

Set CF=1 ; No way !

Mov ax, [v] ; a NASM specific – using ‘[]’ = DEREFERENCING operator = ‘*’ from C

A[7] = *(A+7) ; [] = addition pointer arithmetic operator

Mov eax, [8:1000h];

- Using assembly language directives

BITS 16

.....

A[7] = *(A+7); “+ = []”

Mov eax, [a1] ;

Mov eax, [8:1000h];

Segment code

Start:

Jmp REAL_START

V db 17

V1 dw 54321

REAL_START: Mov ax, [v]

Add ebx, eax

Mul [v1]

.....

Flags instructions do NOT have explicit operands !

In 16 bits programming the address of a memory location is handled explicitly by the programming with total freedom (segment + offset). In 32 bits programming only offsets are allowed to be handled by the programmer.

Limit = size of a segment

At the level of a source code, a programmer can use only ADDRESS SPECIFICATIONS, NEVER final effective addresses ! These may only be handled EXCLUSIVELY by the ADR component from BIU.

In contrast with the segment starting address, on which we don't have any control as programmers in 32 bits programming, offsets must be handled by us !!

For running a program mandatory are the CODE segment and the STACK segment. In writing a source code the only mandatory segment is the CODE segment ! (because stack segment is automatically generated)

The binary form (machine code) of a .exe program does NOT contain ONLY the equivalent of the instructions to be run, but ALSO the WHOLE data segment generated correspondingly by the assembler/compiler !!!

In other words, the DATA segment is also saved together with the code segment on the disk as a part of the **.exe program !!**

Operators ! and ~ usage (page illustrating the way in which we TOGETHER obtained the results !)

In C - !0 = 1 (0 = false, anything different from 0 = TRUE, but a predefined function will set TRUE =1)

In ASM - !0 = ? It the same mechanism as in C

- ! Logic Negation: !X = 0 when X ≠ 0, otherwise = 1 (X-bit)
- ~ 1's Complement: mov al, ~0 => mov AL, 0ffh

a is defined... RESB

Mov eax, ![a] - Expression syntax error ! [a] – is not a SCALAR value...

Mov eax, [!a] - !a is NOT a SCALAR – is a POINTER !!! a is an offset, it is determinable at assembly time, but IT IS NOT A SCALAR !!!!

Mov eax, !a - !a is NOT a SCALAR – syntax error

Mov eax, !(a+7) - !(a+7) is NOT a SCALAR – syntax error

Mov eax, !(b-a) – OK !!! because the difference of 2 pointers IS A SCALAR !!! (usually you will obtain a zero !)

Mov eax, ![a+7] - Expression syntax error !

Mov eax, !7 - EAX = 0

Mov eax, !0 – EAX = 1

Mov eax, ~7 ; 7 = 00 00 00 07h = ... 00000111b, so ~7 = 0 ff ff ff f8h

Mov eax, !ebx ; syntax error !

aa equ 2

mov ah, !aa ; AH = 0

Mov AH, 17^(~17) ; AH = 0 ffh = -1

Mov ax, value ^ (~value); eax= 0 ff ffh = -1

Mov eax, value ^ (~value); eax= 0 ff ff ff ffh = -1

(in the general case we can say that we obtain -1)

Operators ! and ~ usage (examples prepared for me in advance – with previously completed answers...)

In C - !0 = 1 (0 = false, anything different from 0 = TRUE, but a predefined function will set TRUE =1)

In ASM - !0 = same as in C, so

! Logic Negation: !X = 0 when X ≠ 0, otherwise = 1 (X-bit)

~ 1's Complement: mov al, ~0 => mov AL, 0ffh (bitwise operator !)

(because a 0 in asm is a binary ZERO represented on 8, 16, 32 or 64 bits the logical BITWISE negation – 1's complement - will issue a binary 8 of 1's, 16 of 1's, 32 of 1's or 64 of 1's...)

Mov eax, ![a] - because [a] is not something computable/determinable at assembly time, this instr. will issue a syntax error ! – (expression syntax error)

Mov eax, [!a] - ! can only be applied to SCALAR values !!

Mov eax, !a - ! can only be applied to SCALAR values !!

Mov eax, !(a+7) - ! can only be applied to SCALAR values

Mov eax, !(b-a) – ok !

Mov eax, ![a+7] - expression syntax error

Mov eax, !7 - EAX = 0

Mov eax, !0 – EAX = 1

Mov eax, ~7 ; 7 = 00000111b , so ~7 = 11111000b = 0f8h,
EAX=0 ff ff ff f8h

Mov eax, !ebx ; syntax error !

aa equ 2

mov ah, !aa ; AH=0

Mov AH, 17^(~17) ; AH = 11111111b = 0ffh = -1

Mov ax, value ^ ~value ax=11111111 11111111 = 0ffffh = 1

Operands data type (discussed with you...)

Push v – stack ← offset v (32 bits)

Push [v] - Syntax error ! – Operation size not specified !

Push byte [v] – syntax error !

Push word [v] – ok !

Push dword [v] – ok !

Push qword [v] – syntax error !

Mov eax,[v] - ok ! EAX=dword ptr [v] = mov eax, dword ptr [DS:v]

Push [eax] - Syntax error ! – Operation size not specified !

Push word/dword [eax] ; ok !

...? – is it a correct, valid and accessible address [DS:EAX] ?? Possible run-time error “Memory violation”... but this is something decided at run-time based on the value from EAX...

Push 15 – PUSH DWORD 15 – ok !

Pop [v] - Syntax error ! – Operation size not specified !

Pop word/dword [v] – ok !

Pop v ; v is an address !! BUT... it is a CONSTANT address... You can not change a CONSTANT address !! It would be exactly like attempting to write 2=3 !!!... v is NOT a L-value !!

Pop [eax] ; Syntax error ! – Operation size not specified !

Pop word/dword [eax] ; ok !

Pop 15 - 15 is NOT a L-value !! (15 = 3 !!!)

Pop [15] - Syntax error ! – Operation size not specified !

Pop word/dword [15] = [DS:15] – most probably will issue a run –time error ...

Mov [v],0 - Syntax error ! – Operation size not specified !

Mov byte [v], 0 ; OK !!

Mov [v], byte 0 ; OK !!

Div [v] – syntax error

Div byte/word/dword [v] – OK !!!

Imul [v+2] – syntax error

Imul byte/word/dword [v+2] – OK !!!

a dd...

b dw...

Mov a,b – ...error

Mov [a], b – syntax error – Op.size not specified !

Mov [a], word b - ok

Mov dword [a], b – ok

Mov byte [a], b – syntax error ! (similar to mov ah, b type of error...)

Mov qword [a], b ; syntax error !

Mov a,[b] – a NOT a L-value !!

Mov [a], [b] – NO asm instruction can have both operands from memory !!

Mov word [a], [b] - NO asm instruction can have both operands from memory !!

Mul v – MUL reg/mem – syntax error because it doesn't follow the syntax of MUL !

Mul [v] – Op.size not specified !

Mul byte/word/dword [v] - ok

Mul eax ; ok !

Mul [eax] ; Op.size not specified !

Mul byte/word/dword [eax] - ok

MUL 15 ; MUL reg/mem – syntax error because it doesn't follow the syntax of MUL !

Operands data type (examples + answers prepared by me in advance...)

Push v – stack ← offset v

Push [v] - Syntax error ! – Operation size not specified !! (a PUSH on a 32 bits programming stack accepts both 16 and 32 bits values as stack operands) ;

Push dword [v] - ok

Push word [v] - ok

Mov eax,[v] - ok ; EAX = dword ptr [v], in Olly dbg “mov eax, dword ptr [DS:v]”

Push [eax] - Syntax error ! – Operation size not specified !!

Push word/dword [eax]

Push 15 – PUSH DWORD 15

Pop [v] - Syntax error ! – Operation size not specified !! (a POP from the stack accepts both 16 and 32 bits values as stack operands) ;

Pop word/dword [v];

Pop v ; Invalid combination of opcode and operands , because v is an offset (R-value) and a R-value CANNOT be the destination of an assignment ! (like attempting 2=3)

Pop [eax] – Op size not specified !

Pop 15 - Invalid combination of opcode and operands , because v is an offset (R-value) and a R-value CANNOT be the destination of an assignment ! (like attempting 2=3)

Mov [v],0 - op size not spec.

Mov byte [v],0 ; ok !!!

Mov [v], byte 0 ; ok !!!!

Div [v] – Op. size not spec. – 3 possibilities ...

Imul [v+2] - Op. size not spec

a d?...

b d?...

Mov a,b – Invalid combination of opcode and operands , because a is an offset (R-value) and a R-value CANNOT be the destination of an assignment ! (like attempting 2=3)

Mov [a], b – Op. size not spec.

Mov word [a], b or mov [a], word b - the lower word from the offset of b will be transferred into the first 2 bytes starting at offset a !

Mov dword [a], b or... - the offset of b will be transferred into the first 4 bytes starting at offset a !

Mov byte [a], b or.... – SYNTAX ERROR ! because AN OFFSET is EITHER a 16 bits value or a 32 bits value, NEVER an 8 bit value !!!!

(the same effect as mov ah, v)

Mov a,[b] - Invalid combination of opcode and operands , because a is an offset (R-value) and a R-value CANNOT be the destination of an assignment ! (like attempting 2=3)

Mov [a], [b] - Invalid combination of opcode and operands, BECAUSE asm doesn't allow both explicit operands to be from memory !!!

Mul v – Invalid combination of opcode and operands, BECAUSE syntax is MUL reg/mem

Mul [v] – op size not spec.

Mul eax ; ok !

Mul [eax] ; op size not spec.

MUL 15 ; Invalid combination of opcode and operands, BECAUSE syntax is MUL reg/mem

Pop byte [v] - Invalid combination of opcode and operands

Pop qword [v] – Instruction not supported in 32 bit mode !

Mov eax,0

Idiv eax; run-time error ! Zero divide...

Eroare de asamblare / assembly error = syntax error !

The need for XLAT emerged from situations like this:

How to generate the STRING of digits corresponding to a numeric value ?

fa26h in AX → 'fa26'

3 + '0' = Ascii code of CHARACTER '3'

I+'0' = ascii code of whatever I is... 0..9...

If the value is between 10..16 → i+'a'-10

November 4th

- main task of an assembler = generating the corresponding bytes
- at any given moment ONLY ONE segment of every type may be ACTIVE
- in 16 bits programming the segment registers CS, DS, SS, ES contained the STARTING ADDRESSES of the currently active segments
- in 32 bits programming the segment registers CS, DS, SS, ES contain the values of the SELECTORS of the currently active segments
- at any given moment during run time the CS:EIP combination of registers expresses /contain the address of the currently executed instruction
- these values are handled exclusively by BIU
- an assembly language instruction doesn't support/allow both of its explicit operands to be from the RAM memory
- that is because BIU may "bring" only one memory operand at a time (for 2 memory operands we would need 2 BIU, 2 segment registers sets etc)

$$\text{offset_address} = [\text{base}] + [\text{index} \times \text{scale}] + [\text{constant}] \\ (\text{SIB}) \quad (\text{displacement} + \text{immediate})$$

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

- the first 2 elements from the offset address computation formula (base and index*scale) are expressed by the SIB byte from the internal format formula
- the third element: the constant, if present, is expressed by the displacement and/or immediate fields
- SIB and displacement participate ONLY to the offset computation of the memory operand, if there is any
 - "immediate" field may be also involved in offset computation, but it can also appear INDEPENDENTLY from a memory operand, expressing in such a case an immediate operand (mov eax, 7 ; 7 is "immediate" and no memory operand present in the instruction)
- if Modr/m tells us that we have a register operand the next 3 fields from the internal format formula are absent (because if the operand is a register it can NOT be in the same time also a memory operand or an immediate value)
- if Modr/m tells us that we have a memory operand => SIB byte is mandatory, followed MAYBE by displacement and/or immediate
 - the field "immediate" may participate to the offset computation of a memory operand (providing the "constant" field from the offset computation formula) or may appear only by itself expressing the immediate value of an operand (example: mov ebx, 12345678h)

- the displacement field expresses the direct addressing memory access

- immediate field = numerical constants

- in the instructions used in our programs we will use almost exclusively only offsets, these being implicitly prefixed by one of the segment registers CS, DS, SS or ES. (ex. in debugger image - push variabila -> DS:[40100...])

- offset = an address

- direct addressing means direct access to the memory operand based on its offset, without needing / specifying any register in the offset specification formula (so no base or index !)

- if registers appear in the offset computation formula (base or index) => indirect addressing

CS:EIP – The FAR (complete, full) address of the currently executing instruction

EIP – automatically incremented by the current execution

CS – contains the segment selector of the currently active segment and it can be changed only if the execution will switch to another segment

Mov cs, [var] - forbidden

Mov eip, eax - forbidden

Jmp FAR somewhere ; CS and EIP will be both modified !

Jmp start1 ; NEAR jmp – only the offset will be modified, so EIP !

For an instruction there are 3 ways to express a required operand:
(*operands specification modes*)

- *register mode*, if the required operand is a register; `mov ax, bx`
- *immediate mode*, when we use directly the operand's value (not its address and neither a register holding it); `mov eax,2`
- *memory addressing mode*, if the operand is located somewhere in memory. In this case, its offset is computed using the following formula:

$$\text{offset_address} = [\text{base_reg}] + [\text{index_reg} \times \text{scale}] + [\text{constant}]$$

(SIB) (displacement+immediate)

constant = constant offset (displacement = direct addressing) or/and immediate value

`mov edx, [var-5]
[EBX+ECX*2 + v -7] – ok
 SIB depl. const.`

`v db 19
add ebx, [EBX+ECX*2 + v + (-7)] ; - ok`

`sub ecx, [EBX+ECX*2 - v-7] – syntax error !! invalid effective address – impossible segment base multiplier`

`mov [EBX+ECX*2 + a+b-7], bx - not allowed ! syntax error ! because of “a+b”
invalid effective address – impossible segment base multiplier`

`add [EBX+ECX*2 + a-b-7], ecx – ok !
 SIB const.`

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 NOT direct addressing mode is named INDIRECT addressing (based and/or indexed). So, an indirect addressing is that for which we have at least one register specified in square brackets ([]).

In the addressing system operations with pointers are performed. Which are the ARITHMETIC operations allowed with pointers in COMPUTER SCIENCE ?...

Answer: Any operation that makes sense... meaning any operation that expresses as a result a correct location in memory useful as an information for the programmer.

- adding a constant value to a pointer $a[7] = *(a+7)$ – useful for going into memory forth and back relative to a starting address
- subtracting $a[-4]$, $a(-4)$...
- multiplying 2 pointers ? – No way ... no practical usage !
- dividing 2 pointers ? - No way ... no practical usage !
- adding/subtracting 2 pointers ?
- ADDING 2 pointers doesn't make sense !! – it is not allowed
- SUBTRACTING 2 pointers !! does makes sense... $q-p = \text{nr. Of elements (in C)} = \text{nr. Of bytes between these 2 addresses in assembly}$ (this can be very useful for determine the length of a memory area).

Pointer arithmetic...? Contains ONLY 3 operations that are possible:

Adress – adress = ok ($q-p = \text{subtraction of 2 pointers} = \text{sizeof(array)}$)

Address - offset = address – address

Adress + numerical constant (identification of an element by indexing – $a[7]$) , $q+9$

Adress - numerical constant - $a[-4]$, $p-7$

- subtraction of 2 pointers = SCALAR VALUE (constant)
- adding a constant to a pointer → a POINTER !!
- subtracting a constant from a pointer → a POINTER !!

POINTER ARITHMETIC OPERATIONS - *Pointer arithmetic* represents the set of arithmetic operations allowed to be performed with pointers, this meaning using arithmetic expressions which have addresses as operands.

- subtraction of 2 addresses – ok, is allowed, $q-p$ = the number of bytes between those 2 addresses... !!!!
- adding a CONSTANT (INTEGER) to an address – $a[7] = *(a+7)$
- subtracting a CONSTANT (INTEGER) from an address – $a[-4] = *(a-4)$ - useful for referring array elements

$p+q = ????$ (allowed in NASM...sometimes...) – but it doesn't mean in the end as we shall see a pointer addition

How do I make the difference between the address of a variable and its contents ?

Var – invoked like that it is an address (offset) ; [var] – is its contents

[] = the dereferencing operator !! (like $*p$ in C)

Assignment: **i:=i+1**

Dereferencing is usually implicit depending on the context !

BLISS language **i←*i+1**

LHS – is always an address (left-value)

RHS - is a contents (is part of an expression)

Symbol := expression (usually in 99% of the cases)

Address_expression := value_expression !!! (the most general)

Address of I \leftarrow value of I + 1

LHS (Left Hand Side of an assignment = L-value = address) := RHS
(Right Hand Side of an assignment = R-Value = CONTENTS !!)

Symbol := expression_value (99% of the cases...)

Address_computation_Expression := expression_value

In C++ f(a, b, 2) = x+y+z

Int& f(i,...) {....return v[i];} – f is a function that will return an L-value !!
f(88,...) = 79; it means that v[88]=79 !!!

Int& j = i; // j becomes ALIAS for i

(a+2?b:c) = x+y+z ; - correct

(a+2?1:c) = x+y+z; - syntax error !!! 1:=n !!???

Case studies

Mov op size dest, op SAME size b,b w,w dw,dw

Mov ax, ebx - syntax error ! “invalid combination of opcode and operands”

Mov ebx, ch - syntax error ! “invalid combination of opcode and operands”

Mov eax, ebx - eax \leftarrow the contents of EBX

Mov eax, [ebx] = mov eax, [ds:ebx] ; eax = the doubleword value from memory starting at the address DS:EBX

Mov ax, [ebx] = mov ax, [ds:ebx] ; ax = the word value from memory starting at the address DS:EBX

Mov edx, [eax+ebx] – EDX := the doubleword value from memory starting at the address [DS:EAX+EBX]

Mov edx, eax+ebx ; SYNTAX ERROR !! – see the diff. between the OPERATOR + and THE INSTRUCTION ADD !!!

Operators can *perform computations only* with *constant values* determinable at assembly time. The *single exception* to this rule is *the offset specification/computation formula*. (we have there the operator ‘+’ which handles registers contents !)

Mov edx, [ebx+eax] – EDX := the doubleword value from memory starting at the address [DS:EAX+EBX]

Mov edx, [esp+ecx] ; EDX := the doubleword value from memory starting at the address [SS:ESP+ECX] - ok ! ESP is always THE BASE !!

Mov edx, [ecx+esp]; - same effect as above ESP – BASE register ; ESP is always THE BASE !! It doesn’t matter the order in which you write it !

Mov edx, [esp+2*ecx] ; correct!; ESP – base register; ECX – index ; 2=scale; EDX ← the doubleword taken from memory address given by the [SS:ESP+2*ECX]

Mov edx, [ecx+2*esp] ; syntax error ! ESP can be only a base register

mov dh, [edx + ecx * 4 + 3] ; DH← from memory address DS:edx+ecx*4+3 ONE byte is taken and transferred into DH

mov dx, [edx + ecx * 4 + 3] ; DX← from memory address DS:edx+ecx*4+3 ONE word is taken and transferred into DX

mov eax, [eax*3] = mov eax, [eax+eax*2] – CORRECT !

mov eax, [ebx*9 + 12] = mov eax, [DS:ebx + ebx*8+12]

mov eax, [esp*5] – syntax error ! ESP cannot be an INDEX register !

mov ax, [a] ; constant is the ADDRESS of the variable a, NOT its contents !!!! So, that is why the operand [a] OBBEYS the offset specification formula !!!

Mov reg, [var] – in which of the operands specification modes does it belong ?

Mov eax, [a] - ??? what value has a ??

a = its address ! but when specifying [a] this means THE CONTENTS of a [] = dereferencing operator in assembly !!

We can access memory values in 2 ways:

- by means of variable names (`mov eax, [a]`)
- or by computing address values applying the offset spec. formula (`mov eax, [ebx + 2 * ecx-7]` in assembler or `var1=*(p-8)` in C)

Var d?

`Mov eax, var ; EAX ← offset (var) – which is ALWAYS a value on 32 bits !!!`
`Mov eax, [var] ; EAX ← 4 bytes from address DS:var – its CONTENTS !!!`

In TASM and MASM we DO NOT have the dereferencing operator and whenever we define a variable the DATA TYPE inferred by the data definition directive is associated strongly to that variable (db, dw, dd really means associating the corresponding data type with that symbol). This will result in

`Mov eax, var ; SYNTAX ERROR in TASM AND MASM !!! if var is NOT a dd`

If we need in TASM /MASM to transfer the address of an operand we must use the OFFSET operator:

`Mov eax, OFFSET var – transfers the offset of var into EAX in TASM/MASM`
`Mov eax, var – transfers the CONTENTS of var into EAX in TASM/MASM (no need of the dereferencing operator – dereferencing is implicit in TASM/MASM)`

In NASM it is a BIG difference !!!

`Mov ax, var ; IS ALLOWED with a WARNING ! (16 bits reloc of 32 bits value) only 16 bits will be taken (the inferior word from the offset) – it is allowed because of 16 bits addressing mode whih has to still be valid in 32 bits programming also`

`Mov ax, [var] ; AX ← 2 bytes from address DS:var`

`Mov ah, var ; syntax error ! (OBJ file can only handle 16- or 32 bits values) – no offsets on 8 bits are allowed !!!`

`Mov ah, [var] ; ok; AH ← 1 byte from address DS:var`

Var db 17, 18, 19, 29, 2ah, 0x2a, -3

Mov [var], eax; the contents of EAX will overwrite the first 4 bytes from var; [] = means the CONTENTS of var ([] = dereferencing operator)

A db 17

B db 19

C db 21

D db 23, 87, 9h

Mov eax, [A]; 4 bytes taken in order (17, 19, 21, 23) and transferring them in EAX

Mov eax, [B-1] = mov eax, [C-2] = mov eax, [D-3]

Mov eax, [D+ebx*2]

Mov ah, ebx ; - syntax error !

Mov ah, [ebx] ; - 1 byte from [DS:EBX] into AH

Mov ax, [ebx]; - 2 bytes from [DS:EBX] into AX

Mov eax, [ebx] ; - 4 bytes from [DS:EBX] into EAX

Offset_spec16 = [BX|BP] + [SI|DI] + [constant] – the offset specification formula on 16 bits !!!

Mov ah, [bx] ; AH:= 1 byte from DS:[BX]

Mov ax, [bx] ; AX:= 2 bytes from DS:[BX]

BX is A PART of EBX !!! it means that EBX = 0000000 BX

Mov eax,[bx] ; EAX:= 4 bytes from DS:[BX]

Mov ah, [bh] ; syntax error !!!! because BH isn't accepted as an indirect register specification (we can use either EBX in spec32 or BX in spec16) !

a db ...
b dw...
c dd....

The task of the data definition directives in NASM is NOT to specify an associated data type for the defined variables, but ONLY to generate the corresponding bytes to those memory areas designated by the variables accordingly to the chosen data definition directive and following the little-endian representation order.

So, a is NOT a byte – but only an offset and that is all... a symbol representing the start of a memory area WITHOUT HAVING AN ASSOCIATED DATA TYPE !

So, b is NOT a word – but only an offset and that is all... a symbol representing the start of a memory area WITHOUT HAVING AN ASSOCIATED DATA TYPE !

So, c is NOT a doubleword – but only an offset and that is all... a symbol representing the start of a memory area WITHOUT HAVING AN ASSOCIATED DATA TYPE !

- their task is only to allocate the required space AND to specify the way in which they have to be initialized !!!!

The **name of a variable** is **associated** in assembly language **with its offset relative to the segment** in which its **definition appears**. The **offsets of the variables** defined in a program are **always constant** values, determinable at assembly/compiling time.

Assembly language and C are **value oriented languages**, meaning that everything is reduced in the end to a numeric value, this is a low level feature.

In a **high-level programming language**, the **programmer can access the memory only** by using **variable names**, in contrast, in **assembly language**, the **memory is/can/must be accessed ONLY** by using the **offset computation formula** (*formula de la doua noaptea*) where **pointer arithmetic** is also used (pointer arithmetic is also used in C !).

mov ax, [ebx] – the source operand **doesn't have an associated data type** (it represents only a start of a memory area) and because of that, in the case of our MOV instruction the **destination operand is the one that decides the data type of the transfer (a word in this case)**, and the transfer will be made accordingly to the little endian representation.

$$\begin{array}{r}
 0110\ 1100+ \\
 1100\ 0100 \\
 1\ 0011\ 0000 = \\
 1\ \ \ 3\ \ \ 0\ h
 \end{array}
 \quad
 \begin{array}{l}
 6ch + c4h = 130h \\
 AF=1
 \end{array}$$

4 bits = 1 semioctet = 1 nibble = 1 hexadecimal digit

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. **CF signals overflow for the case of UNSIGNED interpretation.**

$$\begin{array}{r}
 1001\ 0011 + \quad 147+ \quad \quad \quad 93h + - 109 + \\
 \underline{0111\ 0011} \quad \underline{115} \quad \quad \quad \underline{73h} \quad \quad \quad \underline{115} \\
 \textcolor{red}{1} 0000\ 0110 \quad \quad 262 \quad \quad \quad 1\ 06h \quad \quad \quad 06 \\
 \textbf{CF=1} \quad \quad \textbf{(unsigned)} \quad \quad \quad \textbf{(hexa)} \quad \quad \quad \textbf{(signed)} \quad \textbf{OF = 0}
 \end{array}$$

byte + byte → byte ($147 + 115 = 6$!!!! – ADC) It follows that in the above example unsigned addition does not function correctly as an arithmetic operation in assembly language and that is why the processor reacts to this situation by setting CF=1 ! The signed interpretation addition functions correctly and that is why we have OF=0.

In such situations is advisable to take into account the value from CF using further ADC instructions if needed.

word + word → word dword + dword → dword
 B+B = B (but what if it doesn't fit ??...) B+B = W

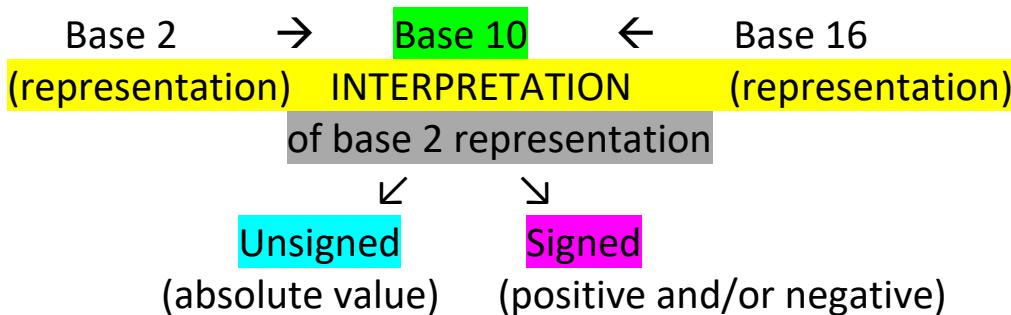
Both CF and OF are referring to addition and subtraction ONLY.

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

Base 2 – 1..... → the 2 interpretations (SIGNED and UNSIGNED) will be
ALWAYS DIFFERENT !!!!

Base 2 – 0..... → the 2 interpretations (SIGNED and UNSIGNED) will be
IDENTICAL !!!

- a). Express the number ab in each of the 2, 10 and 16 nummeration bases, in each of the two possible interpretations (signed and unsigned) 1001 0011b, 147, -109, 93h (4 values – NOT 6 !!!!!)
 - b). Express the number 73h in each of the 2, 10 and 16 nummeration bases, in each of the two possible interpretations (signed and unsigned)
 - c). Express the number 129 in each of the 2, 10 and 16 nummeration bases, in each of the two possible interpretations (signed and unsigned)
 - d). Express the number -37 in each of the 2, 10 and 16 nummeration bases, in each of the two possible interpretations (signed and unsigned)
- b,c,d – also 4 values each – NOT 6 !!!!!



CF and OF express overflow situations ONLY for ADDITION and SUBTRACTION !!!!

(The 2 flags OF and CF are set exactly for letting you come up with a correction if you want...)

- If set to 1, both of them are expressing INCORRECT MATHEMATICAL OPERATIONS !!! (That is why in fact they are set to 1 !)
- What about MULTIPLICATION and DIVISION ?

Op1 – m positions, op2 – n positions , the result of op1 * op2 will be represented on m+n positions. Fortunately, the assembly language IS ALWAYS providing multiplications with enough space for the result.

Multiplication in assembly language WILL NEVER issue a REAL overflow !!!

Anyway, CF and OF will have a role in case of multiplication

$B*b = \text{word}$; $b*b=b$ ($2*3=6$) $\rightarrow OF=CF=0$ (no “overflow”)

$B*b = w$ ($255 * 3 = \dots$) $\rightarrow OF=CF=1$ (“overflow”)

DIVISION ???

- WE DO HAVE overflow in case of division and IT IS THE WORST CASE OF ALL !!!

w/b = byte ; $1002/3 = 334$, but... 334 is a byte ??? No, it isn't and IT IS FATAL !!!
 \rightarrow RUN TIME ERROR Divide overflow, Zero Divide, Division by zero

Number/0 = Zero divide – this is a forbidden operation ! Why this is a forbidden operation ???

Mathematical analysis tells us that:

Number/(epsilon that approaches zero) = a result that approaches infinit !!! – this is a correct mathematical operations

Number/0 is assembly language will issue a DIVIDE OVERFLOW (ZERO DIVIDE, DIVISION by ZERO) because infinit doesn't fit a byte, a word, a doubleword

If I intend to make $1002/3 = \dots$ the assembly language will issue a DIVIDE OVERFLOW because 334 doesn't fit a byte. But from above the processor already considered that 3 messages equivalent... so, it still will issue the same things Why ? because from the mP point of view it the same...

From a technical point of view an INT 0 will be issued in ALL situations of this type !!

- **Do I really need 2 flags for overflow ?**
- What means IMUL and IDIV ? ...
- Where are IADD and ISUB ?? ... They do not exist because

**IADD = ADD, ISUB = SUB ... because they work exactly the same way in base 2.
Addition and subtraction are VALID - UNDER BOTH INTERPRETATIONS
SIMULTANEOUSLY !!!!**

When a base 2 addition or subtraction is performed, in fact 2 different simultaneously operations in base 10 are performed !! One for the unsigned interpretation (and that is why we need CF) and the other one for the signed interpretation (and that is why we need OF).

What values will be associated with CF and OF in case of DIVISION ??
In case of division CF and OF are UNDEFINED !!!

1 addition in base 2 = 2 different SIMULTANEOUS additions in base 10 !!

1 subtraction in base 2 = 2 different SIMULTANEOUS subtractions in base 10 !!

That is why CF and OF are NECESSARY to be TOGETHER present and associated with this operations

- Why DO I NEED IMUL AND IDIV ?

Because in contrast from addition and subtraction (which operate identically in base 2 independently of interpretation – signed or unsigned) multiplication and division function DIFFERENTLY ! Here there is mandatory to specify the interpretation of the operands (signed or unsigned) BEFORE the actual operation is performed... and the microprocessor distinguish that by using DIV vs IDIV and MUL vs. IMUL

In case of addition and subtraction there is no need to make such a distinction before performing it, so the decision regarding a possible interpretation (signed or unsigned) is taken only AFTER the actual operation is performed. That is why we do NOT need IADD or ISUB...

Examples - implicit rules for prefixing an offset with the corresponding segment register

(material discussed and developed together with you)

Mov eax, [ebx+esp] ; - ESP – base , EBX – index ;...SS

Mov eax, [esp + ebx] ; - ESP – base , EBX – index ;...SS

Mov eax, [ebx+esp*2] ; - syntactic error !!

Mov eax, [ebx+ebp*2] ; - ok ! ...DS

Mov eax, [ebx+ebp] ; ok ! ...DS

Mov eax, [ebp+ebx] ; ok ! ...SS

Mov eax, [ebx*2+ebp] ; ok ! - ...SS

Mov eax, [ebx*1+ebp] ; ...SS

Mov eax, [ebp*1+ebx] ; ...DS

Mov eax, [ebx*1+ebp*1] ; ok !...SS

Mov eax, [ebp*1+ebx*1] ; ... DS

Mov eax, [ebx*1+ebp*2] ; ?????

Mov eax, [ebp*1+ebx*2] ; ?????

Jmp et1 ; ...CS:et1...

Jmp eax;

Jmp [DS:et1] ; 4 bytes (short / NEAR jmp) will be taken from [DS:et1] and will be considered as a POINTER to which the jmp will be made IN THE SAME CODE SEGMENT... The jmp will be made to CS:[DS:et1]

JMP FAR [et1]

Jmp 5 ; syntax error because it does not follow the syntax : JMP label/register/memory address

- **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;
-

Examples - implicit rules for prefixing an offset with the corresponding segment register

(material prepared by me in advance) – I left them both here for being parsed and analyzed comparatively if it helps you...

Mov eax, [ebx+esp] ; ESP – base... EBX – index ;EAX ← ...SS:...

Mov eax, [esp + ebx] ; ESP – base... EBX – index ;EAX ← ...SS:...

Mov eax, [ebx+esp*2] ; syntactic error BECAUSE ESP can be ONLY a base register !

Mov eax, [ebx+ebp*2] ; mov eax, DWORD PTR [DS:EBX+EBP*2]

Mov eax, [ebx+ebp] ; ...DS...

Mov eax, [ebp+ebx] ; ...SS...

Mov eax, [ebx*2+ebp] ; ...SS...

Mov eax, [ebx*1+ebp] ;...SS...

Mov eax, [ebp*1+ebx] ; ...DS...

Mov eax, [ebx*1+ebp*1] ; ;...SS...

Mov eax, [ebp*1+ebx*1] ; ...DS...

Jmp et1 ; ...CS:et1...

Jmp [et1] ; JMP short [DS:0f6795B4] - I have to take 4 bytes as the needed correct offset to be referred to the current CS !!! JMP DWORD PTR [DS...] – to be performed at CS:the correct identified offset ; JMP CS:correct_offset (taken relatively to DS) will result usually in “Access violation” run-time error ! (to be checked by you!)

- I go in memory to the address DS:0f6795B4 , because of [] I will take THE CONTENTS from this address (for example 0BA2F5C4) and BECAUSE of JMP this contents will be THE TARGET OFFSET to which I (the processor) will perform this JMP (this offset being relative to the current CS). So, the JMP will be made to the address CS: 0BA2F5C4 !!!!

What you will be as programmers confronted within your checkings will be that DS=ES=SS=GS, a different value for CS and a different value for FS (due to the FLAT MEMORY MODEL).

Jmp 5 ; syntax error BECAUSE it does not obey the JMP syntax , 5 is not a **label**, nor a **register** and nor **a memory address** !!! - Relative call to absolute address not supported by OBJ format

- **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;

[eax+ebx] – indirect addressed operand ; [v] – direct addressed operand (the contents !!!)
V – is determinable at assembly time as an offset !

Bitwise operations and operators

Attention to the difference between operators and instructions !!!

Mov ah, 01110111 << 3 ; AH :=10111000b Vs.

Mov ah, 01110111

Shl ah, 3

& - bitwise AND operator	x AND 0 = 0	; x AND x = x
AND – instruction	x AND 1 = x	; x AND ~x = 0

Operation useful for FORCING THE VALUES OF CERTAIN BITS TO 0 !!!!

- bitwise OR operator	x OR 0 = x	; x OR x = x
OR – instruction	x OR 1 = 1	; x OR ~x = 1

Operation useful for setting the values of some bits to 1 !!!

^ - bitwise EXCLUSIVE OR operator;	x XOR 0 = x ;	x XOR x = 0
XOR – instruction	x XOR 1 = ~x;	x XOR ~x = 1

Operation useful for COMPLEMENTING the value of some bits !

XOR ax, ax ; AX=0 !!! = 00000000 0000000b

Reported Error types in Computer Science

- Syntax error – diagnosed by assembler/compiler !
- Run-time error (execution error) – program crashes – it stops executing
- Logical error = program runs until its end or remains blocked in an infinite loop ... if it functions until its end, it functions LOGICALLY WRONG obtaining totally different results/output than the envisioned ones
- Fatal: Linking Error !!! (for example in the case of a variable defined multiple times in a multimodule program ... if we have 17 modules, a variable must be defined ONLY in a SINGLE module ! If it is defined in 2 or more modules , a “Fatal: Linking Error !!! – Duplicate definition for symbol” Will be obtained.

-]

iRegisters – storage capacities – very small in terms of sizes (8,16, 32 or 64 bits) but very fast as access speed used for temporary store the operands (data, commands codes, ADDRESSES !!!!!!!) with which a processor currently.

3654

76532

1234876453

7464646464098234098209842083490238409283048283428342438

Computer/processor “on N bits”:

- a). software perspective – the size of the memory word = the size of the (majority of) the registers (in our case = 32 bits)
- b).engineering perspective – the size of the communication buses (channels)
- ABUS, CBUS, DBUS

A[i] ; I = INDEX

A[7] = *(A+7) * = the DEREFERENCING operator in C

A – the name of an array in C is its starting address – it's a pointer = THE BASE (the starting address)

Byte + byte (ADD) = byte ;

MULTIPLY op1(M positions) * op2 (N positions) → M+N positions

B*B → W(ord); W*W → DW ; DW*DW → QW (EDX:EAX)

(DX:AX)

Data structures – array, list, queue (FIFO), stack (LIFO)

WHY is the stack SO important ?????

RUN-TIME Mechanism of ANY program in Computer Science FOLLOWS ALWAYS THE LIFO ORDER of activating and running the involved programming units (subroutines = functions + procedures).

A user defined type in C is defined by TYPEDEF (which is INCORRECT, because typedef is in fact defining only the structure)

C, Java, VB, Pascal, Fortran – were IMPERATIVE language , because they rely as a central element on the INSTRUCTIONS (commands).

DATA TYPE = structure + associated OPERATIONS !!!

(essential in this definition is ASSOCIATED – we did not have until OOP AN ENCAPSULATION mechanism)

- You also have in OOP inheritance + polymorphism;

OOP = DATA ORIENTED PROGRAMMING (everything is built having as the central figure the notion of DATA).

From the point of view of the mP – which is its understanding of DATA TYPE notion ?

DATA TYPE for the mP (and in Assembly language) – The size of representation of that element;

On 32 bits these can be – byte, word, dword and qword (these are the assembly language DATA TYPES); You can define variables/operands in the RAM memory by using DATA DEFINITION DIRECTIVES – DB, DW, DD, DQ.

=====

RAM (Random Access Memory) – who is RANDOM ?

- The access time at any given location from the RAM is THE SAME independently of the position (randomely far from the beginning of the memory...)
- In contrast from ROM (read only memories) a RAM supports/allows any number of R/W and in any ORDER (Randomely... reads and writes in a randomely order... The order in which R/W appear is RANDOM...)

=====

101101100101 – representation (in base 2) of what do I need as a base 10 value as a user/human being

So, we need a TRANSFORMATION in base 10 !!!

REPRESENTATION (in base 2) vs. INTERPRETATION (in base 10) !!!!!!!

UNSIGNED vs SIGNED – how can we REPRESENT them ?

The answer – develop 2's complement representation !!!!!

Contents of a computer memory ---→ correct and consistent INTERPRETATION in base 10 !!!!!

Base 2 – 1..... → the 2 interpretations (SIGNED and UNSIGNED) will be ALWAYS DIFFERENT !!!!

Base 2 – 0..... → the 2 interpretations (SIGNED and UNSIGNED) will be IDENTICAL !!!!

CHAPTER 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 msrvct.dll ;printf is a standard C function from msrvct.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 variables are declared here (the segment is called data)  
string: db "Hello from ASMI!", 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 which these instructions work, by using an equivalent sequence of MOV and ADD or SUB instructions:

push eax	\Leftrightarrow	sub esp, 4 ; prepare (allocate) space in order to store the value mov [esp], eax ; store the value in the allocated space
pop eax	\Leftrightarrow	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 cause 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 \leftarrow < 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 <i>general_reg, contents of a memory_operand</i>	<i>general_reg</i> \leftarrow offset(<i>mem_operand</i>)	-
---	--	---

LEA (Load Effective Address) 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.

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).

$$\begin{array}{r} 1\ 0000\ 0000 - \\ 1001\ 0011 \\ \hline 01101101 \end{array} = 6Dh = 96+13 = 109 \text{ (so the 2'complement on 8 bits of 147 is 109)}$$

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} values = 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 $-147 \notin [-256..+255]$ 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, $512-147 = 365 = 1\ 6Dh = 1\ 0110\ 1101 \dots$)

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

$1\ 0110\ 1101 = -(2\text{'s complement of } 1\ 0110\ 1101) = -(0\ 1001\ 0011) = -(093h) = -147$

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

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
 - o (a) 64K for 16-bit processors
 - o (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:

S₃S₂S₁S₀ : 0706050403020100

In this case, the selector s₃s₂s₁s₀ shows a segment access which has the base address as b₇b₆b₅b₄b₃b₂b₁b₀ and a limit l₇l₆l₅l₄l₃l₂l₁l₀. 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:

$$0706050403020100 \leq l_7l_6l_5l_4l_3l_2l_1l_0.$$

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

$$a_7a_6a_5a_4a_3a_2a_1a_0 := b_7b_6b_5b_4b_3b_2b_1b_0 + 0706050403020100$$

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 $s_3s_2s_1s_0 : 0706050403020100$ will result in the following linear address:

$$a_7a_6a_5a_4a_3a_2a_1a_0 := 00000000 + 0706050403020100$$

$$a_7a_6a_5a_4a_3a_2a_1a_0 := 0706050403020100 \\ \Rightarrow$$

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

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

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

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

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

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

The x86 architecture allows 4 types of segments:

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

Every program is composed by one or more segments of one or more of the above specified types. At any given moment during run time there is only at most one 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*):

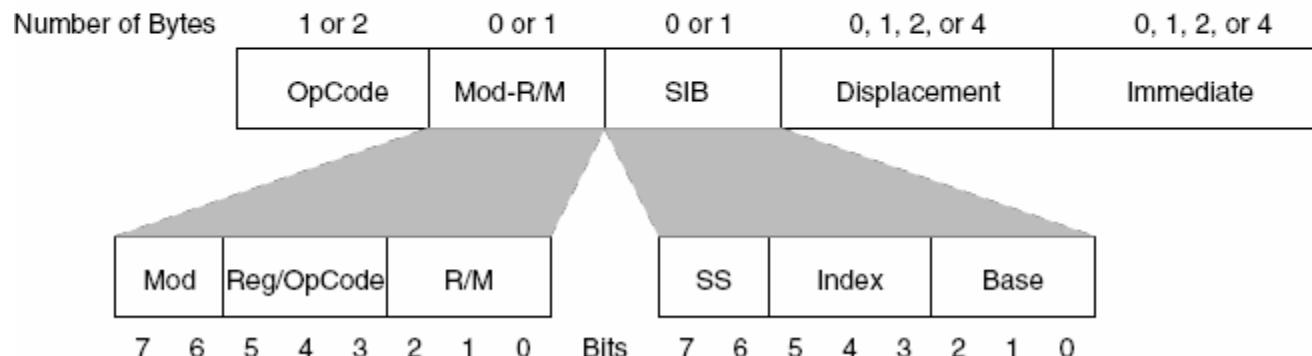
[prefixes] + code + [ModeR/M] + [SIB] + [displacement] + [immediate]

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.

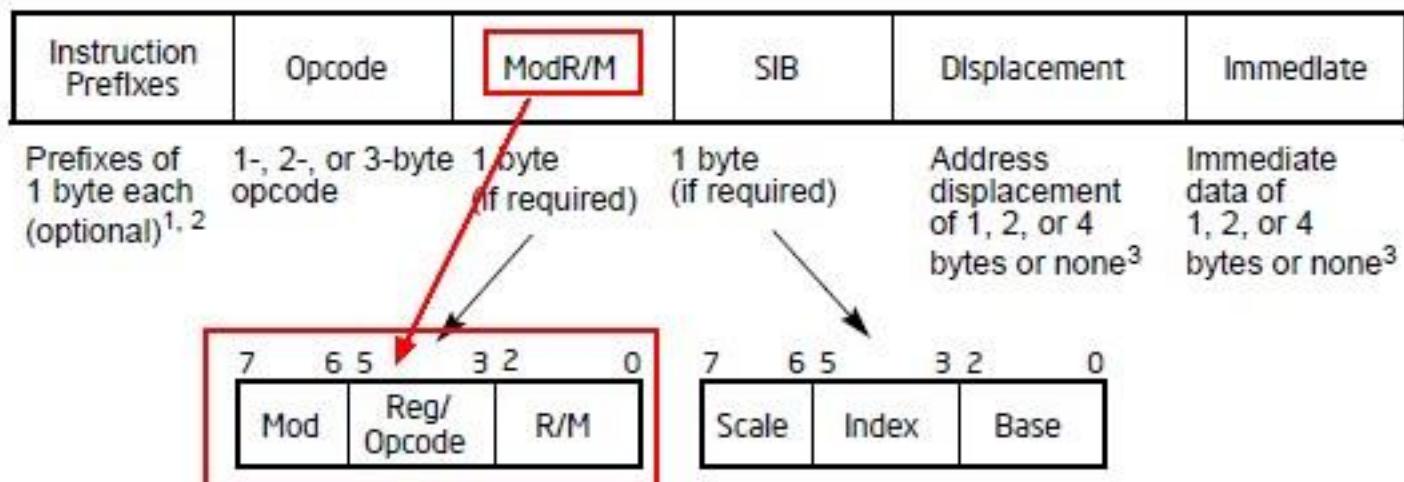
Number of Bytes	0 or 1	0 or 1	0 or 1	0 or 1
	Instruction prefix	Address-size prefix	Operand-size prefix	Segment override

(a) Optional instruction prefixes



(b) General instruction format

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>):

$$\begin{array}{c} [\text{base}] + [\text{index} \times \text{scale}] + [\text{constant}] \\ (\text{SIB}) \qquad \qquad \qquad (\text{displacement+immediate}) \end{array}$$

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:

- $s_3s_2s_1s_0 : \text{offset_specification}$ 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:

$$\text{offset_address} = [\text{base}] + [\text{index} \times \text{scale}] + [\text{constant}]$$

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

CHAPTER 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 *mnenomics*: *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:

$$[\text{base_register} + \text{index_register} * \text{scale} + \text{constant}]$$

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 symmetry to „-“): $+X = X$
7	~	unary, prefix	One's complement: <code>mov al, ~0 => mov AL, 0xFF</code>
7	!	unary, prefix	Logic negation: $!X = 0$ when $X \neq 0$, 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: $-24 // 4 // 2 = -3$ ($-24 / 4 / 2 \neq -3!$)
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	<code><<</code>	Binary, infix	Bitwise left shift: $1 << 4 = 16$
4	<code>>></code>	Binary, infix	Bitwise right shift: $0xFE >> 4 = 0x0F$
3	<code>&</code>	Binary, infix	AND: $0xF00F \& 0x0FF6 = 0x0006$
2	<code>^</code>	Binary, infix	Exclusive OR: $0xFF0F ^ 0xFOFF = 0x0FF0$
1	<code> </code>	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
<code>~</code>	<code>~ expresie</code>	Bits complement
<code>&</code>	<code>expr1 & expr2</code>	Bitwise AND
<code> </code>	<code>expr1 expr2</code>	Bitwise OR
<code>^</code>	<code>expr1 ^ expr2</code>	Bitwise XOR

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

<code>~11110000b</code>	<code>; output result is 0000111b</code>
<code>01010101b & 11110000b</code>	<code>; output result is 01010000b</code>
<code>01010101b 11110000b</code>	<code>; output result is 11110101b</code>
<code>01010101b ^ 11110000b</code>	<code>; output result is 10100101b</code>
<code>! – logical negation (similar with C) ; !0 = 1 ; !(anything different from zero) = 0</code>	

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 continuous 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_attributes) = 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 accessed 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)

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

$$\begin{array}{r} 1001\ 0011 + \quad 147 + \\ 0111\ 0011 \quad \underline{115} \quad \text{there is transport and CF is set therefore to 1} \\ \hline \textcolor{red}{1}\ 0000\ 0110 \quad (=6?) \end{array} \qquad \begin{array}{r} 93h + \quad -109 + \\ \underline{73h} \quad \underline{115} \\ 106h \quad 06 \end{array}$$

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 +
<u>1011 0011</u>	<u>179</u>	a carry of the most significant digit occurs	<u>B3h</u>
1 0100 0110	326	so the value 1 is placed in CF	146h
(unsigned)			(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 +
<u>1011 0011</u>	<u>179</u>	a carry of the most significant digit occurs	<u>B3h</u>
1 0100 0110	70	so the value 1 is placed in CF	146h
(unsigned)			(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 +
<u>0111 0011</u>	<u>115</u>	a carry DOES NOT occur so CF=0	<u>73h</u>
1100 0110	198		C6h
(unsigned)			(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 $b+b \rightarrow b$, so what we obtain as interpretations on 1 byte are :

$$\begin{array}{r} 0101\ 0011 \\ 0111\ 0011 \\ \hline 1100\ 0110 \end{array} + \begin{array}{r} 83 \\ 115 \\ \hline 198 \end{array}$$

(unsigned)
CF=0

$$\begin{array}{r} 53h \\ 73h \\ \hline C6h \end{array} + \begin{array}{r} 83 \\ 115 \\ \hline -58 \end{array} !!!!$$

(hexa)
(signed)
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:

$$\begin{array}{ll} 0.....+ & \text{or} \\ 0..... & 1....+ \\ \hline \text{-----} & \text{-----} \\ 1..... & 0..... \end{array} \quad \begin{array}{l} (\text{Semantically, the two situations denote the impossibility of mathematical acceptance}) \\ (\text{of the 2 operations: we cannot add two positive numbers and obtain a negative result}) \\ (\text{and we cannot add two negative numbers and obtain a positive result}). \end{array}$$

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

$$\begin{array}{ll} 1.....- & \text{sau} \\ 0..... & 1....- \\ \hline \text{-----} & \text{-----} \\ 0..... & 1..... \end{array} \quad \begin{array}{l} (\text{Semantically, the two situations denote the impossibility of mathematical acceptance}) \\ (\text{of the two operations: we cannot subtract a positive number from a negative one and}) \\ (\text{obtaining a positive one, neither can we subtract a negative number from a positive one}) \\ (\text{and obtaining a negative one}). \end{array}$$

1	0110 0010 -	98 -	62h -	98 -
	<u>1100 1000</u>	<u>200</u>	We need significant digit borrowing for performing	<u>C8h</u>
	1001 1010	154	the subtraction, therefore CF is set to 1	9Ah
				(hexa)
				(signed)
				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’.

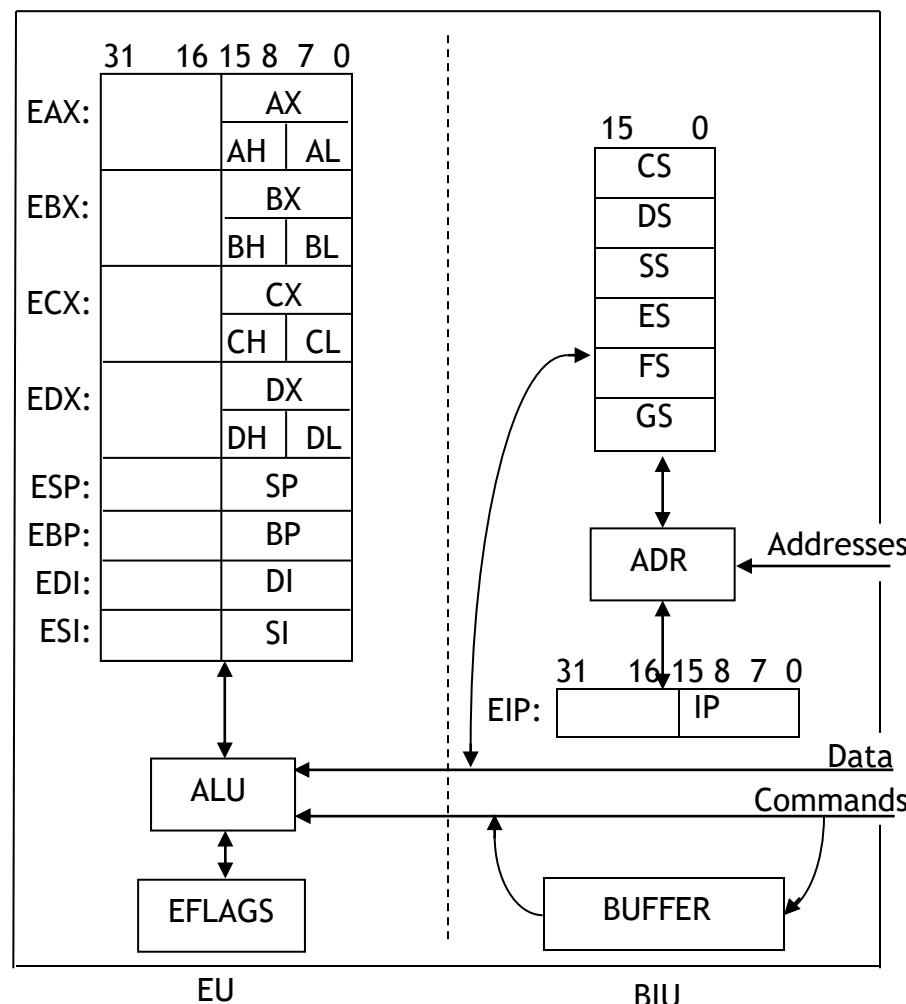
In the case of a correct division CF and OF are undefined. If we have a division overflow, the program crashes, the execution stops and of course it doesn’t matter which are the values from CF and OF...

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 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 instructions – 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

$$\begin{array}{r} 1001\ 0011 + 147 + \\ 0111\ 0011 \quad \underline{115} \quad \text{there is transport and CF is set therefore to 1} \\ \hline \textcolor{red}{1}\ 0000\ 0110 \quad 262 \end{array} \qquad \begin{array}{r} 93h + -109 + \\ \underline{73h} \quad \underline{115} \\ 106h \quad 06 \end{array}$$

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 programmer, 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 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 !!

„,Location Counter

Segment data

a db 17, -2, 0ffh, ‘xyz’,...

db

db....

;lga db \$-a (mov [lga],....) ok – lga is a variable

;lga EQU \$-a (mov [lga],... - illegal !!!) – lga is a CONSTANT

;lga dw \$-\$\$; ok ! – IF a is THE FIRST identifier to be defined in the data segment !!!!

;lga dw lga-a ; !!!!!!

b EQU 27

c dd 12345678h

lg dw b-a ; NO !!! b is NOT an address !!! syntax error

lg db c-a ; OK !!!

lga dw \$-a-4 ; ok !!!

lg dw \$-a ; length (a) + 4 !!!

If no SECTION directive is explicitly used, The \$\$ symbol will evaluate implicitly to the offset of the beginning of the current segment.

“:” are mandatory to be present when DEFINING CODE labels (ex: “start:”) but they must be absent when defining DATA labels (ex: defining variables “a db 17”)

The format of a source line isn’t specific only to the code segment, but is general applicable for any source line independently of the type of that segment (inclusively a data segment)

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

The offset of any label is a constant value determinable at assembly time. In any programming language the location of an allocated variable (its address) remain fixed; that is why the offsets of variables represents constant values determinable at assembly/compile time.

The SEGMENT address is also fixed but determinable ONLY at loading time.

Any offset used only by itself in a program (without the segment part) will be finally completed BY THE ASSEMBLER to a FAR address by prefixing it with a corresponding segment value. This IMPLICIT value will be always one of the CS, DS or SS segment registers and the rules for these implicit associations 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;

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 obtainsyntax 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 cunoasuta 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 incepaturilor 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 diferență între numele de variabile (al căror offset poate fi determinat la assembleare) și numele de segmente (al căror offset NU poate fi determinat la momentul assemblearii ca și o constantă, aceasta valoare fiind cunoscută la fel ca și adresa de segment doar la momentul încarcării programului pt execuție - loading time).

Dacă avem mai mulți operanți pointeri, assemblerul va încerca să incadreze expresia într-o combinație validă de operații cu pointeri:

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

...însă nu trebuie să uităm că fiind vorba despre ADRESARE INDIRECTĂ rezultatul final al expresiei cu operanți pointeri din paranteze trebuie să furnizeze în final UN POINTER !!!... din aceasta cauză în cazul adresării indirecte nu vor fi niciodată acceptate expresii cu operanți adrese de formă “a+b”

Dacă nu, vom obține eroarea de sintaxă “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 directă** vs. **indirectă** este mai permisivă ca operații aritmetice în NASM (din cauză acceptarea expresiilor de tip “a+b”) însă asta nu înseamnă că este mai permisivă IN OPERAȚIILE CU POINTERI !!!

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

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] = ??

Data definition directives (discussed together)

Always your data segment starts at offset 00401000 in OllyDbg

Segment data

a1 db 0,1,2,'xyz' ; 00 01 02 'x' 'y' 'z' ; offset(a1- determinat la incarcarea lui OllyDbg)=00401000; offset(a1) determinat la asamblare de catre NASM = 0 !!!

78 79 7A

db 300, "F"+3 ; 2C ascii code for 'F' + 3 49

a2 TIMES 3 db 44h ; 44 44 44

a3 TIMES 11 db 5,1,3 ; 05 01 03...11 times (total 33 bytes)

a41 db a2+1 – syntax error !!!

a4 dw a2+1, 'bc' ; offset(a2)=00401008h; a2+1=1009h; 09 00?

a44 dw 1009h ; 09 10 (correct, BUT... this particular 10h value it is only finally computable after LOADING !!!)

a5 dd a2+1, 'bcd' ; 09 10 40 00| 62 63 64 00

a6 TIMES 4 db '13' ; equiv with a6 TIMES 4 db '1','3' ; 31 33 31 33 31 33 31 33

a6bis TIMES 4 dw '13' ; 31 33 31 33 31 33 31 33 !!!!!!!!!!!!!!!

d dd [eax] ; syntax error !!

d dd eax ; syntax error !

a7 db a2 ; Syntax error !! Why ? (similar mov ah,a2)

a8 dw a2 ; 08 10

a9 dd a2 ; 08 10 40 00

a10 dq a2 ; 08 10 40 00 00 00 00 00

a11 db [a2] ; syntax error ! A CONTENTS of a memory area or the contents of a registers are NOT values accessible at assembly time ! These are accessible ONLY at run-time !!

a12 dw [a2] ; syntax error !

a13 dd dword [a2] ; syntax error !

a14 dq [a2] ; syntax error !

mov ax, v ; Warning...

Data definition directives (prepared by me in advance...)

Always your data segment starts at offset 00401000 in OllyDbg

Segment data

a1 db 0,1,2,’xyz’ ; 00 01 02 ‘x’ ‘y’ ‘z’ (ascii codes for these chars)

78 79 7A

db 300, “F”+3 ; 2C ‘ascii code for F + 3”; Warning – byte data (300) exceeds bounds !

a2 TIMES 3 db 44h ; 44 44 44

a3 TIMES 11 db 5,1,3 ; 05 01 03... 11 times (33 bytes)

a4 dw a2+1, ‘bc’ ; offset a2 = 00401008 ; 09 10 ‘b’ ‘c’

a5 dd a2+1, ‘bcd’ ; 09 10 40 00 62 63 64 00

a6 TIMES 4 db ‘13’ ; 31 33 31 33 31 33 31 33

a6bis TIMES 4 dw ‘13’ ; 31 33 31 33 31 33 31 33

a7 db a2 ; syntax err. – OBJ format can only handle 16 or 32 bits relocation (equiv. mov ah,a2)

a8 dw a2 ; 08 10

a9 dd a2 ; 08 10 40 00

a10 dq a2 ; 08 10 40 00 00 00 00 00

a11 db [a2] ; expression syntax error !

a12 dw [a2] ; expression syntax error !

a13 dd dword [a2] ; expression syntax error !

a14 dq [a2] ; expression syntax error !

mov ax, v ; Warning – 32 bit offset in 16 bit field !

The steps followed by a program from source code to run-time:

- Syntactic checking (done by assembler/compiler/interpreter)
- OBJ files are generated by the assembler/compiler
- Linking phase (performed by a LINKER = a tool provided by the OS, which checks the possible DEPENDENCIES between this OBJ files/modules); The result → .EXE file !!!
- You (the user) are activating your exe file by clicking or enter-ing...
- The LOADER of the OS is looking for the required RAM memory space for your EXE file. When finding it, it loads the EXE file AND performs ADDRESS RELOCATION !!!!
- In the end the loader gives control to the processor by specifying THE PROGRAM’s ENTRY POINT (ex: the start label) !!! The run-time phase begins NOW...

Segment code (starts always at offset 00402000 – WHO DECIDES THAT ?)

The linker makes such decisions. The base address for loading PEs, at least the default one (set by the Microsoft linkeditor and not only) is 0x400000 for executables (respectively 0x10000000 for libraries). Alink complies with this convention and fills in the ImageBase field of the IMAGE_OPTIONAL_HEADER structure in the P.E. newly built value 0x400000. As each “segment” / section of the program can provide different access rights (the code is executable, we can have read-only segments etc ...), these are planned to start each one at the address of a new memory page (4KiB, so multiple of 0x1000), each memory page can be configured with specific rights by the program loader. In the case of small programs, the implication is that you will get the following map of the program in memory (at run time):

- the program is planned to be loaded into memory at exactly the address 0x400000 (but here will reach the metadata structures of the file, not the code or the data of the program itself)
- the first "segment" will be loaded at 0x401000 (quotation marks because it is not a segment itself but only a logical division of the program, the "segment" of a segment register is not directly associated - for this reason the name is often preferred section instead of segment)
- the second "segment" will be loaded at 0x402000 (the segment that the processor will use for segmentation starts at address 0 and has a limit of 4GiB, regardless of the addresses and section sizes)
- will be prepared "segment" (section) of imports, "segment" of exports and "segment" of stack in the order decided by the likeditor (and of dimensions also provided by him), segments that will be loaded from 0x403000, 0x404000 and so on (increments of 0x1000 as long as they are small enough, otherwise need to be used for increment the smallest multiple of 0x1000 which allows enough space for the contents of the entire segment)

According to the decision logic of the start addresses of the sections, we can conclude that here we have a section (probably data) before the code, containing less than 0x1000 bytes, which is why the code starts immediately after, from 0x402000, the program map being at the end : metadata (headers) from 0x400000, data to 0x401000 and code to 0x402000 (followed of course by other "segments" for stack, imports and, optionally, exports).

Linkeditorul ia decizile de acest tip. Adresa de baza pentru incarcarea PE-urilor, cel putin cea implicita (setata de catre linkeditorul de la Microsoft si nu numai) este 0x400000 in cazul executabilelor (respectiv 0x10000000 pentru biblioteci). Alink respecta aceasta convenie si completeaza in campul ImageBase al structurii IMAGE_OPTIONAL_HEADER din fisierul P.E. nou construit valoarea 0x400000. Cum fiecare “segment”/sectiune din program poate prevedea drepturi diferite de acces (codul este executabil, putem avea segmente read-only etc...), acestea sunt planificate sa inceapa fiecare la adresa cate unei noi pagini de memorie (4KiB, deci multiplu de 0x1000), fiecare pagina de memorie putand fi configurata cu drepturi specifice de catre incarcatorul de programe. In cazul unor programe de mici dimensiuni, implicatia este ca se va obtine urmatoarea harta a programului in memorie (la executare):

- programul este planificat a fi incarcat in memorie la exact adresa 0x400000 (insa aici vor ajunge structurile de metadate ale fisierului, nu codul sau datele programului in sine)
- primul “segment” va fi incarcat la 0x401000 (pun ghilimele deoarece nu este un segment propriu-zis ci doar o diviziune logica a programului, nu este asociat direct “segmentul” unui regisztr de segment – din aceasta pricina se prefera de multe ori denumirea de sectiune in loc de cea de segment)
- al doilea “segment” va fi incarcat la 0x402000 (segmentul pe care il va folosi procesorul pentru segmentare incepe la adresa 0 si are limita de 4GiB, indiferent de adresele si dimensiunile sectiunilor)
- va fi pregatit “segment” (sectiune) de importuri, “segment” de exporturi si “segment” de stack in ordinea decisa de catre likeditor (si de dimensiuni prevazute tot de catre acesta), sectiuni ce vor fi incarcate de la 0x403000, 0x404000 si asa mai departe (incremente de 0x1000 cat timp au dimensiune suficient de mica, in caz contrar fiind nevoie a se folosi pentru increment cel mai mic multiplu de 0x1000 care permite suficient spatiu pentru continutul intregului segment)

Conform logicii de decizie a adreselor de inceput ale sectiunilor, putem concluziona ca aici avem o sectiune (de date probabil) inaintea celei de cod, continand sub 0x1000 octeti, motiv pentru care codul porneste imediat dupa, de la 0x402000, harta programului fiind la final: metadate (antete) de la 0x400000, date la 0x401000 si cod la 0x402000 (urmat bineintele de alte “sectiuni” pentru stiva, importuri si, optional, exporturi).

Segment code (starts always at offset 00402000)

Start:

```
Jmp Real_start  (2 bytes) - 00402000
a db 17           - 00402002
b dw 1234h       - 00402003
c dd 12345678h  - 00402005
```

Real_start:

.....

```
Mov eax, c ; eax = 402005
```

Mov edx, [c] ; mov edx, DWORD PTR DS:[402005]; in mod normal asta inseamna ca in EDX will be assigned with the doubleword of offset 00402005 taken from DS !!!!

.....

```
Mov edx, [CS:c] ; mov edx, DWORD PTR CS:[402005]
```

```
Mov edx, [DS:c] ; mov edx, DWORD PTR DS:[402005]
```

```
Mov edx, [SS:c] ; mov edx, DWORD PTR SS:[402005]
```

```
Mov edx, [ES:c] ; mov edx, DWORD PTR ES:[402005]
```

The output will be in all of the 5 cases the same **EDX:=12345678h** WHY ??

The explanation is directly related to the **flat memory model** - all segments actually describe the entire memory, from 0 to the end of the first 4GiB of memory. As such, [CS: c] or [DS: c] or [SS: c] or [ES: c] will access the same memory location but with different access rights. Although all selectors indicate identical segments in address and size, they may differ in how other control and access fields of the segment descriptors indicated by them are completed.

The flat model assures us that the segmentation mechanism is transparent to us, we do not notice differences between segments and, as such, we completely get rid of the segmentation worry, but we are interested in the logical division into segments of the program, which is why we use separate **sections** / "segments". for data code). This is true but only as long as we limit ourselves to CS / DS / ES and SS! The FS and GS selectors point to special segments that do not follow the flat pattern (reserved for the interaction of the program with the S.O.), more precisely, [FS: c] does not indicate the same memory as [CS: c]!

Explicatia este direct legata de **modelul de memorie flat** – toate segmentele descriu in realitate intreaga memorie, incepand de la 0 si pana la capatul primilor 4GiB ai memoriei. Ca atare, [CS:c] sau [DS:c] sau [SS:c] sau [ES:c] vor accesa aceeasi locatie de memorie insa cu drepturi de acces potential diferite. Desi toti selectorii indica segmente identice ca adresa si dimensiune, acestia pot avea diferente in cum le sunt completate alte campuri de control si de acces ale descriptorilor de segment indicati de catre ei.

Modelul flat ne asigura ca mecanismul de segmentare este transparent pentru noi, noi nu sesizam diferente intre segmente si, ca atare, scapam complet de grija segmentarii (insa ne intereseaza impartirea logica in segmente a programului, motiv pentru care folosim sectiuni/"segmente" separate pentru cod date). Acest lucru este valabil insa doar cat timp ne limitam la CS/DS/ES si SS! Selectorii FS si GS indica inspre segmente speciale care nu respecta modelul flat (rezervate interactiunii programului cu S.O-ul), mai precis, [FS:c] sau [GS:c] NU indica aceeasi memorie ca si [CS:c]!

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 $2^{32} = 4\text{GB}$ of memory can be addressed using this memory model. The operating system then translates these linear addresses to physical addresses using paging schemes.

Call code (THE CALLER):

- a). Saving the volatile resources (EAX, ECX, EDX, EFLAGS)
- b). Passing parameters
- c). Saving the returning address and performing the call

Entry code (THE CALLEE – called subroutine):

- a). Building the new stackframe PUSH EBP,
 MOV EBP, ESP
- b). Allocating space for local variables SUB ESP, nr_bytes
- c). Saving non-volatile resources exposed to be modified

Exit code (THE CALLEE):

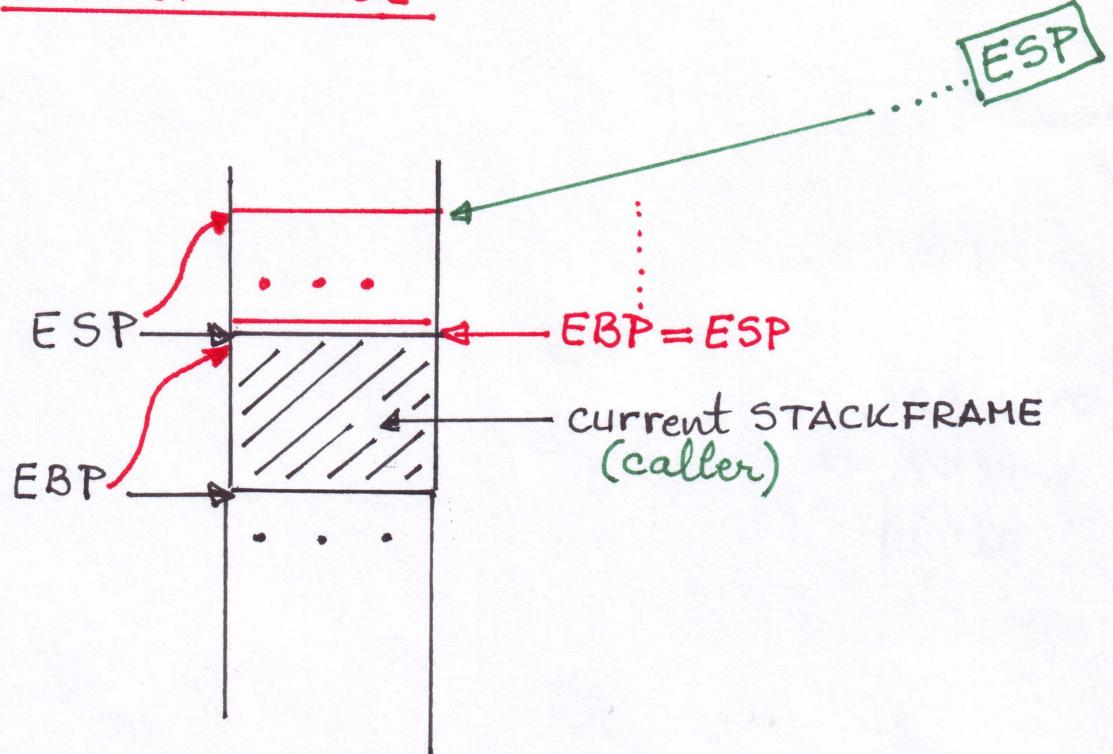
- a). Restoring non-volatile resources
- b). Freeing the space allocated for local variables [ADD ESP, nr_bytes_locals] – mentioned here just as a reverse for the above b) from the entry code, but not really necessary because deallocating the stackframe (mov esp, ebp) includes this action anyway from a practically point of view.
- c). Deallocating the stackframe MOV ESP, EBP (if we know exactly the size of the stackframe , ADD ESP, sizeof(stackframe) solves similarly...) and restoring the base of the POP EBP caller stackframe (old EBP) (a, b c – the reverse of the entry code)
- d). Returning from the subroutine (RET) and deallocating passed parameters (if we have a STDCALL function) - (reverse of b + c from the call code)

It is still to be done the reverse of a) from call code. It is the task of the CALLER to do it together with a possible parameters take out from the stack (if it is a CDECL function).

Responsabilities for generating the call code, entry code and exit code

		Function/proc call	{	}
CALLER	CALLEE	Call code	Entry code	Exit code
C	C	C compiler	C compiler	C compiler
C	asm	C compiler	ASM programmer	ASM programmer
asm	C	ASM programmer	C compiler	C compiler
asm	asm	Call (saving the return address)	NOTHING mandatory	RET (grab the returning address and jmp)

ENTRY PHASE



- involves the creation of a NEW STACKFRAME for the CALLED subroutine:

push EBP ; for restoring the base of the CURRENT STACKFRAME when returning

MOV EBP,ESP ; This is the BIRTH of the NEW STACKFRAME (sizeof initial = 0)
... - - - - - [ESP] will start to "grow" by currently performed PUSHES

Volatile vs. non-volatile resources

- **volatile** resources are represented by those registers that **the calling convention is defining them as belonging to the called subroutine**, thus, the caller being responsible **as part of the call code** to save their values (if the called subroutine is using them) and after that, at the end of the call to restore the initial (old) values. So: who is saving the volatile resources ? **The caller** (as part of the call code) . Who is restoring in the end those values ? Also **the caller but NOT as part of a certain call/entry or exit code**. Just restore them after the call in the regular code as a mandatory responsibility.
- **non-volatile** resources are any memory addresses or registers which do not belong explicitly to the called subroutine, but if this one needs to modify those resources, it is necessary that the called subroutine to save them at the entry as part of the entry code and restore them back at exit, as part of the exit code. So: who is saving the non-volatile resources ? **The callee** (apelatul = the **called** subroutine, as part of the entry code) . Who is restoring in the end these values ? Also **the callee** (as part of the exit code).

MNEMONICA	SEMNIFICATIE (salt dacă..<<relație>>)	Condiția 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 și 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 \neq 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 \neq OF
JG JNLE	este mai mare decât nu este mai mic sau egal	ZF=0 și 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 depășire	OF=1
JNO	nu există depășire	OF=0

Tabelul 4.1. Instrucțiunile de salt condiționat

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 := offset 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)

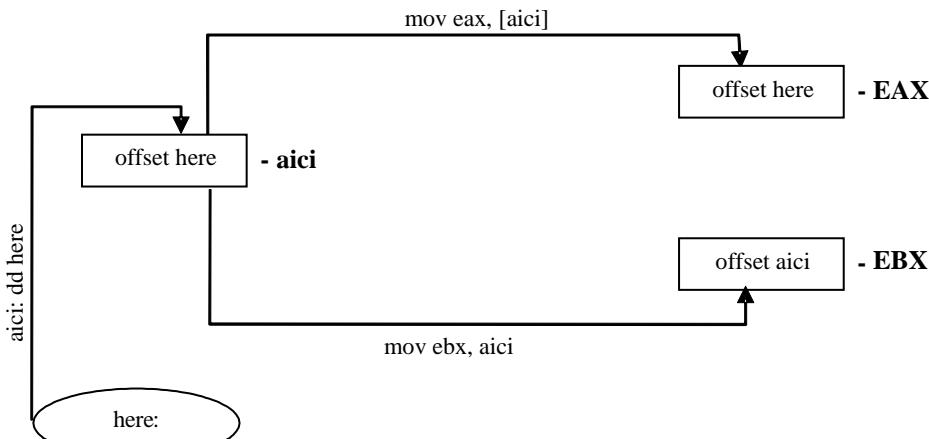


Fig. 4.4. Initializing variable aici and registers EAX and EBX.

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**

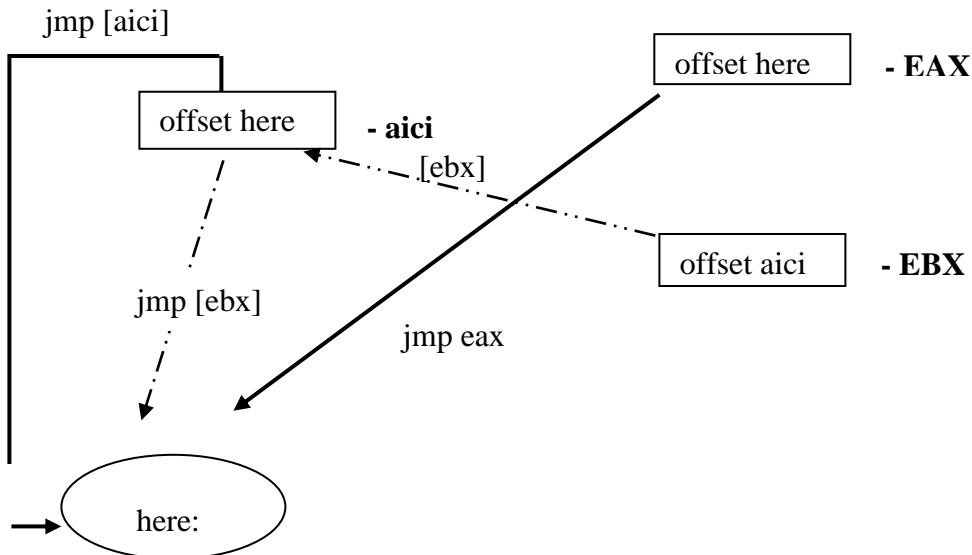


Fig. 4.5. Alternative ways for performing jumps to the label *here*.

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/specify/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 program: [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

36h = SS

3Eh = DS

26h = ES

64h = FS

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