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’.

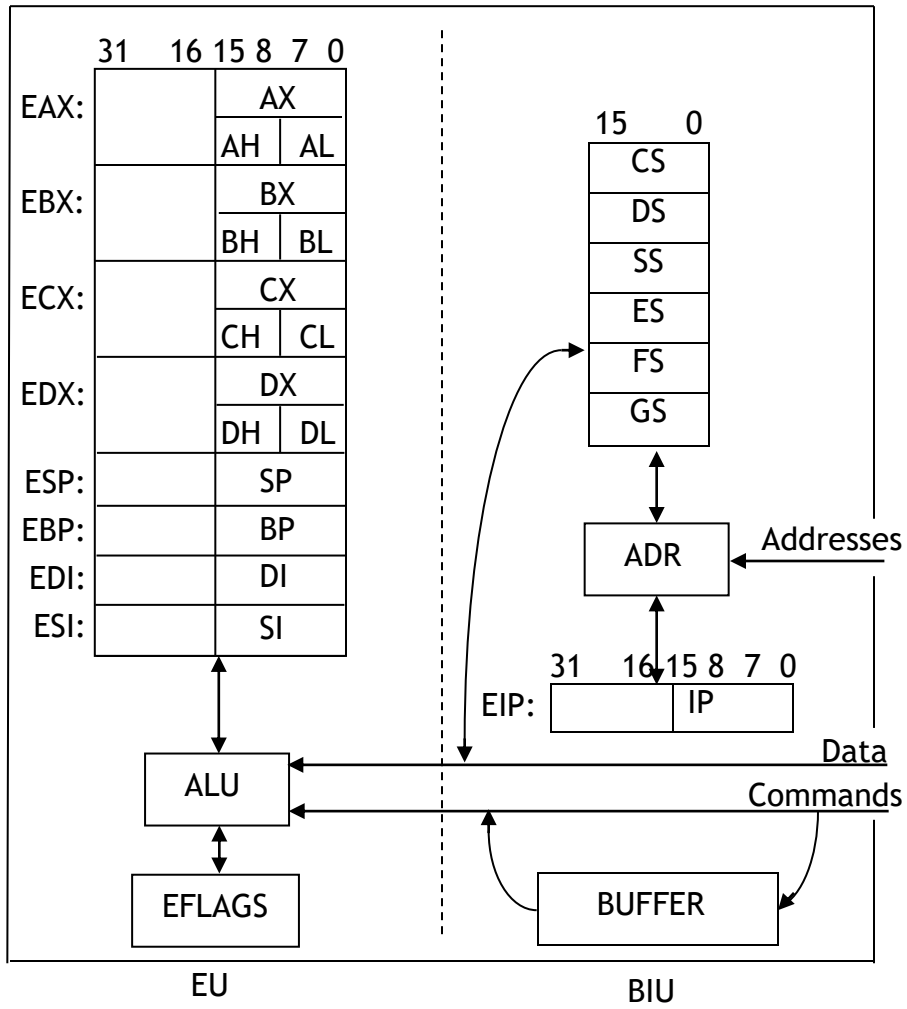
In the case of a correct division CF and OF are undefined. If we have a division overflow, the program crashes, the execution stops and of course it doesn’t matter which are the values from CF and OF…

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:

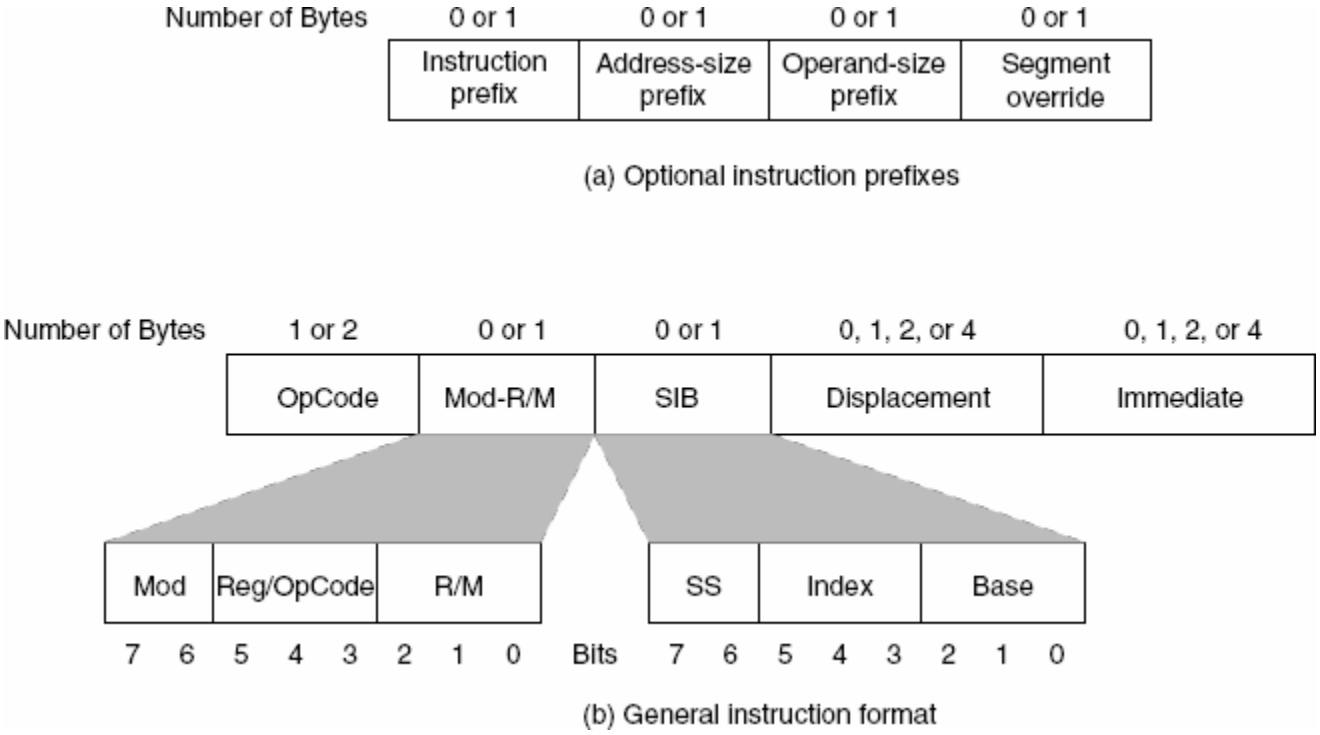
instruction_name *destination**,** source*

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

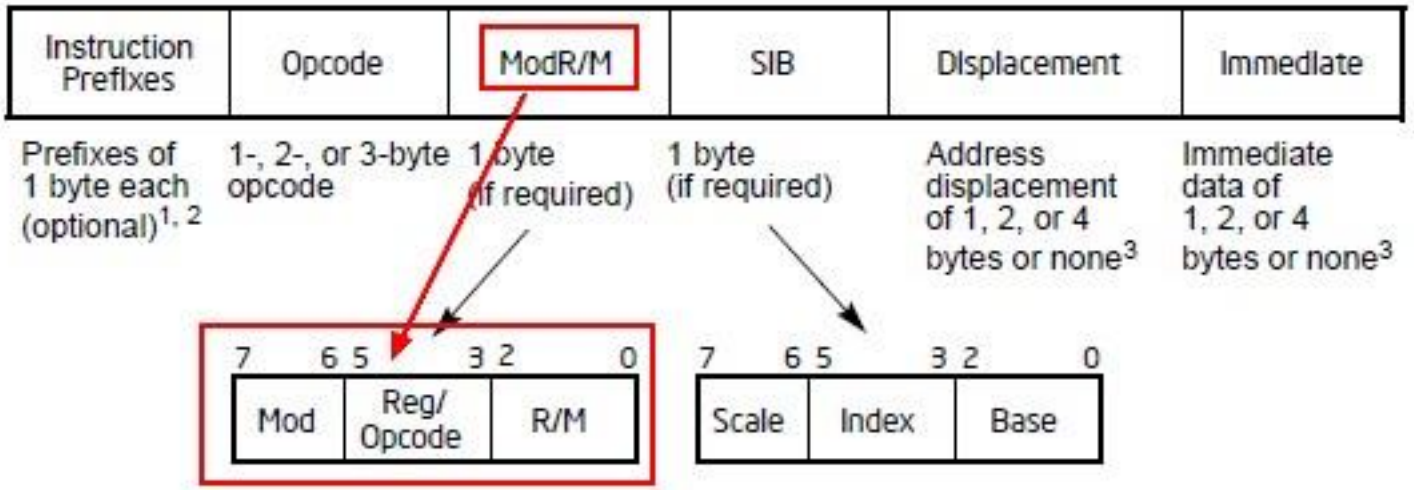
[prefixes] + code + [ModeR/M] + [SIB] + [displacement] + [immediate]

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)
```

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 EDI, ESI, EBP, ESP, EBX, EDX, ECX and EAX in the stack	-
POPA / POPAD	Pops EAX, ECX, EDX, EBX, ESP, EBP, ESI and EDI 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