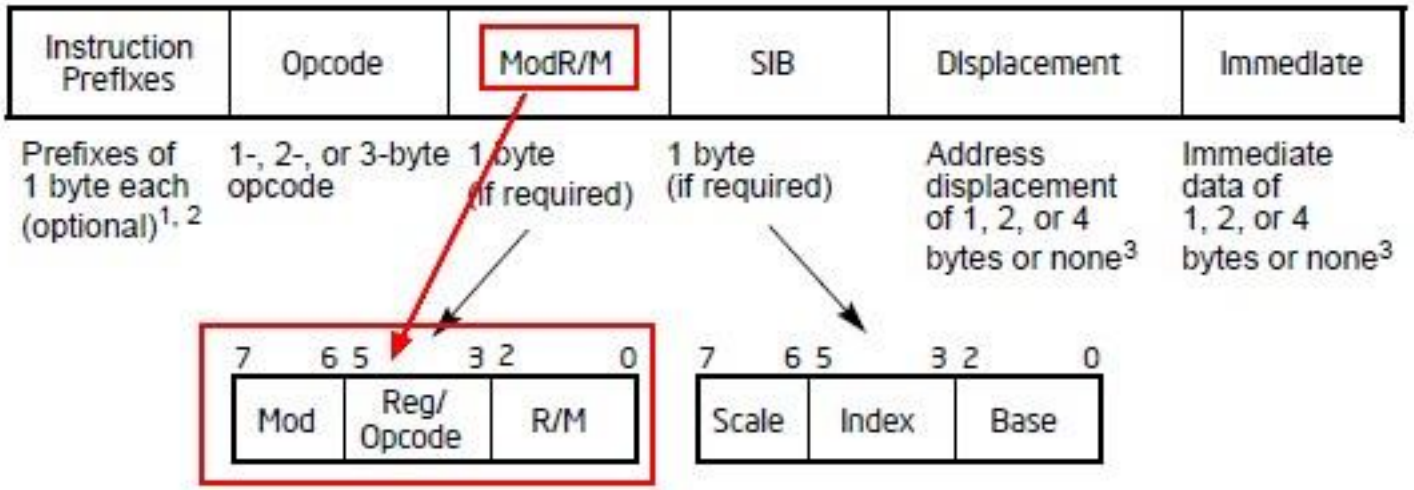
[prefixes] + code + [ModeR/M] + [SIB] + [displacement] + [immediate]

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. Computing the offset of an operand. Addressing modes.

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

C H A P T E R 3

ASSEMBLY LANGUAGE BASICS

Machine Language of a Computing System (CS) – the set of the machine instructions to which the processor directly reacts. These are represented as bit strings with predefined semantics.

Assembly Language – a programming language in which the basic instructions set corresponds with the machine operations and which data structures are the machine primary structures. This is a **symbolic language**. **Symbols - Mnemonics + labels.**

The basic elements with which an assembler works with are:

* **labels** – user-defined names for pointing to data or memory areas.

* **instructions** - mnemonics which suggests the underlying action. The assembler generates the bytes that codifies the corresponding instruction.

* **directives** - indications given to the assembler for correctly generating the corresponding bytes. Ex: relationships between the object modules, segment definitions, conditional assembling, data definition directives.

* **location counter** – an integer number managed by the assembler for every separate memory segment. At any given moment, the value of the location counter is the number of the generated bytes correspondingly with the instructions and the directives already met in that segment (the current offset inside that segment). The programmer can use this value (read-only access!) by specifying in the source code the '$' symbol.

NASM supports two special tokens in expressions, allowing calculations to involve the current assembly position: the $ and $$ tokens. $ evaluates to the assembly position at the beginning of the line containing the expression; so you can code an infinite loop using JMP $.
$$ evaluates to the start of the current section; so you can tell how far into the section are by using ($-$$).

Directive SECTION

```
section .data
   db 'hello'
   db  'h', 'e', 'l','l','o'
   data_segment_size  equ $-$$
```

$-$$ = the distance from the beginning of the segment AS A SCALAR (constant numerical value)!!!!!!!!!!

$ - is an offset = POINTER TYPE !!!! It is an address !!!!
$$ - is an offset =POINTER TYPE !!!! It is an address !!!!

$ means "address of here".
$$ means "address of start of current section".
So $-$$ means "current size of section".

For the example above, this will be 10, as there are 10 bytes of data given.

3.1. SOURCE LINE FORMAT

In the x86 assembly language the source line format is:

[***label[:]***] [***prefixes***] [***mnemonic***] [***operands***] [***;comment***]

We illustrate the concept through some examples:

```
here:    jmp here          ; label + mnemonic + operand + comment
repz cmpsd                 ; prefix + mnemonic + comment
start:                     ; label + comment
                           ; just a comment (which could be missed)
a   dw  19872, 42h         ; label + mnemonic + 2 operands + comment

 len equ $-a  ; label + mnemonic + $-a (operand) + comment
```

The allowed characters for a *label* are:

- Letters: A-Z, a-z;
- Digits: 0-9;
- Characters _, $, $$, #, @, ~, . and ?

A valid variable name starts with a letter, _ or ?.
These rules are valid for all valid *identifiers* (symbolic names, such as variable names, label names, macros, etc).

All identifiers are *case sensitive*, the language making the distinction between upper and lower case letters while analyzing user defined identifiers. This means that the Abc identifier is different from the abc identifier. For implicit names which are part of the language (such as keywords, mnemonics, registers) there are no differences between upper and lower case letters (they are *case insensitive*).

The assembly language offers two categories of labels:

1). ***Code labels***, present at the level of instructions sequences for defining the destinations of the control transfer during a program execution. **They can appear also in data segments!**

2). ***Data labels***, which provide symbolic identification for some memory locations, from a semantic point of view being similar with the *variable* concept from the other programming languages. **They can appear also in code segments!**

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

The distinction between accessing a variable's address or its associated content is made as follows:

- When specified in ***straight brackets, the variable name denotes the value of the variable***; for example, [p] specifies accessing the value of the variable, in the same way in which *p represents dereferencing a pointer (accessing the content indicated by the pointer) in C;
- In any other context, ***the name of the variable represents the address of the variable***; for example, p is always the address of the variable p;

Examples:

```
mov EAX, et      ; loads into EAX register the address (offset) of data or code starting at label et
mov EAX, [et]    ; loads into EAX register the content from address et (4 bytes)
lea eax, [v]     ; loads into EAX register the address (offset) of variable v (4 bytes)
```

(similar as effect with MOV eax, v)

As a generalization, ***using straight brackets always indicates accessing an operand from memory***. For example, mov EAX, [EBX] means the transfer of the memory content whose address is given by the value of EBX into EAX (4 bytes are taken from memory starting at the address specified in EBX as a pointer).

There are 2 types of *mnemonics*: *instructions names* and *directives names*. *Directives* guide the assembler. They specify the particular way in which the assembler will generate the object code. *Instructions* are actions that guide the processor.

Operands are parameters which define the values to be processed by the instructions or directives. They can be **registers, constants, labels, expressions, keywords** or **other symbols**. Their semantics depends on the mnemonic of the associated instruction or directive.

3.2. EXPRESSIONS

expression - operands + operators. *Operators* indicate how to combine the operands for building an expression. **Expressions are evaluated at assembly time** (their values are computable at assembly time, except for the operands representing registers contents, that can be evaluated only at run time – the offset specification formula).

3.2.1. Operands specification modes

Instructions operands may be specified in 3 different ways, called ***specification modes.***

The 3 operand types are: ***immediate operands, register operands*** and ***memory operands***. Their values are computed at assembly time for the immediate operands and for the direct addressed operands (the offset part only!), at loading time for memory operands in direct addressing mode (as a complete FAR address – segment address is determinable here so the whole FAR address is known now !) – this step involves a so called ADDRESS RELOCATION PROCESS (adjusting an address by fixing its segment part), and at run time for the registers operands and for indirectly accessed memory operands.

??:offset (assembly time) 0708:offset (loading time)

3.2.1.1. Immediate operands

Immediate operands are constant numeric data computable at assembly time.

Integer constants are specified through binary, octal, decimal or hexadecimal values. Additionally, the use of the _ (underscore) character allows the separation of groups of digits. The numeration base may be specified in multiple ways:

- Using the H or X suffixes for hexadecimal, D or T for decimal, Q or O for octal and B or Y for binary; in these cases the number must start with a digit between 0 and 9, to eliminate confusions between constants and symbols, for example 0ABCH is interpreted as a hexadecimal number, but ABCH is interpreted as a symbol.
- Using the C language convention, by adding the 0x or 0h prefixes for hexadecimal, 0d or 0t for decimal, 0o or 0q for octal, and 0b or 0y for binary.

Examples:

- the hexadecimal constant B2A may be expressed as: 0xb2a, 0xb2A, 0hb2a, 0b12Ah, 0B12AH

- the decimal value 123 may be specified as: 123, 0d123, 0d0123, 123d, 123D, ...
- 11001000b, 0b11001000, 0y1100_1000, 001100_1000Y represent various ways of expressing the binary number 11001000

The offsets of data labels and code labels are values computable at assembly time and they remain constant during the whole program's run-time.

```
mov eax, et     ; transfer into the EAX register the offset associated to the et label
```

will be evaluated at assembly time as (for example):

```
mov eax, 8          ; 8 bytes „distance" relative to the beginning of the data segment
mov eax, [var] – in OllyDBG you will find   mov eax, DWORD PTR [DS:004027AB]
```

These values are constant because of the allocation rules in programming languages in general. These rules state that the memory allocation order of declared variables (more precisely the distance relative to the start of the data segment in which a variable is allocated) as well as the distances of destination jumps in the case of **goto** - style instructions are constant values during the execution of a program.

In other words, a variable once allocated in a memory segment will never change its location (i.e. its position relative to the start of that segment). This information is determinable at assembly time based upon the order in which variables are declared in the source code and due to the dimension of representation inferred from the associated type information.

3.2.1.2. Register operands

Direct using - **mov eax, ebx**
Indirect usage and addressing – used for pointing to memory locations - **mov eax, [ebx]**

3.2.1.3. Memory addressing operands

There are 2 types of memory operands: *direct addressing operands* and *indirect addressing operands*.

The *direct addressing operand* is a constant or a symbol representing the address (segment and offset) of an instruction or some data. These operands may be *labels* (for ex: jmp et), *procedures names* (for ex: call proc1) or *the value of the location counter* (for ex: b db $-a).

The offset of a direct addressing operand is computed at assembly time. The address of every operand relative to the executable program's structure (establishing the segments to which the computed offsets are relative to) is computed ***at linking time***. The actual physical address is computed ***at loading time***.

The effective address always refers to a segment register. This register can be explicitly specified by the programmer, or otherwise a segment register is implicitly associated by the assembler. The implicit rules for performing this association WITH AN EXPLICIT SPECIFIED OFFSET OPERAND are:

- **CS** for code labels target of the control transfer instructions (jmp, call, ret, jz etc);
- **SS** in SIB addressing when using EBP or ESP as *base* (no matter of *index* or *scale*);
- **DS** for the rest of data accesses;

Explicit segment register specification is done using the segment prefix operator "**:**"
ES can be used only in explicit specifications (like ES:[Var] or ES:[ebx+eax*2-a]) or IN CERTAIN STRING INSTRUCTIONS (MOVSB)

JMP FAR CS:…
JMP FAR DS:… or JMP FAR [label2]

3.2.1.4. Indirect addressing operands

Indirect addressing operands use registers for pointing to memory addresses. Because the actual registers values are known only at run time, indirect addressing is suited for dynamic data operations.

The general form for indirectly accessing a memory operand is given by the offset computing formula:

[base_register + index_register* scale + constant**]**

Constant is an expression which value is computable at assembly time. For ex. [ebx + edi + table + 6] denotes an indirect addressed operand, where both *table* and *6* are constants.

The operands *base_register* and *index_register* are generally used to indicate a memory address referring to an array. In combination with the scaling factor, the mechanism is flexible enough to allow direct access to the elements of an array of records, with the condition that the byte size of one record to be 1, 2, 4 or 8. For example, the upper byte of the DWORD element with the index given in ECX, part of a record vector which address (of the vector) is in edx can be loaded in dh by using the instruction

```
mov dh, [edx + ecx * 4 + 3]
```

From a syntactic point of view, when the operand is not specified by the complete formula, some of the components missing (for example when "* scale" is not present), the assembler will solve the possible ambiguity by an analysis process of all possible equivalent encoding forms, choosing the shortest finally. For example, having

```
push dword [eax + ebx]   ; saves on the stack the doubleword from the address eax+ebx
```

the assembler is free to consider eax as the base and ebx as an index or vice versa, ebx as the basis and eax as index.

In a similar way, for

```
pop DWORD [ecx] ; restores the top of the stack in the variable which address is given in ecx
```

the assembler can interpret ecx either as a base or as an index. What is really important to keep in mind is that all codifications considered by the assembler are equivalent and its final decision has no impact on the functionality of the resulted code.

Also, in addition to solving such ambiguities, the assembler also allows non-standard expressions, with the condition to be in the end transformable into the above standard form. Other examples:

```
lea eax, [eax*2]          ; load in eax the value of eax*2 (which is, eax becomes 2*eax)
```

In this case, the assembler may decide between coding as base = eax + index = eax and scale = 1 or index = eax and scale = 2.

```
lea eax, [eax*9 + 12]     ; eax will be eax * 9 + 12
```

Although the scale cannot be 9, the assembler will not issue an error message here. This is because it will notice the possible encoding of the address like: base = eax + index = eax with scale = 8, where this time the value 8 is correct for the scale. Obviously, the statement could be made clearer in the form

```
lea eax, [eax + eax * 8 + 12]
```

For indirect addressing it is essential to specify between square brackets at least one of the components of the offset computation formula.

3.2.2. Using operators

Operators – used for combining, comparing, modifying and analyzing the operands. Some operators work with integer constants, others with stored integer values and others with both types of operands.

It is very important to understand the difference between operators and instructions. **Operators perform computations only with constant SCALAR values computable at assembly time (scalar values = constant immediate values), with the exception of adding and/or subtracting a constant from a pointer (which will issue a pointer data type) and with the exception of the offset computation formula (which supports the '+' operator).** Instructions perform computations with values that may remain unknown (and this is generally the case) until run time. Operators are relatively similar with those in C. Expression evaluation is done on 64 bits, the final results being afterwards adjusted accordingly to the sizeof available in the available usage context of that expression.

For example the addition operator (+) performs addition at assembly time and the ADD instruction performs addition during run time. We give below the operators that are used by the x86 assembly language expressions in NASM !.

Priority	Operator	Type	Result
7	-	unary, prefix	Two's complement (negation): -X = 0 – X
7	+	unary, prefix	No effect (provides simetry to „-"): +X = X
7	~	unary, prefix	One's complement: mov al, ~0 => mov AL, 0xFF
7	!	unary, prefix	Logic negation: !X = 0 when X ≠ 0, else 1
6	*	Binary, infix	Multiplication: 1 * 2 * 3 = 6
6	/	Binary, infix	Result (quotient) of unsigned division: 24 / 4 / 2 = 3
6	//	Binary, infix	Result (quotient) of signed division: -24 // 4 // 2 = -3 **(-24 / 4 / 2 ≠ -3!)**
6	%	Binary, infix	Remainder of unsigned division: 123 % 100 % 5 = 3
6	%%	Binary, infix	Remainder of signed division: -123 %% 100 %% 5 = -3
5	+	Binary, infix	Sum: 1 + 2 = 3
5	-	Binary, infix	Subtraction: 1 – 2 = -1

4	<<	Binary, infix	Bitwise left shift: 1 << 4 = 16
4	>>	Binary, infix	Bitwise right shift: 0xFE >> 4 = 0x0F
3	&	Binary, infix	AND: 0xF00F & 0x0FF6 = 0x0006
2	^	Binary, infix	Exclusive OR: 0xFF0F ^ 0xF0FF = 0x0FF0
1	|	Binary, infix	OR: 1 | 2 = 3

The indexing operator has a widespread use in specifying indirectly addressed operands from memory. The role of the [] operator regarding indirect addressing has been explained in Paragraph 3.2.1.

3.2.2.3. Bit shifting operators

expression >> how_many *and* *expression << how_many*

```
mov ax, 01110111b << 3;  AX =  00000011 10111000b
add bh,  01110111b >> 3;  the source operator is 00001110b
```

3.2.2.4. Bitwise operators

Bitwise operators perform bit-level logical operations for the operand(s) of an expression. The resulting expressions have constant values.

OPERATOR	SYNTAX	MEANING
~	~ expresie	Bits complement
&	expr1 & expr2	Bitwise AND
|	expr1 | expr2	Bitwise OR
^	expr1 ^ expr2	Bitwise XOR

Examples (we assume that the expression is represented on a byte):

```
~11110000b                  ; output result is   00001111b
01010101b  &  11110000b     ; output result is   01010000b
01010101b  |  11110000b     ; output result is   11110101b
01010101b  ^  11110000b     ; output result is   10100101b
```

! – logical negation (similar with C) ; !0 = 1 ; !(anything different from zero) = 0

3.2.2.6. The segment specification operator

The segment specifier operator (:) performs the FAR address computation of a variable or label relative to a certain segment. Its syntax is: **segment:expression**

```
[ss: ebx+4]  ; offset relative to SS;
[es:082h]    ; offset relative to ES;
10h:var      ; the segment address is specified by the 10h selector,
               the offset being the value of the var label.
```

3.2.2.7. Type operators

They specify the types of some expressions or operands stored in memory. Their syntax is:

type expression

where the type specifier is one of the following: **BYTE**, **WORD**, **DWORD**, **QWORD** or **TWORD**

This syntactic form causes ***expression*** to be treated temporarily (limited to that particular instruction) as having ***„type"*** sizeof without destructively modifying its initial value. That is why these type operators are also called „non-destructive temporary conversion operators". For memory stored operators, *type* may be **BYTE**, **WORD**, **DWORD, QWORD or TWORD** having the size of 1, 2, 4, 8 and 10 bytes respectively. For code labels *type* is either **NEAR** (4 bytes address) or **FAR** (6 bytes address).

For example, **byte [**A] takes only the first byte from the memory location designated by A. Similar, **dword [**A] will consider the doubleword starting at address A.

3.3. DIRECTIVES

Directives direct the way in which code and data are generated during assembling.

3.3.1.1. The SEGMENT directive

SEGMENT directive allows targeting the bytes of code or of data emitted by an assembler to a given segment, having a name and some specific characteristics.

SEGMENT *name* [*type*] [ALIGN=*alignment*] [*combination*] [*usage*] [CLASS=*class*]

The numeric value assigned to the segment name is the segment address (32 bits) corresponding to the memory segment's position during run-time. For this purpose, NASM offers the special symbol $$ which is equal with the current segment's address, this having the advantage that can be used in any context, without knowing the current segment's name.

Except the name, all the other fields are optional both regarding their presence or the order in which they are specified.

The optional arguments *alignment*, *combination*, *usage* and *'class'* give to the link-editor and the assembler the necessary information regarding the way in which segments must be loaded and combined in memory.

The type allows selecting the usage mode of the segment, having the following possible values:

- **code** (or **text**) - the segment will contain code, meaning that the content cannot be written but it can be executed
- **data** (or **bss**) - data segment allowing reading and writing but not execution (implicit value).
- **rdata** - the segment that it can only be read, containing definitions of constant data

The optional argument *alignment* specifies the multiple of the bytes number from which that segment may start. The accepted alignments are powers of 2, between 1 and 4096.

If *alignment* is missing, then it is considered implicitly that ALIGN=1, i.e. the segment can start from any address.

The optional argument *combination* controls the way in which similar named segments from other modules will be combined with the current segment at linking time. The possible values are:

- **PUBLIC -** indicates to the link editor to concatenate this segment with other segments with the same name, obtaining a single segment having the length the sum of concatenated segments' lengths.
- **COMMON -** specifies that the beginning of this segment must overlap with the beginning of all segments with the same name, obtaining a segment having the length equal to the length of the larger segment with the same name.
- **PRIVATE** - indicates to the link editor that this segment cannot be combined with others with the same name.
- **STACK** - the segments with the same name will be concatenated. During run time the resulted segment will be the stack segment.

Implicitly, if no combination method is specified, any segment is PUBLIC.

The argument *usage* allows choosing another word size than the default 16 bits one.

The argument *'class'* has the task to allow choosing the order in which the link editor puts the segments in memory. All the segments that have the same class will be placed in a contiguous block of memory whatever their order in the source code. No implicit value exists, it being undefined when its specification is missing, leading though to NOT concatenating all the program's segments defined so in a continous memory block.

```
segment code use32 class=CODE          segment data use32 class=DATA
```

3.3.2. Data definition directives

Data definition = *declaration* (attributes specification) + *allocation* (reserving required mem. space).
(UNIQUE !!!) (it is NOT unique !!!) (UNIQUE !!!!)

In C – 17 modules (separate files) ; A1- global variable (int A1 + 16 data declarations extern int A1)
LINKER is responsible for checking the DEPENDENCIES between the modules !

The structure of a Variable = ([name], set_of_attributes, [address/reference, value])

Dynamic variable DOES NOT HAVE A NAME !!!!!

P=new(…); p=malloc(…); …free… (Diferenta between POINTER and DYNAMIC variables !!!!)

(Name, set_of_atrributes) = Formal parameters !!!!

Set_of_attributes = (type, Domeniu_de_vizibilitate (scope), lifetime (extent), memory class)
Memory class (in C) = (auto, register, static, extern)

data type = *size of representation* – byte, word, doubleword or quadword

The general form of a data definition source line is:

[*name*] *data_type* *expression_list* [**;***comment*]
or
[*name*] *allocation_type* *factor* [**;***comment*]
or
[*name*] **TIMES** *factor* *data_type* *expression_list* [**;***comment*]

where *name* is a label for data referral. The data type is the size of representation and its value will be the address of its first byte.

factor is a number which shows how many times the *expression_list* is repeated.

Data_type is a data definition directive, one of the following:

DB - byte data type (BYTE)
DW - word data type (WORD)
DD - doubleword data type (pointer - DWORD)
DQ - 8 bytes data type (QWORD – 64 bits)
DT - 10 bytes data type (TWORD – used to store BCD constants or real constants for extended precision)

For example, the following sequence defines and initializes 5 memory variables:

```
segment data
      var1  DB   'd'     ;1 byte
        .a  DW   101b ;2 bytes
      var2  DD   2bfh  ;4 bytes
        .a  DQ   307o ;8 bytes (1 quadword)
        .b  DT   100   ;10 bytes
```

Var1 and var2 variables are defined using common labels, visible in the entire source code, while .a and .b are local labels, the access to these local variables being restricted in the sense that:

- these variables can be accessed with the local name, i.e. .a or .b, until another common label is defined (they are local to preceding label);
- they can be accesed from anywhere by their complete name: var1.a, var2.a or var2.b.

The initialization value may be also an expression, as for example vartest DW (1002/4+1)

After a data definition directive may appear more than one value, thereby allowing declaration and initialization of arrays. For example, the declaration:

```
Tablou      DW  1,2,3,4,5
```

creates an array with 5 integers represented as words and having their values 1,2,3,4,5. If the values supplied after the directive don't fit on a single line, we can add as many lines as necessary, which shall contain only the directive and the desired values. Example:

```
Tabpatrate DD   0, 1, 4, 9, 16, 25, 36
           DD   49, 64, 81
           DD   100, 121, 144, 169
```

allocation_type is a uninitialized data reservation directive:

RESB - byte data type (BYTE)
RESW - word data type (WORD)
RESD - doubleword data type (DWORD)
RESQ - 8 bytes data type (QWORD - 64 bits)
REST - 10 bytes data type (TWORD – 80 bits)

factor is a number showing how many times the specified data type allocation is repeated.

For example Tabzero RESW 100h reserves 256 words for *Tabzero* array.

NASM does not support the MASM/TASM syntax of reserving uninitialized space by writing DW ?. The operand to a RESB-type pseudo-instruction is a **critical expression (all operands involved in computations must be known at expression evaluation time)**. Ex:

```
buffer:         resb    64      ; reserve 64 bytes
wordvar:        resw    1       ; reserve a word
realarray       resq    10      ; array of ten reals
```

TIMES directive allows repeated assembly of an instruction or data definition.

TIMES *factor data_type expression*

For example Tabchar TIMES 80 DB ‘a’

creates an ”array” of 80 bytes, every one of them being initialized with the ASCII code of 'a'.

matrice10x10 times 10*10 dd 0

will provide 100 doublewords stored continuously in memory starting from address associated with *matrice10x10* label.

TIMES can also be applied to instructions:

TIMES 32 add eax, edx ; having as effect EAX = EAX + 32*EDX

3.3.3. EQU directive

EQU directive allows assigning a numeric value or a string during assembly time to a label without allocating any memory space or bytes generation. The EQU directive syntax is:

name **EQU** *expression*

Examples:

```
END_OF_DATA      EQU '!'
BUFFER_SIZE      EQU 1000h
INDEX_START      EQU (1000/4 + 2)
VAR_CICLARE      EQU i
```

By use of such equivalence, the source code may become more readable.

You can see the similarity between the labels defined by the EQU directive and the symbolic constants defined in high level programming languages.

The expressions used when defining labels by EQU may also contain labels defined by EQU:

```
TABLE_OFFSET      EQU 1000h
INDEX_START       EQU (TABLE_OFFSET + 2)
DICTIONAR_STAR    EQU (TABLE_OFFSET + 100h)
```

String constants. Memory layout and using them in data transfer instructions.

When initializing a memory area with string type constants (sizeof > 1), the data type used in definition (dw, dd, dq) does only the reservation of the required space, the "filling" order of that memory area being the order in which the characters (bytes) appear in that string constant:

```
a6 dd '123', '345','abcd'    ; 3 doublewords are defined, their contents being
                                31 32 33 00|33 34 35 00|61 62 63 64|

a6 dd '1234'  ; 31 32 33 34

a6 dd '12345' ;    31 32 33 34|35 00 00 00|


a71 dw '23','45'   |32 33| 34 35|   - 2 words = 1 doubleword
a72 dw '2345'     - 2 words -  32 33|34 35|
a73 dw '23456'   - 3 words -  32 33|34 35|36 00|

mov eax, [a73] ; EAX = 35 34 33 32
mov ah, [a73] ; AH = 32
mov ax, [a73] ; AX = 33 32
```

The same happens for a71 and a72.

'...' = "..." In C, '…' ≠"…", in assembly NO !

In C, ASCIIZ implies that 'x' = ASCII code for x (1 byte) – character ;
"x" = 'x','\0' (2 bytes) – string;

```
a8 dw '1', '2', '3'    - 3 words  -  31 00|32 00|33 00
a9 dw '123'    - 2 words -          31 32|33 00
```

The following definitions provide the same memory configuration:

```
        dd    'ninechars'           ; doubleword string constant
        dd    'nine','char','s'     ; 3 doublewords
        db    'ninechars',0,0,0  ; "filling" memory area by a bytes sequence
```

From official documentation we have:

A character constant with more than one byte will be arranged (WHERE ???) with little-endian order in mind: if you code

```
mov eax,'abcd' (EAX = 0x64636261)
```

then the constant generated is not `0x61626364`, but `0x64636261` so that if you were then to store the value into memory, it would read `abcd` rather than `dcba`. This is also the sense of character constants understood by the Pentium's CPUID instruction.

The main idea of this definition is that THE CHARACTER REPRESENTATION VALUE associated to a string constant 'abcd' is in fact 'dcba' (this being the STORAGE format in the CONSTANTS TABLE).

Mov dword [a], '2345' will be in OllyDBG mov dword ptr DS:[401000],35343332

And the effect on the memory area reserved for a is |32 33 34 35|

….but if you code a data definition like a7 dd '2345' the corresponding memory layout will be NO little-endian representation, but |32 33 34 35|

So, comparatively and in short:

a7 dd '2345' ; |32 33 34 35|
a8 dd 12345678h ; |78 56 34 12|
…….
mov eax, '2345' → EAX = '5432' = 35 34 33 32 (in OllyDbg you will find mov eax, 35343332)

mov ebx, [a7] → EBX = '5432' = 35 34 33 32

DIFFER from the behavior of the numeric constant when they appear in a data transfer instruction:

mov ecx, 12345678h ; ECX = 12345678h
mov edx, [a8] → EDX = 12345678h

In the case when DB is used as a data definition directive it is normal that the bytes order given in the constant to be also kept in memory in similar way (little-endian representation being applied only to data types bigger than a byte!), so this case doesn't need an extra analysis.

a66 TIMES 4 db '13' ; 31 33 31 33 31 33 31 33

a67 TIMES 4 dw '13' ; 31 33 31 33 31 33 31 33 - those two DIFFERENT definitions provide the same output result !!!

a68 TIMES 4 dw '1','3' ; 31 00 33 00 31 00 33 00 31 00 33 00 31 00 33 00

a69 TIMES 4 dd '13' ; 31 33 00 00 | 31 33 00 00 | 31 33 00 00 | 31 33 00 00

So, a constant string in NASM behaves as there is a previously allocated "memory area" (IT IS AND IT IS CALLED CONSTANT TABLE) to these constants, where these are stored using the little-endian representation !!

From a REPRESENTATION point of view the value associated with a string constant IS ITS INVERSE !!!! (see the official definition above)

From a NUMERIC VALUE point of view ??? (true both in C and assembler ???)

"abcd" = THE ADDRESS FROM MEMORY OF THIS CONSTANT (in C)

"abcd"[1] = 'b'

"abcd"[251] = ??

Location counter and pointer arithmetic (discussed with you)

```
SEGMENT data

a db 1,2,3,4 ; 01 02 03 04 (offset (a) =0)
lg db $-a       ; 04; offset (lg) = 4
lg db $-data ; in TASM will be the same effect as above because in TASM, MASM offset(segment-name)=0 !!!; in NASM NO WAY, offset(data) = 00401000 !!!
    -    Here I will have syntax error !
lg db data-$ ; syntax error !

lg2 dw data-a; este acceptata de care asamblor !!!! in schimb da eroare la link-editor !!!

   db a-$      ; 0-5 = -5 = FB

c  equ  a-$    ;  0-6 = -6 = FA
d equ  a-$     ;  0-6 = -6 = FA
e  db a-$      ;  0-6 = -6 = FA
x dw x         ; 07 10 !!!!!
x db x         ; syntax error !

   db lg-a        ; 4-0 = 4
   db a-lg        ; 0-4 = -4
   db [$-a]             ; syntax error !
   db [lg-a]             ; syntax error !

 ;x dw x     ; x = offset(x)  09 00 la asamblare si in final 09 10

 lg1  EQU lg1          ;  lg1=0 !!!  (because it is a NASM BUG !!!)
 lg1  EQU lg1-a       ;   lg1=0 !!! (because offset(a) =0); if we define something before a we will obtain …..syntax error !

   b  dd  a-start   ;  syntax error ! (a – defined here, start – defined somewhere else)
      dd  start-a  ;   OK !!!! IT WORKS !!! (start – defined here, a – defined somewhere else)
      dd  start-start1 ; OK !!! (because both labels belong to the same segment !!!)
```

```
segment code use32

    start:

        mov ah, lg1    ; AH=0
        mov bh, c      ; c is a constant def by EQU, so BH=-6

        mov ch, lg         ; syntax error
        mov ch, lg-a       ; 4
        mov ch, [lg-a]     ;  mov ch, byte ptr DS:[4]    - most probably memory access violation

        mov cx, lg-a       ;  4
        mov cx, [lg-a]     ; mov ch, word ptr DS:[4]    - most probably memory access violation


        mov cx, $-a  ;  syntax error ! ($ – CODE SEGMENT's location counter, a – defined somewhere else)
        mov cx, a-$  ; ok !!! Possible warning…

        mov ch, $-a  ;  syntax error ! ($ – CODE SEGMENT's location counter, a – defined somewhere else)
        mov ch, a-$  ;  a-$ IS OK, BUT a-$ IS A POINTER !!!!! – syntax error !!!

        mov cx, $-start  ; ok !
        mov cx, start-$  ; ok !

        mov ch, $-start  ; ok !! – because it is a SCALAR !!!!!
        mov ch, start-$  ; ok !! – because it is a SCALAR !!!!!


        mov cx, a-start   ; ok !!
        mov cx, start-a   ; syntax error ! (start – defined here, a – defined somewhere else)


        start1:

        mov ah, a+b   ; NASM accepts THIS !!! NO SYNTAX ERROR !!! MERGE – DAR NU E ADUNARE POINTERI !!!!!
```

a+b = (a-$$) + (b-$$) = SCALAR + SCALAR = SCALAR
mov ax, b+a ; idem – ok
mov ax, [a+b] ; Invalid effective address – THIS IS REALLY A POINTER ADDITION !!!

Location counter and pointer arithmetic (prepared by me in advance)

SEGMENT data

a db 1,2,3,4 ; 01 02 03 04

lg db $-a ; 04
lg db $-data ; syntax error – expression is not simple or relocatable
lg db a-data ; syntax error – expression is not simple or relocatable
lg2 dw data-a; the assembler allows it but we obtain a linking error – "Error in file at 00129 – Invalid fixup recordname count…."
db a-$; = -5 = FB
c equ a-$; = -6 = FA
db lg-a ; 04
db a-lg ; = -4 = FC
db [$-a] ; expression syntax error – a memory contents is not a constant computable at assembly time
db [lg-a] ; expression syntax error – a memory contents is not a constant computable at assembly time

lg1 EQU lg1 ; lg1 = 0 ! (BUG NASM !!!!)
lg1 EQU lg1-a ; lg1 = 0 – WHY ????? It's a BUG !!! It works like that only because offset(a) = 0; if we put something else before a in data segment we will obtain a syntax error : "Recursive EQUs, macro abuse"
g34 dw c-2 ; -8

b dd a-start ; expression is not simple or relocatable !!!
dd start-a ; OK !!!!!!!! - POINTER DATA TYPE!!!!! (because it represents a FAR pointers subtraction !!!!)

dd start-start1 ; OK !!! – because they are 2 labels from the same segment! – Result will be a SCALAR DType !!!!

```
segment code use32
start:

    mov ah, lg1    ; AH = 0
    mov bh, c      ; BH = -6 = FA

    mov ch, lg         ; OBJ format can handle only 16 or 32 byte relocation
    mov ch, lg-a       ; CH = 04
    mov ch, [lg-a]     ;  mov byte ptr DS:[4] – Access violation – most probably…

    mov cx, lg-a        ;  CX = 4
    mov cx, [lg-a]      ; mov WORD ptr DS:[4] – Access violation – most probably…

    mov cx, $-a  ;  invalid operand type !!!!!
    mov cx, a-$  ;  OK !!!!!!

    mov ch, $-a  ;  invalid operand type !!!!!
    mov ch, a-$  ;  OBJ format can handle only 16 or 32 byte relocation

    mov cx, $-start  ; ok !!!
    mov cx, start-$  ; ok !!!

    mov ch, $-start  ; ok !!!
    mov ch, start-$  ; ok !!!

    mov cx, a-start   ; ok !!!
    mov cx, start-a   ; invalid operand type !!!

    start1:

    mov ah, a+b   ; NO syntax error !!!!!!!!!! BUT …. It is NO pointer arithmetic, NO pointers addition
                      a+b = (a-$$) + (b-$$)
    mov ax, b+a   ; AX = (b-$$) + (a-$$) – SCALARS ADDITION !!!!

    mov ax, [a+b]  ; INVALID EFFECTIVE ADDRESS !!!! – THIS represents REALLY a pointers addition which is FORBIDDEN !!!!!!
```

Conclusion:

Expressions of type et1 – et2 (where et1 and et2 are labels – either code or data) are syntactically accepted by NASM,

- Either if both of them are defined in the same segment
- Either if et1 belongs to a different segment from the one in which the expression appears and et2 is defined in this latter one. In such a case, the expression is accepted and the data type associated to the expression et1-et2 is POINTER and NOT SCALAR (numeric constant) as in the case of an expression composed of labels belonging to the same segment.

  Subtracting offsets specified relative to the same segment = SCALAR
  Subtracting pointers belonging to different segments = POINTER

The name of a segment is associated with "Segment's address in memory at run-time", but this value isn't accessible to us, this being decided only at loading-time by the OS loader. That is why, if we try to use the name of a segment explicitly in our programs we will get either a syntax error (in situations like $/a-data) either a linking error (in situations like data-a). In 16 bits programming a segment's name is associated to its offset and this offset is zero ! In 32 bits programming, the offset of a segment is NOT a constant computable at assembly time and that is why we cannot use it in pointer arithmetic expressions !

Mov eax, data ; Segment selector relocations are not supported in PE files (relocation error occurred)

You can check for instance that NASM and OllyDbg always provide for start offset (DATA) = 00401000, and offset(CODE) = 00402000. So here the offsets aren't zero as in 16 bits programming !! So from this point of view there is a big difference between VARIABLES NAMES (which offsets can be determined at assembly time) and SEGMENTS NAMES (which offsets can NOT be determined at assembly time as a constant, this value being known exactly like the segment's address ONLY at loading time).

If we have multiple pointer operands, the assembler will try to fit the expression into a valid combination of pointer arithmetic operations:

Mov bx, [(v3-v2) ± (v1-v)] – OK !!! (adding and subtracting immediate values is correct !!)

… but we must not forget that in the case of INDIRECT ADDRESSING the final result of the expression with pointer operands in parentheses must finally provide A POINTER !!! from this reason in the case of indirect addressing operands in the form “a + b” will never be accepted !

If not, we will obtain “Invalid effective address ”syntax error”:

Mov bx, [v2-v1-v] ; Invalid effective address !

Mov bx, v2-v1-v ; Invalid operand type ! mov bx, v2-(v1+v)

Mov bx, v2+v1-v ; ok

Mov bx, [v3-v2-v1-v] ; Invalid effective address !

Mov bx, v3-v2-v1-v ; OK !!! – because... Mov bx, v3-v2-v1-v = Mov bx, (v3-v2)-(v1+v) = subtracting immediate values

The direct vs. indirect addressing variant is more permissive as arithmetic operations in NASM (due to the acceptance of “a + b” type expressions) but this does not mean that it is more permissive IN POINTER OPERATIONS !!!

Mov bx, (v3±v2) ± (v1±v) – OK !!! (adding and subtracting immediate values is correct!!)

Cee ace apare in paranteze () atat in variant cu galben cat si cea verde SUNT SCALARI !!!!!!!!

Numele unui segment este asociat cu "Adresa segmentului in memorie in faza de executie" insa aceasta nu ne este disponibila/accesibila, fiind decisa la momentul incarcarii programului pt executie de catre incarcatorul de program al SO. De aceea, daca folosim nume de segmente in expresii din program in mod explicit vom obtine fie eroare de sintaxa (in situatii de tipul $/a-data) fie de link-editare (in situatii de tipul data-a), deoarece practic numele unui segment este asociat cu adresa FAR al acestuia" (adresa care nu va fi cunoascuta decat la momentul incarcarii programului, deci va fi disponibila doar la run-time) si NU este asociat cu "Offset-ul segmentului in faza de asamblare, fiind o constanta determinata la momentul asamblarii" (asa cum este in programarea sub 16 biti de exemplu, unde nume segment in faza de asamblare = offset-ul sau = 0). Ca urmare, se constata ca in programarea sub 32 de biti numele de segmente NU pot fi folosite in expresii !!

Mov eax, data ; Segment selector relocations are not supported in PE files (relocation error occurred)

Sub 32 biti offset (data) = vezi OllyDbg – Intotdeauna DATA segment incepe la offset-ul 00401000, iar CODE segment la offset 00402000. Deci offset-urile inceputurilor de segment nu sunt 0 la fel ca si la programarea sub 16 biti. Din acest punct de vedere sub 32 biti este o mare diferenta intre numele de variabile (al caror offset poate fi determinat la asamblare) si numele de segmente (al caror offset NU poate fi determinat la momentul asamblarii ca si o constanta, aceasta valoare fiind cunoscuta la fel ca si adresa de segment doar la momentul incarcarii programului pt executie - loading time).

Daca avem mai multi operanzi pointeri, asamblorul va incerca sa incadreze expresia intr-o combinatie valida de operatii cu pointeri:

Mov bx, [(v3-v2) ± (v1-v)] – OK !!! (adunarea si scaderea de valori imediate este corecta !!)

…insa nu trebuie sa uitam ca fiind vorba despre ADRESARE INDIRECTA rezultatul final al expresiei cu operanzi pointeri din paranteze trebuie sa furnizeze in final UN POINTER !!!... din aceasta cauza in cazul adresarii indirecte nu vor fi niciodata accepate expresii cu operanzi adrese de forma “a+b”

Daca nu, vom obtine eroarea de sintaxa “Invalid effective address”:

Mov bx, [v2-v1-v] ;
Mov bx, v2-v1-v ;
Mov bx, [v3-v2-v1-v] ; Invalid effective address !
Mov bx, v3-v2-v1-v ; OK !!! – deoarece ?... Mov bx, v3-v2-v1-v = Mov bx, (v3-v2)-(v1+v) = scadere de valori immediate

Varianta de adresare directa vs.indirecta este mai permisiva ca operatii aritmetice in NASM (din cauza acceptarii expresiilor de tip “a+b”) insa asta nu inseamna ca este mai permisiva IN OPERATIILE CU POINTERI !!!

Mov bx, (v3±v2) ± (v1±v) – OK !!! (adunarea si scaderea de valori imediate este corecta !!)

JMP instruction analysis. NEAR and FAR jumps.

We present below a very relevant example for understanding the control transfer to a label, highlighting the differences between a direct transfer vs. an indirect one.

```
segment data
        aici   DD  here        ;equiv. with  aici := offsetul of label here from code segment

segment code
        .       .       .
        mov   eax, [aici]       ;we load EAX with the contents of variable aici (that is the offset of here
                                  inside the code segment – same effect as  mov eax, here)

        mov  ebx, aici          ;we load EBX with the offset of aici inside the data segment
                                ; (probable 00401000h)
```

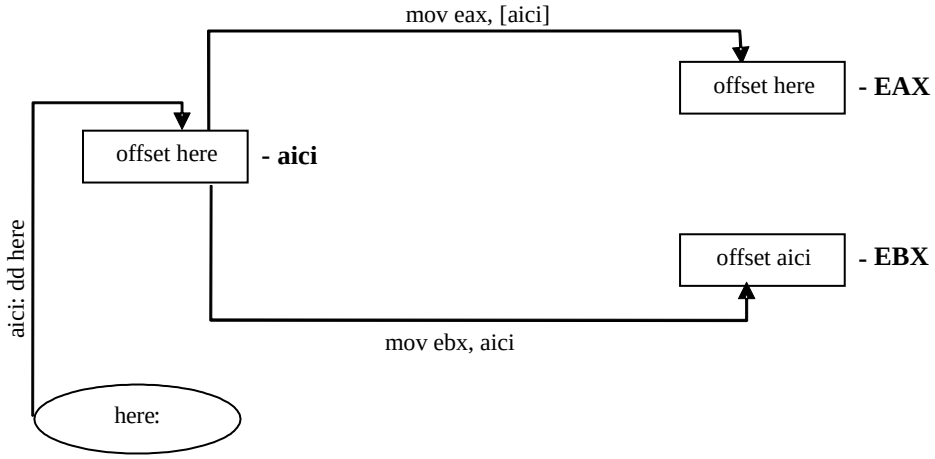


Fig. 4.4. Initializing variable aici and registers EAX and EBX.

jmp [aici] ;jump to the address indicated by the value of variable aici (which is the address of here), so this is an instruction equiv with jmp here ; what does in contrast **jmp aici** ??? – the same thing as jmp ebx ! **jump to CS:EIP with EIP=offset (aici) from SEGMENT DATA (00401000h)** ; jump to some instructions going until the first „access violation"

jmp here ;jump to the address of here (or, equiv, jump to label here); **jmp [here]** ?? – JMP DWORD PTR DS:[00402014] – most probably „Access violation"....

jmp eax ;jump to the address indicated by the contents of EAX (accessing register in direct mode), that is to label here ; in contrast, what does **jmp [eax]** ??? JMP DWORD PTR DS:[EAX] – most probably „Access violation"....

jmp [ebx] ;jump to the address stored in the memory location having the address the contents of EBX (indirect register access – the only indirect access from this example) – what does in contrast **jmp ebx** ??? - jump to **CS:EIP with EIP=offset (aici) from the SEGMENT DATA (00401000h)** ; jump to some instructions going until the first „access violation"

;in EBX we have the address of variable aici, so the contents of this variable will be accessed. In this memory location we find the offset of label here, so a jump to address here will be performed - **consequently, the last 4 instructions from above are all equivalent to jmp here**

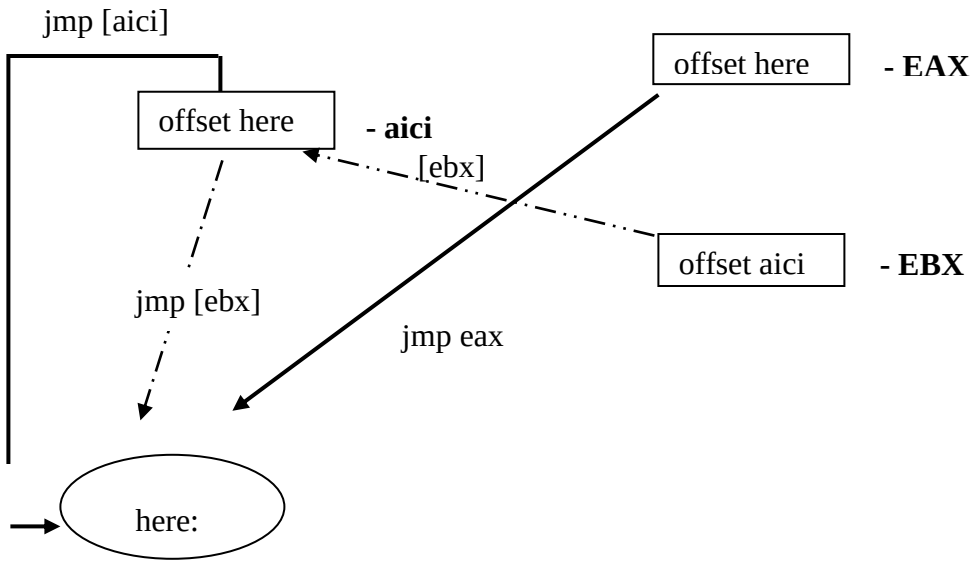


Fig. 4.5. Alternative ways for performing jumps to the label *here.*

```
        jmp   [ebp]   ;  JMP DWORD PTR SS:[EBP]
.....................................................................
        here:
                mov ecx, 0ffh
```

Explanations on the interaction between the implicit association rules of an offset with the corresponding segment register and performing the corresponding jump to the specified offset

JMP [var_mem] **; JMP DWORD PTR DS:[004010??]**

JMP [EBX] **; JMP DWORD PTR DS:[EBX]**
JMP [EBP] **; JMP DWORD PTR SS:[EBP]**

Accordingly to the implicit association rules of an offset with the corresponding segment register

In this case do we have to expect that the jump to be made in the data segment at the specified offset and the processor to interpret that data as instructions code ? (taking into consideration the implicit offset positioning relative to DS) ??. That is, the FAR jumping address to be DS:[00401000] and DS:[reg] respectively ?

NO ! NOT AT ALL ! Something like that doesn't happen. First of all, the IMPLICIT jump type is NEAR (check in OllyDbg how NASM is generating the corresponding object code by DWORD PTR, so it takes into consideration JUST the OFFSET - relative to DS or SS !). Being NEAR jump instructions, these will be made IN THE SAME MEMORY (CODE) SEGMENT in which those three instructions appear ,so the jumps will be made in fact to CS:[var_mem] and CS:[reg] respectively.

So... a NEAR jump, even if it is implicitly prefixed with DS or SS doesn't make a jump into the data segment or in the stack segment! From that segment it will only **load the corresponding offset** to which to perform the jump inside the current CODE SEGMENT !!! That is why we call it in fact a NEAR JUMP !!!... isn't it ? **Prefixing is useful here only for allowing to load the corresponding OFFSET value of the pointer,** offset which can be placed/present/specified/stored in any (other) segment.

It follows that even if we use explicit prefixing, like for example

jmp [ss: ebx + 12]
jmp [ss:ebp +12] equiv with jmp [ebp +12]

because the jump is still a NEAR one, it will be made inside the same (current) code segment, that is to CS : EIP, namely to CS : offset value taken from the stack

So, prefixing a target of a jump with a segment register has only the task of correctly indicate the actual segment from which the required offset will be loaded, offset to which the NEAR jump will be performed inside the current code segment !

If we wish to make a jump to a different segment (namely to perform a FAR jump) we must EXPLICITLY specify that by using the FAR type operator:

jmp far [ss: ebx + 12] => CS : EIP <- the far address (48 bits) taken from the stack segm.

The FAR operator specifies here that not only EIP must be loaded with what we have at the specified address, but also CS must be loaded with a new value (this makes in fact the jump to be a FAR one).

In short and as a conclusion we have :
- the value of the pointer representing the address where the jump has to be made can be stored anywhere in memory, this meaning that any offset specification that is valid for a MOV instruction can be also present as an operator for a JMP instruction (for example **jmp [gs:ebx + esi*8 - 1023]**)
- the contents of the pointer (the bytes taken from that specific memory address) may be near or far, depending on how the programmer specifies or not the FAR operator, being thus applied either only to EIP (if the jump is NEAR), either to the pair CS:EIP if the jump is FAR.

Any jump can be considered hypothetically equivalent to MOV instructions such as:
- jmp [gs:ebx + esi * 8 - 1023] <=> "mov EIP, DWORD [gs:ebx + esi * 8 - 1023]"
- jmp FAR [gs:ebx + esi * 8 - 1023] <=> "mov EIP, DWORD [gs:ebx + esi * 8 - 1023]" +
"mov CS, WORD [gs:ebx + esi * 8 - 1023 + 4]"

[gs:ebx + esi * 8 - 1023]

7B 8C A4 56 D4 47 98 B7.....

"Mov CS:EIP, [memory]" → EIP = 56 A4 8C 7B
CS = 47 D4

JMP through labels – always NEAR !

```
segment data use32 class=DATA

   a db 1,2,3,4
    start3:
     mov edx, eax   (to be verified !!!)  - !!!!

segment code use32 class=code
   start:
         mov edx, eax
         jmp start2  - ok – NEAR jmp -   JMP  00403000 (offset code segment = 00402000)
         jmp start3 – ok – NEAR jmp –   JMP  00401004 (offset data segment = 00401000)
         jmp far start2  - Segment selector relocations are not supported in PE file
         jmp far start3  - Segment selector relocations are not supported in PE file

         ;The two jumps above jmp start2 and jmp start3 respectively will be done to the specified
         ;labels, start2 and start3 but they will not be considered FAR jumps (the proof is that the
         ;specification of this attribute below in the other two variants of the instructions will lead
         ;to a syntax error!)
         ;They will be considered NEAR jumps due to the FLAT memory model used by the OS

         add eax,1
   final:
     push    dword 0
     call    [exit]

 segment code1 use32 class=code

   start2:
      mov eax, ebx

   push dword 0
   call [exit]
```

Why ?... Because of the **"Flat memory model"**

Final conclusions.

- **NEAR jumps – can be accomplished through all of the three operand types (label, register, memory addressing operand)**
- **FAR jumps (this meaning modifying also the CS value, not only that from EIP) – can be performed ONLY by a memory addressing operand on 48 bits (pointer FAR). Why only so and not also by labels or registers ?**
- **- by labels, even if we jump into another segment (an action possible as you can see above) this is not considered a FAR jump because CS is not modified (due to the implemented memory model – Flat Memory Model). Only EIP will be changed and the jump is technically considered to be a NEAR one.**
- **Using registers as operands we may not perform a far jump, because registers are on 32 bits and we may so specify maximum an offset (NEAR jump), so we are practically in the case when it is impossible to specify a FAR jump using only a 32 bits operand.**

24. x86 Instruction Prefix Bytes

- x86 instruction can have up to 4 prefixes.
- Each prefix adjusts interpretation of the opcode:

  String manipulation instruction prefixes (prefixes provided EXPLICITLY by the programmer !)

  F3h = REP, REPE
  F2h = REPNE

  where

  - REP repeats instruction the number of times specified by *iteration count* ECX.
  - REPE and REPNE prefixes allow to terminate loop on the value of **ZF** CPU flag.

- 0xF3 is called REP when used with MOVS/LODS/STOS/INS/OUTS (instructions which don't affect flags)
  0xF3 is called REPE or REPZ when used with CMPS/SCAS
  0xF2 is called REPNE or REPNZ when used with CMPS/SCAS, and is not documented for other instructions.
  Intel's insn reference manual REP entry only documents F3 REP for MOVS, not the F2 prefix.

  Related string manipulation instructions are:

  - MOVS, move string
  - STOS, store string
  - SCAS, scan string
  - CMPS, compare string, etc.

  See also string manipulation sample rogram: rep_movsb.asm

Segment override prefix causes memory access to use *specified segment* instead of *default segment* designated for instruction operand. **(provided EXPLICITLY by the programmer).**

2Eh = CS	**26h = ES**
36h = SS	**64h = FS**
3Eh = DS	**65h = GS**

Operand override, **66h**. Changes size of data expected by default mode of the instruction e.g. 16-bit to 32-bit and vice versa.

Address override, **67h**. Changes size of address expected by the default mode of the instruction. 32-bit address could switch to 16-bit and vice versa.

These two last prefix types appear as a result of some particular ways of using the instructions (examples below), which will cause the generation of these prefixes by the assembler in the internal format of the instruction. These prefixes are NOT provided EXPLICITLY by the programmer by keywords or reserved words of the assembly language.

Instruction prefix - **REP MOVSB**

Segment override prefix - **ES xlat or mov ax, [cs:ebx]**
(ES:EBX)

mov ax, DS:[ebx] – implicit prefix
mov ax, [cs:ebx] – explicit prefix given by the programmer

Operand size prefix –
Bits 32 - default mode of the below code

……..
cbw ; 66:98 - because rez is on 16 bits (AX)
cwd ; 66:99 - because rez is composed by 2 reg on 16 bits (DX:AX)
cwde ; 98 - because we follow the default mode on 32 biti – rez in EAX
push ax ; 66:50 – because a 16 bits value is loaded onto the stack, the stack being organized on 32 bits
push eax ; 50 - ok – consistent usage with default mode
mov ax, a ; 66:B8 0010 – because rez is on 16 bits

Address size prefix – 0x67

Bits 32
mov eax, [bx] ; 67:8B07 - because DS:[BX] is 16 bits addressing

Bits 16
mov BX, [EAX] ; 67:8B18 – because DS:[EAX] is 32 bits addressing

Bits 16
push dword[ebx] ; 66:67:FF33 – Here the default mode is Bits 16; because the addressing [EBX] is on 32 bits 67 will be generated and because a push is made to a DWORD operand instead of one on 16 bits 66 will be generated as a prefix

67:8B07 mov eax, [bx] ; Offset_16_bits = [BX|BP] + [SI|DI] + [const]

Bits 16 - default mode of the below code
……..
cbw ; 98 - ok, because result is on 16 bits (AX)
cwd ; 99 - ok, because result is composed by a combination of 2 registers on 16 bits (DX:AX)
cwde ; 66:98 - because here the 16 bits default mode is not followed – result in EAX

MNEMONICA	SEMNIFICAŢIE (salt dacă..<<relaţie>>)	Condiţia verificată
JB **JNAE** **JC**	este inferior nu este superior sau egal există transport	CF=1
JAE **JNB** **JNC**	este superior sau egal nu este inferior nu există transport	CF=0
JBE **JNA**	este inferior sau egal nu este superior	CF=1 sau ZF=1
JA **JNBE**	este superior nu este inferior sau egal	CF=0 şi ZF=0
JE **JZ**	este egal este zero	ZF=1
JNE **JNZ**	nu este egal nu este zero	ZF=0
JL **JNGE**	este mai mic decât nu este mai mare sau egal	SF≠OF
JGE **JNL**	este mai mare sau egal nu este mai mic decât	SF=OF
JLE **JNG**	este mai mic sau egal nu este mai mare decât	ZF=1 sau SF≠OF
JG **JNLE**	este mai mare decât nu este mai mic sau egal	ZF=0 şi SF=OF
JP **JPE**	are paritate paritatea este pară	PF=1
JNP **JPO**	nu are paritate paritatea este impară	PF=0
JS	are semn negativ	SF=1
JNS	nu are semn negativ	SF=0
JO	există depăşire	OF=1
JNO	nu există depăşire	OF=0

Tabelul 4.1. Instrucţiunile de salt condiţionat

4.3.2.3. **Exemple comentate**

Modul de acţiune a instrucţiunilor de salt condiţionat şi instrucţiunea **cmp**

```
mov al,80h  ;al := 128 = 10000000b = -128 ! (Interesant! – remarcăm faptul că datorită ;regulilor de reprezentare în cod complementar 128 şi -128 au aceeaşi ;reprezentare binară şi anume 10000000b!)

(*)  cmp al,0  ;instrucţiunea cmp nu interpretează în nici un fel valoarea din AL (ca fiind ;cu semn sau fără semn) ci doar realizează scăderea fictivă al-0 şi afectează ;corespunzător flagurile: SF=1 , CF=ZF=OF=PF=AF=0.

     jl et  ;utilizarea instrucţiunii JL (Jump if Less than) provoacă interpretarea ;comparaţiei al<0 cu semn (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă SF≠OF şi cum SF=1 iar OF=0 se decide îndeplinirea condiţiei ;şi saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a ;stat la latitudinea programatorului care prin utilizarea instrucţiunii JL a ;decis că doreşte să compare -128 cu 0 şi cum -128 este “less than” 0 ;condiţia a fost îndeplinită (echiv. cu jnge et). În contrast, jnl et sau jge et ;(care vor testa dacă SF=OF) NU vor fi îndeplinite şi NU vor provoca saltul ;la eticheta specificată.

     jb et  ;utilizarea instrucţiunii JB (Jump if Below) provoacă interpretarea ;comparaţiei al<0 fără semn (vezi tabelul 4.2), adică cf. tabelului 4.1 se ;testează dacă CF=1 şi cum CF=0 se decide neîndeplinirea condiţiei deci nu ;se va face saltul la eticheta et. Deducem deci că interpretarea valorilor comparate a stat la latitudinea programatorului care prin utilizarea instrucţiunii JB a decis că doreşte să compare 128 cu 0 şi cum 128 NU este “below” 0 condiţia NU a fost îndeplinită (echivalent cu jnae et sau jc et).

     jae et1  ;se testează fără semn dacă al ≥ 0 (128 ≥ 0?) - CF=0 deci condiţie ;îndeplinită (echivalent cu jnc et1 sau jnb et1) – se efectuează saltul la ;eticheta et1

     jbe et2  ;se testează fără semn dacă al ≤ 0 (128 ≤ 0?) – CF = ZF = 0 deci condiţia ;(CF=1 sau ZF=1) NU este îndeplinită şi ca urmare nu se va face saltul la ;eticheta et2 – rezultat consistent cu jb et, deoarece jbe implică jb ;(echivalent cu jna et2)

     ja et3  ;se testează fără semn dacă al > 0 (128 > 0?) – CF = ZF = 0 deci condiţia ;(CF=0 şi ZF=0) este îndeplinită şi ca urmare se
```

va face saltul la eticheta et3 ;(echivalent cu jnbe et3) şi rezultat consistent cu jbe et2, deoarece dacă ;jbe nu este îndeplinită atunci ja trebuie să fie.

je et4 ;se testează dacă al = 0 (128 = 0 ?) – nu se pune problema semnului dacă se ;testează egalitatea! – cum ZF=0, condiţia ZF=1 nu este îndeplinită deci nu ;se va efectua saltul la eticheta et4 (echivalent cu jz et4). În contrast, jne ;et4 sau jnz et4 (care vor testa dacă ZF=1) vor fi îndeplinite şi vor provoca ;saltul la eticheta specificată.

jle et5 ;se testează cu semn dacă al ≤ 0 (-128 ≤ 0?) – OF = ZF = 0 şi SF=1 deci ;condiţia (ZF=1 sau SF≠OF) este îndeplinită şi ca urmare se va face saltul la ;eticheta et5 (echivalent cu jng et5) şi rezultat consistent cu jl et, deoarece ;jle implică jl.

jg et6 ;se testează cu semn dacă al > 0 (-128 > 0?) – OF = ZF = 0 şi SF=1 deci ;condiţia (ZF=0 şi SF=OF) NU este îndeplinită şi ca urmare NU se va face ;saltul la eticheta et6 (echivalent cu jnle et6) şi rezultat consistent cu jle ;et5, deoarece dacă jg nu este îndeplinită atunci jle trebuie să fie.

jp et7 ;se testează dacă PF=1 - PF=0 deci condiţie neîndeplinită – nu se efectuează ;saltul (echivalent cu jpe et7 – *Jump if Parity Even*). În contrast, jnp et7 ;(care testează dacă PF=0 – echivalentă cu jpo et7 – *Jump if Parity Odd*) va ;fi îndeplinită şi saltul se va efectua.

jo et8 ;se testează dacă OF=1 - OF=0 deci condiţie neîndeplinită – nu se efectuează

;saltul (nu există depăşire). În contrast, jno et8 (care testează dacă OF=0) va ;fi îndeplinită şi saltul se va efectua.

js et9 ;se testează dacă în interpretarea cu semn **rezultatul** comparaţiei are semn ;negativ (deoarece aşa cum specificam în cadrul prezentării instrucţiunii ;CMP, nu este vorba de a interpreta cu semn sau fără semn ***operanzii*** ;scăderii fictive *d-s*, ci ***rezultatul*** final al acesteia !) adică testăm dacă SF=1 -;condiţie îndeplinită în cazul nostru şi ca urmare saltul se va efectua !. În ;contrast, jns et9 (care testează dacă SF=0) NU va fi îndeplinită şi saltul ;NU se va efectua.

cmp 0,al ;eroare de sintaxă : *"Illegal immediate*" deoarece sintaxa instrucţiunii cmp ;interzice specificarea ca prim operand a unei valori imediate (constante). ;dacă totuşi dorim forţarea unei

comparaţii de acest tip (0-al) putem utiliza ;pe post de prim operand un registru iniţializat cu valoarea 0.

```
mov bl,0
```

cmp bl, al ;realizează scăderea fictivă bl-al (0-al = 0-80h = 0-10000000b = ;10000000b) şi afectează corespunzător flagurile: CF=SF=OF=1, ;ZF=PF=AF=0.

Exerciţiu propus: Reluaţi discuţia efectului tuturor instrucţiunilor de salt condiţionat de mai sus (analizate pentru cazul comparaţiei (*) cmp al,0) în condiţiile în care această comparaţie e înlocuită de ultimele două instrucţiuni prezentate, adică în cazul în care se efectuează cmp bl,al cu bl=0.

Care ar fi însă justificarea faptului că în cazul cmp bl,al avem CF = OF = SF = 1 iar în cazul *cmp al,0* doar SF=1 iar CF = OF = 0 ?

Pentru a justifica modurile de setare diferite ale flag-urilor trebuie să luăm în discuţie regulile practice de setare a acestor flag-uri. Aceste reguli generale sunt:

- SF ia valoarea bitului de semn al rezultatului obţinut;
- CF ia valoarea cifrei de transport : dacă e vorba despre o adunare se analizează dacă rezultatul obţinut a provocat (CF=1) sau nu (CF=0) un transport în afara spaţiului de reprezentare; dacă e vorba despre o scădere *d-s*, avem: dacă $|d| \geq |s|$ atunci CF=0 (nu e nevoie de cifră de împrumut pentru efectuarea scăderii) iar dacă $|d| < |s|$ atunci CF=1 (este nevoie de cifră de împrumut pentru efectuarea scăderii şi acest lucru se reflectă în CF)
- OF este setat la valoarea 1 dacă există depăşire în interpretarea cu semn a rezultatului ("*OF is set if there exists a signed overflow*"), adică dacă rezultatul obţinut nu se încadrează în intervalul de interpretare admis (acesta fiind [-128..+127] dacă este vorba despre octeţi şi respectiv [-32768..+32767] pentru cuvinte interpretate cu semn).

Ultimele două reguli derivă de fapt din modul de implementare a conceptului de **depăşire** (*overflow*) la nivelul procesorului 80x86.

În cazul operaţiilor/operanzilor fără semn depăşirea va fi semnalată prin setarea indicatorului CF (*carry flag*). În cazul operaţiilor/operanzilor cu semn depăşirea va fi semnalată prin setarea indicatorului OF (*overflow flag*).

Cum să detectăm însă situaţiile de depăşire în cazul operaţiilor de adunare şi scădere ? Care sunt regulile practice de aplicat pentru a înţelege şi a putea justifica corect setările de flag-uri pe care le remarcăm în cadrul programelor rulate ? În discuţiile ce urmează ne vom concentra în principal pe justificarea modului de setare a flag-ului OF (*overflow flag*) deoarece şi datorită numelui său acesta este principalul factor răspunzător de caracterizarea unei situaţii din partea programatorilor ca fiind depăşire sau nu.

Atragem însă atenţia asupra a ceea ce se ignoră de multe ori în acest context şi anume faptul că o situaţie de tipul CF=1 (cu OF=0) semnalează la rândul ei o depăşire, însă pentru cazul numerelor interpretate fără semn.
Pentru ADUNARE: dacă se adună două numere de acelaşi semn şi rezultatul este de semn diferit atunci se semnalează depăşire (OF=1), **în caz contrar nu** (OF=0). Aceasta este deci ceea ce am putea numi *regula depăşirii la adunare* (RDA) în cazul interpretării cu semn.

De exemplu, la nivel de octet, dacă vom considera adunarea 100 + 50 = 150 vom obţine depăşire (!) cu semn (pare surprinzător, nu-i aşa ?). Justificare: 100 (= 64h = 01100100b) + 50 (= 32h = 00110010b) = 150 (= 96h = 10010110b). Operanzii au acelaşi semn dar rezultatul este de semn diferit, deci conform RDA vom avea OF=1. Intuitiv, depăşirea se poate justifica prin faptul că 150 $\notin$ [-128..127] deci se obţine o eroare de tip *"out of range"*. Deşi s-ar putea replica faptul că 150 = 10010110b = -106 (în interpretarea cu semn), iar -106 $\in$ [-128..127], această ultimă interpretare nu poate fi acceptată deoarece operanzii (100 şi 50) au valori pozitive în ambele interpretări (bitul de semn fiind 0). Ca urmare, suma a două numere pozitive nu poate da un număr negativ şi astfel singura interpretare ce poate fi acceptată în acest context pentru 10010110b este 150 $\notin$ [-128..127] deci se setează OF=1.

Pe de altă parte, CF = 0 (nu există cifră de transport în afara spaţiului de reprezentare) deci nu avem depăşire în interpretarea fără semn: rezultatul adunării 100 + 50 = 150 $\in$ [0, 255] (intervalul de interpretare admis pentru numere fără semn).

Analog, în interpretarea cu semn, suma a două numere negative nu poate furniza un număr pozitiv. Luăm exemplul:

```
  10010110 +
  10000010
 1 00011000
```

Se observă din reprezentarea binară că există un transport de cifră 1 în afara spaţiului de reprezentare admis al celor 8 biţi, deci intuitiv este suficient de justificat depăşirea. Din punct de vedere al aplicării RDA obţinem pe 8 biţi în interpretarea cu semn că suma a două numere negative (ele sunt negative deoarece bitul de semn este 1 pentru ambele numere) ar trebui să furnizeze un număr pozitiv: 00011000b = 18h = 24. Această valoare este de fapt o trunchiere a valorii binare corecte (pe 9 biţi!) ce ar fi trebuit obţinută (100011000b = 118h), iar trunchierea are loc tocmai datorită depăşirii. Ca urmare, nu se poate obţine un număr pozitiv prin adunarea a două numere negative (decât printr-o trunchiere iar necesitatea trunchierii înseamnă de fapt depăşire!). Se observă că o astfel de trunchiere înseamnă întotdeauna şi apariţia unei cifre de transport 1 în afara dimensiunii de reprezentare a rezultatului, deci vom avea automat şi CF=1.

10010110b = 96h = -106 (în interpretarea cu semn) = +150 (în interpretarea fără semn)
10000010b = 82h = -126 (în interpretarea cu semn) = +130 (în interpretarea fără semn)

În interpretarea fără semn avem 150 + 130 = 280 ∉ [0..255] (justificarea intuitivă a depăşirii). Tehnic, am văzut deja că CF = 1 şi rezultă astfel clar că avem depăşire în interpretarea fără semn.

Nu putem avea deci -106 + (-126) = 24! (pentru că 00011000b = 18h = 24 în ambele interpretări) Acesta este sensul în care se aplică RDA aici. Un alt mod de justificare intuitivă a depăşirii în acest tip de situaţie este:

În interpretarea cu semn avem -106 + (-126) = -232 ∉ [-128..127] deci OF=1.

Această ultimă motivaţie este mai intuitivă pentru justificarea depăşirii însă astfel de justificări sunt mai greu de exprimat la nivelul unui algoritm. Tehnic vorbind, RDA rămâne "cea mai rapid aplicabilă regulă practică din punct de vedere algoritmic" dacă ne putem exprima aşa… (şi iată că am putut!)

Rezultă că în cazul în care adunăm două numere de semne diferite nu se va semnala niciodată depăşire. De asemenea, dacă adunăm două numere de acelaşi semn dar rezultatul are acelaşi semn cu operanzii nu se va semnala nici în acest caz depăşire (înseamnă că nu a fost nevoie de trunchiere pentru reprezentarea rezultatului pe aceeaşi dimensiune ca şi cea a operanzilor). Se poate verifica uşor din punct de vedere matematic că în nici unul din aceste cazuri nu ieşim din intervalul de interpretare admis.

Pentru SCĂDERE: se interpretează **operanzii** respectivi cu semn, se efectuează scăderea solicitată asupra configuraţiilor corespunzătoare de biţi şi dacă rezultatul obţinut interpretat cu semn nu se încadrează în intervalul de interpretare admis (intervalul [-128..127] pentru octeţii cu semn şi respectiv [-32768..32767] pentru cuvinte interpretate cu semn) atunci se semnalează depăşire (*overflow*) şi astfel OF=1. Această formulare o putem numi *regula depăşirii la scădere* (RDS) pentru cazul interpretării cu semn.

În cazul depăşirii la scădere fără semn: necesitatea efectuării unei scăderi cu împrumut de cifră este semnalată de către procesor prin setarea CF=1, pe care o putem interpreta semnatic drept "depăşire la scădere în interpretarea fără semn".

Să analizăm în continuare mai multe exemple menite să clarifice aplicarea regulilor de mai sus precum şi impactul lor asupra modului de setare al flag-urilor.

Exemple:

i).
```
mov ah,82h  ;82h = 130 (interpretarea fără semn) = -126 (interpretarea cu semn)
            ; = 10000010b (bitul de semn fiind 1 cele două interpretări diferă)
mov bh,2ah  ;2ah = 42 (atât în interpretarea cu semn cât şi în cea fără semn)
            ; = 00101010b (bitul de semn fiind 0 cele două interpretări coincid)
cmp ah,bh   ;se realizează scăderea fictivă ah-bh=10000010b - 00101010b = 01011000b
            ; = 58h = 88 (atât în interpretarea cu semn cât şi în cea fără semn
              deoarece ;bitul de semn este 0)
```

Această scădere setează flag-urile astfel:

SF = 0 (deoarece bitul de semn pentru rezultatul 58h = 01011000b este 0)
CF = 0 (deoarece |82h| > |2ah| nu se pune problema unei scăderi cu împrumut de cifră; deci
nu vom avea depăşire în interpretarea fără semn care se efectuează: 130 – 42 = 88)
OF = 1 (se efectuează scăderea în interpretarea cu semn, adică ah-bh = -126 – 42 = -168 şi
cum -168 ∉ [-128..127] se semnalează *signed overflow* şi ca urmare OF=1)

```
cmp bh,ah   ;se realizează scăderea fictivă bh-ah = 00101010b-10000010b = 10101000b
            ; = A8h = 168 (în interpretarea fără semn) = -88 (în interpretarea cu semn)
```

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul A8h = 10101000b este 1)
CF = 1 (deoarece |2ah| < |82h| se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine 42 – 130 = 168 (!) provenită de fapt din necesitatea unei scăderi de tipul (256 + 42) – 130 = 168 şi ca urmare a necesităţii împrumutului se va semnala depăşire în interpretarea fără semn, înţeleasă aici ca "nu se poate efectua corect această scădere fără utilizarea unei cifre de împrumut")
OF = 1 (se efectuează scăderea în interpretarea cu semn, adică bh-ah = 42-(-126) = +168 şi
cum +168 ∉ [-128..127] se semnalează *signed overflow* şi ca urmare OF=1)

ii).

```
mov ah,126  ;echivalent cu mov ah,7eh deoarece 126 = 7Eh = 01111110b (bitul de ;semn fiind 0 cele două interpretări coincid, ca urmare conţinutul lui AH este ;126 atât în interpretarea cu semn cât şi în cea fără semn)
mov bh,2ah  ;2ah = 42 (atât în interpretarea cu semn cât şi în cea fără semn)
            ; = 00101010b (bitul de semn fiind 0 cele două interpretări coincid)
cmp ah,bh   ;se realizează scăderea fictivă ah-bh= 01111110b-00101010b = 01010100b
            ; = 54h = 84 = 126 - 42  (atât în interpretarea cu semn cât şi în cea fără semn deoarece bitul de semn al rezultatului este 0)
```

Această scădere setează flag-urile astfel:
SF = 0 (deoarece bitul de semn pentru rezultatul 54h = 01010100b este 0)
CF = 0 (deoarece |126| > |42| nu se pune problema unei scăderi cu împrumut de cifră, deci nu se va semnala depăşire în interpretarea fără semn)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică ah-bh = 126 – 42 = 84 şi

cum 84 ∈ [-128..127] NU se semnalează *signed overflow* şi ca urmare OF=0)

```
cmp bh,ah   ;se realizează scăderea fictivă bh-ah = 00101010b-01111110b = 10101100b
            ; = 42-126 = ACh = 172 (în interpretarea fără semn) = -84 (în interpretarea ;cu semn)
```

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul ACh = 10101100b este 1)
CF = 1 (deoarece |42| < |126| se pune problema unei scăderi cu împrumut de cifră ; în interpretarea fără semn scăderea devine 42 – 126 = 172 (!) provenită de fapt din necesitatea unei scăderi de tipul (256 + 42) – 126 = 172 şi ca urmare a necesităţii împrumutului se va semnala depăşire în interpretarea fără semn prin setarea flagului carry)
OF = 0 (se efectuează scăderea în interpretarea cu semn, adică bh-ah = 42-126 = -84 şi

cum -84 ∈ [-128..127] NU se semnalează *signed overflow* şi ca urmare OF=0)

Ca regulă generală să observăm că din punctul de vedere al reprezentării binare, dacă rezultatul scăderii a-b ∈ [-127..127] atunci şi b-a ∈ [-127..127] (situaţia particulară în care a-b = -128 o tratăm mai jos). Analog pentru reprezentări de tip cuvânt la nivelul intervalului [-32767..32767] cu discuţie asupra cazului particular -32768. Ca urmare se poate concluziona faptul că instrucţiunile **cmp a,b** şi **cmp b,a** vor furniza întotdeauna aceeaşi valoare pentru OF.

iii). - discuţie asupra cazurilor cmp 80h,0 şi cmp 0,80h

```
mov ah,80h  ;80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)
            ; = 10000000b (bitul de semn fiind 1 cele două interpretări diferă)
mov bh,0    ;bh:=0

cmp ah,bh   ;se realizează scăderea fictivă ah-bh= 10000000b-00000000b = 10000000b
            ; = 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)
```

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul 80h = 10000000b este 1)
CF = 0 (deoarece |80h| > |0| nu se pune problema unei scăderi cu împrumut de cifră, deci nu

poate fi vorba despre depăşire în interpretarea fără semn)

OF = 0 (se efectuează scăderea în interpretarea cu semn, adică ah-bh = -128 – 0 = -128 şi

cum -128 ∈ [-128..127] NU se semnalează *signed overflow* şi ca urmare OF=0)

cmp bh,ah ;se realizează scăderea fictivă bh-ah= 00000000b-10000000b = 10000000b

; = 80h = 128 (interpretarea fără semn) = -128 (interpretarea cu semn)

Această scădere setează flag-urile astfel:

SF = 1 (deoarece bitul de semn pentru rezultatul 80h = 10000000b este 1)
CF = 1 (deoarece |0h| < |80h| se pune problema unei scăderi cu împrumut de cifră; în interpretarea fără semn scăderea devine 0 – 128 = 128 (!) provenită de fapt din necesitatea unei scăderi de tipul (256 + 0) – 128 = 128 şi ca urmare a necesităţii împrumutului se va semnala depăşire în interpretarea fără semn prin setarea flagului carry)

OF = 1 (se efectuează scăderea în interpretarea cu semn, adică bh-ah = 0 – (-128) = +128 şi

cum +128 ∉ [-128..127] se semnalează *signed overflow* şi ca urmare OF=1)

CF = 1 în cazul cmp 0,80h deoarece se efectuează o scădere cu împrumut de tipul :

$$\begin{array}{rl} 0 - 10000000b = & \mathbf{1}\,00000000 - \\ & \underline{\quad 10000000} \\ & \mathbf{0}10000000 \end{array}$$

şi cifra de împrumut se transferă în CF.

Să analizăm în acest context ce înseamnă şi cum s-a ajuns la domeniul "numerelor cu semn posibil a fi reprezentate pe 1 octet" respectiv domeniul "numerelor cu semn posibil a fi reprezentate pe 1 cuvânt".

Pe 1 octet se pot reprezenta 256 de valori, indiferent că vorbim despre interpretarea cu semn sau interpretarea fără semn. În interpretarea fără semn aceste valori sunt cele din intervalul [0..255]. Care sunt însă cele 256 de valori reprezentabile în interpretarea cu semn ? Este vorba despre intervalul [-128..127] sau despre intervalul [-127..128] ? Pentru că nu poate fi vorba despre intervalul [-128..128] deoarece în acest interval sunt 257 de valori ! Cu alte cuvinte cineva a trebuit să aleagă una dintre cele două variante şi totodată să facă precizarea că numerele -128 şi +128 nu pot coexista între limitele aceluiaşi interval de reprezentare al aceluiaşi tip de dată! (reamintim că în limbaj de asamblare *tip de dată = dimensiune de reprezentare*)

În acest sens este de observat şi impactul acestui mod de reprezentare asupra limbajelor de nivel înalt: de exemplu atât **shortint** cât şi **byte** în Turbo Pascal acceptă valoarea 80h (-128 ca *shortint* şi +128 ca *byte*) însă 80h **nu poate avea două interpretări distincte în**

cadrul aceluiaşi tip de dată ! Nu vom întâlni la nivelul nici unui limbaj de programare de nivel înalt valorile -128 şi +128 ca fiind prezente în cadrul aceluiaşi tip de dată !

Ca urmare, s-a luat decizia ca intervalul acceptat al valorilor cu semn reprezentabile pe 1 octet să fie intervalul [-128..+127] (care este exact domeniul de valori şi a tipului de dată **shortint** din Turbo Pascal): deci **+128 nu este acceptat ca valoare cu semn reprezentabilă pe 1 octet !**

Totuşi, după cum putem verifica foarte uşor, instrucţiunile **mov ah, 128** şi **mov ah,-128** sunt amândouă acceptate de către asamblor, efectul fiind în ambele cazuri încărcarea în *ah* a configuraţiei binare 10000000b ! Aceasta deoarece în primul caz va fi vorba de fapt despre interpretarea fără semn pentru 80h iar în al doilea caz va fi vorba despre interpretarea cu semn. Simpla încărcare a unui registru cu o anumită configuraţie binară nu presupune şi necesitatea interpretării respectivei configuraţii într-un anumit fel. Sarcina interpretării acelei configuraţii drept cu semn sau fără semn va cădea în sarcina instrucţiunilor ce urmează şi care vor folosi ca operanzi aceste valori. De exemplu, utilizarea lui IMUL în loc de MUL va provoca interpretarea configuraţiei binare respective drept un operand cu semn în loc de unul fără semn. Analog, utilizarea lui DIV în loc de IDIV va provoca interpretarea aceluiaşi operand ca fără semn ş.a.m.d.

În cazul cmp 80h,0 se efectuează 80h–0 = 80h = 10000000b (128 - 0 = 128 în interpretarea fără semn) fără a fi nevoie de o cifră de transport împrumutată pentru a putea efectua scăderea, deci nu avem depăşire în interpretarea fără semn şi astfel CF = 0. În interpretarea cu semn a operanzilor şi a rezultatului final avem -128 - 0 = -128 ∈ [-128..127] deci nu avem depăşire nici în interpretarea cu semn şi astfel OF = 0.
Pe de altă parte, avem evident în ambele cazuri SF=1. Justificarea *intuitivă*: în interpretarea cu semn valoarea 10000000b reprezintă un număr strict negativ adică -128. Justificarea *tehnică*: bitul de semn al reprezentării binare 10000000b este 1 deci SF=1.

iv). Să analizăm în continuare modurile în care putem compara valorile 0 şi 1 (şi apoi 0 şi -1) şi ce efecte are asupra flagurilor instrucţiunea cmp în fiecare dintre situaţii.

Situaţia **cmp 1,0** (evidenţiată la nivelul unui text sursă de exemplu prin cmp ah,0 cu ah=1) va efectua scăderea fictivă 1-0 = 1 = 00000001b. Efectul asupra flag-urilor va fi CF = SF = OF = ZF = PF = AF = 0. Justificările sunt evidente pe baza discuţiilor din exemplele anterioare.

Situaţia **cmp 0,1** (evidenţiată la nivelul unui text sursă de exemplu prin cmp ah,1 cu ah=0) va efectua scăderea fictivă 0-1 = -1 = 11111111b:

$$0 - 00000001b = \begin{array}{r} 1\ 00000000 - \\ \underline{\quad 00000001} \\ 0\ 11111111 \end{array}$$

Efectul asupra flag-urilor va fi CF = SF = PF = AF = 1 şi ZF = OF = 0. Justificarea valorilor din CF şi SF este şi aici evidentă pe baza discuţiilor din exemplele anterioare iar OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar -1 ∈ [-128..127].

Situaţia **cmp -1,0** (evidenţiată la nivelul unui text sursă de exemplu prin cmp ah,0 cu ah = -1) va efectua scăderea fictivă -1- 0 = -1 = 11111111b. Efectul asupra flag-urilor va fi SF = PF = 1 şi CF = OF = ZF = AF = 0. SF=1 deoarece bitul de semn este 1. OF=0 deoarece rezultatul în interpretarea cu semn este -1, iar -1 ∈ [-128..127]. CF=0 deoarece nu se impune efectuarea unei scăderi cu împrumut.

Situaţia **cmp 0,-1** (evidenţiată la nivelul unui text sursă de exemplu prin cmp ah,-1 cu ah = 0) va efectua scăderea fictivă 0 – (-1) = +1 = 00000001b:

```
0 - 11111111b   =   1 00000000 –
                      11111111
                    0 00000001
```

Efectul asupra flag-urilor va fi CF = AF = 1 şi OF = SF = ZF = PF = 0. SF = 0 deoarece bitul de semn este 0. OF=0 deoarece 0 – (-1) = +1 ∈ [-128..127]. CF = 1 deoarece se impune efectuarea unei scăderi cu împrumut. Putem justifica şi aşa: în interpretarea fără semn această scădere înseamnă de fapt 0 - 255 = 1 (!), care trebuie justificată prin (256+0) – 255 = 1, deci e nevoie de cifră de împrumut şi astfel se semnalează depăşire în cazul interpretării fără semn, deci CF = 1.

v). Cazurile studiate anterior (i-iv) s-au referit la operaţii de scădere datorită analizei pe care am avut-o în vedere asupra efectelor instrucţiunii cmp. Să analizăm în continuare şi cazul unei depăşiri furnizate de operaţia de adunare revenind astfel la discuţia asupra aplicării regulii RDA:

```
        mov ah,126  ;126 = 01111110b = 7eh (aceeaşi valoare 126 în ambele interpretări)
        add ah, 2   ; 2 = 2h = 00000010b ; AH := 01111110b + 00000010b  = 7eh + 02h
=
                    ; 10000000b = 80h (= 128 fără semn = -128 cu semn)
CF = 0  deoarece:     01111110 +
                      00000010
                      10000000    - nu există transport în afara spaţiului de reprezentare
            al rez.
```

SF = 1 deoarece bitul de semn al rezultatului este 1 (în interpretarea cu semn rezultatul operaţiei efectuate este strict negativ = -128).

OF = 1 deoarece:
- justificare *tehnică* - conform RDA se adună două numere de acelaşi semn (bitul de semn este 0 pentru amândouă) iar rezultatul este de semn diferit (bitul de semn este 1).

- justificare *intuitivă* - adunăm două numere fără semn a căror sumă este 126 + 2 = 128. Însă numărul +128 ∉ [-128..127] deci se semnalează *signed overflow* şi ca urmare OF=1.

vi). Unul dintre efectele surprinzătoare ale interpretărilor cu semn sau fără semn se referă la situaţia în care programatorul îşi iniţializează operanzii cu anumite valori iniţiale dorite (cu semn sau fără semn, conform necesităţilor problemei în cauză) şi se aşteaptă la obţinerea unor rezultate sau reacţii în conformitate cu valorile furnizate. Atenţie însă! De obicei aceste valori au o dublă interpretare posibilă şi nu vor fi interpretate în orice situaţie sub forma furnizată la iniţializare!

Utilizarea ulterioară a unor instrucţiuni care forţează prin modul de lor de acţiune interpretarea complementară (cu semn/fără semn) celei de la iniţializare poate provoca apariţia unor situaţii în care un utilizator la prima vedere fie să suspecteze erori din partea asamblorului (!) fie din punct de vedere al exprimării în baza 10 să se ajungă la interpretări hilare… Aceasta se întâmplă dacă nu se ţine cont în permanenţă de dubla interpretare posibilă a configuraţiilor binare manipulate. Să luăm un exemplu:

```
mov al, 200  ; al = 11001000b = 0C8h = 200 (fără semn) = -56 (cu semn)
mov bl, -1   ; bl = 11111111b = 0FFh = 255 (fără semn) = -1 (cu semn)
cmp al, bl   ; al-bl = 11001001b = C9h = -55 (cu semn) = 201 (fără semn)
               (şi se setează corespunzător OF=ZF=0 şi CF=SF=1)
```

Deci pe cine comparăm de fapt aici? Pe 200 cu -1 aşa cum precizează valorile de la iniţializare?
Sau poate pe 200 cu 255? Sau pe -56 cu -1 ? Sau pe -56 cu 255?

Răspuns: comparăm întotdeauna pe 0C8h cu 0FFh sau în exprimare binară pe 11001000 cu 11111111. Efectul va fi unul singur: afectarea corespunzătoare a flag-urilor în urma efectuării scăderii fictive AL-BL. Modul de exprimare corect al comparaţiei efectuate în baza 10 nu este dedus din acţiunea instrucţiunii CMP (care nu distinge absolut de loc între cele 4 variante posibile de comparare de mai sus) ci pe baza unor eventuale instrucţiuni ulterioare care vor avea ele rolul de a interpreta în unul din cele 4 moduri de mai sus comparaţia efectuată. Să urmărim în acest sens variantele de comparare de mai jos identificate prin utilizarea instrucţiunilor corespunzătoare de salt condiţionat:

```
jl et1  ; evident că 200≮-1 deci la prima vedere pare că nu este îndeplinită condiţia
```

necesară pentru efectuarea saltului… să nu uităm însă faptul că JL (Jump If Less) interpretează rezultatul comparaţiei ca fiind cu semn (deci -55) aceasta însemnând implicit şi faptul că scăderea este interpretată ca (-56 – (-1)) deci şi operanzii vor fi amândoi interpretaţi cu semn… cum -56 < -1 iată că şi intuitiv condiţia se verifică (pe lângă justificarea tehnică a îndeplinirii condiţiei de salt SF≠OF) şi deci saltul se va efectua ! Deci chiar dacă programatorul a furnizat la iniţializare valorile 200 şi -1, utilizarea instrucţiunii JL a provocat interpretarea comparaţiei ca fiind între -55 şi -1 şi nu între 200 şi -1! (explicaţia de aici şi faptul că saltul se va efectua vă poate ajuta să "demonstraţi" unor colegi cum 200 poate fi mai mic decât -1 !!!)

ja et2 ; deoarece 200 > -1 în acest caz ne-am aştepta ca saltul să se efectueze… însă utilizarea instrucţiunii JA (Jump if Above) impune interpretarea fără semn, deci varianta de comparaţie corectă aici este comparaţia lui 200 cu 255 şi cum 200 $\not>$ 255 condiţia nu este îndeplinită şi deci saltul nu se va efectua (iată deci cum se poate "demonstra" că 200 nu este superior valorii -1 !!!). Ca o confirmare, se poate vedea că nici condiţia tehnică impusă de JA nu este îndeplinită: ar trebui să avem CF=ZF=0, însă în cazul nostru CF=1 deci saltul nu se va efectua.

jb et3 ; intuitiv 200 < 255, iar tehnic CF=1 deci saltul se efectuează

jg et4 ; intuitiv -56 $\not>$ -1, iar tehnic deşi ZF=0 nu este îndeplinită şi condiţia SF = OF deci

; saltul nu se va efectua

Ca urmare din cele 4 situaţii teoretic posibile de mai sus, vom întâlni concret numai două:

- comparaţie fără semn (200 cu 255) - impusă de "above" sau "below"
- comparaţie cu semn (-56 cu -1) – impusă de "less than" sau "greater than"

Nu putem aşadar compara de fapt pe 200 cu -1 aşa cum au fost specificate valorile la iniţializare şi nici pe -56 cu 255 deoarece **interpretarea este ori cu semn ori fără semn pentru ambii operanzi!**

vii). Am studiat în exemplele anterioare modalitatea de reacţie (de interpretare) a procesorului 80x86 legată de noţiunea de depăşire în cazul operaţiilor de adunare şi de scădere. Când şi cum semnalează însă procesoarele din familia 80x86 depăşirea la înmulţire şi respectiv la împărţire ?

"Depăşirea" la înmulţire. Instrucţiunile MUL şi IMUL setează CF=1 şi OF=1 dacă "jumătatea" superioară a produsului (octetul superior dacă este vorba despre produs-cuvânt sau cuvântul superior dacă este vorba despre produs-dublucuvânt) este o valoare diferită de zero. Aceasta este definiţia noţiunii de "depăşire la înmulţire" în cazul arhitecturii 80x86. Să remarcăm faptul că nu se face distincţie între MUL şi IMUL şi de aceea nici între CF şi OF. Ori vor fi amândouă flag-urile setate la valoarea 1 cu semnificaţia de "depăşire la înmulţire" în sensul precizat mai sus, ori vor primi amândouă valoarea 0. Iată un exemplu pe 8 biţi:

```
mov al, 5
mov bl,170
mul  bl        ;AX := AL * BL = 5 * 170 = 850 = 0352h şi vom avea CF=1 şi
OF=1                    ;deoarece octetul superior AH = 03 ≠ 0.
```

Varianta cu IMUL va furniza:

```
mov al, 5
mov bl,170  ;170 = 0aah = -86 în interpretarea cu semn
imul bl     ;AX := AL * BL = 5 * (-86) = - 430 = 0fe52h şi vom avea CF=1 şi
```

;OF=1 deoarece octetul superior AH = 0feh ≠ 0.

În cazul unor operanzi pe 16 biţi putem avea de exemplu:

```
val1 DW 2000h
val2 DW 0100h
…
mov ax, val1
mul val2        ;DX:AX = 00200000h şi vom avea CF=1 şi OF=1 deoarece
jumătatea ;superioară a produsului DX:AX, adică registrul DX conţine valoarea
0020h ≠ 0.
```

Aceste setări nu trebuie să le interpretăm drept erori. Nu este în nici un caz vorba despre o potenţială pierdere de informaţie ca şi în cazul celorlalte depăşiri - adunare, scădere sau împărţire. Aceasta deoarece chiar dacă înmulţim valorile maximale posibil a fi reprezentate pe dimensiunea operanzilor (255 * 255 pentru octeţi şi respectiv 65535 * 65535 pentru cuvinte) tot nu se depăşeşte dublul dimensiunii de reprezentare a operanzilor, adică spaţiul pe care îl avem oricum la dispoziţie prin definiţie, deoarece 255 * 255 = 65025 < 65535 (numărul maximal fără semn reprezentabil pe un cuvânt) iar 65535 * 65535 = 4 294 836 225 < 4 294 967 295 (numărul maximal fără semn reprezentabil pe un dublucuvânt).

În cazul înmulţirii cu semn (instrucţiunea IMUL) justificarea este similară: 127 * 127 = 16129 < 32767 (numărul maximal cu semn ce poate fi reprezentat pe 1 cuvânt), iar 32767 * 32767 = 1 073 676 289 < 2 147 483 647 (numărul maximal cu semn reprezentabil pe un dublucuvânt).

Depăşirea în cazul înmulţirii la nivelul limbajului de asamblare 80x86 este doar o semnalare a faptului că plecându-se de la operanzi octeţi (respectiv cuvinte) produsul nu încape tot într-un octet (respectiv într-un cuvânt) ci este realmente nevoie de o dimensiune dublă pentru memorarea rezultatului. În acest sens, a se vedea şi capitolul 1, în care din punct de vedere matematic s-a specificat clar că înmulţirea nu provoacă de fapt depăşire, tocmai din cauza alocării unui spaţiu suficient pentru reprezentarea produsului. În concluzie, se poate spune că din punct de vedere matematic singura operaţie care nu provoacă depăşire este înmulţirea, însă procesoarele 80x86 promovează totuşi noţiunea de “depăşire la înmulţire” pentru a diferenţia între situaţiile în care produsul încape într-un spaţiu de dimensiunea operanzilor şi în care nu.

Situaţiile în care produsul încape pe dimensiunea operanzilor vor fi caracterizate de setările CF = OF = 0 (nu avem deci depăşire la înmulţire). Iată un exemplu:

```
mov al, 5
mov bl, 51
mul  bl         ; AX := AL * BL = 5 * 51 = 255 = 00ffh şi vom avea CF=0 şi
OF=0                    ; deoarece octetul superior AH = 0.
```

Depăşirea la împărţire. În cazul împărţirii, specificarea acestei operaţii sub forma

(I)DIV *operand*

presupune că operandul specificat este împărţitorul (posibil a fi reprezentat fie pe 8 fie pe 16 biţi) iar deîmpărţitul este considerat implicit în AX (dacă *operand* este octet) sau în DX:AX (dacă împărţitorul este cuvânt). Efectuarea operaţiei are ca efect:

AX : operand pe 8 biţi = câtul în AL şi restul în AH;
DX:AX / operand pe 16 biţi = câtul în AX şi restul în DX;

În cazul împărţirii depăşirea apare atunci când rezultatul împărţirii nu încape în spaţiul rezervat conform definiţiei pentru reprezentare, mai exact, când câtul nu încape în AL sau respectiv AX. Într-o astfel de situaţie, procesorul 80x86 emite o întrerupere 0, execuţia terminându-se cu un mesaj furnizat de către rutina de tratare a întreruperii 0, de genul "Divide by zero", "Zero divide" sau "Divide overflow" (în funcţie de tipul de procesor şi/sau de SO instalat). Pare ciudat la prima vedere că o împărţire prin 0 (de genul *div bh* cu bh = 0) ce practic nu se poate efectua din punct de vedere matematic este tratată similar ca efect din punct de vedere al limbajului de asamblare cu o împărţire care matematic se poate efectua. Secvenţa

```
mov ax,60000
mov bl,2
div bl
```

ar trebui să furnizeze din punct de vedere matematic câtul 30000. Însă conform definiţiei împărţirii DIV acest cât trebuie memorat în registrul AL, de dimensiune octet. Cum cea mai mare valoare reprezentabilă pe 1 octet este 255, este evident astfel că din punct de vedere al limbajului de asamblare împărţirea de mai sus nu se poate nici ea efectua (similar cu o situaţie de tip div 0) şi ca urmare înţelegem acum decizia proiectanţilor de a trata tot prin emiterea unei întreruperi 0 şi o situaţie de genul celei de mai sus. Să remarcăm în acest sens şi faptul că mesajul "*Divide overflow*" (depăşire la împărţire) este acceptat în acest context ca similar unui "*Divide by zero*".

viii). Una dintre erorile logice frecvente pe care o fac programatorii neexperimentaţi este de a confunda exprimările *"numere cu semn"* şi *"numere fără semn"* cu exprimările *"numere negative"* şi respectiv *"numere pozitive"*. Numere cu semn nu înseamnă automat numere negative ! Numerele cu semn sunt fie pozitive, fie negative. Numerele fără semn sunt întotdeauna pozitive.
Ce concluzii vom trage relativ la modul de interpretare (cu semn sau fără semn) din enunţul unei probleme care cere efectuarea unei anumite acţiuni *"dacă numărul v este (strict) negativ"*? În primul rând vom concluziona că este vorba despre interpretarea cu semn. Se pune însă întrebarea: cum vom testa practic dacă un număr cu semn este negativ sau nu? (să presupunem că *v* este octet). Fiind vorba despre interpretarea cu semn, dacă primul bit al configuraţiei binare este 1 atunci numărul este negativ. Deci totul se reduce la un test asupra

primului bit din reprezentarea numărului. Iată două alternative pentru realizarea unui astfel de test:

a). Realizăm o deplasare a primului bit în CF şi testăm valoarea sa printr-o instrucţiune adecvată de salt condiţionat. Secvenţa

```
mov al,v            ;pentru a nu afecta destructiv conţinutul variabilei v
shl al,1            ;shift stânga cu 1 poziţie pentru ca primul bit să treacă în CF.
jc este_negativ     ;dacă CF=1 atunci salt la eticheta este_negativ
```

asigură testarea faptului dacă variabila *v* este sau nu un număr negativ.

b). Utilizăm instrucţiunea **cmp** pentru o comparaţie în raport cu 0:

```
cmp v,0             ;scădere fictivă v-0
jl  este_negativ    ;dacă v<0 atunci salt la eticheta este_negativ
```

sau alternativ

```
cmp 0,v             ; scădere fictivă 0-v
jg  este_negativ    ;dacă 0>v atunci salt la eticheta este_negativ
```

ix). Am văzut ca la nivelul efectuării operaţiilor de adunare sau scădere procesorul 80x86 nu diferenţiază între adunări/scăderi cu semn sau fără semn (tehnic vorbind ele se efectuează drept operaţii binare cu rezultat interpretabil **ulterior** drept cu semn sau fără). Totuşi, în momentul în care se pune problema exprimării în baza 10 a unei operaţii de adunare sau scădere ne punem întrebarea: cum să exprimăm semantic corect operanzii operaţiei respective pentru ca aceste exprimări să fie consistente cu interpretarea rezultatului final obţinut ? Mai concret:

```
    00000101 +          (= 5 în ambele interpretări)
    11111110    (= 254 fără semn şi -2 în interpretarea cu semn)
(1) 00000011    (= 3 în ambele interpretări ale configuraţiei pe 8 biţi)
```

reprezintă 5 + 254 = 259 (= 1 00000011 – configuraţie pe 9 biţi !) sau reprezintă 5 + (-2) = 3 ? După cum vom vedea şi aici răspunsul este că putem interpreta în ambele moduri şi să justificăm astfel ca două reacţii separate modul de setare al flag-urilor CF şi respectiv OF.

Datorită cifrei de transport vom avea CF=1 (independent de interpretarea operanzilor sau a rezultatului final drept cu semn sau fără semn, deoarece este vorba despre o consecinţă tehnică a modului de efectuare a operaţiei binare de adunare). Ca urmare în interpretarea fără semn avem depăşire (evident, deoarece 259 > 255, adică decât numărul maxim reprezentabil pe 1 octet).

Ce se întâmplă cu OF ? Rularea secvenţei

```
mov al, 5      ; = 5 în ambele interpretări
mov bl, 254    ; = -2 în interpretarea cu semn
add al, bl     ; AL := AL+BL = 5+(-2) = 3
```

nu setează flagul OF la valoarea 1, deci situaţia de mai sus nu este considerată "depăşire" în interpretarea cu semn! Din punct de vedere al justificării modului de setare a flag-ului OF secvenţa de mai sus ar fi mai corectă dacă ar fi scrisă:

```
mov al, 5
mov bl, -2
add al, bl     ; deci  5 + (-2) = 3
```

şi este evident că în această interpretare nu este vorba despre nici o depăşire (şi de aceea şi OF = 0).

Să ne reamintim în acest context şi exemplele date la prezentarea RDA şi RDS de la paginile ??-??: adunarea 100 + 50 = 150 va semnala depăşire (*signed overflow* - conform RDA), iar scăderile 130 - 42 (interpretată ca -126 - 42 = -168 $\notin$ [-128..127]) şi 42 -130 (interpretată ca scăderea 42-(-126) = +168 $\notin$ [-128..127]) produc la rândul lor *signed overflow* şi ca urmare OF=1.

C H A P T E R 4
ASSEMBLY LANGUAGE INSTRUCTIONS

General form of an ASM program in NASM + short example:

```
global start                    ; we ask the assembler to give global visibility to the symbol called start
                                ;(the start label will be the entry point in the program)

extern ExitProcess, printf      ; we inform the assembler that the ExitProcess and printf symbols are foreign
                                ;(they exist even if we won't be defining them),
                                ; there will be no errors reported due to the lack of their definition

import ExitProcess kernel32.dll  ;we specify the external libraries that define the two symbols:
                    ; ExitProcess is part of kernel32.dll library (standard operating system library)
import printf msvcrt.dll            ;printf is a standard C function from msvcrt.dll library (OS)

bits 32                     ; assembling for the 32 bits architecture

segment code use32 class=CODE          ; the program code will be part of a segment called code
        start:
                ; call printf(”Hello from ASM”)
                push dword string   ; we send the printf parameter (string address) to the stack (as printf requires)
                call   [printf]          ; printf is a function (label = address, so it must be indirected [])

                ; call ExitProcess(0), 0 represents status code: SUCCESS
                push dword  0
                call   [ExitProcess]

segment data use32 class=DATA   ; our cariables are declared here (the segment is called data)
        string:    db "Hello from ASM!", 0
```

4.1. DATA MANAGEMENT / 4.1.1. Data transfer instructions

4.1.1.1. General use transfer instructions

MOV *d,s*	<d> <-- <s> (b-b, w-w, d-d)	-
PUSH *s*	ESP = ESP - 4 and transfers („pushes") <s> in the stack (s – doubleword)	-
POP *d*	Eliminates („pops") the current element from the top of the stack and transfers it to d (d – doubleword) ; ESP = ESP + 4	-
XCHG *d,s*	<d> ↔ <s> ; s,d – have to be L-values !!!	-
[reg_segment] XLAT	AL ← < DS:[EBX+AL] > or AL ← < segment:[EBX+AL] >	-
CMOVcc d, s	<d> ← <s> if cc (conditional move) is true	-
PUSHA / PUSHAD	Pushes in the stack EAX, ECX, EDX, EBX, ESP, EBP, ESI and EDI	-
POPA / POPAD	Pops EDI, ESI, EBP, ESP, EBX, EDX, ECX and EAX from stack	-
PUSHF	Pushes EFlags in the stack	-
POPF	Pops the top of the stack and transfers it to Eflags	-
SETcc d	<d> ← 1 if cc is true, otherwise <d> ← 0 (byte set on condition code)	-

If the destination operand of the MOV instruction is one of the 6 segment registers, then the source must be one of the eight 16 bits EU general registers or a memory variable. The loader of the operating system initializes automatically all segment registers and changing their values, although possible from the processor point of view, does not bring any utility (a program is limited to load only selector values which indicates to OS preconfigured segments without being able to define additional segments).

PUSH and **POP** instructions have the syntax **PUSH** *s* and **POP** *d*

Operands d and s MUST be doublewords, because the stack is organized on doublewords. The stack grows from big addresses to small addresses, 4 bytes at a time, ESP pointing always to the doubleword from the top of the stack .

We can illustrate the way in ehich these instructions works, by using an equivalent sequence of MOV and ADD or SUB instructions:

```
push eax  ⇔     sub esp, 4          ; prepare (allocate) space in order to store the value
                mov [esp], eax      ; store the value in the allocated space

pop eax   ⇔     mov eax, [esp]      ; load in eax the value from the top of the stack
                add esp, 4          ; clear the location
```

These instructions only allow you to deposit and extract values represented by word and double. Thus, PUSH AL is not a valid instruction (syntax error), because the operand is not allowed to be a byte value. On the other hand, the sequence of instructions

```
PUSH  ax     ; push ax in the stack
PUSH  ebx    ; push ebx in the stack
POP   ecx    ; ecx <- the doubleword from the top of the stack (the value of ebx)
POP   dx     ; dx <- the word from the stack (the value of ax)
```

Is a valid sequence of instructions and is equivalent as an effect with:

```
MOV   ecx,   ebx
MOV   dx,    ax
```

In addition to this constraint (which is inherent in all x86 processors), the operating system requires that stack operations be made only through doublewords or multiple of doublewords accesses, for reasons of compatibility between user programs and the kernel and system libraries. The implication of this constraint is that the PUSH operand16 or POP operand16 instructions (for example, PUSH word 10), although supported by the processor and assembled successfully by the assembler, is not allowed by the operating system, might causing what is named the incorrectly aligned stack error: the stack is correctly aligned if and only if the value in the ESP register is permanently divisible by 4!

The **XCHG** instruction allows interchanging the contents of two operands having the same size (byte, word or doubleword), at least one of them having to be a register (the other one being either a register or a memory address). Its syntax is

XCHG *operand1, operand2*

XLAT "translates" the byte from AL to another byte, using for that purpose a user-defined correspondence table called *translation table.* The syntax of the XLAT instruction is

[reg_segment] XLAT

translation_table is the direct address of a string of bytes. The instruction requires at entry the far address of the translation table provided in one of the following two ways:

- DS:EBX (implicit, if the segment register is missing)
- segment_register:EBX, if the segment register is explicitly specified.

The effect of **XLAT** is the replacement of the byte from AL with the byte from the translation table having the index the initial value from AL (the first byte from the table has index 0). EXAMPLE: pag.111-112 (course book).

For example, the sequence

```
mov  ebx, Table
mov  al,6
ES  xlat                    AL ← < ES:[EBX+6] >
```

transfers the content of the 7th memory location (having the index 6) from *Table* into AL.

The following example translates a decimal value "number" between 0 and 15 into the ASCII code of the corresponding hexadecimal digit :

```
segment data use32
        .       .       .
TabHexa     db      '0123456789ABCDEF'
        .       .       .
segment code use32
mov   ebx, TabHexa
        .       .       .
mov   al, numar
xlat                        ; AL ← < DS:[EBX+AL] >
 ES xlat                    ; AL ← < ES:[EBX+AL] >
```

This strategy is commonly used and proves useful in preparing an integer numerical value for printing (it represents a conversion *register numerical value – string to print*).

How many ACTIVE code segments can we have ?... 1 – CS
How many ACTIVE stack segments can we have ?... 1 - SS
How many ACTIVE data segments can we have ?... 2 – DS and ES BOTH are reffering to DATA segments

4.1.1.3. Address transfer instruction - LEA

LEA *general_reg*, *contents of a memory_operand*	general_reg ← offset(mem_operand)	-

LEA (***L**oad **E**ffective **A**ddress*) transfers the offset of the *mem* operand into the destination register. For example

```
lea   eax,[v]
```

loads into EAX the offset of the variable v, the instruction equivalent to `mov  eax, v`

But **LEA** has the advantage that the source operand may be an addressing expression (unlike the mov instruction which allows as a source operand only a variable with direct addressing in such a case). For example, the instruction:

```
lea   eax,[ebx+v-6]
```

is not equivalent to a single MOV instruction. The instruction `mov eax, ebx+v-6`

is syntactically incorrect, because the expression ebx+v-6 cannot be determined at assembly time.

By using the values of offsets that result from address computations directly (in contrast to using the memory pointed by them), LEA provides more versatility and increased efficiency: versatility by combining a multiplication with additions of registers and/or constant values and increased efficiency because the whole computation is performed in a single instruction, without occupying the ALU circuits, which remain available for other operations (while the address computation is performed by specialized circuits in BIU)

Example: multiplying a number with 10

```
mov eax, [number]        ; eax <- the value of the variable number
lea eax,   [eax * 2]     ; eax <- number * 2
lea eax,   [eax * 4 + eax]   ; eax <- (eax * 4) + eax  = eax * 5 = (number * 2) * 5
```

4.1.1.4. Flag instructions

The following four instructions are *flags transfer instructions*:

LAHF (***Load register AH from Flags***) copies SF, ZF, AF, PF and CF from FLAGS register in the bits 7, 6, 4, 2 and 0, respectively, of register AH. The contents of bits 5, 3 and 1 are undefined. Other flags are not affected (meaning that LAHF does not generate itself other effects on some other flags – it just transfers the flags values and that's all).

SAHF (***Store register AH into Flags***) transfers the bits 7, 6, 4, 2 and 0 of register AH in SF, ZF, AF, PF and CF respectively, replacing the previous values of these flags.

PUSHF transfers all the flags on top of the stack (the contents of the EFLAGS register is transferred onto the stack).The values of the flags are not affected by this instruction. The **POPF** instruction extracts the word from top of the stack and transfer its contents into the EFLAGS register.

The assembly language provides the programmer with some instructions to set the value of the flags (the condition indicators) so that the programmer can influence the operation mode of the instructions which exploits these flags as desired.

CLC	CF=0	CF
CMC	CF = ~CF	CF
STC	CF=1	CF
CLD	DF=0	DF
STD	DF=1	DF

CLI, STI – they are used on the Interrupt Flag. They have effect only on 16 bits programming, on 32 bits the OS blocking the programmer's access to this flag.

4.1.2. **Type conversion instructions** (destructive)

CBW	converts the byte from AL to the word in AX (sign extension)	-
CWD	converts the word from AX to the doubleword in DX:AX (sign extension)	-
CWDE	converts the word from AX to the doubleword in EAX (sign extension)	-
CDQ	converts the doubleword from EAX to the quadword in EDX:EAX (sign extension)	
MOVZX d, s	loads in d (REGISTER !), which must be of size larger than s (reg/mem), the UNSIGNED contents of s (zero extension)	-
MOVSX d, s	load in d (REGISTER !), which must be of size larger than s (reg/mem), the SIGNED contents of s (sign extension) http://www.c-jump.com/CIS77/ASM/DataTypes/T77_0270_sext_example_movsx.htm !!!!!!!!!!!	-

CBW converts the signed byte from AL into the signed word AX (extends the sign bit of the byte from AL into the whole AH, thus destroying the previous content of AH). For example,

```
mov al, -1      ; AL=0FFh
cbw             ;extends the byte value -1 from AL to the word value -1 in AX (0FFFFh).
```

Similarly, for the signed conversion word - doubleword, the **CWD** instruction extends the signed word from AX into the signed doubleword in DX:AX. Example:

```
mov ax,-10000       ; AX = 0D8F0h
cwd         ;obtains the value  -10000  in  DX:AX (DX = 0FFFFh ; AX = 0D8F0h)
cwde        ; obtains the value  -10000  in  EAX     (EAX = 0FFFFD8F0h)
```

The **unsigned** conversion is done by „zerorizing" the higher byte or word of the initial value (for example, by `mov ah,0` or `mov dx,0` – a similar effect like applying the `MOVZX` instruction)

Why CWD coexists with CWDE ? The CWD instruction must remain for backwards compatibility reasons, but also to assure the proper functioning of the (I)MUL and (I)DIV instructions.

4.1.3. The impact of the little-endian representation on accessing data (pag.119 – 122 – coursebook)

If the programmer uses data consistent with the size of representation established at definition time (for example accessing bytes as bytes and not as bytes sequences interpretted as words or doublewords, accesing words as words and not as bytes pairs, accessing doubewords as doublewords and not as sequences of bytes or words) then the assembly language instructions will automatically take into account the details of representation (they will manage automatically the little-endian memory layout). If so, the programmer must NOT provide himself any source code measures for assuring the correctness of data management. Example:

```
a   db   'd', -25, 120
b   dw     -15642, 2ba5h
c    dd   12345678h
…
mov  al, [a]      ;loads in  AL the  ASCII code of  'd'
mov  bx, [b]     ;loads in BX the value -15642; the order of bytes in BX will be reversed compared to the
                  memory representation of b, because only the memory representation uses little-endian! At
                  register level data is stored according to the usual structural representation (equiv.to a big
                  endian representation).

mov edx, [c]     ;loads in  EDX the value of the doubleword 12345678h
```

If we need accessing or interpreting data in a different form than that of definition then we must use explicit type conversions. In such a case, the programmer must assume the whole responsability of correctly accessing and interpreting data. In such cases the programmer must be aware of the little-endian representation details (the particular memory layout corresponding to that variable/memory area) and use proper and consistent accesing mechanisms **Ex pag.120-122.**

```
segment data
    a dw 1234h          ;because of the little-endian representation, in memorie the bytes have the
                        following placement:
    b dd 11223344h      ;34h   12h   44h   33h   22h   11h
                ; address    a     a+1    b     b+1    b+2    b+3

    c db -1

segment code

    mov al, byte [a+1] ;accessing a as a byte, calculating the address a+1, selecting the byte from the address
                        a+1 (the byte with the value of 12h) and transfer it in the AL register

    mov dx, word [b+2]      ;dx:=1122h

    mov dx, word [a+4]      ;dx:=1122h  because b+2 = a+4, these pointer type expressions compute the same
                            address, specifically the address of the byte 22h.

    mov dx, [a+4]           ;this instruction is equivalent to the previous one, specifying the conversion operator
                            WORD not being required.

    mov bx,  [b]            ;bx:=3344h

    mov bx,  [a+2]          ;bx:=3344h, because the following addresses are equal: b = a+2.

    mov ecx, dword [a]      ;ecx:=33441234h, because the doubleword that starts at the address of a is
                            composed of the following bytes: 34h  12h  44h  33h, which (because of the little-
                            endian representation) form the following doubleword: 33441234h.

    mov ebx, [b]                ; ebx := 11223344h

    mov ax, word [a+1]          ; ax := 4412h
```

```
mov eax, dword [a+1]        ; eax := 22334412h

mov dx, [c-2]               ; DX := 1122h because c-2 = b+2 = a+4

mov bh, [b]                 ;bh := 44h

mov ch, [b-1]               ;ch := 12h

mov cx, [b+3]               ;CX := 0FF11h
```

4.2. OPERATIONS.

4.2.1. Arithmetic operations

Operands are represented in complementary code (see 1.5.2.). The microprocessor performs additions and substractions "seeing" only bits configurations, NOT signed or unsigned numbers. The rules of binary adding or substracting two numbers do not impose previously considering the operands as signed or unsigned, because independently of interpretation, additions and subtractions works the same way. So, at the level of these operations, the signed or unsigned interpretation depends on a further context and is left to the programmer.

The addition and the subtraction are evaluated in the same way (adding or subtracting the binary configurations) not taking into account the sign (interpretation) of these configurations! This does not apply to multiplication and division. When using these operations we need to know beforehand if the operands will be interpreted as signed or unsigned.

For example, if A and B are bytes:

A = 9Ch = 10011100b (= 156 in the unsigned interpretation and -100 in the signed interpretation)
B = 4Ah = 01001010b (= 74 , both in signed and unsigned interpretation)

The microprocessor performs the addition C = A + B obtaining

C = E6h = 11100110b (= 230 in the unsigned interpretation and -26 in the signed one)

We though notice that the simple addition of the bits configuration (without taking into account a certain interpretation at the moment of addition) assures the result correctness, both in signed and unsigned interpretation.

ARITHMETIC INSTRUCTIONS – page 123 (coursebook)
4.2.1.3. Examples – page 129-130 (coursebook)

4.2.2. Logical bitwise operations (AND, OR, XOR and NOT instructions).
AND is recommended for isolating a certain bit or for forcing the value of some bits to 0.
OR is suitable for forcing certain bits to 1.
XOR is suitable for complementing the value of some bits.

4.2.3. Shifts and rotates.
Bit shifting instructions can be classified in the following way:

- Logic shifting instructions
  - left - **SHL**
  - right - **SHR**
- Arithmetic shifting instructions
  - left - **SAL**
  - -right - **SAR**

Bit rotating instructions can be classifed in the following way:

- Rotating instructions without carry
  - left - **ROL**
  - right - **ROR**
- Rotating instructions with carry
  - left - **RCL**
  - right - **RCR**

For giving a suggestive definition for shifts and rotates let's consider as an initial configuration one byte having the value X = abcdefgh, where a-h are binary digits, h is the least significant bit, bit 0, a is the most significant one, bit 7, and k is the actual value from CF (CF=k).

We then have:

SHL X,1 ;has the effect X = bcdefgh0 and CF = a
SHR X,1 ;has the effect X = 0abcdefg and CF = h
SAL X,1 ; identically to SHL
SAR X,1 ;has the effect X = aabcdefg and CF = h
ROL X,1 ;has the effect X = bcdefgha and CF = a
ROR X,1 ;has the effect X = habcdefg and CF = h
RCL X,1 ;has the effect X = bcdefghk and CF = a
RCR X,1 ;has the effect X = kabcdefg and CF = h

INSTRUCTIONS FOR BITS SHIFTING AND ROTATING – pag.134 (coursebook)

4.3. BRANCHING, JUMPS, LOOPS

4.3.1. Unconditional jump

Three instructions fall into this cathegory: JMP (equiv. to GOTO from other languages), CALL (a procedure call means a control transfer from the call's point to the first instruction from the called routine) and RET (control transfer back to the first executable instruction after the CALL).

JMP *operand*	Unconditional jump to the address specified by operand	-
CALL *operand*	Transfers control to the procedure identified by operand	-
RET [*n*]	Transfers control to the first instruction after CALL	-

4.3.1.1. JMP instruction

Syntax: **JMP** *operand*

where *operand* is a label, register or a memory address containing an address. Its effect is the unconditional control transfer to the instruction following the label, to the address contained in the register or to the address specified by the memory variable respectively. For example, after running the sequence

```
            mov  ax,1
            jmp  AdunaDoi
AdunaUnu:   inc  ax
            jmp  urmare
AdunaDoi:   add  ax,2
urmare:     .    .    .
```

AX will hold the value 3. **inc** şi **jmp** betwwen *AdunaUnu* and *AdunaDoi* will not be executed, unless a jump to *AdunaUnu* will be done from another step of the program.

As mentioned above, the jump may be made to an address stored in a register or in a memory variable. Examples:

(1)
```
mov  eax, etich

jmp  eax  ;register operand
          ; jmp [eax] ?
etich:    . . .
```

(2)
```
segment data
      Salt  DD  Dest ;Salt := offset Dest
      .     .   .
segment code
      .     .   .
      jmp  [Salt]      ;NEAR jump
.     ;memory variable operand
Dest :    . . .
```

If in case (1) we wish to replace the register destination operand with a memory variable destination operand, a possible solution is:

```
            b   resd  1
(1')            . . .
            mov  [b], DWORD etich ; b := offset etich
            jmp [b]          ; NEAR jump – memory variable operand
                             ;  JMP DWORD PTR DS:[offset_b]
```

Exemplul 4.3.1.2. – pag.142-143 (coursebook) – control transfer to a label. Analysis and comparison.

4.3.2. Conditional jump instructions

4.3.2.1. Comparisions between operands

CMP *d,s*	compares the operands values (does not modify them - fictious subtraction *d* - *s*)	OF,SF,ZF,AF,PF and CF
TEST *d,s*	non-destructive *d* **AND** *s*	OF = 0, CF = 0 SF,ZF,PF - modified, AF - undefined

Conditional jump instructions are usually used combined with comparision instructions. Thus, the semantics of jump instructions follows the semantics of a comparision instruction. Besides the equality test performed by a CMP instruction we need frequently to determine the exact order relationship between 2 values. For example we have to answer to: nr. 11111111b (= FFh = 255 = -1) is bigger than 00000000b(= 0h = 0)? The answer is .... IT DEPENDS !!!! This answer can be either YES or NO ! If we perform an unsigned comparision, then the first one is 255 and is obvious bigger than 0. If the 2 values are compared in the signed interpretation, then the first is -1 and is less than 0.

The CMP instruction does not make any difference between the two above cases, because as we mentioned in 4.2.1.1 addition and subtraction are performed always in the same way (adding or subtracting binary configurations) no matter their interpretations (signed or unsigned). So it's not the matter to interpret the operands of CMP as being signed or unsigned, but to further interpret the RESULT of the subtraction ! Conditional jump instructions are responsible to do that (Section 4.4.2.2).

4.3.2.2. Conditional jumps

Table 4.1. (pag.146 – coursebook) presents the conditional jump instructions together with their semantics and according to which flags values the jumps are made. For all the conditional jump instructions the general syntax is

<conditional_jump_instruction> label

The effect of the conditional jump instructions is expressed as ***"jump if operand1 <<relationship>> operand2"*** (where on the two operands a previously CMP or SUB instruction is supposed to have been applied) or relative to the actual value set for a certain flag. As easy can be noticed based on the conditions that must be verified, instructions on the same line in the table have similar effect.

When two signed numbers are compared, **"less than"** and **"greater than"** terms are used and when two unsigned numbers are compared *"below"* and *"above"* terms are respectively used.

4.3.2.3. Examples along with comments..... pag.148-162 (coursebook).

- comparative analysis and discussion of the concepts of: signed vs. unsigned representations, overflow, actual effects of conditional jump instructions, specific flags (CF, OF, SF, ZF)

4.3.3. Repetitive instructions (coursebook pag.162 – 164)

These are: **LOOP, LOOPE, LOOPNE** and **JECXZ**. Their syntax is

<instruction> label

LOOP performs the repetitive run of the instructions block starting at *label*, as long as the value of CX register is different from 0. **It first performs decrementation of ECX, then the test and eventually the jump.** The jump is "short" (max. 127 bytes – so pay attention to the "distance" between LOOP and the label!). – PAY ATTENTION !! CHECK IT !!! (short jump is out of range!)

When the end of loop conditions are more complex **LOOPE** and **LOOPNE** may be used. **LOOPE** (***LOOP*** *while* ***E****qual*) differ from LOOP by ending condition, loop is ended either if ECX=0, either if ZF=0. In the case of **LOOPNE** (***LOOP*** *while* ***N****ot* ***E****qual*) the loop will end either if ECX=0, either if ZF=1. Even if the loop exit shall

be based on value of ZF, CX decrementation is done anyway. **LOOPE** is also known as **LOOPZ** and **LOOPNE** is also known as **LOOPNZ**. These are usually used preceeded by a CMP or SUB instruction.

JECXZ (***Jump if ECX is Zero***) performs the jump to the operand label only if ECX=0, being useful when we want to test the value in ECX before entering in a loop. In the following example, JECXZ instruction is used to avoid entering the loop if ECX=0:

```
            .       .       .
        jecxz MaiDeparte        ;if ECX=0 a jump over the loop is made
    Bucla:
        Mov   BYTE [esi],0      ;initializing the current byte
        inc   si                ;passing to the next byte
        loop  Bucla             ;resume the loop or ending it
    MaiDeparte:     .       .       .
```

If a loop is entered with ECX=0, ECX is first decremented, obtaining the value 0FFFF FFFFh (= -1, so a value different from 0), the loop being resumed until 0 in ECX will be reached, namely **2^32** = 4.294.967.296 more times !

It's important to say here that <u>none of the presented repetitive instructions affects the flags</u>.

```
                            dec  ecx
    loop Bucla    and       jnz  Bucla
```

although semantic equivalent, they do not have the same effect, because DEC modifies OF, ZF, SF and PF, while LOOP doesn't affect any flag.

4.3.4. <u>CALL and RET instructions</u>

A procedure call is done by using the **CALL** instruction, it can be a *direct* or an *indirect* call. The direct call has the syntax:

CALL *operand*

Similar to JMP, **CALL** transfers the control to the address specified by the operand. In addition to JMP, before performing the jump, CALL saves to the stack the address of the instruction following CALL (the returning address). In other words, we have the equivalence:

```
CALL operand              push dword A
A: . . .          ⇔       jmp  operand
```

The end of the called sequence is marked by a **RET** instruction. This pops from the stack the returning address stored there by CALL, transferring the control to the instruction from this address. The RET syntax is:

RET [*n*]

where *n* is an optional parameter. It indicates freeing from the stack *n* bytes below the returning address.

RET instruction can be illustrated by this equivalence:

```
                          B   resd 1
RET n                     .     .     .
(near return)     ⇔       pop dword [B]
                          add  esp,n
                          jmp  [B]
```

Usually, as it is natural, CALL and RET are used in the following context:

```
        procedure_label:
.       .       .
        ret  n
.       .       .
```

```
CALL  procedure_label
```

CALL may also take the transfer address from a register or from a memory variable. Such a call is identified as an *indirect call*. Example:

```
call  ebx       ;address taken from a register
call  [vptr]    ;address taken from a memory variable
```

Concluding, the destination operand of a CALL instruction may be:

- a procedure name
- the name of a register containing an address
- a memory address

Conversions classification

a). Destructive – cbw, cwd, cwde, cdq, movzx, movsx, mov ah,0; mov dx,0; mov edx,0 ; INSTRUCTIONS !!!

Non-destructive – Type operators: byte, word, dword, qword

b). Signed - cbw, cwd, cwde, cdq, movsx

Unsigned – movzx, mov ah,0; mov dx,0; mov edx,0, byte, word, dword, qword

c). by enlargement – all the destructive ones ! + word, dword, qword

by narrowing – byte, word, dword

Implicit vs explicit conversions

e = a+b+c e,b = float , a,c – integer (integer to float – implicit conversions)

i=c //only in C NOT in C++

float → integer ? How can you do this conversion ? Float to integer – NOT by conversions but by applying predefined functions of the language (floor, ceil, trunc etc). Alternatively you must ASSUME as a programmer THE RESPONSIBILITY OF CUTTING OUT INFORMATION by using predefined special functions (floor, ceil, round, trunc)

Flat memory model or linear **memory model** refers to a **memory** addressing paradigm in which "**memory** appears to the program as a single contiguous address space." The CPU can directly (and linearly) address all of the available **memory** locations without having to resort to any sort of **memory** segmentation or paging schemes.

Linear Memory Model

A **linear memory model**, also known as the **flat memory model** refers to a memory addressing technique in which memory is organized in a single contiguous address space. This means that the processing unit can access these memory locations directly as well as linearly.

To better understand a linear memory model, we should understand two basic components: **address** and **data**. Address is a hexadecimal number which is used to denote the exact place of a memory chunk. Data is the value stored in that memory. In a linear memory model, the entire memory space is linear, sequential and contiguous. The address ranges from 0 to *MaxByte -1*, where *MaxBytes* is the maximum limit of memory. Each program uses one 32-bit linear memory space, that means 2^32 = 4GB of memory can be addressed using this memory model. The operating system then translates these linear addresses to physical addresses using paging schemes.